Strong Connectivity in Directed Hypergraphs and its Application to the Atomic Decomposition of Ontologies

Francisco José Martín-Recuerda Moyano

E.T.S de Ingenieros Informáticos
Universidad Politécnica de Madrid (UPM)

This dissertation is submitted for the degree of
Doctor of Informatics

Departamento de Inteligencia
Artificial

February 2016

Title: *“Strong Connectivity in Directed Hypergraphs and its Application to the Atomic Decomposition of Ontologies”*

Author: Francisco José Martín-Recuerda Moyano

Supervisor: Prof. Asunción Gómez Pérez

Supervisor: Dr. Dirk Walther

Tribunal nombrado por el Sr. Rector Magnífico de la Universidad Politécnica de Madrid,
el día 18 de Enero de 2016.

Presidente: *D.^a María Nieves Rodríguez Brisaboa*

Vocal: *D. Daniel Borrajo Millán*

Vocal: *D.^a María Jesús Taboada Iglesias*

Vocal: *D.^a María de los Santos Pérez Hernández*

Secretario: *D. Mariano Fernández López*

Suplente: *D. Rafael Berlanga Llavorí*

Suplente: *D.^a María Belen Díaz Agudo*

Realizado el acto de defensa y lectura de la Tesis el día 05 de 02 de 2016 en la Escuela
Técnica Superior de Ingenieros Informáticos.

Calificación: _____

EL PRESIDENTE

VOCAL 1

VOCAL 2

VOCAL 3

SECRETARIO

To my family and to all who loves to fight for *lost causes* ...
(A mi familia y a los defensores de las *causas perdidas* ...)

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and acknowledgements.

Francisco José Martín-Recuerda Moyano

February 2016

Acknowledgements

I gratefully acknowledge the advice and help of my supervisors, Asunción Gómez-Pérez and Dirk Walther. Without their assistance the work embodied in this thesis would not have been possible.

Thanks are also due to Ernesto Jiménez-Ruiz, Ignazio Palmisano and Chiara del Vescovo for their support on how to compute syntactic locality-based modules and the atomic decomposition of OWL ontologies.

Finally, I would like to thank all people who provide support during this research.

Abstract

Directed hypergraphs are an intuitive modelling formalism that have been used in problems related to propositional logic, relational databases, computational linguistic and machine learning. Directed hypergraphs are also presented as an alternative to directed (bipartite) graphs to facilitate the study of the interactions between components of complex systems that cannot naturally be modelled as binary relations. In this context, they are known as hyper-networks.

A directed hypergraph is a generalization of a directed graph suitable for representing many-to-many relationships. While an edge in a directed graph defines a relation between two nodes of the graph, a hyperedge in a directed hypergraph defines a relation between two sets of nodes. Strong-connectivity is an equivalence relation that induces a partition of the set of nodes of a directed hypergraph into strongly-connected components. These components can be collapsed into single nodes. As result, the size of the original hypergraph can significantly be reduced if the strongly-connected components have many nodes. This approach might contribute to better understand how the nodes of a hypergraph are connected, in particular when the hypergraphs are large. In the case of directed graphs, there are efficient algorithms that can be used to compute the strongly-connected components of large graphs. For instance, it has been shown that the macroscopic structure of the World Wide Web can be represented as a “*bow-tie*” diagram where more than 70% of the nodes are distributed into three large sets and one of these sets is a large strongly-connected component. This particular structure has been also observed in complex networks in other fields such as, e.g., biology. Similar studies cannot be conducted in a directed hypergraph because there does not exist any algorithm for computing the strongly-connected components of the hypergraph.

In this thesis, we investigate ways to compute the strongly-connected components of directed hypergraphs. We present two new algorithms and we show their correctness and computational complexity. One of these algorithms is inspired by Tarjan’s algorithm for directed graphs. The second algorithm follows a simple approach to compute the strongly-connected components. This approach is based on the fact that two nodes of a graph that are strongly-connected can also reach the same nodes. In other words, the connected component of each node is the same.

Both algorithms are empirically evaluated to compare their performances. To this end, we have produced a selection of random directed hypergraphs inspired by existent and well-known random graphs models like Erdős-Rényi and Newman-Watts-Strogatz.

Besides the application examples that we mentioned earlier, directed hypergraphs have also been employed in the field of knowledge representation. In particular, they have been used to compute the modules of an ontology. An ontology is defined as a collection of axioms that provides a formal specification of a set of terms and their relationships; and a module is a subset of an ontology that completely captures the meaning of certain terms as defined in the ontology. In particular, we focus on the modules computed using the notion of syntactic locality. As ontologies can be very large, the computation of modules facilitates the reuse and maintenance of these ontologies. Analysing all modules of an ontology, however, is in general not feasible as the number of modules grows exponentially in the number of terms and axioms of the ontology. Nevertheless, the modules can succinctly be represented using the Atomic Decomposition of an ontology. Using this representation, an ontology can be partitioned into atoms, which are maximal sets of axioms that co-occur in every module. The Atomic Decomposition is then defined as a directed graph such that each node correspond to an atom and each edge represents a dependency relation between two atoms.

In this thesis, we introduce the notion of an axiom dependency hypergraph which is a generalization of the atomic decomposition of an ontology. A module in the ontology corresponds to a connected component in the hypergraph, and the atoms of the ontology to the strongly-connected components. We apply our algorithms for directed hypergraphs to axiom dependency hypergraphs and in this manner, we compute the atoms of an ontology. To demonstrate the viability of this approach, we have implemented the algorithms in the application HyS which computes the modules of ontologies and calculate their atomic decomposition. In the thesis, we provide an experimental evaluation of HyS with a selection of large and prominent biomedical ontologies, most of which are available in the NCBO Bioportal. HyS outperforms state-of-the-art implementations in the tasks of extracting modules and computing the atomic decomposition of these ontologies.

Resumen

Los hipergrafos dirigidos se han empleado en problemas relacionados con lógica proposicional, bases de datos relacionales, lingüística computacional y aprendizaje automático. Los hipergrafos dirigidos han sido también utilizados como alternativa a los grafos (bipartitos) dirigidos para facilitar el estudio de las interacciones entre componentes de sistemas complejos que no pueden ser fácilmente modelados usando exclusivamente relaciones binarias. En este contexto, este tipo de representación es conocida como hiper-redes.

Un hipergrafo dirigido es una generalización de un grafo dirigido especialmente adecuado para la representación de relaciones de muchos a muchos. Mientras que una arista en un grafo dirigido define una relación entre dos de sus nodos, una hiperarista en un hipergrafo dirigido define una relación entre dos conjuntos de sus nodos. La conexión fuerte es una relación de equivalencia que divide el conjunto de nodos de un hipergrafo dirigido en particiones y cada partición define una clase de equivalencia conocida como componente fuertemente conexo. El estudio de los componentes fuertemente conexos de un hipergrafo dirigido puede ayudar a conseguir una mejor comprensión de la estructura de este tipo de hipergrafos cuando su tamaño es considerable. En el caso de grafo dirigidos, existen algoritmos muy eficientes para el cálculo de los componentes fuertemente conexos en grafos de gran tamaño. Gracias a estos algoritmos, se ha podido averiguar que la estructura de la WWW tiene forma de “pajarita”, donde más del 70% del los nodos están distribuidos en tres grandes conjuntos y uno de ellos es un componente fuertemente conexo. Este tipo de estructura ha sido también observada en redes complejas en otras áreas como la biología. Estudios de naturaleza similar no han podido ser realizados en hipergrafos dirigidos porque no existe algoritmos capaces de calcular los componentes fuertemente conexos de este tipo de hipergrafos.

En esta tesis doctoral, hemos investigado como calcular los componentes fuertemente conexos de un hipergrafo dirigido. En concreto, hemos desarrollado dos algoritmos para este problema y hemos determinado que son correctos y cual es su complejidad computacional. Ambos algoritmos han sido evaluados empíricamente para comparar sus tiempos de ejecución. Para la evaluación, hemos producido una selección de hipergrafos dirigidos generados de forma aleatoria inspirados en modelos muy conocidos de grafos aleatorios como Erdős-Rényi, Newman-Watts-Strogatz and Barabási-Albert. Varias optimizaciones para ambos algoritmos

han sido implementadas y analizadas en la tesis. En concreto, colapsar los componentes fuertemente conexos del grafo dirigido que se puede construir eliminando ciertas hiperaristas complejas del hipergrafo dirigido original, mejora notablemente los tiempos de ejecución de los algoritmos para varios de los hipergrafos utilizados en la evaluación.

Aparte de los ejemplos de aplicación mencionados anteriormente, los hipergrafos dirigidos han sido también empleados en el área de representación de conocimiento. En concreto, este tipo de hipergrafos se han usado para el cálculo de módulos de ontologías. Una ontología puede ser definida como un conjunto de axiomas que especifican formalmente un conjunto de símbolos y sus relaciones, mientras que un módulo puede ser entendido como un subconjunto de axiomas de la ontología que recoge todo el conocimiento que almacena la ontología sobre un conjunto específico de símbolos y sus relaciones. En la tesis nos hemos centrado solamente en módulos que han sido calculados usando la técnica de localidad sintáctica.

Debido a que las ontologías pueden ser muy grandes, el cálculo de módulos puede facilitar las tareas de re-utilización y mantenimiento de dichas ontologías. Sin embargo, analizar todos los posibles módulos de una ontología es, en general, muy costoso porque el número de módulos crece de forma exponencial con respecto al número de símbolos y de axiomas de la ontología. Afortunadamente, los axiomas de una ontología pueden ser divididos en particiones conocidas como átomos. Cada átomo representa un conjunto máximo de axiomas que siempre aparecen juntos en un módulo. La descomposición atómica de una ontología es definida como un grafo dirigido de tal forma que cada nodo del grafo corresponde con un átomo y cada arista define una dependencia entre una pareja de átomos.

En esta tesis introducimos el concepto de “axiom dependency hypergraph” que generaliza el concepto de descomposición atómica de una ontología. Un módulo en una ontología correspondería con un componente conexo en este tipo de hipergrafos y un átomo de una ontología con un componente fuertemente conexo. Hemos adaptado la implementación de nuestros algoritmos para que funcionen también con axiom dependency hypergraphs y poder de esa forma calcular los átomos de una ontología. Para demostrar la viabilidad de esta idea, hemos incorporado nuestros algoritmos en una aplicación que hemos desarrollado para la extracción de módulos y la descomposición atómica de ontologías. A la aplicación la hemos llamado HyS y hemos estudiado sus tiempos de ejecución usando una selección de ontologías muy conocidas del área biomédica, la mayoría disponibles en el portal de Internet NCBO. Los resultados de la evaluación muestran que los tiempos de ejecución de HyS son mucho mejores que las aplicaciones más rápidas conocidas.

Table of contents

List of figures	xix
List of tables	xxi
1 Introduction	1
1.1 Thesis Outline and Relevant Contributions	2
2 Preliminaries I: Directed Graphs and Hypergraphs	7
2.1 Directed Graphs	8
2.1.1 Representation of Directed Graphs	11
2.2 Directed Graph Traversal	13
2.2.1 Breadth-first Search Algorithm	13
2.2.2 Depth-first Search Algorithm	17
2.3 Strong Connectivity in Directed Graphs	22
2.3.1 Naïve Algorithm	22
2.3.2 Tarjan's Algorithm	27
2.4 Directed Hypergraphs	35
2.4.1 Representation of Directed Hypergraphs	43
2.5 Directed Hypergraph Traversal	45
2.5.1 Breadth-first Search Algorithm	45
2.5.2 Depth-first Search Algorithm	57
2.6 Strong Connectivity in Directed Hypergraphs	64
2.7 Summary	67
3 Strong Connectivity in Directed Hypergraphs	69
3.1 Directed Hypergraph Approximation	70
3.2 Depth-first Search Algorithm for Directed Hypergraphs	73
3.3 Computing Strongly- B -Connected Components in Directed Hypergraphs . .	83
3.3.1 Adapting Naïve Algorithm	83

3.3.2	Adapting Tarjan's Algorithm	88
3.3.3	Condensing the Graph Fragment of a Directed Hypergraph	93
3.4	Evaluation	96
3.5	Summary	98
4	Preliminaries II: Syntactic Locality and Atomic Decomposition	99
4.1	Syntactic Locality-based Modules	99
4.1.1	Extracting modules based on locality	102
4.1.2	Signature Containment for Module Extraction	104
4.2	Atomic Decomposition	105
4.3	Reference Implementations	107
5	Module Extraction and Atomic Decomposition of OWL Ontologies	109
5.1	Axiom Dependency Hypergraphs	109
5.2	Locality-based Modules	111
5.2.1	Syntactic $\perp\top^*$ -local modules	117
5.3	Atomic Decomposition	121
5.4	Summary	127
6	HyS	129
6.1	Directed Labelled Graph Representation	130
6.2	Normalization and Denormalization of OWL Ontologies	135
6.3	Directed Labelled Graph Traversal	148
6.3.1	Breadth-first Search Algorithm	149
6.3.2	Depth-first Search Algorithm	159
6.4	Locality-based Modules	166
6.5	Atomic Decomposition	169
6.5.1	Naïve Algorithm	170
6.5.2	Tarjan's Algorithm	172
6.5.3	Computing the Condensed Axiom Dependency Hypergraph	172
6.6	Implementation	175
6.6.1	Architecture	176
6.6.2	Interfaces	178
6.7	Evaluation	181
6.8	Summary	186

7

Conclusions and Future Work

189

7.1

Research Methodology

190

7.2

Significance of Main Contributions

191

7.3

Future Work

192

References

193

List of figures

2.1	A strongly connected directed graph with three directed cycles	9
2.2	(Example 2.7): Condensing a directed graph	11
2.3	(Example 2.10): Representation of a directed graph	13
2.4	BFS algorithm for directed graphs.	15
2.5	DFS algorithm for directed graphs.	18
2.6	(Example 2.19): DFS algorithm	20
2.7	Naïve algorithm to compute the SCCs of a directed graph.	23
2.8	Tarjan for directed graphs.	28
2.9	(Example 2.30): Tarjan’s algorithm for directed graphs	33
2.10	(Example 2.40): Connectivity notions in directed hypergraphs [28].	41
2.11	(Example 2.46): Adjacency-matrix of a directed hypergraph.	44
2.12	Function Bvisit for directed hypergraphs [18, 31].	46
2.13	BFS main function for directed hypergraphs.	50
2.14	BFS visit function for directed hypergraphs.	51
2.15	Examples of directed hypergraphs for testing algorithms	52
2.16	(Example 2.59): node to node connectivity in BFS-graphs	55
2.17	Node to node connectivity in new BFS-graphs	56
2.18	DFS main function for directed hypergraphs.	58
2.19	DFS visit function for directed hypergraphs.	59
2.20	(Example 2.64): node to node connectivity in DFS-graphs	63
2.21	Algorithm for computing terminal SCCs in directed hypergraphs [2].	65
2.22	(Example 2.65): computing terminal SCCs [2].	66
3.1	DFS/Approximation/Tarjan main function for B -hypergraphs.	74
3.2	DFS/Approximation/Tarjan visit function for B -hypergraphs.	75
3.3	Example 3.10: DFS for B -hypergraphs	76
3.4	DFS/Approximation/Tarjan visit function for directed hypergraphs.	82
3.5	Naïve algorithm that computes the SCCs of a directed hypergraph.	84

3.6	Naïve algorithm that computes the SCCs of a directed hypergraph [28]. . . .	87
3.7	Approximating graph using the new version of the naïve algorithm	87
3.8	Example 3.8: Tarjan for B -hypergraphs	90
3.9	Algorithm that computes the partially condensed directed hypergraph. . . .	94
4.1	Module Extraction Algorithm [11]	102
4.2	Module Extraction Algorithm for $\perp\top^*$ -local modules [36]	103
5.1	(Example 5.7): Σ -extension of a $\perp$ -ADH.	113
5.2	(Example 5.7): An alternative representation of the Σ -extension of a $\perp$ -ADH.	116
5.3	Module Extraction Algorithm for $\perp\top^*$ -local modules.	117
5.4	(Example 5.14): Computing a $\perp\top^*$ -local module using the function $\text{extract}^*(\Sigma, \mathcal{O})$	119
5.5	Example 5.24: From the $\perp$ -ADH to the condensed $\perp$ -ADH	125
5.6	Example 5.25: $\top$ -locality vs. $\top$ -reachability	126
6.1	(Example 6.10): Labelled graph encoding of $\mathcal{H}_{\mathcal{O}}^{\perp}$	132
6.2	(Examples 6.13 & 6.14): Normalization and B -connectivity.	137
6.3	BFS main function for directed labelled graphs.	150
6.4	BFS auxiliary functions for directed labelled graphs.	151
6.5	BFS visit function for directed labelled graphs.	152
6.6	(Example 6.31): BFS-graph for directed labelled graphs.	155
6.7	DFS/Approximation/Tarjan main function for directed labelled graphs. . . .	159
6.8	DFS/Approximation/Tarjan auxiliary functions for directed labelled graphs.	160
6.9	DFS/Approximation/Tarjan visit function for directed labelled graphs. . . .	161
6.10	(Example 6.38): DFS-graph for directed labelled graphs.	162
6.11	Module extraction algorithm for the labelled graph-encoding of an ADH . . .	168
6.12	Naïve algorithm to compute the SCCs of a directed labelled graph.	170
6.13	Function for collapsing the SCCs of an ADH encoded by a labelled graph. . .	173
6.14	Function for condensing an ADH encoded by a labelled graph.	174
6.15	Example of the <i>HyS</i> API	179
6.16	Commands available in <i>HyS</i>	180

List of tables

3.1	Selected hypergraphs for the evaluation Naïve and Tarjan algorithms.	97
3.2	Evaluation of Naïve and Tarjan algorithms.	98
6.1	Selected ontologies for the evaluation of <i>HyS</i>	182
6.2	Evaluation of <i>HyS</i> for Atomic Decomposition.	183
6.3	Size of the labelled graph-encoding of an ADH for $\mathcal{O}$	185
6.4	Evaluation of <i>HyS</i> for module extraction.	185

Chapter 1

Introduction

Directed hypergraphs [18] are an intuitive modelling formalism that have been used in problems related to propositional logic [4, 35], relational databases [3], computational linguistic [23] and machine learning [20]. Directed hypergraphs are also presented as an alternative to directed (bipartite) graphs to facilitate the study of the interactions between components of complex systems that cannot naturally be modelled as binary relations [29, 22]. In this context, they are known as hyper-networks [17].

Strong-connectivity is an equivalence relation that induces a partition of the set of nodes of a directed hypergraph into strongly-connected components. These components can be collapsed into single nodes. As result, the size of the original hypergraph can significantly be reduced if the strongly-connected components have many nodes. This approach might contribute to a better understanding of how the nodes of a hypergraph are connected, in particular when the hypergraphs are large. In the case of directed graphs, there are efficient algorithms that can be used to compute the strongly-connected components of large graphs. For instance, it has been shown that the macroscopic structure of the World Wide Web can be represented as a “*bow-tie*” diagram where more than 70% of the nodes are distributed into three large sets and one of these sets is a large strongly-connected component [9]. This particular structure has been also observed in complex networks in other fields such as, e.g., biology [24, 45]. Similar studies cannot be conducted in a directed hypergraph because there does not exist any algorithm for computing the strongly-connected components of the hypergraph.

In this thesis, we investigate ways to compute the strongly-connected components of directed hypergraphs. We present two new algorithms and we show their correctness and computational complexity. One of these algorithms is inspired by Tarjan’s algorithm [39] for directed graphs. The second algorithm follows a simple approach to compute the strongly-connected components. This approach is based on the fact that two nodes of a graph that are

strongly-connected can also reach the same nodes. In other words, the connected component of each node is the same.

Both algorithms are empirically evaluated to compare their performances. To this end, we have produced a selection of random directed hypergraphs inspired by existent and well-known random graphs models like Erdős-Rényi [15, 16, 19] and Newman-Watts-Strogatz[44, 30].

Besides the application examples that we mentioned earlier, directed hypergraphs have also been employed in the field of knowledge representation. In particular, they have been used to compute the modules of an ontology [38, 33, 34]. An ontology is defined as a collection of axioms that provides a formal specification of a set of terms and their relationships; and a module is a subset of an ontology that completely captures the meaning of certain terms as defined in the ontology. In particular, we focus on the modules computed using the notion of syntactic locality [12]. As ontologies can be very large, the computation of modules facilitates the reuse and maintenance of these ontologies. Analysing all modules of an ontology, however, is in general not feasible as the number of modules grows exponentially in the number of terms and axioms of the ontology. Nevertheless, the modules can succinctly be represented using the Atomic Decomposition of an ontology [14]. Using this representation, an ontology can be partitioned into atoms, which are maximal sets of axioms that co-occur in every module. The Atomic Decomposition is then defined as a directed graph such that each node corresponds to an atom and each edge represents a dependency relation between two atoms.

In this thesis, we introduce the notion of an axiom dependency hypergraph which is a generalization of the atomic decomposition of an ontology. A module in the ontology corresponds to a connected component in the hypergraph, and the atoms of the ontology to the strongly-connected components. We apply our algorithms for directed hypergraphs to axiom dependency hypergraphs and in this manner, we compute the atoms of an ontology. To demonstrate the viability of this approach, we have implemented the algorithms in the application *HyS* which extracts modules and computes the atomic decomposition of ontologies. In the thesis, we provide an experimental evaluation of *HyS* with a selection of large and prominent biomedical ontologies, most of which are available in the NCBO Bioportal. *HyS* outperforms state-of-the-art implementations in the tasks of extracting modules and computing the atomic decomposition of these ontologies.

1.1 Thesis Outline and Relevant Contributions

The remainder of the thesis is organized as follows:

Chapter 2. This chapter provides an overview on directed graphs and hypergraphs. The main focus of the chapter is to explain the notions of connectivity and strong-connectivity for both representations and to study well-known algorithms for traversing directed graphs and computing their strongly-connected components.

This chapter does not only review the state of art but it also includes the following minor contributions:

- (1) A detailed proof of correctness of the algorithm by Tarjan [39] based on the proof sketch suggested by the author.
- (2) An extended version of the function $Bvisit(\cdot, \cdot)$ [18, 31] for completely traversing a directed hypergraph similar to how Breadth-first search works.
- (3) A new version of depth-first search for directed hypergraphs.

In the case of (2) and (3), we show the correctness and complexity of each algorithm.

Chapter 3. We introduce in this chapter, the new notion of approximating graph and some relevant properties. An approximating graph is a directed graph that preserves node-to-node connectivity of a directed hypergraph. Based on this notion, we define and implement a new depth-first search algorithm for traversing directed hypergraphs, and furthermore, we explain how it can also be used to compute approximating graphs. In addition, we show that this algorithm can also identify the strongly-connected components of a directed hypergraph similar to the algorithm by Tarjan [39] for directed graphs.

To compare the complexity and performance of the new depth-first search algorithm, we design and implement a new simple algorithm that computes the strongly-connected components of a directed hypergraph. Besides the study of complexity, we also show the correctness of both algorithms.

To speed up the performances of the simple algorithm for computing the strongly-connected components of a directed hypergraph, we discuss an optimization that collapses the strongly-connected components of the graph fragment of the hypergraph. To evaluate both algorithms, we generate a collection of artificial and random hypergraphs. The random hypergraphs are based on Erdős-Rényi [15, 16, 19] and Newman-Watts-Strogatz[44, 30] models for random graphs.

Chapter 4. In this chapter, we provide a concise overview on relevant notions related to description logics [7], syntactic locality [12] and atomic decomposition [14].

Chapter 5. In this chapter, we introduce the notion of an axiom dependency hypergraph [25, 27] which is a generalization of the atomic decomposition of an ontology. We show that

in this hypergraph representation, a module in the ontology corresponds to a connected component in the hypergraph, and the atoms of the ontology to the strongly-connected components. By condensing the strongly-connected components of an axiom dependency hypergraph and computing the approximating graph, we conclude that this graph is equivalent to the atomic decomposition of an ontology.

In the paper [25], we presented the following contributions:

- (1) The notion of an axiom dependency hypergraph.
- (2) The relation between (strongly-) connected components in an axiom dependency hypergraph and modules (atoms, respectively) in the ontology.
- (3) A prototype implementation to compute the atoms of ontologies that can be represented using an axiom dependency hypergraph where its hyperedges are directed edges.

Chapter 6. In this chapter, we introduce a labelled graph representation of an axiom dependency hypergraph [27]. This representation is more compact and facilitates the implementation of applications for axiom dependency hypergraphs. We also show that existent normalization rules [33, 34] for OWL 2 ontologies might produce incorrect results when the atomic decomposition of the normalized ontology and their modules are computed. Based on the algorithms for traversing directed hypergraphs that we discussed in Chapters 2 & 3, we define two algorithms for traversing axiom dependency hypergraphs encoded in directed labelled graphs. One algorithm traverses a labelled graph in a Breadth-first search fashion and the other is very similar to the depth-first search algorithm presented in Chapter 3. We show how these algorithms can be also used to compute the strongly-connected components of an axiom dependency hypergraph encoded in a labelled graph. For all the algorithms presented in this chapter, we show their correctness and complexity. To test the suitability of these algorithms in real ontologies, we have implemented the application HyS for computing the atomic decomposition and the modules of ontologies. We provide an experimental evaluation of HyS with a selection of large and prominent biomedical ontologies, most of which are available in the NCBO Biportal.

In the papers [26, 27], we presented the following contributions:

- (1) A notion of labelled graph-encoding of an axiom dependency hypergraph.
- (2) A new version of the Breadth-first search algorithm for labelled graphs.
- (3) An algorithm for computing modules in the labelled-graph representation of an axiom dependency hypergraph.

- (4) A simple algorithm for computing the strongly-connected components in the labelled-graph representation of an axiom dependency hypergraph.
- (5) The application HyS for computing the atomic decomposition and the modules of ontologies.
- (6) An evaluation of HyS against state-of-the-art implementations using a selection of large and prominent biomedical ontologies , most of which are available in the NCBO Bioportal.

Chapter 7. In this chapter, the significance of the major results of the thesis is summarised, outstanding issues are analysed and directions for future work are suggested. We also provide a brief discussion on the methodology followed during this research.

Chapter 2

Preliminaries I: Directed Graphs and Hypergraphs

This chapter provides an overview on directed graphs and hypergraphs. The main focus of the chapter is to explain the notions of connectivity and strong-connectivity for both representations and to study well-known algorithms for traversing directed graphs and computing their strongly-connected components.

This chapter does not only review the state of the art but also includes the following minor contributions:

- (1) A detailed proof of correctness of the algorithm by Tarjan [39] based on the proof sketch suggested by the author.
- (2) An extended version of the function $Bvisit(\cdot, \cdot)$ [18, 31] for completely traversing a directed hypergraph similar to how Breadth-first search works.
- (3) A new version of Depth-first search for directed hypergraphs.

In the case of (2) and (3), we show the correctness and complexity of each algorithm.

Chapter 2 is organized as follows. Section 2.1 provides an overview of relevant notions on directed graphs and discusses common approaches to store these graphs. The algorithms breadth-first search and depth-first search for traversing directed graphs are reviewed in Section 2.2. An analysis of two well-known algorithms for computing the strongly-connected components of a directed graph is included in Section 2.3. An overview of relevant notions on directed hypergraphs and known approaches to store these hypergraphs can be found in Section 2.4. Two new algorithms for traversing directed hypergraphs completely are discussed in Section 2.5. With respect to strong-connectivity for directed hypergraphs, we are only aware of an algorithm for computing the "*terminal*" strongly-connected components

of a directed hypergraph [2]. This algorithm is discussed in Section 2.6. A summary of the chapter is provided in Section 2.7.

2.1 Directed Graphs

In this section, we review basic notions on directed graphs as they are introduced in [37, 10]. A *directed graph* (or digraph) is a tuple $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a finite non-empty set of nodes and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of directed edges. A *directed edge* $e = (T(e), H(e))$ is an ordered pair, where $T(e) \subseteq \mathcal{V}$, $|T(e)| = 1$, $H(e) \subseteq \mathcal{V}$ and $|H(e)| = 1$. $T(e)$ ($H(e)$) is known as the *tail* (*head*) of the directed edge and it represents where the edge starts (ends).

A *directed path* in a directed graph is a sequence of nodes and edges such that each edge pointing from a node in the sequence to its successor in the sequence.

Definition 2.1 (Directed Path). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $s, t \in \mathcal{V}$. A directed path from s to t in $\mathcal{G}$ is a sequence of the form $\pi_{s \rightarrow t} = (v_1, e_1, v_2, e_2, \dots, e_{n-1}, v_n)$, $n > 0$, such that $v_1 = s$, $v_n = t$, $e_i = (v_i, v_{i+1}) \in \mathcal{E}$, for all $i \in \{1, \dots, n-1\}$.*

The directed path $\pi_{s \rightarrow t}$ in $\mathcal{G}$ is acyclic if $s \neq t$, and $\pi_{s \rightarrow t}$ is simple if $v_i \neq v_j$ for all $i, j \in \{1, \dots, n-1\}$ with $i \neq j$. $\dashv$

A *directed cycle* is a directed path such that the first and last node are the same node. A *simple cycle* is a simple path such that the first and last node are the same node.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $v_1, v_i \in \mathcal{V}$ be two nodes. v_i is *connected* (or *reachable*) from v_1 , written $v_1 \geq_{\mathcal{G}} v_i$, if there is a directed path from v_1 to v_i . Connectivity (or reachability) in directed graphs can equivalently be defined as follows. A node v_i is connected (or reachable) from a node v_1 if (i) $v_i = v_1$ (v_i is connected from itself), or (ii) there is a direct edge $e = (v_{i-1}, v_i)$ such that $v_1 \geq_{\mathcal{G}'} v_{i-1}$, where $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$, $\mathcal{V}' = \mathcal{V}$ and $\mathcal{E}' = \mathcal{E} \setminus \{e\}$ [28].

Let $v \in \mathcal{V}$ be a node in a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. A *connected component* (CC) of v is a maximal set $\mathcal{V}' \subseteq \mathcal{V}$ of nodes connected from v . We denote with $CC(v)$ the connected component of v .

In a directed graph $\mathcal{G}$, two nodes v and w are *strongly-connected* if v is connected to w and vice versa. Moreover, a directed graph is strongly-connected if all its vertices are strongly-connected to one another. Notice that there is a close relation between strong-connectivity and directed cyclic paths, as it is shown in the following proposition from [37].

Proposition 2.2. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $v, w \in \mathcal{V}$ be two nodes. v and w are strongly-connected if, and only if, there exists at least one directed cycle that contains them both.*

This proposition can readily be seen. Just compose the paths from v to w and from w to v (the other direction can be shown similarly).

Strong-connectivity is an equivalence relation over the set of nodes of a directed graph, as it satisfies the following properties:

- *Reflexivity* : Every vertex v is strongly-connected to itself.
- *Symmetry* : If v is strongly-connected to w , then w is strongly connected to v .
- *Transitivity* : If v is strongly-connected to w and w is strongly-connected to x , then v is also strongly-connected to x

As an equivalence relation over the set of nodes of a directed graph, strong-connectivity induce a partition of the nodes of the graph into equivalence classes. The equivalence classes based on strong-connectivity are called strongly-connected components.

A *strongly-connected component (SCC)* of a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a maximal set $\mathcal{V}' \subseteq \mathcal{V}$ of nodes such that for every pair v and w of nodes from $\mathcal{V}'$, it holds that v and w are strongly-connected. Let $\text{SCCs}(\mathcal{G})$ be the set of strongly-connected components of $\mathcal{G}$.

As suggested by Proposition 2.2, there is a close relationship between directed cycles and strongly-connected components. Figure 2.1 provides an example of a strongly connected component (SCC_1) with three directed cycles in one .

$$\text{SCC}_1 = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

Fig. 2.1 A strongly connected directed graph with three directed cycles

The following proposition can be readily be seen and it makes precise the relation between the notions of a connected component and a strongly-connected component.

Proposition 2.3. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $v, w \in \mathcal{V}$ be two nodes. Let $\mathcal{S} \in \text{SCCs}(\mathcal{G})$ be a strongly-connected component. Then: $v, w \in \mathcal{S}$ iff $\text{CC}(v) = \text{CC}(w)$. $\dashv$

Proof. ‘ $\Rightarrow$ ’: Suppose that the nodes v and w are strongly-connected. Then, v is connected from w and vice versa: $v \geq_{\mathcal{G}} w$ and $w \geq_{\mathcal{G}} v$. As a result, any node in $\mathcal{G}$ connected from v is also connected from w (and vice versa). This implies that $\text{CC}(v) = \text{CC}(w)$.

‘ $\Leftarrow$ ’: Let $\text{CC}(v) = \text{CC}(w)$. This means that v is connected from w and vice versa: $v \geq_{\mathcal{G}} w$ and $w \geq_{\mathcal{G}} v$. Therefore, the nodes v and w are strongly-connected. This is v and w are in the same strongly-connected component. $\square$

The partition of a directed graph under an equivalence relation over the nodes of the graph induces a *quotient* directed graph. We make this notion more precise in the following definition.

Definition 2.4 (Quotient Graph). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $\simeq$ be an equivalence relation over $\mathcal{V}$. The quotient graph of $\mathcal{G}$ under $\simeq$, written $\mathcal{G}/\simeq$, is the graph $\mathcal{G}/\simeq = (\mathcal{V}/\simeq, \mathcal{E}_{\simeq})$, where*

- $\mathcal{V}/\simeq = \{[x]_{\simeq} \mid x \in \mathcal{V}\}$; and
- $e = ([x]_{\simeq}, [y]_{\simeq}) \in \mathcal{E}_{\simeq}$ iff there is an $e' \in \mathcal{E}$ such that $e' = (x, y)$ and $[x]_{\simeq} \cap [y]_{\simeq} = \emptyset$. $\dashv$

As we mentioned earlier in this section, strong-connectivity is an equivalence relation over the nodes of a directed graph. The quotient graph derived from the strong-connectivity relation over the nodes of a directed graph is known as the *condensed* directed graph.

Definition 2.5 (Condensed Directed Graph). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $\simeq$ be the strong-connectivity relation over $\mathcal{V}$. The condensed directed graph of $\mathcal{G}$ is defined as the quotient graph of $\mathcal{G}$ under $\simeq$, written $\mathcal{G}/\simeq$, is the graph $\mathcal{G}/\simeq = (\mathcal{V}/\simeq, \mathcal{E}_{\simeq})$, where $\mathcal{V}/\simeq = \{[x]_{\simeq} \mid x \in \mathcal{V}\}$ is the set of all strongly-connected components of $\mathcal{G}$ ($\text{SCCs}(\mathcal{G})$). $\dashv$*

The condensed version of a directed graph is the result of collapsing the strongly-connected components of the original graph into single new nodes. Therefore, we obtain a *directed acyclic graph (DAG)*, a directed graph with no directed cycles.

Proposition 2.6. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $\simeq$ be the strong-connectivity relation over $\mathcal{V}$. Let $\mathcal{G}/\simeq = (\mathcal{V}/\simeq, \mathcal{E}_{\simeq})$ be the condensed directed graph of $\mathcal{G}$.*

Then: $\mathcal{G}/\simeq$ is a directed acyclic graph. $\dashv$

By Proposition 2.2, this proposition can readily be seen. If each strongly-connected component of a directed graph has been collapsed into single new nodes then there is no pair of nodes in the condensed directed graph such that both nodes are strongly-connected. By Proposition 2.2, the condensed directed graph has no directed cycles (it is a directed acyclic

graph). Moreover, if a directed graph has no cycles (it is a directed acyclic graph) then it is not possible (by Proposition 2.2) to find a pair of nodes that are strongly-connected.

We say that a strongly-connected component of a directed graph is *terminal* if its out-degree in the condensed version of the original graph is zero. This means that there are no edges in the condensed directed graph that has a terminal strongly-connected component in their tail.

Example 2.7. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, where $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $\mathcal{E} = \{e_1 = (v_1, v_2), e_2 = (v_1, v_4), e_3 = (v_2, v_5), e_4 = (v_3, v_5), e_5 = (v_3, v_6), e_6 = (v_4, v_2), e_7 = (v_5, v_4), e_8 = (v_6, v_3)\}$

After computing the strongly-connected components and collapsing them into single nodes, we produce the following equivalence classes: $[v_1]_{\simeq} = \{v_1\}$, $[v_2]_{\simeq} = \{v_2, v_4, v_5\}$ and $[v_3]_{\simeq} = \{v_3, v_6\}$. We can refer to the strongly-connected component v_2, v_4, v_5 using any member v in v_2, v_4, v_5 with $[v]$. We can refer to the strongly-connected component $\{v_2, v_4, v_5\}$ using any member $v \in \{v_2, v_4, v_5\}$ with $[v]$.

$\mathcal{G}/_{\simeq} = (\mathcal{V}/_{\simeq}, \mathcal{E}_{\simeq})$ is the condensed directed graph of $\mathcal{G}$ such that $\mathcal{V}/_{\simeq} = \{[v_1]_{\simeq}, [v_2]_{\simeq}, [v_3]_{\simeq}\}$ and $\mathcal{E}_{\simeq} = \{e_9 = ([v_1]_{\simeq}, [v_2]_{\simeq}), e_{10} = ([v_3]_{\simeq}, [v_2]_{\simeq})\}$. Notice that $[v_2]_{\simeq}$ represents a terminal strongly-connected component. $\triangleleft$

Fig. 2.2 (Example 2.7): Condensing a directed graph

2.1.1 Representation of Directed Graphs

There are two common approaches to describe a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$: the *adjacency-list* representation and the *adjacency-matrix* (also known as incidence-matrix) representation. The adjacency-list provides for each node of a directed graph a list of successors. Each set

stores the nodes that are neighbours of a particular node. In consequence, there is one set for each node of the graph.

The adjacency-matrix representation describes a directed graph as a matrix of binary numbers. The number of rows and the number of columns of the matrix are equal to the number of nodes of the graph ($|\mathcal{V}| \times |\mathcal{V}|$). It is assumed that the vertices are numbered $1, 2, \dots, |\mathcal{V}|$ in some arbitrary order. Then, an element $a_{i,j}$ of the matrix has value 1 if, and only if, there is an edge (v_i, v_j) in the graph and 0 otherwise.

Notice that for *sparse* directed graphs ($|\mathcal{E}| \ll |\mathcal{V}|^2$), the adjacency-list representation is often more optimal (in terms of space consumption) than the adjacency-matrix.

Definition 2.8 (Adjacency-list for directed graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $v \in \mathcal{V}$. We denote with $AL(v)$ the adjacency-list of v in $\mathcal{G}$ that represents the set of all nodes in $\mathcal{V}$ adjacent to v . Formally, $AL(v) = \{w \in \mathcal{V} \mid (v, w) \in \mathcal{E}\}$.*

The adjacency-list representation of $\mathcal{G}$, denoted with $AL(\mathcal{G})$, is the set of $|\mathcal{V}|$ adjacency-lists, one for each node in $\mathcal{V}$, where $AL(\mathcal{G}) = \bigcup \{AL(v') \mid v' \in \mathcal{V}\}$. $\dashv$

Definition 2.9 (Adjacency-matrix for directed graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $id\text{-}node: \mathcal{V} \rightarrow \mathbb{N}$ be a total function assigning to every node $v \in \mathcal{V}$ a natural number $id\text{-}node(v) \in \{1, \dots, |\mathcal{V}|\}$. Let $v, w \in \mathcal{V}$ such that $id\text{-}node(v) = i$ and $id\text{-}node(w) = j$, where $i, j \in \{1, \dots, |\mathcal{V}|\}$. Let $a_{i,j}$ be the entry in a binary matrix such that it is located in the i -th row and j -th column.*

The adjacency-matrix representation of $\mathcal{G}$, denoted with $AM(\mathcal{G})$, is a $|\mathcal{V}| \times |\mathcal{V}|$ binary matrix such that for all $a_{i,j} \in AM(\mathcal{G})$, $a_{i,j} = 1$ iff $(v, w) \in \mathcal{E}$, and $a_{i,j} = 0$ otherwise. $\dashv$

The following example illustrates how a directed graph can be described using an adjacency-list or an adjacency-matrix representation.

Example 2.10. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the directed graph depicted in Figure 2.3(1), where $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$ and $\mathcal{E} = \{e_1 = (v_1, v_2), e_2 = (v_1, v_3), e_3 = (v_2, v_3), e_4 = (v_3, v_2), e_5 = (v_3, v_4), e_6 = (v_4, v_1)\}$.*

The adjacency list (Figure 2.3(2)) includes 4 lists, one for each node. For instance, the list related to the node v_1 has two nodes, v_2 and v_3 . This implies that there are two directed edges in $\mathcal{E}$, (v_1, v_2) and (v_1, v_3) .

The adjacency matrix (Figure 2.3(3)) is of size 4×4 . This means that the number of rows and the number of columns correspond to the number of nodes of the directed graph. The element $a_{1,2}$ has value one and this indicates that there is a directed edge (v_1, v_2) . $\triangleleft$

Fig. 2.3 (Example 2.10): Representation of a directed graph

2.2 Directed Graph Traversal

In this section, we study two well-known algorithms for traversing directed graphs: breadth-first search and depth-first search.¹

2.2.1 Breadth-first Search Algorithm

Breadth-first search (BFS) is an algorithm that has been commonly used for traversing directed graphs. For a given node, v , BFS discovers the head nodes of the directed edges that have v in the tail. The algorithm follows the same approach for each of the head nodes discovered. That is, it discovers all nodes at distance k from v before discovering any node at distance $k + 1$. Therefore, BFS computes the distance (smallest number of edges) from v to each reachable node in the directed graph [10]. A representation of the BFS algorithm is depicted in Figure 2.4. Lines in red and labelled with the symbol ‘+’ have been added to explicitly store the BFS-graph, a directed labelled graph that indicates the order in which the input directed graph was traversed by BFS (Definition 2.13).

We use the following auxiliary notion to refer to the set of successors at a given node. Let $\mathcal{E} : \mathcal{V} \rightarrow 2^{\mathcal{V}}$ be a function that returns the set of nodes (successors) that are reachable from a node $v \in \mathcal{V}$ using only direct edges in $\mathcal{E}$:

$$\mathcal{E}(v) := \{v' \in \mathcal{V} \mid e = (v, v') \in \mathcal{E}\}.$$

The order in which the directed graph is traversed by BFS can be represented as a directed labelled graph commonly known as *BFS-graph*. Every node of a BFS-graph is labelled with two timestamps: one timestamp indicating when the node was *discovered* (i.e., when it was

¹The material presented in this section has been adapted from [1, 10, 37].

visited for the first time), and a second timestamp stating when the node has been *explored* (i.e., when all its direct successors have been discovered). We use $\text{discovered-order}(v)$ and $\text{explored-order}(v)$ to refer to the timestamps that indicate when the node v was discovered and explored, respectively. We make these notions more precise in the following definitions.

Definition 2.11 (Discovered-order). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $\text{discovered-order}: \mathcal{V} \rightarrow \mathbb{Z}$ be a total function assigning to every node $v \in \mathcal{V}$ an integer value $\text{discovered-order}(v) \in \{-1, 1, \dots, n\}$, where $n = 2 \times |\mathcal{V}|$, stating the position of node v in the order in which the nodes of $\mathcal{G}$ are discovered. Value -1 means that the node has not been discovered yet. $\dashv$*

Definition 2.12 (Explored-order). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $\text{explored-order}: \mathcal{V} \rightarrow \mathbb{Z}$ be a total function assigning to every node $v \in \mathcal{V}$ an integer value $\text{explored-order}(v) \in \{-1, 1, \dots, n\}$, where $n = 2 \times |\mathcal{V}|$, stating the position of node v in the order in which the nodes of $\mathcal{G}$ are explored. Value -1 means that the node has not been explored yet. $\dashv$*

We define a BFS-graph computed by BFS on directed graphs as follows.

Definition 2.13 (BFS-graph). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $\text{main}(\cdot)$ be the function depicted in Figure 2.4. A breadth-first-search graph (BFS-graph) of $\mathcal{G}$ is a directed labelled graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ with $\mathcal{G}' = \text{main}(\mathcal{G})$. $\dashv$*

In general, there are several BFS-graphs of a directed graph depending on the order in which the nodes are traversed. The traversal order is stored in the node labels of a BFS-graph.

Lemma 2.14 (Properties of BFS for directed graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ be the functions depicted in Figure 2.4. Let $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{L})$ be a BFS-graph of $\mathcal{G}$, and let Q be the queue data structure defined in function $\text{visit}(\cdot)$. Then the following holds:*

- (i) *for all $v \in \mathcal{V}$: $\text{visit}(v)$ enqueues v in Q exactly one time;*
- (ii) *for all $v \in \mathcal{V}$: $\text{visit}(v)$ iterates over each outgoing edge from v exactly once;*
- (iii) *$\mathcal{V}' = \mathcal{V}$ and $\mathcal{E}' = \mathcal{E}$; and*
- (iv) *for all $x, y \in \mathcal{V}$: $x \geq_{\mathcal{G}} y$ iff $x \geq_{\mathcal{G}'} y$. $\dashv$*

Proof. We start the proof of Lemma 2.14 by showing item (i). The value $\text{discovered-order}(v)$ for each node $v \in \mathcal{V}$ is initialized to -1 by the function $\text{main}(\cdot)$ (Lines 3-6). The function $\text{main}(\cdot)$ iterates over each node $v \in \mathcal{V}$ (Lines 8-12) and when $\text{discovered-order}(v) = -1$, it calls the function $\text{visit}(v)$ (Line 10). This ensures that each node will be visited at least one

```

function main( $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ )
1:  $\mathcal{V}' := \{\}$  +
2:  $\mathcal{E}' := \{\}$  +
3: for every  $v \in \mathcal{V}$  do
4:   discovered-order( $v$ ) := -1
5:   explored-order( $v$ ) := -1
6: end-for
7: order-counter := 0
8: for every  $v \in \mathcal{V}$  do
9:   if (discovered-order( $v$ ) = -1) then
10:    visit( $v$ )
11:   end-if
12: end-for
13:  $\mathcal{L} := \{(\text{discovered-order}(v), \text{explored-order}(v)) \mid v \in \mathcal{V}\}$  +
14: return  $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{L})$  +

function visit( $v$ )
1: order-counter := order-counter + 1
2: discovered-order( $v$ ) := order-counter
3:  $Q := \{\}$ 
4: enqueue( $Q, v$ )
   // Explored all direct successors of v that are still undiscovered
5: while  $Q \neq \{\}$  do
6:    $u := \text{dequeue}(Q)$ 
7:    $\mathcal{V}' := \mathcal{V}' \cup \{u\}$  +
8:   for every  $s \in \mathcal{E}(u)$  do // visit all successors of v
9:      $\mathcal{E}' := \mathcal{E}' \cup \{(u, s)\}$  +
10:    if (discovered-order( $s$ ) = -1) then
11:      // s has been discovered
12:      order-counter := order-counter + 1
13:      discovered-order( $s$ ) := order-counter
14:      enqueue( $Q, s$ )
15:    end-if
16:  end-for
17:  // u has been explored
18:  order-counter := order-counter + 1
19:  explored-order( $u$ ) := order-counter
20: end-while

```

Fig. 2.4 BFS algorithm for directed graphs.

time. In addition, the function $\text{visit}(\cdot)$ enqueues a node v in Q if $\text{discovered-order}(v) = -1$. (Lines 4 and 13). When a node v is enqueued, the value in $\text{discovered-order}(v)$ is also updated with the value in order-counter which is larger than -1 (Lines 2 and 12). Therefore, it is not possible that a node is enqueued (or dequeued) more than one time. As a result, item (i) holds.

Consider item (ii). By item (i), each node of the graph will be enqueued (and dequeued) exactly one time by the function $\text{visit}(\cdot)$. Each time that a node is dequeued, the function $\text{visit}(\cdot)$ iterates only one time over each direct successors using a for-loop (Lines 8-15). This means that the function $\text{visit}(\cdot)$ iterates over all edges of the input graph. Therefore, item (ii) holds.

Item (iii) follows from items (i) and (ii). By item (i), each node of $\mathcal{G}$ is enqueued exactly one time in Q by the function $\text{visit}(\cdot)$. This implies that each node of Q will be also dequeued one time and stored in $\mathcal{V}'$. Therefore, $\mathcal{V}' = \mathcal{V}$. By item (ii), the function $\text{visit}(\cdot)$ iterates over each outgoing edge of the node that has been currently dequeued just one time and in each iteration, the edge that connects the node that has been dequeued and its successor is added to $\mathcal{E}'$. This implies that all edges in $\mathcal{E}$ are also stored in $\mathcal{E}'$. Therefore, item (iii) holds.

Item (iv) can be readily be seen because by item (iii), $\mathcal{G}$ and $\mathcal{G}'$ have the same nodes and edges. This implies that node-to-node connectivity in $\mathcal{G}$ is preserved in $\mathcal{G}'$. $\square$

The proposition coming next follows from Lemma 2.14.

Proposition 2.15 (Correctness of BFS for directed graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $\text{main}(\cdot)$ be the function depicted in Figure 2.4. Then:*

(i) *The function $\text{main}(\cdot)$ terminates.*

(ii) *The function $\text{main}(\cdot)$ traverses $\mathcal{G}$ completely.* $\dashv$

We show that the time complexity of BFS is linear with respect to the size of the input graph.

Proposition 2.16 (Time complexity of BFS for directed graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $\text{main}(\cdot)$ be the function depicted in Figure 2.4. Then: the time complexity of function $\text{main}(\cdot)$ is $O(|\mathcal{V}| + |\mathcal{E}|)$.* $\dashv$

Proof. The loops of the function $\text{main}(\cdot)$ includes two for-loops (Lines 3-6 and Lines 8-12) that iterate over all nodes of the directed graph. Moreover, this function must store the labels of each node (Line 13). This can be done using an extra for-loop that iterates over all nodes of the graph. This implies $3 \times |\mathcal{V}|$ iterations inside the function $\text{main}(\cdot)$.

We continue with the study of the time complexity of the function $\text{visit}(\cdot)$. By item (i) of Lemma 2.14, each node of the graph will be enqueued (and dequeued) exactly one time by the function $\text{visit}(\cdot)$. By item (ii), the function $\text{visit}(\cdot)$ iterates over each outgoing edge of the node that has been currently enqueued just one time. Assuming that the operations inside the for-loop in $\text{visit}(\cdot)$ are executed in constant time (as the rest of operations outside the for-loop), the time complexity of the execution of the function $\text{visit}(\cdot)$ is $\mathcal{O}(|\mathcal{E}|)$. As a conclusion, the time complexity of the function $\text{main}(\cdot)$ is $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$. $\square$

Notice that a directed graph can have at most $|\mathcal{V}|^2$ edges. Therefore, the time complexity of the function $\text{main}(\cdot)$ depicted in Figure 2.4 is $\mathcal{O}(|\mathcal{V}|^2)$ in this case. This means that the time complexity of the BFS algorithm for directed graphs is quadratic with respect to the number of nodes of the graph.

2.2.2 Depth-first Search Algorithm

Depth-first search (DFS) is a popular algorithm for traversing directed graphs. As the name of the algorithm suggests, it searches “deeper” in the graph whenever possible.

A representation of the DFS algorithm is depicted in Figure 2.5. Lines in red and labelled with the symbol ‘+’ have been added to explicitly store the *DFS-graph*, a directed labelled graph that represents the order in which the directed graph is traversed by DFS. Like in the case of a BFS-graph, every node of a DFS-graph is labelled with two timestamps: one timestamp indicating when the node was *discovered* (i.e., when it was visited for the first time), and a second timestamp stating when the node has been *explored* (i.e., when all its successors have been discovered and explored). Notice that the notion of being *discovered* is similar in BFS and DFS. This is not the case with the notion of being *explored*. During BFS, a node has been explored if all its direct successors have been visited. In DFS, a node has been explored if all its successors (all nodes that can be reached from it) have been also explored.

We use also $\text{discovered-order}(v)$ and $\text{explored-order}(v)$ to refer to the timestamps that indicates when the node v was discovered and explored, respectively.

We define a DFS-graph computed by DFS on directed graphs as follows.

Definition 2.17 (DFS-graph). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $\text{main}(\cdot)$ be the function depicted in Figure 2.5. A depth-first-search graph (DFS-graph) of $\mathcal{G}$ is a directed labelled graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ with $\mathcal{G}' = \text{main}(\mathcal{G})$.* $\dashv$

In general, there are several DFS-graphs of a directed graph depending on the order in which the nodes are traversed. The traversal order is stored in the node labels of a DFS-graph.

```

function main( $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ )
1:  $\mathcal{V}' := \{\}$  +
2:  $\mathcal{E}' := \{\}$  +
3: for every  $v \in \mathcal{V}$  do
4:   discovered-order( $v$ ) := -1
5:   explored-order( $v$ ) := -1
6: end-for
7: order-counter := 0
8: for every  $v \in \mathcal{V}$  do
9:   if (discovered-order( $v$ ) = -1) then
10:    visit( $v$ )
11:   end-if
12: end-for
13:  $\mathcal{L} := \{(\text{discovered-order}(v), \text{explored-order}(v)) \mid v \in \mathcal{V}\}$  +
14: return  $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{L})$  +

function visit( $v$ )
1: order-counter := order-counter + 1
2: discovered-order( $v$ ) := order-counter
3:  $\mathcal{V}' := \mathcal{V}' \cup \{v\}$  +
   // Visit all successors of v
4: for every  $s \in \mathcal{E}(v)$  do
5:    $\mathcal{E}' := \mathcal{E}' \cup \{(v, s)\}$  +
6:   if (discovered-order( $s$ ) = -1) then
7:     visit( $s$ ) // recursive call
8:   end-if
9: end-for
   // v has been explored
10: order-counter := order-counter + 1
11: explored-order( $v$ ) := order-counter

```

Fig. 2.5 DFS algorithm for directed graphs.

Suppose $e = (v, s)$ is an directed edge in a DFS-graph. Based on the timestamps of each of the nodes of e , we can distinguish four types of edges: (1) *tree edge* (s was discovered for the first time), (2) *forward edge* (s was already discovered but not before v and none of them have been explored yet), (c) *back edge* (s was discovered before v but none of them have been explored yet), and (d) *cross edge* (s was discovered and explored before v was discovered).

The following proposition can be readily be seen and it makes precise the relation between the timestamps of the nodes of a directed edge and its type.

Proposition 2.18. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be a DFS-graph. Let $e = (v, s) \in \mathcal{E}$ be a directed edge in $\mathcal{G}$. Let discovered-order and explored-order be the functions determined by $\mathcal{L}$. Then:*

- (a) *If $(\text{discovered-order}(v) < \text{discovered-order}(s))$ and $(\text{explored-order}(v) > \text{explored-order}(s))$, then $e = (v, s)$ is a tree edge or a forward edge*
- (b) *If $(\text{discovered-order}(v) > \text{discovered-order}(s))$ and $(\text{explored-order}(v) < \text{explored-order}(s))$, then $e = (v, s)$ is a back edge*
- (c) *If $(\text{discovered-order}(v) > \text{discovered-order}(s))$ and $(\text{explored-order}(v) > \text{explored-order}(s))$, then $e = (v, s)$ is a cross edge* ⊢

Example 2.19. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, where $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $\mathcal{E} = \{e_1 = (v_1, v_2), e_2 = (v_1, v_4), e_3 = (v_2, v_5), e_4 = (v_3, v_5), e_5 = (v_3, v_6), e_6 = (v_4, v_2), e_7 = (v_5, v_4)\}$*

The depth-first search algorithm can traverse the nodes and edges of $\mathcal{G}$ following different orders. A possible DFS-graph that can be produced by the depth-first search algorithm is represented in Figure 2.6, where root nodes (nodes that are not connected from any other node) are identified by dashed circles. The labels $\mathcal{L}(v) := (\text{discovered-order}(v), \text{explored-order}(v))$, for every node $v \in \mathcal{V}$ are represented between angle brackets, $< ./ >$.

In Figure 2.6, the DFS-graph has two main sub-graphs. The first one has v_1 as its root node and includes also the nodes v_2 , v_4 and v_5 . The second one has two nodes, v_3 and v_6 , and v_3 is the root node.

We can verify in this example Proposition 2.18. For instance, $e_1 = (v_1, v_2)$ is a tree edge. So, $\text{discovered-order}(v_1) < \text{discovered-order}(v_2)$ and $\text{explored-order}(v_1) < \text{explored-order}(v_2)$. Similarly, $e_2 = (v_1, v_4)$ is a forward edge. So, $\text{discovered-order}(v_1) < \text{discovered-order}(v_4)$ and $\text{explored-order}(v_1) < \text{explored-order}(v_4)$. Notice that in this case, v_4 was discovered before the edge e_2 is added to the depth-first tree. There is also a back edge, $e_6 = (v_4, v_2)$, and $\text{discovered-order}(v_4) > \text{discovered-order}(v_2)$ and $\text{explored-order}(v_4) < \text{explored-order}(v_2)$.

To cover all types of edges, the example also includes a cross edge, $e_4 = (v_3, v_5)$. Thus, $\text{discovered-order}(v_3) > \text{discovered-order}(v_5)$ and $\text{explored-order}(v_3) > \text{explored-order}(v_5)$.

Notice that the first root node can be v_3 instead of v_1 and the the DFS-graph would look slightly different. For instance, e_4 would not be a cross edge, e_6 would not be a back edge and e_2 would not be a forward edge. Moreover, e_1 and e_2 become cross edges and e_3 a backward edge.

Independently of the DFS-graph produced by the depth-first search algorithm, the nodes v_2 , v_4 and v_5 are connected by two tree edges and one back edge. This configuration correspond to a cycle in the depth-first tree where the three nodes are strongly-connected. This means that the three nodes form a strongly connected component (SCC).

◁

Fig. 2.6 (Example 2.19): DFS algorithm

In Proposition 2.21, we prove correctness of the function $\text{main}(\cdot)$ for DFS. We start by showing some properties of a DFS-graph in the following lemma.

Lemma 2.20 (Properties of DFS for directed graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $\mathcal{G}' = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be a DFS-graph of $\mathcal{G}$. Let $\text{visit}(\cdot)$ be the function depicted in Figure 2.5. Then the following holds:*

- (i) *for all $v \in \mathcal{V}$: $\text{visit}(v)$ is called exactly once;*
- (ii) *for all $v \in \mathcal{V}$: $\text{visit}(v)$ iterates over each outgoing edge from v exactly once;*
- (iii) *$\mathcal{V}' = \mathcal{V}$ and $\mathcal{E}' = \mathcal{E}$; and*

(iv) for all $x, y \in \mathcal{V}$: $x \geq_{\mathcal{G}} y$ iff $x \geq_{\mathcal{G}'} y$. ⊢

Proof. We start the proof of Lemma 2.20 by showing item (i). Every call of $\text{visit}(\cdot)$ is guarded by an if-statement. Therefore, the call of function $\text{visit}(\cdot)$ for a node v is executed if v has not been discovered before ($\text{discovered-order}(v) = -1$). In Line 2 of $\text{visit}(v)$, $\text{discovered-order}(v)$ is set to a value larger than -1 (to be precise the value stored in order-counter). This ensures that $\text{visit}(v)$ will not be called again. The for-loop from Line 4 to Line 9 in $\text{main}(\cdot)$ ensures that $\text{visit}(v)$ is called for all nodes $v \in \mathcal{V}$.

Consider item (ii). By item (i), the function $\text{visit}(v)$ is called exactly once, for all $v \in \mathcal{V}$. The for-loop from Line 4 to Line 9 in $\text{visit}(v)$ iterates over each outgoing edge of v just one time.

Item (iii) follows from items (i) and (ii). By item (i), the function $\text{visit}(\cdot)$ is called exactly one time for each node of the graph. This implies that each node of the graph will be also stored in $\mathcal{V}'$ by the function $\text{visit}(\cdot)$ (Line 3). By item (ii), the function $\text{visit}(\cdot)$ iterates over each outgoing edge of the node that is currently visited and in each iteration, the edge is added into $\mathcal{E}'$. Therefore, item (iii) holds.

Item (iv) can be readily be seen because by item (iii), $\mathcal{G}$ and $\mathcal{G}'$ have the same nodes and edges. This implies that node-to-node connectivity in $\mathcal{G}$ is preserved in $\mathcal{G}'$. □

The next proposition follows from Lemma 2.20.

Proposition 2.21 (Correctness of DFS for directed graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $\text{main}(\cdot)$ be the function depicted in Figure 2.5. Then:*

- (i) *The function $\text{main}(\cdot)$ terminates.*
- (ii) *The function $\text{main}(\cdot)$ traverses $\mathcal{G}$ completely.* ⊢

We show that the time complexity of DFS is linear with respect to the size of the input graph.

Proposition 2.22 (Time Complexity of DFS for Directed Graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $\text{main}(\cdot)$ be the function depicted in Figure 2.5. Then: the time complexity of the function $\text{main}(\cdot)$ is $O(|\mathcal{V}| + |\mathcal{E}|)$.* ⊢

Proof. The function $\text{main}(\cdot)$ includes two for-loops (Lines 3-6 and Lines 8-12) that iterates over all nodes of the directed graph. Moreover, this function must store the labels of each node (Line 13). This can be done using an extra for-loop that iterates over all nodes of the graph. This implies $3 \times |\mathcal{V}|$ iterations inside the function $\text{main}(\cdot)$.

We continue with the study of the time complexity of the function $\text{visit}(\cdot)$. By item (i) of Lemma 2.20, $\text{visit}(v)$ is called exactly once, for all $v \in \mathcal{V}$. By item (ii) of Lemma 2.20, $\text{visit}(v)$ iterates over each outgoing edge of v exactly once, for all $v \in \mathcal{V}$. Assuming that the operations inside the for-loop in $\text{visit}(\cdot)$ are executed in constant time (as the rest of operations outside the for-loop), the time complexity of the execution of $|\mathcal{V}|$ calls (recursive or not) of the function $\text{visit}(\cdot)$ is $\mathcal{O}(|\mathcal{E}|)$. As a conclusion, the time complexity of the function $\text{main}(\cdot)$ is $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$. $\square$

Notice that a directed graph can have at most $|\mathcal{V}|^2$ edges. Therefore, the time complexity of the function $\text{main}(\cdot)$ depicted in Figure 2.5 is $\mathcal{O}(|\mathcal{V}|^2)$ in this case. This means that the time complexity of the DFS algorithm for directed graphs is quadratic with respect to the number of nodes of the graph.

2.3 Strong Connectivity in Directed Graphs

2.3.1 Naïve Algorithm

The algorithm depicted in Figure 2.7 computes the strongly-connected components of a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. In short, the algorithm computes the connected component of each node of the graph and identifies which of these connected components are the same. The nodes with the same connected component are in the same strongly-connected component (Proposition 2.3).

The algorithm has been defined using two functions: $\text{main}(\cdot)$ and $\text{visit}(\cdot)$, where the latter is nearly equivalent to the function $\text{visit}(\cdot)$ of the BFS algorithm (Figure 2.4). Notice that the function $\text{visit}(\cdot)$ of the naïve algorithm can compute the connected component of a node if all nodes of the graph are initialized as unvisited: $\text{visited}(v) = \text{false}$, for all $v \in \mathcal{V}$ (Lines 4–6 in function $\text{main}(\cdot)$).

The function $\text{main}(\cdot)$ starts by fixing a total order $<$ on the set of nodes in $\mathcal{G}$ (Line 1). The order is used to represent a set $\mathcal{S}_v$ of all nodes connected from a node v as a sequence π_v of nodes (Line 8). The sequence π_v for every node v in $\mathcal{G}$ is then introduced as a branch in a labelled tree T (Lines 9–13). Note that each prefix of π_v is considered as a node in T . If the node is not already contained in T , then the node is labelled with the empty set (Line 10). Then the node v is added to the final node of the branch π_v (Line 12). Two different sequences of ordered nodes are introduced as different branches in T . A sequence of ordered nodes that is contained in another sequence of nodes already included in the tree might be part of the same branch of the tree (this depends of the total order $<$ considered). The function

```

function main( $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ )
1: fix total order  $<$  on  $\mathcal{V}$ 
2:  $T := \emptyset$ 
3: for every  $v \in \mathcal{V}$  do
4:   for every  $v \in \mathcal{V}$  do
5:     visited( $v$ ) := false
6:   end-for
7:    $\mathcal{S}_v := \text{visit}(v)$  // collect all successors of  $v$ 
8:    $\pi_v := \text{sort}_{<}(\mathcal{S}_v)$ 
      // store all successors of  $v$  in  $T$ 
9:   for every  $\pi' \in \text{prefix}(\pi_v)$  do
10:    if  $\pi' \notin \text{dom}(T)$  then  $T := T \cup \{(\pi', \emptyset)\}$ 
11:  end-for
12:   $T(\pi_v) := T(\pi_v) \cup \{v\}$ 
13: end-for
      // return SCCs( $\mathcal{G}$ )
14: return  $\{T(\pi) \mid \pi \in \text{dom}(T) \text{ and } T(\pi) \neq \emptyset\}$ 

function visit( $v$ )
1: visited( $v$ ) := true
2:  $\mathcal{V}' := \{v\}$ 
3:  $Q := \{v\}$ 
4: enqueue( $Q, v$ )
      // visit all successors of  $v$ 
5: while  $Q \neq \{v\}$  do
6:    $u := \text{dequeue}(Q)$ 
7:    $\mathcal{V}' := \mathcal{V}' \cup \{u\}$ 
8:   for every  $s \in \mathcal{E}(u)$  do
9:     if (visited( $s$ ) = false) then
10:       visited( $s$ ) := true //  $s$  has been visited
11:       enqueue( $Q, s$ )
12:     end-if
13:   end-for
14: end-while
      // return the connected component of  $v$ 
15: return  $\mathcal{V}'$ 

```

Fig. 2.7 Naïve algorithm to compute the SCCs of a directed graph.

$\text{main}(\cdot)$ returns the labels of the tree T that are not empty (Line 14). Each non-empty label corresponds to a strongly-connected component of $\mathcal{G}$.

Proposition 2.23 (Correctness of the naïve algorithm for directed graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ be the functions depicted in Figure 2.7. Then:*

- (i) *for all $v \in \mathcal{V}$: $\text{visit}(v) = \text{CC}(v)$,*
- (ii) *$\text{main}(\mathcal{G})$ terminates,*
- (iii) *$\text{main}(\mathcal{G}) = \text{SCCs}(\mathcal{G})$.* ⊥

Proof. We start the proof of Proposition 2.23 by showing item (i). The function $\text{visit}(\cdot)$ is very similar to the function $\text{visit}(\cdot)$ of BFS (Figure 2.4). One small difference is that $\text{visit}(\cdot)$ uses a boolean label to establish for each node $v \in \mathcal{V}$, if v has been visited ($\text{visited}(v) = \text{true}$) or not ($\text{visited}(v) = \text{false}$). This avoids that the same node is visited more than one time. Therefore, it can readily be seen that item (i) and item (ii) in Lemma 2.14 also hold for the function $\text{visit}(\cdot)$ of the naïve algorithm for computing strongly-connected components. As a result, $\text{visit}(v)$ visits all nodes connected from v and only one time. This implies that $\text{visit}(v)$ returns $\text{CC}(v)$ (Line 15).

Consider item (ii). The major for-loop of $\text{main}(\cdot)$ iterates over each node of the graph (Lines 3&13). In each iteration, a node $v \in \mathcal{V}$ is selected and the following steps should be completed: (1) all nodes are marked as unvisited (Lines 4–6), (2) the connected component of node v is computed and stored in $\mathcal{S}_v$, and (3) $\mathcal{S}_v$ is included in T . It can readily be seen that $\text{main}(\cdot)$ terminates. The number of iterations of the major for-loop is bounded by the number of nodes of the graph. Moreover, the number of steps inside of each iteration is also limited.

Consider item (iii). By Proposition 2.3, two nodes $v, w \in \mathcal{V}$ are in the same strongly-connected component iff $\text{CC}(v) = \text{CC}(w)$. The function $\text{main}(\cdot)$ calls the function $\text{visit}(\cdot)$ to compute for each node of the graph, its connected component (Line 7). In Lines 9–13, the function $\text{main}(\cdot)$ uses a tree data structure, T , to compare each of the connected component computed by $\text{visit}(\cdot)$. It can readily be seen that two connected components that are the same will be stored in the same branch of T . It is also apparent that the non-empty labels of the nodes of T correspond to the strongly-connected components of $\mathcal{G}$. Therefore, item (iii) holds. □

We show that the time complexity of the naïve algorithm for computing strongly-connected components in directed graphs is not linear with respect to the size of the graph. In fact, it is cubic (in the worst-case) with respect to the number of nodes of the graph.

Proposition 2.24 (Time complexity of the naïve algorithm for directed graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let $\text{main}(\cdot)$ be the function depicted in Figure 2.7. Then: the time complexity of function $\text{main}(\cdot)$ is $\mathcal{O}(|\mathcal{V}|^3)$.* $\dashv$

Proof. We start the proof by analysing the time complexity of $\text{visit}(\cdot)$ of the naïve algorithm. This function is very similar to the function $\text{visit}(\cdot)$ of BFS (Figure 2.4). Therefore, it can readily be seen that item (i) and item (ii) in Lemma 2.14 also hold for the function $\text{visit}(\cdot)$ of the naïve algorithm. By item (i) of Lemma 2.14, each node of the graph will be enqueued (and dequeued) exactly one time by the function $\text{visit}(\cdot)$. By item (ii), the function $\text{visit}(\cdot)$ iterates over each outgoing edge of the node that has been currently enqueued just one time. Assuming that the operations inside the for-loop in $\text{visit}(\cdot)$ are executed in constant time (as the rest of operations outside the for-loop), the time complexity of the execution of the function $\text{visit}(\cdot)$ is $\mathcal{O}(|\mathcal{E}|)$.

We continue the proof by analysing the time complexity of $\text{main}(\cdot)$ of the naïve algorithm. In Line 1, a total order on $\mathcal{V}$ is fixed. This can be done in linear time with respect to the number of nodes. After this task is executed, a for-loop over each of the nodes of the graph must be completed (Lines 3&13). Most of the processing done by $\text{main}(\cdot)$ occurs inside of this for-loop which requires $|\mathcal{V}|$ iterations. The first task to be achieved inside of the loop is to set all nodes of the graph as unvisited. This is done by another for-loop that iterates over each node of the graph (Lines 4–6). This requires $|\mathcal{V}|$ iterations and each iteration is completed in constant time. The next task is to compute the connected component of the node currently selected by the main for-loop (Line 7). This is done by the function $\text{visit}(\cdot)$ in $|\mathcal{E}|$ steps in the worst-case. Next, the nodes of the connected component are ordered based on the total order that has been defined. Depending on the algorithm implemented by the function $\text{sort}_{<}(\cdot)$, this can be done in $|\mathcal{V}| \times \log |\mathcal{V}|$ steps in the worst-case. The task that comes after aims to identify if the connected component computed earlier already exist (Lines 9–13). This is done by extracting each node of the connected component and by adding the node into a labelled tree which represents all connected components that has been computed so far. The order of the nodes of each connected component is preserved in the labelled tree. For this kind of trees, the operations $\text{insert}()$ and $\text{search}()$ can be executed in $\log |\mathcal{V}|$ steps [10]. Therefore, Line 10 can be completed in $2 \times \log |\mathcal{V}|$ steps in the worst-case. Consequently, the for-loop in Lines 9–11 can be completed in $|\mathcal{V}| \times 2 \times \log |\mathcal{V}|$ steps. The last task in the main for-loop is to add a node in the label of the last node of the connected component that has been included in T (Line 12). This requires to search for the last node added in T . As it was explained earlier, this operation can be done in $\log |\mathcal{V}|$ steps.

As a summary, the number of steps to be completed inside of the main for-loop (Lines 3&13) is the following: $|\mathcal{V}| + |\mathcal{E}| + |\mathcal{V}| \times \log |\mathcal{V}| + |\mathcal{V}| \times 2 \times \log |\mathcal{V}| + \log |\mathcal{V}|$. This means that

the number of steps to complete the execution of the function $\text{main}(\cdot)$ is: $|\mathcal{V}| + |\mathcal{V}| \times (|\mathcal{V}| + |\mathcal{E}| + |\mathcal{V}| \times \log |\mathcal{V}| + |\mathcal{V}| \times 2 \times \log |\mathcal{V}| + \log |\mathcal{V}|)$. A directed graph can have at most $|\mathcal{V}|^2$ edges. Then, the time complexity of $\text{main}(\cdot)$ can be defined with respect to the number of nodes of the graph as follows: $\mathcal{O}(|\mathcal{V}|^3)$. $\square$

Example 2.25 shows how the naïve algorithm computes the strongly-connected components of a directed graph. In particular, the example illustrates how the nodes of each strongly-connected components are stored in the labelled tree T and how the total order can affect the size of T .

Example 2.25. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $\mathcal{E} = \{e_1 = (v_1, v_2), e_2 = (v_1, v_3), e_3 = (v_2, v_3), e_4 = (v_2, v_4), e_5 = (v_4, v_3), e_6 = (v_4, v_5), e_7 = (v_5, v_2), e_8 = (v_5, v_6), e_9 = (v_6, v_3), e_{10} = (v_6, v_4)\}$.

Line 1 in function $\text{main}(\cdot)$ fix a total order $<$ on $\mathcal{V}$. In this example, let $<$ be a total order such that $v_1 < v_2 < v_3 < v_4 < v_5 < v_6$. Notice that $<$ is equal to the transitive closure of the relation $\{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_6)\}$.

For each node of the graph, the function $\text{visit}(\cdot)$ returns the connected component (Line 7). These are the connected components computed in each iteration of the for-loop (Lines 3&13): $CC(v_1) = \mathcal{V}$, $CC(v_3) = \{v_3\}$, and $CC(v_2) = CC(v_4) = CC(v_5) = CC(v_6) = \{v_2, v_3, v_4, v_5, v_6\}$.

Each connected component will be stored in a labelled tree T to identify which of them are the same connected component. Given the fixed order of this example, the tree consists of the following branches b_1 , b_2 and b_3 , where

$$\begin{aligned} b_1 : & (v_1, \{\}) \longrightarrow (v_1 v_2, \{\}) \longrightarrow (v_1 v_2 v_3, \{\}) \longrightarrow (v_1 v_2 v_3 v_4, \{\}) \longrightarrow \\ & (v_1 v_2 v_3 v_4 v_5, \{\}) \longrightarrow (v_1 v_2 v_3 v_4 v_5 v_6, \{v_1\}) \\ b_2 : & (v_2, \{\}) \longrightarrow (v_2 v_3, \{\}) \longrightarrow (v_2 v_3 v_4, \{\}) \longrightarrow (v_2 v_3 v_4 v_5, \{\}) \longrightarrow \\ & (v_2 v_3 v_4 v_5 v_6, \{v_2, v_4, v_5, v_6\}) \\ b_3 : & (v_3, \{v_3\}) \end{aligned}$$

The function $\text{main}(\cdot)$ returns the following strongly-connected components (one for each non-empty label in T): $\{\{v_1\}, \{v_2, v_4, v_5, v_6\}, \{v_3\}\}$.

Notice that the order $<$ can affect the size of T . Let $<$ be a total order such that $v_3 < v_2 < v_4 < v_5 < v_6 < v_1$. Then, T only has one branch b_1 , where

$$\begin{aligned} b_1 : & (v_3, \{v_3\}) \longrightarrow (v_3 v_2, \{\}) \longrightarrow (v_3 v_2 v_4, \{\}) \longrightarrow (v_3 v_2 v_4 v_5, \{\}) \longrightarrow \\ & (v_3 v_2 v_4 v_5 v_6, \{v_2, v_4, v_5, v_6\}) \longrightarrow (v_3 v_2 v_4 v_5 v_6 v_1, \{v_1\}) \end{aligned}$$

$\triangleleft$

2.3.2 Tarjan's Algorithm

In this section, we discuss the algorithm by Tarjan (or simply *Tarjan*) [39] that extends DFS to compute strongly-connected components.² Tarjan is depicted in Figure 2.8. The lines in blue and labelled with the symbol ‘*’ are only for Tarjan. The lines in black are for DFS and Tarjan.

To better understand how Tarjan works, we first define the notion of a root-node which corresponds to the first node of a strongly-connected component that is discovered during the execution of the function $\text{visit}(\cdot)$ (Figure 2.8).

Definition 2.26 (Root-node of a strongly-connected component). *Let $\mathcal{G}'$ be a DFS-graph of a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Let $\mathcal{S} \subseteq \mathcal{V}$ be a strongly-connected component of $\mathcal{G}$. A node $v \in \mathcal{V}$ is the root node of $\mathcal{S}$ in $\mathcal{G}'$ iff for all $w \in \mathcal{S} \setminus \{v\}$: $\text{discovered-order}(v) < \text{discovered-order}(w)$, where $\text{discovered-order}(\cdot)$ is determined by $\mathcal{G}'$. $\dashv$*

The problem of finding the strongly-connected components of a directed graph can be reduced to the problem of finding the root-nodes of these strongly connected components. We therefore need to show how to easily identify a root-node. This can be done by using the function $\text{min-order}(\cdot)$ (also known as $\text{lowlink}(\cdot)$ [39]).

The function $\text{min-order}(v)$ maps a node v to the value $\text{discovered-order}(w)$ of some node w that has been discovered before v (i.e., it holds that $\text{discovered-order}(w) \leq \text{discovered-order}(v)$) such that v and w are contained in the same strongly-connected component. That is, there is a cross edge or back edge from a descendant of v to w , and the root node of the strongly-connected component that contains v and w is an ancestor of both nodes.

In order to define the function $\text{min-order}(\cdot)$, we introduce the notion of descendant of a node first. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $\mathcal{G}' = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be a directed labelled graph such that $\mathcal{G}' = \text{main}(\mathcal{G})$ is a DFS-graph of $\mathcal{G}$. Let $v \in \mathcal{V}$ be two nodes. A node s is a *descendant of v in $\mathcal{G}'$* if the following conditions are satisfied: (i) $v \geq_{\mathcal{G}'} s$, (ii) $\text{discovered-order}(v) \leq \text{discovered-order}(s)$, and (iii) $\text{explored-order}(s) \leq \text{explored-order}(v)$. We denote with $\text{descendants}(v)$ the set of descendants of v in $\mathcal{G}'$. Similarly, we defined the notion of an ancestor. A node s is an *ancestor of v in $\mathcal{G}'$* if the following conditions are satisfied: (i) $s \geq_{\mathcal{G}'} v$, (ii) $\text{discovered-order}(s) \leq \text{discovered-order}(v)$, and (iii) $\text{explored-order}(v) \leq \text{explored-order}(s)$. We denote with $\text{ancestors}(v)$ the set of ancestors of v in $\mathcal{G}'$.

Notice that for the definitions of descendants and ancestors, we are considering a DFS-graph $\mathcal{G}'$ such that $\mathcal{G}'$ has already been computed. Before the computation of DFS-graph starts, the value provided by $\text{discovered-order}(\cdot)$ and $\text{explored-order}(\cdot)$ for each node of the

²The material presented in this section has been adapted from [1, 39].

```

function main( $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ )
1: list-of-sccs :=  $\{\{\}\}$  *
2: stack :=  $\square$  *
3: for every  $v \in \mathcal{V}$  do
4:   discovered-order( $v$ ) :=  $-1$ 
5:   explored-order( $v$ ) :=  $-1$ 
6:   min-order( $v$ ) :=  $-1$  *
7: end-for
8: order-counter := 0
9: for every  $v \in \mathcal{V}$  do
10:  if (discovered-order( $v$ ) =  $-1$ ) then
11:    visit( $v$ )
12:  end-if
13: end-for
14: return list-of-sccs *

function visit( $v$ )
1: order-counter := order-counter + 1
2: discovered-order( $v$ ) := order-counter
3: min-order( $v$ ) := order-counter *
4: push(stack,  $v$ ) *
5: for every  $s \in \mathcal{E}(v)$  do // visit all successors of  $v$ 
6:  if (discovered-order( $s$ ) =  $-1$ ) then
7:    visit( $s$ ) // recursive call
8:    min-order( $v$ ) := min(min-order( $v$ ), min-order( $s$ )) *
9:  else *
10:   if stack contains  $s$  then *
11:     min-order( $v$ ) := min(min-order( $v$ ), discovered-order( $s$ )) *
12:   end-if *
13: end-if
14: end-for
15: order-counter := order-counter + 1
16: explored-order( $v$ ) := order-counter
17: if (min-order( $v$ ) = discovered-order( $v$ )) then // compute SCC if  $v$  is a root *
18:  scc :=  $\{\}$  *
19:  repeat *
20:     $s := \text{pop}(\text{stack})$  *
21:    add(scc,  $s$ ) *
22:  until ( $s = v$ ) *
23:  add(list-of-sccs, scc) *
24: end-if *

```

Fig. 2.8 Tarjan for directed graphs.

DFS-graph is -1 (Lines 3-7 in function $\text{main}(\cdot)$). For instance, it is possible to reach an ancestor of a node such that this ancestor has not been explored yet.

Definition 2.27 (Minimum discovered-order). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be a DFS-graph. We define the function $\text{min-order}: \mathcal{V} \rightarrow \mathbb{Z}$ as follows: For all $v \in \mathcal{V}$,*

$$\text{min-order}(v) = \min\{ \text{discovered-order}(w) \mid w, d \in \mathcal{V} \text{ s.t. (i) - (iv)} \},$$

where

- (i) d is a descendant of v in $\mathcal{G}$,
- (ii) $(d, w) \in \mathcal{E}$,
- (iii) w is connected to v in $\mathcal{G}$, and
- (iv) $\text{discovered-order}(w) \leq \text{discovered-order}(v)$. ⊥

The edge (d, w) selected in Item (ii) of Definition 2.27 represents a back or cross edge that connects a descendant of v (Item (i)) with w (Item (iii)), which is a node that has been discovered before v (Item (iv)). If w has not been explored yet (or w is explored after v is explored), then (d, w) is a back edge. If w has already been explored, then (d, w) is a cross edge. The nodes w and v are mutually connected and, thus, they are contained in the same strongly-connected component.

We can now use the function $\text{min-order}(\cdot)$ to identify the root-node of each strongly-connected component of a directed graph as follows.

Proposition 2.28 (Characterization of root-nodes). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $\mathcal{G}' = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be a directed labelled graph such that $\mathcal{G}' = \text{main}(\mathcal{G})$ is a DFS-graph of $\mathcal{G}$. Let $\mathcal{S} \subseteq \mathcal{V}$ be a strongly-connected component of $\mathcal{G}'$.*

Then: v is the root-node of $\mathcal{S}$ iff $\text{discovered-order}(v) = \text{min-order}(v)$ ⊥

Proof. ‘ $\Rightarrow$ ’: Suppose v is the root-node of $\mathcal{S}$. By Definition 2.27, $\text{min-order}(v) \leq \text{discovered-order}(v)$. Suppose $\text{min-order}(v) < \text{discovered-order}(v)$. Then there exists an ancestor w of v in $\mathcal{S}$ such that $\text{discovered-order}(w) = \text{min-order}(v)$. Therefore, there is a path from v to w and vice versa. That is, v and w are in the same strongly connected component $\mathcal{S}$ but w was discovered before v . Thus, v cannot be a root-node of $\mathcal{S}$, which is a contradiction. Hence, $\text{min-order}(v) = \text{discovered-order}(v)$.

‘ $\Leftarrow$ ’: Suppose that $\text{discovered-order}(v) = \text{min-order}(v)$ and that v is not the root-node of $\mathcal{S}$. If v is not the root-node of $\mathcal{S}$, then some proper ancestor r of v is the root-node of $\mathcal{S}$. Thus,

there is a path P from the node v to the root-node r . Consider the first edge of the path P from a descendant of the node v to a node w which is not a descendant of v . The edge is either a back edge or a cross edge to a node that has been discovered before the node v . In either case $\text{discovered-order}(w) < \text{discovered-order}(v)$.

It remains to show that the root-node r and the node w are in the same strongly connected component S . There must be a path from the root-node r to the node v since r is an ancestor of v . The path P goes from v to w to r . Hence r and w are in the same strongly connected component S . Therefore, $\text{min-order}(v) \leq \text{discovered-order}(w) < \text{discovered-order}(v)$, which is a contradiction. $\square$

The proofs for termination (Proposition 2.21, Item (i)) and time complexity of DFS (Proposition 2.22) for directed graphs are also valid for this version of Tarjan's algorithm. To show that this algorithm is correct, we have to prove that the strongly-connected components returned by $\text{main}(\cdot)$ in list-of-sccs (Line 14) are correct. To this aim, we introduce the function $\text{node}(\cdot)$. Let $\text{node} : \mathbb{N} \rightarrow \mathcal{V}$ be the function that maps a $\text{discovered-order}(\cdot)$ number $i \in \{-1, 1, \dots, 2 \times |\mathcal{V}|\}$ to a node $\text{node}(i)$ such that $i = \text{discovered-order}(\text{node}(i))$. In other words, $\text{node}(\cdot)$ is the inverse function of $\text{discovered-order}(\cdot)$.

Theorem 2.29 (Correctness of Tarjan for directed graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph, and let $\text{main}(\cdot)$ be the function depicted in Figure 2.8. Then: $\text{main}(\mathcal{G}) = \text{SCCs}(\mathcal{G})$. $\dashv$*

Proof. In this proof [39, 1], we refer to $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ in Figure 2.8. Let $\mathcal{G}' = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be a directed labelled graph such that $\mathcal{G}' = \text{main}(\mathcal{G})$ is a DFS-graph of $\mathcal{G}$. By Item (i) of Lemma 2.20, $\text{visit}(\cdot)$ is called once for every node $v \in \mathcal{V}$.

To prove correctness of Theorem 2.29, it suffices to prove, by induction on the number of terminated calls of the function $\text{visit}(\cdot)$, that when the execution of $\text{visit}(v)$ ends, $\text{min-order}(v)$ has been computed correctly for a node $v \in \mathcal{V}$. To make this condition more precise, we show the induction hypothesis (IH) that after the call of $\text{visit}(v)$, for $v \in \mathcal{V}$, has terminated it holds that

$$\{v, \text{node}(\text{min-order}(v))\} \subseteq S$$

for some strongly connected component $S \in \text{SCCs}(\mathcal{G})$.

We show IH by induction on the number n of calls to $\text{visit}(\cdot)$ that have terminated. In the base case, we have that $n = 1$. Suppose $v \in \mathcal{V}$ is a node such that $\text{visit}(v)$ has terminated. Consider the execution of $\text{visit}(v)$. By Lines 2&3 it holds that $\text{min-order}(v) = \text{discovered-order}(v)$. We distinguish two cases depending on the number of successors of v : (a) $|\mathcal{E}(v)| = 0$, and (b) $|\mathcal{E}(v)| > 0$. Consider Case (a). As $|\mathcal{E}(v)| = 0$, no successor is selected in the for-loop from Line 5 to Line 14. Therefore, $\text{min-order}(v)$ is not modified

as Lines 8&11 are not executed. As $\text{min-order}(v) = \text{discovered-order}(v)$, it holds that $v = \text{node}(\text{min-order}(v))$. It holds that $v \in S$ for some $S \in \text{SCCs}(\mathcal{G})$. Hence, IH follows.

Consider Case (b). Let $w \in \mathcal{E}(v)$. Then w is selected in the for-loop from Line 5 to Line 14. Since $\text{visit}(v)$ is the only call that has terminated, $\text{visit}(w)$ has not terminated. Therefore, $\text{discovered-order}(w) \neq -1$ and $w \in \text{stack}$. It holds that $\text{discovered-order}(w) < \text{discovered-order}(v)$; otherwise $\text{visit}(w)$ would have terminated before $\text{visit}(v)$. Because $\text{discovered-order}(w) \neq -1$, the condition of the if-statement in Line 6 is not satisfied. The algorithm proceeds to Line 10. Since $w \in \text{stack}$, the condition of the if-statement in Line 10 is satisfied. By Line 11 it holds that $\text{min-order}(v) = \text{discovered-order}(w)$. It holds that $w = \text{node}(\text{min-order}(v))$. From $w \geq_{\mathcal{G}} v$ and $(v, w) \in \mathcal{E}$ it follows that $\{v, w\} \subseteq S$ for some $S \in \text{SCC}(\mathcal{G})$. After the for-loop from Line 5 to Line 14 has finished, $\text{min-order}(v)$ is set to the smallest value of $\text{discovered-order}(w)$, where w ranges over $\mathcal{E}(v)$. Hence IH holds. This finishes the induction base.

Consider the induction step. Suppose we have shown the IH for n . We now show the IH for $n+1$ terminated calls of $\text{visit}(\cdot)$. Suppose $v \in \mathcal{V}$ is the node such that $\text{visit}(v)$ is the $n+1$ -st call that has terminated. Consider the execution of $\text{visit}(v)$. We distinguish two cases depending on the number of successors of v : (a) $|\mathcal{E}(v)| = 0$, and (b) $|\mathcal{E}(v)| > 0$. Case (a) can be shown the same way as Case (a) of the induction base. Consider Case (b). Suppose $w \in \mathcal{E}(v)$. Then w is selected in the for-loop from Line 5 to Line 14. We proceed to Line 6. We distinguish two cases depending on whether or not w has been already discovered: (b1) $\text{discovered-order}(w) \neq -1$, and (b2) $\text{discovered-order}(w) = -1$.

Consider Case (b1). The condition in the if-statement in Line 6 is not satisfied. We proceed to Line 10. We distinguish two cases depending on whether or not w is on the stack: (b1)(a) $w \in \text{stack}$, and (b1)(b) $w \notin \text{stack}$.

Consider Case (b1)(a). As $w \in \text{stack}$, the condition in the if-statement in Line 10 is satisfied. We proceed to Line 11. We show that $\{w\} \geq_{\mathcal{G}} v$. Then $\text{discovered-order}(v)$ will be set in Line 11 to the value of $\text{discovered-order}(w)$ of some w . Consequently, v and $\text{node}(\text{min-order}(v))$ are contained in the same strongly connected component of $\mathcal{G}$. Hence IH holds.

To show $\{w\} \geq_{\mathcal{G}} v$ in Case (b1)(a), we distinguish three cases depending on when w was discovered: (b1)(a1) $\text{discovered-order}(w) < \text{discovered-order}(v)$, (b1)(a2) $\text{discovered-order}(w) = \text{discovered-order}(v)$, and (b1)(a3) $\text{discovered-order}(w) > \text{discovered-order}(v)$. Consider Case (b1)(a1). We distinguish two new cases depending on whether $\text{visit}(w)$ has terminated: (b1)(a1)(i) $\text{explored-order}(w) = -1$, and (b1)(a1)(ii) $\text{explored-order}(w) \neq -1$. In Case (b1)(a1)(i), w is an ancestor of v . Then, (v, w) is a back-edge. Therefore, $\{w\} \geq_{\mathcal{G}} v$. Consider Case (b1)(a1)(ii). Then the call $\text{visit}(w)$ has terminated. Therefore, (v, w) is a cross

edge. As $w \in \text{stack}$, there is a node $r \in \text{stack}$ such that r is a common ancestor of v and w . Then, $\{w\} \geq_{\mathcal{G}} v$ as $\{w\} \geq_{\mathcal{G}} r$ and $\{r\} \geq_{\mathcal{G}} v$.

Consider Case (b1)(a2), i.e. $\text{discovered-order}(w) = \text{discovered-order}(v)$. It holds that $v = w$. Then (v, w) is a reflexive edge. Then $\{w\} \geq_{\mathcal{G}} v$.

Consider Case (b1)(a3), i.e. $\text{discovered-order}(w) > \text{discovered-order}(v)$. As $w \in \text{stack}$, w can reach a common ancestor, say r , of v and w . It follows that $\{w\} \geq_{\mathcal{G}} v$ as $\{w\} \geq_{\mathcal{G}} r$ and $\{r\} \geq_{\mathcal{G}} v$. Note that there is an alternative path from v to w that was already considered and $\text{min-order}(v)$ was updated with the value in $\text{discovered-order}(r)$. Then, w was explored earlier and (v, w) is a forward edge.

Consider Case (b1)(b), i.e. $w \notin \text{stack}$. Then, the call $\text{visit}(w)$ has terminated. We have that $\{w\} \not\geq_{\mathcal{G}} v$ as w is not an ancestor of v , and w is not connected to a common ancestor of v and w . Although (v, w) is a cross edge, v and w are not in the same strongly-connected component. Consequently, the condition of the if-statement in Line 10 is not satisfied and the value $\text{min-order}(v)$ is not modified.

Consider Case (b2), i.e. $\text{discovered-order}(w) = -1$. The node w was not discovered before and the condition of the if-statement in Line 6 is satisfied. Note that (v, w) is a tree edge. The algorithm proceeds to Line 7 and executes a recursive call of $\text{visit}(w)$. We distinguish three cases depending on the value of $\text{min-order}(w)$: (b2)(a1) $\text{min-order}(w) < \text{min-order}(v)$, (b2)(a2) $\text{min-order}(w) = \text{min-order}(v)$, and (b2)(a3) $\text{min-order}(w) > \text{min-order}(v)$.

Consider Case (b2)(a1). Then, $\{w\} \geq_{\mathcal{G}} v$ as w is connected to an ancestor of v via a back edge (and possible a cross edge) from a descendant of w . The value in $\text{min-order}(v)$ is set to $\text{min-order}(w)$ in Line 8. Therefore, v and $\text{node}(\text{min-order}(v))$ are contained in the same strongly connected component of $\mathcal{G}$.

Consider Case (b2)(a2). Suppose that $\text{min-order}(w) = \text{min-order}(v)$. Then, $\{w\} \geq_{\mathcal{G}} v$ as w is connected to v via a back edge from w or a descendant of w . The value in $\text{min-order}(v)$ is not changed in Line 8. Therefore, v and $\text{node}(\text{min-order}(v))$ are contained in the same strongly connected component of $\mathcal{G}$.

Consider Case (b2)(a3), i.e. $\text{min-order}(w) > \text{min-order}(v)$. Then, $\{w\} \not\geq_{\mathcal{G}} v$ as w is not connected to v or an ancestor of v . Therefore, it is not possible that w and v are mutually connected. The value in $\text{min-order}(v)$ is not changed in Line 8.

This concludes the IH. We have shown that for all cases considered the IH follows as every node is contained in some SCC. $\square$

Example 2.30 shows how `tarjanDG` works in detail. In particular, the example illustrates how the use of the information stored in the stack and in the labels $\text{min-order}(\cdot)$ and $\text{discovered-order}(\cdot)$ contributes to identify the SCCs of a directed graph.

Fig. 2.9 (Example 2.30): Tarjan's algorithm for directed graphs

Example 2.30. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $\mathcal{E} = \{e_1 = (v_1, v_2), e_2 = (v_1, v_3), e_3 = (v_2, v_3), e_4 = (v_2, v_4), e_5 = (v_4, v_3), e_6 = (v_4, v_5), e_7 = (v_5, v_2), e_8 = (v_5, v_6), e_9 = (v_6, v_3), e_{10} = (v_6, v_4)\}$.

In Lines 1&2, the function $\text{main}(\cdot)$ initializes list-of-sccs (stores the list of SCCs that have been found) as an empty list of lists of nodes and stack as an empty stack. The data structure stack stores the nodes that have been visited by the function $\text{visit}(\cdot)$ and not yet explored or they have been explored but $\text{min-order}(\cdot) < \text{discovered-order}(\cdot)$ (they belong to a new SCC). In Lines 4-7, the function $\text{main}(\cdot)$ initializes the value of the labels $\text{discovered-order}(\cdot)$, $\text{explored-order}(\cdot)$, and $\text{min-order}(\cdot)$ of each node to -1. This step is represented in Figure 2.9 (a) with the value pair $\langle \text{discovered-order}(\cdot), \text{min-order}(\cdot) \rangle$ that

labels each node. In Line 8, order-counter (a counter that determines the order in which a node has been discovered or explored) is initialized to zero.

According to Figure 2.9 (b), the first node that it is visited by the function $\text{visit}(\cdot)$ is the node v_1 . This is why $\text{discovered-order}(v_1)$ and $\text{min-order}(v_1)$ are equal to one ($< 1, 1 >$). The node v_1 is also added into the stack (Line 4 of the function $\text{visit}(\cdot)$) as the first visited node.

Following depth-first search, the function $\text{visit}(\cdot)$ visits the nodes v_2 , v_4 and v_5 as it is indicated by the stack in Figure 2.9 (b). From node v_5 , the function $\text{visit}(\cdot)$ reaches again the node v_2 which is in the stack. As a result, $\text{min-order}(v_5)$ is updated with the value stored in $\text{discovered-order}(v_2)$ which is 2.

From v_5 , v_6 can be visited and from v_6 , the node v_3 is reached by the function $\text{visit}(\cdot)$, as it is indicated by Figure 2.9 (c). The node v_3 does not have successors and the function $\text{visit}(\cdot)$ discovers the first SCC, $\text{scc}_1 = \{v_3\}$, which is included in the list of SCCs: $\text{list-of-sccs} = \{\{v_3\}\}$. The only node that can be visited from v_6 is v_4 which is in the stack. As a result, $\text{min-order}(v_6)$ is updated with the value stored in $\text{discovered-order}(v_4)$ which is 3.

After all successors of v_6 has been visited the function $\text{visit}(\cdot)$ backtracks to continue the exploration of successors of v_5 . All successors of v_5 has been explored and the function $\text{visit}(\cdot)$ backtracks again to continue the exploration of successors of v_4 . The value of $\text{min-order}(v_4)$ is updated with the value of $\text{min-order}(v_5)$ which is 2. From the node v_4 , it can be reached the node v_3 which has been visited before but it is not in the stack. So, the value of $\text{min-order}(v_4)$ is not updated. No more successors can be explored from v_4 and the function $\text{visit}(\cdot)$ backtracks to continue the exploration of successors of v_2 .

From v_2 , it can be reached the node v_3 which has been visited before but it is not in the stack. So, the value of $\text{min-order}(v_2)$ is not updated. All successors of v_2 has been explored and $\text{discovered-order}(v_2) = \text{min-order}(v_2)$. The function $\text{visit}(\cdot)$ finds a new SCC. The node v_2 and all nodes in the stack stored after v_2 are removed and included in a new SCC: $\text{scc}_2 = \{v_2, v_4, v_5, v_6\}$ and $\text{stack} = \{v_1\}$.³ The scc_2 is also included in the list of SCCs: $\text{list-of-sccs} = \{\{v_3\}, \{v_2, v_4, v_5, v_6\}\}$. The result of this step is indicated in Figure 2.9 (d).

The function $\text{visit}(\cdot)$ backtracks again to continue the exploration of direct successors of v_1 . The node v_3 is a successor of v_1 that has been visited before but it is not in the stack. Therefore, the value of $\text{min-order}(v_1)$ is not updated. The values stored in $\text{discovered-order}(v_1)$ and $\text{min-order}(v_1)$ are the same and the function $\text{visit}(\cdot)$ finds a new SCC: $\text{scc}_3 = \{v_1\}$. The scc_3 is also included in the list of SCCs: $\text{list-of-sccs} = \{\{v_3\}, \{v_2, v_4, v_5, v_6\}, \{v_1\}\}$. The result of this step is indicated in Figure 2.9 (d). $\triangleleft$

³Notice that $\text{min-order}(v_4) = \text{min-order}(v_5) = \text{discovered-order}(v_2)$ but this is not the case $\text{min-order}(v_6)$. However, the function $\text{visit}(\cdot)$ computes the SCCs correctly even if the value of $\text{min-order}(v_6)$ is incorrect.

2.4 Directed Hypergraphs

A directed hypergraph [18] is a tuple $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a non-empty finite set and $\mathcal{E} \subseteq 2^{\mathcal{V}} \times 2^{\mathcal{V}}$. $\mathcal{V}$ is known as the set of nodes of $\mathcal{H}$ and $\mathcal{E}$ is the set of directed hyperedges.⁴ An ordered pair $e = (T(e), H(e))$ is a directed hyperedge, $e \in \mathcal{E}$, where $|T(e)| > 0$, $|H(e)| > 0$ and $T(e) \cap H(e) = \emptyset$. $H(e)$ ($T(e)$) is known as the *head* (*tail*) and represents a set of nodes where the hyperedge ends (starts). We call a directed hyperedge e *simple* if $|T(e)| = 1$ and $|H(e)| = 1$ (i.e., if e corresponds to a directed edge); otherwise, if $|T(e)| > 1$ or $|H(e)| > 1$, e is called *complex*. A *B-hyperedge* (*F-hyperedge*) is a directed hyperedge with only one node in the head (tail): $|H(e)| = 1$ ($|T(e)| = 1$). Directed hypergraphs containing *B-hyperedges* (*F-hyperedges*) only are called *B-hypergraphs* (*F-hypergraphs*). A *BF-hypergraph* is a directed hypergraph that has both types of hyperedges. The size of a directed hypergraph $\mathcal{H}$ is defined as the sum of the tail and head nodes of each hyperedge together with the number of nodes of the hypergraph: $|\mathcal{H}| = |\mathcal{V}| + \sum_{e \in \mathcal{E}} (|T(e)| + |H(e)|)$. Notice that there are $2^{|\mathcal{V}|}$ different possible tails or heads. Suppose that the tails and heads of a directed hyperedge can be empty and can have some nodes in common (but not all) and let $n = |\mathcal{V}|$. Then, the number of different hyperedges in a directed hypergraph can be computed as follows [40]:

$$\sum_{k=0}^n \binom{n}{k} 2^k = 3^n.$$

The previous result implies that the number of hyperedges in a directed hypergraph is exponential (in the worst-case) with respect to the number of nodes whereas in a directed graph is only quadratic. From the implementation point of view, it might be required the design of succinct representations of directed hypergraphs [40, 27].

Given a directed hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and a node $v \in \mathcal{V}$, we define the *backward star* of v ($BS_{\mathcal{H}}(v)$) and the *forward star* ($FS_{\mathcal{H}}(v)$) of v as follows: $BS_{\mathcal{H}}(v) = \{e \in \mathcal{E} \mid v \in H(e)\}$ and $FS_{\mathcal{H}}(v) = \{e \in \mathcal{E} \mid v \in T(e)\}$. This means that $BS_{\mathcal{H}}(v)$ is the set of all hyperedges that have v in their heads and $FS_{\mathcal{H}}(v)$ is the set of all hyperedges that have v in their tails. If $|BS_{\mathcal{H}}(v)| = 0$ ($|FS_{\mathcal{H}}(v)| = 0$), v is known as *root* (*leaf*) node which means that v does not have any incoming (outgoing) hyperedges. To preserve the notation already introduced in Section 2.2, we denote with $\mathcal{E}^-(v)$, the set of all hyperedges that have v in their heads and we denote with $\mathcal{E}(v)$, the set of all hyperedges that have v in their tails: $\mathcal{E}^-(v) = BS_{\mathcal{H}}(v)$ and $\mathcal{E}(v) = FS_{\mathcal{H}}(v)$.

Definition 2.31 (Graph fragment of a directed hypergraph). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and let $\mathcal{E}' \subseteq \mathcal{E}$ be the set of F-hyperedges in $\mathcal{H}$.*

⁴The material presented in this section has been adapted from [18, 31, 40, 5, 2, 28].

The graph fragment of $\mathcal{H}$ is a directed graph $\mathcal{G}(\mathcal{H}) = (\mathcal{V}, \mathcal{E}'')$ such that $\mathcal{E}'' = \{(t, h) \mid (t, H) \in \mathcal{E}', h \in H \text{ and } t, h \in \mathcal{V}\}$. $\dashv$

Next, we define the notion of B -connectivity in directed hypergraphs. This is the only notion of connectivity that we will consider in this document.

Definition 2.32 (B -connectivity in directed hypergraphs). Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, and let $v \in \mathcal{V}$ be a node. Let $\mathcal{V}' \subseteq \mathcal{V}$ be a set of nodes. v is B -connected in $\mathcal{H}$ from a set of nodes $\mathcal{V}'$ (written $\mathcal{V}' \geq_{B(\mathcal{H})} v$) if

(i) $v \in \mathcal{V}'$, or

(ii) there is a hyperedge e such that $v \in H(e)$ and for all $w \in T(e)$: $\mathcal{V}' \geq_{B(\mathcal{H})} w$.

We denote with $\geq_{B(\mathcal{H})}(\mathcal{V}')$ the B -connected component of $\mathcal{V}'$ in $\mathcal{H}$ which is the maximal set of nodes B -connected from $\mathcal{V}'$ in $\mathcal{H}$: $\geq_{B(\mathcal{H})}(\mathcal{V}') = \{v \in \mathcal{V} \mid \mathcal{V}' \geq_{B(\mathcal{H})} v\}$. ⁵ $\dashv$

According to [28], condition (ii) in Definition 2.32 is not precise enough and it should be modified as follows: (ii) there is a hyperedge e such that $v \in H(e)$ and for all $w \in T(e)$: $\mathcal{V}' \geq_{B(\mathcal{H}_{-e})} w$, where $\mathcal{H}_{-e} = (\mathcal{V}_{-e}, \mathcal{E}_{-e})$, $\mathcal{E}_{-e} = \mathcal{E} \setminus \{e\}$ and $\mathcal{V}_{-e} = \mathcal{V} \setminus \{v \in \mathcal{V} \mid BS_{\mathcal{H}}(v) \cup FS_{\mathcal{H}}(v) = \{e\}\}$. ⁶

B -connectivity has popularized the use of directed hypergraphs to represent n -ary relations in problems related to relational databases, propositional logic, biological networks and ontologies [18, 31, 22, 38]. In B -connectivity, the tail nodes of a hyperedge have a prominent role because only when all of them are reached, it is possible to reach the head nodes. Besides B -connectivity, there are different notions of connectivity in directed hypergraphs. Among others, F -connectivity and BF -connectivity are also very popular in the literature [18, 31]. Like tail nodes in B -connectivity, these notions stress the importance of head nodes in a directed hyperedge. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, and let $v, w \in \mathcal{V}$ be two nodes. Let $\mathcal{H}^R = (\mathcal{V}^R, \mathcal{E}^R)$ be the *reverse* (transpose or converse) of the directed hypergraph $\mathcal{H}$ such that $\mathcal{V}^R = \mathcal{V}$ and $\mathcal{E}^R = \{(H, T) \mid (T, H) \in \mathcal{E}\}$. v is F -connected from a node w in $\mathcal{H}$ (written $w \geq_{F(\mathcal{H})} v$) if and only if w is B -connected from v in $\mathcal{H}^R$ ($v \geq_{B(\mathcal{H}^R)} w$). Furthermore, v is BF -connected from w (written $w \geq_{BF(\mathcal{H})} v$) in $\mathcal{H}$ if and only if $w \geq_{B(\mathcal{H})} v$ and $w \geq_{F(\mathcal{H})} v$.

A *directed path* in a directed hypergraph is a sequence of nodes and hyperedges such that each edge pointing from a node in the sequence to its successor in the sequence. This is made precise in the following definition.

⁵We might also refer to the B -connected component of $\mathcal{V}' \subseteq \mathcal{V}$ in $\mathcal{H}$ as $CC_{B(\mathcal{H})}(\mathcal{V}')$.

⁶It is unclear why this condition improves the original condition (ii) in Definition 2.32.

Definition 2.33 (Directed path for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, and let $x, y \in \mathcal{V}$. A directed path from s to t in $\mathcal{H}$ is a sequence $\pi_{s,t}$ of the form $\pi_{s \rightarrow t} = (v_1, e_1, v_2, e_2, \dots, e_{n-1}, v_n)$, $n > 0$, such that*

- $v_i \in \mathcal{V}$, for all $i \in \{1, \dots, n\}$,
- $e_j \in \mathcal{E}$, for all $j \in \{1, \dots, n-1\}$,
- $v_1 = s$ and $v_1 \in T(e_1)$,
- $v_n = t$ and $v_n \in H(e_{n-1})$, and
- $v_i \in H(e_{i-1}) \cap T(e_i)$, for all $i \in \{2, \dots, n-1\}$.

A directed path $\pi_{s \rightarrow t}$ from s to t in $\mathcal{H}$ is acyclic if $s \neq t$. A directed path $\pi_{s \rightarrow t}$ in $\mathcal{H}$ is simple if $v_i \neq v_j$ for all $i, j \in \{1, \dots, n-1\}$ with $i \neq j$. $\dashv$

Notice that having a directed path between two nodes does not imply that the two nodes are B -connected in the sense of Definition 2.32.

Example 2.34. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, where $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$ and $\mathcal{E} = \{e_1 = (\{v_1\}, \{v_2\}), e_2 = (\{v_2, v_3\}, \{v_4\})\}$. Let $\pi_{v_1 \rightarrow v_4} = (v_1, e_1, v_2, e_2, v_4)$ be a directed path in $\mathcal{H}$. By Definition 2.33, $\pi_{v_1 \rightarrow v_4}$ is a valid directed path starting on node v_1 and finishing on node v_4 . By Definition 2.32, however, node v_4 is not B -connected from node v_1 . $\triangleleft$*

The notion of directed path for directed hypergraphs seems not to be sufficient. It is needed a different notion that at least ensures that the first and last node of the path is B -connected. This turns out not to be simple to achieve. The original definition of B -hyperpath [18] has been reconsidered in later publications [32, 40]. The problem was that Gallo et al. [18] provide a topological characterization of B -hyperpaths not directly related to Definition 2.32. As it is reported by [32, 28], it is possible to find examples of B -hyperpaths that fulfil the definition provided by [18] but not Definition 2.32.

Before we define what a B -hyperpath is, we introduce the notion of subhypergraph.

Definition 2.35 (Subhypergraph for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph. $\mathcal{H}' = (\mathcal{V}', \mathcal{E}')$ is a subhypergraph of $\mathcal{H}$ if: (i) $\mathcal{V}' \subseteq \mathcal{V}$, (ii) $\mathcal{E}' \subseteq \mathcal{E}$, and (iii) for all $(T(e), H(e)) \in \mathcal{E}'$, $T(e) \subseteq \mathcal{V}'$ and $H(e) \subseteq \mathcal{V}'$. $\dashv$*

Given a directed hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, we define a B -hyperpath from a set of nodes $S \subseteq \mathcal{V}$ to a node $v \in \mathcal{V}$ as a subhypergraph of $\mathcal{H}$ similar to it was done by [32, 5].

Definition 2.36 (*B-hyperpath for directed hypergraphs*). Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph. Let $S \subseteq \mathcal{V}$ be a non-empty set of nodes, and let $t \in \mathcal{V}$ be a node. A *B-hyperpath* from S to t in $\mathcal{H}$ is a hypergraph $\mathcal{H}' = (\mathcal{V}', \mathcal{E}')$ such that

- (i) $\mathcal{H}'$ is a subhypergraph of $\mathcal{H}$;
- (ii) $S \geq_{B(\mathcal{H}')} t$;
- (iii) If $|\mathcal{E}'| = 1$ and $e_1 \in \mathcal{E}'$, then $t \in H(e_1)$ and $T(e_1) \subseteq \{S\}$; and
- (iv) If $|\mathcal{E}'| = k$ and $k > 1$, then $\mathcal{E}'$ can be ordered in a sequence $e_1, e_2, \dots, e_k$ such that, $t \in H(e_k)$ and for every $e_i \in \mathcal{E}'$, it holds that $T(e_i) \subseteq \{S\} \cup H(e_1) \cup H(e_2) \cup \dots \cup H(e_{i-1})$.

We denote with $\mathcal{H}_{B(S \rightarrow t)} = (\mathcal{V}_{B(S \rightarrow t)}, \mathcal{E}_{B(S \rightarrow t)})$, the *B-hyperpath* from S to t .

A *B-hyperpath* from S to t produced by condition (i) of Definition 2.32 (reflexivity) is a subhypergraph $\mathcal{H}_{B(S \rightarrow t)} = (\mathcal{V}_{B(S \rightarrow t)}, \mathcal{E}_{B(S \rightarrow t)})$ such that $\mathcal{V}_{B(S \rightarrow t)} = S$ and $\mathcal{E}_{B(S \rightarrow t)} = \{\}$.⁷

A *B-hyperpath* $\mathcal{H}' = (\mathcal{V}', \mathcal{E}')$ is cyclic if there are at least two nodes $u, w \in \mathcal{V}'$ such that both nodes are mutually *B-connected* in $\mathcal{H}'$: $u \geq_{B(\mathcal{H}')} w$ and $w \geq_{B(\mathcal{H}')} u$. $\dashv$

The following example illustrates the notion of *B-hyperpath* and how it differs from the notion of directed path.

Example 2.37. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, where $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5\}$ and $\mathcal{E} = \{e_1 = (\{v_1\}, \{v_3\}), e_2 = (\{v_2\}, \{v_4\}), e_3 = (\{v_3, v_4\}, \{v_5\})\}$. Let $S \subseteq \mathcal{V}$ be a set of nodes such that $S = \{v_1, v_2\}$.

By Definition 2.36, there is a *B-hyperpath* from S to v_5 , $\mathcal{H}_{S \rightarrow v_5} = (\mathcal{V}_{S \rightarrow v_5}, \mathcal{E}_{S \rightarrow v_5})$, where $\mathcal{V}_{S \rightarrow v_5} = \mathcal{V}$ and $\mathcal{E}_{S \rightarrow v_5} = \mathcal{E}$. From v_1 to v_5 , it is not possible to find a *B-hyperpath* in $\mathcal{H}$ but it is possible to find the following directed path: $\pi_{v_1 \rightarrow v_5} = (v_1, e_1, v_3, e_3, v_5)$. $\triangleleft$

Notice that it is also possible to represent a *B-hyperpath* as labelled tree, where the nodes correspond to the hyperedges of the *B-hyperpath*. By traversing the labelled tree from the starting nodes of the *B-hyperpath* to the target node, we can produce a sequence of hyperedges. This tree representation is named as *unfolded B-hyperpath* [5].

Definition 2.36 does not require that *B-hyperpaths* are represented as a sequence of nodes and hyperedges (*linear property* [40]) like in directed paths. In fact, it might not be always possible to produce such sequence because the tail and head of a hyperedge might contain more than one node and each tail node of a directed hyperedge might be reached from different hyperedges. However, there are applications where it is required to produce

⁷The hyperpath produced by reflexivity (Definition 2.32 (i)) is a directed hypergraph without hyperedges. This means that the definition of directed hypergraphs should allow hypergraphs without hyperedges.

B -hyperpaths that preserve the linear property (or linearity). This can be achieved with the notion of L -hyperpath (*Linear-hyperpath*) defined as follows [40].

Definition 2.38. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, and let $s, t \in \mathcal{V}$ be two nodes. Let $\mathcal{V}' = \{v_1, v_2, \dots, v_m, v_{m+1}\}$ be a set of distinct nodes, where $\mathcal{V}' \subseteq \mathcal{V}$, $v_1 = s$, $v_{m+1} = t$, and $v_i \neq v_j$ for all $i, j \in \{1, \dots, m+1\}$ with $i \neq j$. Let $\mathcal{E}' = \{e_1, e_2, \dots, e_m\}$ be a set of distinct hyperedges, where $\mathcal{E}' \subseteq \mathcal{E}$, and $e_i \neq e_j$ for all $i, j \in \{1, \dots, m\}$ with $i \neq j$.

An L -hyperpath $\pi_{L(s \rightarrow t)} = (v_1, e_1, v_2, e_2, \dots, v_m, e_m, v_{m+1})$ is a sequence of distinct nodes and hyperedges such that

- (i) $s = v_1$;
- (ii) $t = v_{m+1}$;
- (iii) for every $v_i \in \mathcal{V}' \setminus \{v_{m+1}\}$, it holds that $v_i \in T(e_i)$ and $v_{i+1} \in H(e_i)$; and
- (iv) for every $e_i \in \mathcal{E}'$, it holds that $T(e_i) \subseteq \{s\} \cup H(e_1) \cup \dots \cup H(e_{i-1})$.

The node t is L -connected from the node s in $\mathcal{H}$ (written $s \geq_{L(\mathcal{H})} t$) if there is a L -hyperpath from s to t in $\mathcal{H}$.

If $v_1 = v_{m+1}$ then $\pi_{L(s \rightarrow t)} = (v_1, e_1, v_2, e_2, \dots, v_m, e_m, v_{m+1})$ is an L -hypercycle in $\mathcal{H}$. $\dashv$

The next proposition follows from Definitions 2.36 and 2.38 [40, 28].

Proposition 2.39. Every L -hyperpath is a B -hyperpath but not vice versa. $\dashv$

The following example illustrates the different notions of connectivity discussed in this section.

Example 2.40. Figure 2.10 shows four different directed hypergraphs selected from [28]. Each directed hypergraph has been designed to illustrate one or more connectivity notions between two nodes (filled in grey).

In Figure 2.10(1), node v_3 is $B/F/BF/L$ -connected from node v_1 . Because from v_1 is possible to reach v_2 using the simple directed hyperedge e_1 , all tail nodes of the directed hyperedge e_2 are B -connected from v_1 . Therefore, v_3 is B -connected from v_1 . Similarly, we can produce the reverse version of this directed hypergraph and show that v_1 is B -connected from v_3 . Consequently, v_3 is F -connected from v_1 and also, v_3 is BF -connected from v_1 . In the same figure, we can define the following L -hyperpath from v_1 to v_3 : $\pi_{L(v_1 \rightarrow v_3)} = (v_1, e_1, v_2, e_2, v_3)$. Therefore, v_3 is L -connected from v_1 .

In Figure 2.10(2), node v_3 is only F -connected from node v_1 . If we produce the reverse directed hypergraph then it becomes obvious that v_1 is B -connected from v_3 . Therefore, v_3 is

Fig. 2.10 (Example 2.40): Connectivity notions in directed hypergraphs [28].

F -connected from v_1 . Notice that in this case, v_3 is not B -connected from v_1 . The reason is that it is required that both tails nodes of the hyperedge e_2 are B -connected from v_1 and this is not possible because v_2 is not B -connected from v_1 (in fact, v_2 does not have any incoming hyperedge). As a result, v_3 is not BF -connected (or L -connected because Proposition 2.39) from v_1 .

It can readily be seen that node v_3 is $B/F/BF$ -connected from node v_1 in Figure 2.10(3). This is not true in the case of L -connectivity because it is not possible to define an L -hyperpath between v_1 and v_3 .

In Figure 2.10(4), node v_3 is only B/L -connected from node v_1 . The node v_1 is B -connected with itself. Thus, all tail nodes of e_1 are B -connected from v_1 and as a result, v_3 is B -connected from v_1 . Notice that in the reverse hypergraph, v_1 is not B -connected from v_3 . Therefore, v_3 is not F -connected from v_1 and consequently, v_3 is not BF -connected from v_1 . It can readily be seen that $\pi_{L(v_1 \rightarrow v_3)} = (v_1, e_1, v_3)$ is a L -hyperpath. Thus, v_3 is L -connected from v_1 . $\triangleleft$

After fixing the notions of connectivity and hyperpath on directed hypergraphs, the notions of strong connectivity and strongly-connected component can be defined similarly as in directed graphs.

In a directed hypergraph $\mathcal{H}$, two nodes v and w are *strongly-connected* if v is connected to w and vice versa. Like in directed graphs, strong-connectivity is an equivalence relation on the set of nodes of a directed hypergraph. Therefore, strong-connectivity partitions the nodes of a directed hypergraph into equivalence classes. The equivalence classes for strong-connectivity are called strongly-connected components.

A *strongly-connected component (SCC)* of a directed hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a maximal set $\mathcal{V}' \subseteq \mathcal{V}$ of nodes such that for every pair v and w of nodes from $\mathcal{V}'$, it holds that v and w are strongly-connected. Let $\text{SCCs}(\mathcal{H})$ be the set of all strongly-connected components of $\mathcal{H}$. Moreover, a directed hypergraph is strongly-connected if all its nodes are strongly-connected to one another. This implies $\text{SCCs}(\mathcal{H}) = \{\mathcal{V}\}$, where $\mathcal{H} = (\mathcal{V}, \mathcal{E})$.

For the computation of strongly-connected components we can apply any of the connectivity notions that we have discussed earlier in this section. If we do not indicate the connectivity notion that we are using then it is assumed that we refer to B -connectivity.

Like in directed graphs, the partition of a directed hypergraph under an equivalence relation over the nodes of the hypergraph induces a *quotient* directed hypergraph. We make this notion more precise in the following definition.

Definition 2.41 (Quotient Hypergraph). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph. Let $\simeq$ be an equivalence relation over $\mathcal{V}$. The quotient of $\mathcal{H}$ under $\simeq$, written $\mathcal{H}/\simeq$, is the graph $\mathcal{H}/\simeq = (\mathcal{V}/\simeq, \mathcal{E}_{\simeq})$, where*

- $\mathcal{V}/\simeq = \{[x]_{\simeq} \mid x \in \mathcal{V}\}$; and
- $e = (T(e), H(e)) \in \mathcal{E}_{\simeq}$ iff there is an $e' \in \mathcal{E}$ such that $T(e) = \{[x]_{\simeq} \mid x \in T(e')\}$, $H(e) = \{[x]_{\simeq} \mid x \in H(e')\}$ and $T(e) \cap H(e) = \emptyset$. $\dashv$

We can now define the notion of *condensed* directed hypergraph as the quotient hypergraph derived from the strong- B -connectivity relation over the nodes of a directed hypergraph.

Definition 2.42 (Condensed Directed Hypergraph). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph. Let $\simeq_B$ be the strong- B -connectivity relation over $\mathcal{V}$ in $\mathcal{H}$. The condensed directed hypergraph of $\mathcal{H}$ is defined as the quotient hypergraph of $\mathcal{H}$ under $\simeq_B$, written $\mathcal{H}/\simeq_B$, is the hypergraph $\mathcal{H}/\simeq_B = (\mathcal{V}/\simeq_B, \mathcal{E}_{\simeq_B})$, where $\mathcal{V}/\simeq_B = \{[x]_{\simeq_B} \mid x \in \mathcal{V}\}$ is the set of all strongly- B -connected components of $\mathcal{H}$ ($\text{SCCs}(\mathcal{H})$). $\dashv$*

The following proposition establishes the relation between B -connectivity in a directed hypergraph $\mathcal{H}$ and B -connectivity in the condensed directed hypergraph $\mathcal{H}/\simeq_B$. This is done

by showing that a B -connected component in $\mathcal{H}$ is equivalent to a B -connected component in $\mathcal{H}/\simeq_B$ after the nodes of the B -connected component in $\mathcal{H}/\simeq_B$ are expanded to retrieve the original nodes that were defined in $\mathcal{H}$.

Proposition 2.43. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and let $\mathcal{V}' \subseteq \mathcal{V}$ be a set of nodes. Let $\geq_B$ be the B -connectivity relation in $\mathcal{H}$ and let $\simeq_B$ be the strong- B -connectivity relation in $\mathcal{H}$. Then: $\geq_B(\mathcal{V}') = \bigcup \geq_{\mathcal{H}/\simeq_B}([\mathcal{V}']_{\simeq_B})$, where $[\mathcal{V}']_{\simeq_B} = \{[v]_{\simeq_B} \mid v \in \mathcal{V}'\}$. $\dashv$*

Proof. “ $\subseteq$ ”: We show the contrapositive. Suppose $[v]_{\simeq_B} \notin \geq_{\mathcal{H}/\simeq_B}([\mathcal{V}']_{\simeq_B})$. Then there is no B -hyperpath in $\mathcal{H}/\simeq_B$ from $[\mathcal{V}']_{\simeq_B}$ to $[v]_{\simeq_B}$. This implies that there is no B -hyperpath in $\mathcal{H}$ from $\mathcal{V}'$ to v , i.e., $v \notin \geq_B(\mathcal{V}')$.

“ $\supseteq$ ”: We show the contrapositive. Suppose $v \notin \geq_B(\mathcal{V}')$. Then there is no B -hyperpath in $\mathcal{H}$ from $\mathcal{V}'$ to v . Then there is no B -hyperpath in $\mathcal{H}/\simeq_B$ from $[\mathcal{V}']_{\simeq_B}$ to $[v]_{\simeq_B}$, i.e. $[v]_{\simeq_B} \notin \geq_{\mathcal{H}/\simeq_B}([\mathcal{V}']_{\simeq_B})$. $\square$

Expanding a collapsed strongly-connected components can be done using set-union because each collapsed strongly-connected component represents a set of nodes. This is why in Proposition 2.43, we refer to the B -connected component of $[\mathcal{V}']_{\simeq_B}$ after it has been expanded using the expression: $\bigcup \geq_{\mathcal{H}/\simeq_B}([\mathcal{V}']_{\simeq_B})$. Note that $\mathcal{V}' \subseteq \bigcup [\mathcal{V}']_{\simeq_B}$, where $[\mathcal{V}']_{\simeq_B} = \{[v]_{\simeq_B} \mid v \in \mathcal{V}'\}$.

The condensed directed hypergraph is the result of collapsing the strongly- B -connected components into single nodes and turning hyperedges between nodes into hyperedges between the newly formed nodes. Therefore, we obtain a *directed acyclic hypergraph (DAH)*, a directed hypergraph where there are no cyclic B -hyperpaths. The following proposition can readily be seen.

Proposition 2.44. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph. Let $\simeq_B$ be the strong- B -connectivity relation over $\mathcal{V}$ in $\mathcal{H}$. Let $\mathcal{H}/\simeq_B = (\mathcal{V}/\simeq_B, \mathcal{E}_{\simeq_B})$ be the condensed directed hypergraph of $\mathcal{H}$.*

Then: $\mathcal{H}/\simeq_B$ is a directed acyclic hypergraph. $\dashv$

We say that a strongly- B -connected component of a directed hypergraph is *terminal* if its out-degree in the condensed version of the original hypergraph is zero. This means that there are not edges in the condensed directed hypergraph that has a terminal strongly- B -connected component in their tail.

2.4.1 Representation of Directed Hypergraphs

The most well-known data-structure for storing directed hypergraphs is the *adjacency-matrix* (also known as incidence-matrix [18]). An adjacency-matrix describes a directed hypergraph

as a matrix where each element can store three possible numbers. These numbers are usually $\{-1, 0, 1\}$. The number of rows of the adjacency-matrix are equal to the number of nodes of the directed hypergraph and the number of columns of the matrix are equal to the number of hyperedges. This means that the size of an adjacency-matrix that represents a directed hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is $|\mathcal{V}| \times |\mathcal{E}|$. It is assumed that the nodes are numbered $1, 2, \dots, |\mathcal{V}|$ in some arbitrary order. The same is also true for the hyperedges that are also numbered $1, 2, \dots, |\mathcal{E}|$ in some arbitrary order. Then, an element $a_{i,j}$ of the matrix has value -1 if and only if the node v_i is in the tail of the hyperedge e_j ($v_i \in T(e_j)$). Similarly, $a_{i,j}$ has value 1 if and only if the node v_i is in the head of the hyperedge e_j ($v_i \in H(e_j)$). Furthermore, $a_{i,j}$ has value 0 if and only if the node v_i is not in the tail or in the head of the hyperedge e_j .

Definition 2.45 (Adjacency-matrix for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph. Let $\text{id-node}: \mathcal{V} \rightarrow \mathbb{N}$ be a total function assigning to every node $v \in \mathcal{V}$ a natural value $\text{id-node}(v) \in \{1, \dots, |\mathcal{V}|\}$. Let $\text{id-edge}: \mathcal{E} \rightarrow \mathbb{N}$ be a total function assigning to every hyperedge $e \in \mathcal{E}$ a natural value $\text{id-edge}(e) \in \{1, \dots, |\mathcal{E}|\}$. Let $v \in \mathcal{V}$ be a node and $e \in \mathcal{E}$ be a hyperedge such that $\text{id-node}(v) = i$ and $\text{id-edge}(e) = j$, where $i \in \{1, \dots, |\mathcal{V}|\}$ and $j \in \{1, \dots, |\mathcal{E}|\}$. Let $a_{i,j}$ be the entry in a matrix such that it is located in the i th row and j th column.*

The adjacency-matrix representation of $\mathcal{H}$, denoted with $AM(\mathcal{H})$, is a $|\mathcal{V}| \times |\mathcal{E}|$ matrix such that for all $a_{i,j} \in AM(\mathcal{H})$, $a_{i,j} = -1$ iff $v \in T(e)$, $a_{i,j} = 1$ iff $v \in H(e)$, and $a_{i,j} = 0$ otherwise. $\dashv$

The following example illustrates how a directed hypergraph can be described using an adjacency-matrix representation.

Example 2.46. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, where $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5\}$ and $\mathcal{E} = \{e_1 = (\{v_1\}, \{v_2\}), e_2 = (\{v_2\}, \{v_3\}), e_3 = (\{v_2, v_3\}, \{v_4\}), e_4 = (\{v_3, v_4\}, \{v_5\}), e_5 = (\{v_4, v_5\}, \{v_1, v_2\})\}$.*

The adjacency matrix (Figure 2.11(2)) is of size 5×5 , where the element $a_{2,3}$, for instance, has value -1 and this indicates that $v_2 \in T(e_3)$. The element $a_{4,3}$ has value 1 and this means that $v_4 \in H(e_3)$. $\triangleleft$

As it was observed on directed graphs, it is not very optimal (from the point of view of memory space consumption) to store a sparse directed hypergraph using an adjacency-matrix. To the best of our knowledge, there is not a data-structure for directed hypergraphs similar to the adjacency-list for directed graphs. However, a directed hypergraph can be represented as a bipartite directed graph or as a *FD*-graph [3] and from these representations it is possible to build an adjacency-list.

(1) Directed hypergraph

	e_1	e_2	e_3	e_4	e_5
v_1	-1	0	0	0	1
v_2	1	-1	-1	0	1
v_3	0	1	-1	-1	0
v_4	0	0	1	-1	-1
v_5	0	0	0	1	-1

(2) Adjacency matrix

Fig. 2.11 (Example 2.46): Adjacency-matrix of a directed hypergraph.

For the practical implementation of algorithms for directed hypergraphs, it might make sense to create several lookups tables that facilitate the identification of specific nodes or hyperedges in the hypergraph. This might include: (1) the list of nodes and hyperedges of the hypergraph; (2) for each tail, the list of hyperedges that have this tail; (3) for each head, the list of hyperedges that have this head; (4) for each tail, the list of its nodes; (5) for each node, the list of tails that contain this node; (6) for each head, the list of its nodes; and (7) for each node, the list of heads that contain this node. A similar representation based on a collection of linked adjacency-lists has been suggested in [28].

2.5 Directed Hypergraph Traversal

In this section, we present two new algorithms to traverse directed hypergraphs. These algorithms are based on the notion of B -connectivity and try to reproduce how breadth-first search and depth-first search traverse directed graphs.

2.5.1 Breadth-first Search Algorithm

To the best of our knowledge, there is not the equivalent version of the breadth-first search algorithm to traverse directed hypergraphs.

There are algorithms that can identify which are the nodes that are connected from a particular node using a specific connectivity notion (L -connectivity [40], B -connectivity, F -connectivity and BF -connectivity [18, 31]). This is for computing the connected component of a node in a directed hypergraph.

Figure 2.12 provides a description of the algorithm for computing B -connected components in directed hypergraphs [18, 31]. Every node of the input directed hypergraph is

labelled with a boolean value and every hyperedge with an integer value. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and let $v \in \mathcal{V}$ be a node and $e \in \mathcal{E}$ be a hyperedge. We use the auxiliary notion $\text{visited}(v)$ to indicate if the node v has been visited ($\text{visited}(v) = \text{true}$) or not ($\text{visited}(v) = \text{false}$) by the function $\text{Bvisit}(\cdot, \cdot)$, where $\text{visited} : \mathcal{V} \rightarrow \{0, 1\}$ is a boolean function. We use also the auxiliary notion $\text{visited-tail-node}(e)$ to specify how many tail nodes of the hyperedge e has been reached by the function $\text{Bvisit}(\cdot, \cdot)$, where $\text{visited-tail-node} : \mathcal{E} \rightarrow \mathbb{N}$ is a function. This is a simple and elegant counter mechanism to detect when a head node of a hyperedge becomes reachable during the execution of the function $\text{Bvisit}(\cdot, \cdot)$.

When the function $\text{Bvisit}(s, \mathcal{H})$ starts (Lines 1&2), it associates to each node of the hypergraph the value `false` (none of the nodes has been visited yet) and to each edge the value `zero` (none of the nodes of the tail of each hyperedge has been reached yet). The queue Q stores the nodes that have been visited but not explored (some of the successors of each of these nodes has not been visited) yet. The while-loop (Lines 7–21) iterates over all nodes in Q . For each node u in Q , the procedure iterates over all hyperedges that have u in their tails (Lines 9–20). For each hyperedge e in $\mathcal{E}(u)$, the counter $\text{visited-tail-node}(e)$ is incremented because a new tail node has been reached (Line 10). If the number of tail nodes reached coincides with the size of the tail then the head of this hyperedge becomes reachable (condition in if-statement in Line 11). In this case, the function iterates over all head nodes and marks each head node w as visited ($\text{visited}(w) = \text{true}$, Line 14) and stores w in Q if w has not been visited before. As a result, each tail node of each tail is only reached once. When the execution of function $\text{Bvisit}(\cdot, \cdot)$ finishes, it returns the set of nodes of the input hypergraphs that has been visited (stored in $\mathcal{V}'$).

Proof of correctness of the function $\text{Bvisit}(\cdot, \cdot)$ is not provided by [18, 31] but it is provided by [28]. The proof where we show that all nodes of $\mathcal{V}'$ are B -connected from node s (Lemma 2.47, item (v)) is similar to the proof of correctness by [28].

Lemma 2.47 (Properties of the function $\text{Bvisit}(\cdot, \cdot)$). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and let $s \in \mathcal{V}$ be a node. Let $\text{Bvisit}(\cdot, \cdot)$ be the function depicted in Figure 2.12 and let $\mathcal{V}' \subseteq \mathcal{V}$ be the result of the execution of this function. Let $\mathcal{E}' \subseteq \mathcal{E}$ be the maximal set of hyperedges such that $\mathcal{E}' = \{e \in \mathcal{E} \mid T(e) \cap \mathcal{V}' \neq \emptyset\}$.*

Then the following holds:

- (i) *for all $v \in \mathcal{V}$: $\text{Bvisit}(s, \mathcal{H})$ enqueues v in Q at most once;*
- (ii) *for all $v \in \mathcal{V}'$: $\text{visited}(v) = \text{true}$;*
- (iii) *for all $e \in \mathcal{E}$: $\text{Bvisit}(s, \mathcal{H})$ iterates over e if $e \in \mathcal{E}'$;*
- (iv) *for all $e \in \mathcal{E}'$: $\text{Bvisit}(s, \mathcal{H})$ iterates over e no more than $|T(e)|$ times; and*

```

function Bvisit( $s, \mathcal{H} = (\mathcal{V}, \mathcal{E})$ )
    // Initialize the labels of the nodes and hyperedges
    1: for every  $v \in \mathcal{V}$  do: visited( $v$ ) := false
    2: for every  $e \in \mathcal{E}$  do: visited-tail-node( $e$ ) := 0
    3:  $Q := \{\}$ 
    4: enqueue( $Q, s$ )
    5: visited( $s$ ) := true
    6:  $\mathcal{V}' := \{s\}^+$ 
    // Identify B-connected nodes from  $s$ 
    7: while  $Q \neq \{\}$  do
    8:    $u :=$  dequeue( $Q$ )
    9:   for every  $e \in \mathcal{E}(u)$  do
    10:    visited-tail-node( $e$ ) := visited-tail-node( $e$ ) + 1
    11:    if (visited-tail-node( $e$ ) =  $|T(e)|$ ) then
    12:      // All nodes in  $H(e)$  are B-connected from  $s$ 
    13:      for every  $w \in H(e)$  do
    14:        if (visited( $w$ ) = false) then
    15:          visited( $w$ ) := true
    16:          enqueue( $Q, w$ )
    17:           $\mathcal{V}' := \mathcal{V}' \cup \{w\}^+$ 
    18:        end-if
    19:      end-for
    20:    end-if
    21:  end-while
    22: return  $\mathcal{V}'^+$ 

```

Fig. 2.12 Function Bvisit for directed hypergraphs [18, 31].

(v) for all $v \in \mathcal{V}'$: $s \geq_{B(\mathcal{H})} v$. ⊢

Proof. We start the proof of Lemma 2.47 by showing item (i). For all nodes $v \in \mathcal{V}$, the function $Bvisit(s, \mathcal{H})$ stores in the label $visited(v)$ the value false (Line 1). This means that all nodes of the hypergraph are labelled as non-visited. Only when a node has not been visited before it is possible to enqueue it into Q . When this happens, the value of the label $visited$ is updated with the value true (Lines 5&14) and it cannot be changed during the rest of the execution of the function. Therefore, it is not possible that a node is enqueued (or dequeued) more than one time. As a result, item (i) holds.

Item (ii) can readily be seen. Only when the label $visited$ of a node is updated with the value true (Lines 5&14), the node is enqueued in Q and stored in $\mathcal{V}'$ (Lines 6&16).

Consider item (iii). The function $Bvisit(\cdot, \cdot)$ iterates over the hyperedges that contains the node u in their tails (for-loop in Lines 9–20), where u is a node that has been dequeued from Q which implies that u has been stored in $\mathcal{V}'$ and $visited(u) = \text{true}$. Therefore, only hyperedges in $\mathcal{E}'$ are considered and item (iii) holds.

Consider item (iv). By item (i), each node of the hypergraph can be enqueued (and dequeued) at most once by the function $Bvisit(\cdot, \cdot)$. By item (iii), the function $Bvisit(\cdot, \cdot)$ only iterates over hyperedges in $\mathcal{E}'$. After a node $u \in \mathcal{V}$ is dequeued from Q (Line 8), the function $Bvisit(\cdot, \cdot)$ iterates over each hyperedge $e \in \mathcal{E}'$ that includes u in its tail (for-loop in Lines 9–20). Then, the function $Bvisit(\cdot, \cdot)$ iterates over the same edge as many times as there are tail nodes dequeued from Q which means that they have been stored in $\mathcal{V}'$. If the tail nodes of a hyperedge e are in $\mathcal{V}'$ ($T(e) \subseteq \mathcal{V}'$) then the function $Bvisit(\cdot, \cdot)$ iterates over e no more than $|T(e)|$ times. As result, item (iv) holds.

Consider item (v). By item (ii), each node $v \in \mathcal{V}$ that is visited ($visited(v) = \text{true}$) during the execution of the function $Bvisit(s, \mathcal{H})$ are included in $\mathcal{V}'$. In particular, each node is added into $\mathcal{V}'$, one by one. Let $\mathcal{V}'_i$ be the version of $\mathcal{V}'$ such that its i th node is the last node that has been added, where $i \leq |\mathcal{V}'|$. We show by induction over i that for all $i \geq 1$, all nodes in $\mathcal{V}'_i$ are B -connected from s : $s \geq_{B(\mathcal{H})} v$, where $v \in \mathcal{V}'_i$ and $1 \leq i \leq |\mathcal{V}'|$. This is the induction hypothesis (IH).

In the base case, we have that $i = 1$. This means that $\mathcal{V}'_1 = \{s\}$. By Definition 2.32, s is B -connected from itself: $s \geq_{B(\mathcal{H})} s$.

Consider the induction step. Suppose that we have shown the induction hypothesis (IH) for $i = n$, where $1 < n < |\mathcal{V}'|$. We now show the IH for $i = n + 1$. Suppose $v \in \mathcal{V}$ is the last node added into $\mathcal{V}'_{n+1}$. This means that there is a hyperedge $e \in \mathcal{E}'$ such that $v \in H(e)$ and $visited\text{-tail-node}(e) = |T(e)|$. This implies $T(e) \subseteq \mathcal{V}'_n$ and by the induction hypothesis, for all $w \in T(e)$, $s \geq_{B(\mathcal{H})} w$. Then by Definition 2.32, $s \geq_{B(\mathcal{H})} v$. Hence IH holds.

□

The next proposition follows from Lemma 2.14.

Proposition 2.48 (Correctness of function $Bvisit(\cdot, \cdot)$). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and $s \in \mathcal{V}$ be a node. Let $Bvisit(s, \mathcal{H})$ be the function depicted in Figure 2.12 and $\mathcal{V}' \subseteq \mathcal{V}$ be the result of the execution of this function. Then:*

(i) *The function $Bvisit(s, \mathcal{H})$ terminates.*

(ii) $\mathcal{V}' = \geq_{B(\mathcal{H})}(s)$, where $\geq_{B(\mathcal{H})}(s) = \{v \in \mathcal{V} \mid s \geq_{B(\mathcal{H})} v\}$. ⊢

By items (i)&(iii)&(iv) of Lemma 2.47, the function $Bvisit(\cdot, \cdot)$ terminates. By item (v) of Lemma 2.47, $\mathcal{V}'$ is the B -connected component from the input node $s \in \mathcal{V}$.

We now show that the time complexity of the function $Bvisit(\cdot, \cdot)$ is linear with respect to the size of the input hypergraph [18, 31].

Proposition 2.49 (Time complexity of function $Bvisit(\cdot, \cdot)$). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and $s \in \mathcal{V}$ be a node. Let $Bvisit(s, \mathcal{H})$ be the function depicted in Figure 2.12. Then: the time complexity of function $Bvisit(s, \mathcal{H})$ is $O(|\mathcal{H}|)$, where $|\mathcal{H}| = |\mathcal{V}| + \sum_{e \in \mathcal{E}} (|T(e)| + |H(e)|)$. ⊢*

Proof. The for-loops in Lines 1&2 are executed in $|\mathcal{V}| + |\mathcal{E}|$ steps. The while-loop (Lines 7–21) iterates over all nodes stored in Q . A node of the hypergraph can only be stored in Q at most once (when the node has not been visited before). This is ensured by the if-condition in Line 13. Therefore, the while-loop iterates $|\mathcal{V}|$ times in the worst-case. For each node $v \in Q$, the function iterates over all hyperedges that contain v in their tails ($|\mathcal{E}(v)|$ times) using a for-loop (Lines 9–20). Let's assume that there is a data-structure that allows to identify each member of $\mathcal{E}(v)$ in constant time.⁸ Let's also assume that the hyperedge selected in the current iteration of the for-loop is e . Inside of the for-loop, the value of $visited_tail_node(e)$ is incremented (Line 10). Because each node of the hypergraph can be stored in Q at most once, the value of $visited_tail_node(e)$, where $e \in \mathcal{E}$ represents any hyperedge in $\mathcal{H}$, can be updated at most $|T(e)|$ times. Only when $visited_tail_node(e) = |T(e)|$, the algorithm can iterate over all nodes in the head of e (for-loop in Lines 12–18) and check if any of these nodes has not been visited. If it is the case (if-condition in Line 13), the node is added into Q (Line 15). Notice that if the hyperedge e is a B -hyperedge ($|H(e)| = 1$) then $visited_tail_node(e)$ will be updated at most $|T(e)|$ times and only one node in $H(e)$ will be examined inside of the while-loop (Lines 7–21). Similarly, if e is a F -hyperedge then $visited_tail_node(e)$ will be

⁸This is implicitly assumed by [18, 31].

updated at most once and $|H(e)|$ nodes will be examined inside of the while-loop. Assuming that adding a node into Q and updating the labels of nodes and hyperedges can be done in constant time, we can summarize the execution of the while-loop in terms of the number of times the label of each hyperedges is updated and the number of head nodes that are examined. Therefore, we can conclude that the execution of the while-loop (Lines 7–21) can be done in $\sum_{e \in \mathcal{E}} (|T(e)| + |H(e)|)$ steps. Assuming that this amount is much larger than $|\mathcal{E}|$, we can conclude that the time complexity of the function $\text{Bvisit}(s, \mathcal{H})$ is $O(|\mathcal{H}|)$ $\square$

To identify the nodes that are B -connected from a particular node, the function $\text{Bvisit}(\cdot, \cdot)$ traverses the input directed hypergraph in a breadth-first search fashion. For a given node, s in a directed hypergraph $\mathcal{H}$, $\text{Bvisit}(s, \mathcal{H})$ tries to discover first all head nodes of the directed hyperedges that have s in their tails. Notice that because of B -connectivity, it might not be possible to reach the head nodes for some of the hyperedges that have s in their tails. The function $\text{Bvisit}(s, \mathcal{H})$ follows the same approach for each of the head nodes discovered. That is, it tries to discover all nodes at distance k from s before discovering any node at distance $k + 1$.

It seems reasonable to extend the function $\text{Bvisit}(\cdot, \cdot)$ to create a BFS algorithm for directed hypergraphs based on the notion of B -connectivity. This is done in Figures 2.13&2.14. The BFS algorithm is defined using two functions: $\text{main}(\cdot)$ and $\text{visit}(\cdot)$. The fundamental objective of function $\text{main}(\cdot)$ is to ensure that all nodes of the input hypergraph are visited (for-loop in Lines 8–14). Function $\text{visit}(\cdot)$ slightly extends function $\text{Bvisit}(\cdot, \cdot)$. The main change is that $\text{visit}(\cdot)$ stores when a node has been visited (or explored) for the first time. Notice that some nodes of the hypergraph might be re-visited and re-explored several times.

Like the BFS algorithm for directed graphs, lines in red and labelled with the symbol ‘+’ have been added to explicitly store the *BFS-graph*, a directed labelled graph that indicates the order in which the nodes of the input directed hypergraph were traversed by BFS (Definition 2.52). Every node of a BFS-graph is labelled with two timestamps: one timestamp indicating when the node was *discovered* (i.e., when it was visited for the first time), and a second timestamp stating when the node has been *explored* (i.e., when all its direct successors have been discovered). We use $\text{discovered-order}(v)$ and $\text{explored-order}(v)$ to refer to the timestamps that indicate when the node v was discovered and explored, respectively.

Similar to directed graphs, we make these notions more precise in the following definitions.

Definition 2.50 (Discovered-order). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph. Let $\text{discovered-order}: \mathcal{V} \rightarrow \mathbb{Z}$ be a total function assigning to every node $v \in \mathcal{V}$ an integer value $\text{discovered-order}(v)$*

```

function main( $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ )
1:  $\mathcal{V}' := \{\}$  +
2:  $\mathcal{E}' := \{\}$  +
3: for every  $v \in \mathcal{V}$  do
4:   discovered-order( $v$ ) := -1
5:   explored-order( $v$ ) := -1
6: end-for
7: order-counter := 0
8: for every  $v \in \mathcal{V}$  do
9:   if (discovered-order( $v$ ) = -1) then
      // Visit the node  $v$ 
10:   for every  $w \in \mathcal{V}$  do: visited( $w$ ) := false
11:   for every  $e \in \mathcal{E}$  do: visited-tail-node( $e$ ) := 0
12:   visit( $v$ )
13:   end-if
14: end-for
15:  $\mathcal{L} := \{(\text{discovered-order}(v), \text{explored-order}(v)) \mid v \in \mathcal{V}\}$  +
16: return  $\mathcal{G} = (\mathcal{V}', \mathcal{E}', \mathcal{L})$  +

```

Fig. 2.13 BFS main function for directed hypergraphs.

```

function visit( $v$ )
    // v has been discovered
1:  $Q := \{\}$ 
2: enqueue( $Q, v$ )
3: order-counter := order-counter + 1
4: discovered-order( $v$ ) := order-counter
5: visited( $v$ ) := true
6:  $\mathcal{V}' := \mathcal{V}' \cup \{v\} +$ 
    // Identify B-connected nodes from v
7: while  $Q \neq \{\}$  do
8:    $u := \text{dequeue}(Q)$ 
9:   for every  $e \in \mathcal{E}(u)$  do
10:    visited-tail-node( $e$ ) := visited-tail-node( $e$ ) + 1
11:    if (visited-tail-node( $e$ ) =  $|T(e)|$ ) then
        // All nodes in  $H(e)$  are B-connected from v
12:      for every  $w \in H(e)$  do
13:         $\mathcal{E}' := \mathcal{E}' \cup \{(u, w)\} +$ 
14:        if (visited( $w$ ) = false) then
15:          visited( $w$ ) := true
16:          enqueue( $Q, w$ )
17:           $\mathcal{V}' := \mathcal{V}' \cup \{w\} +$ 
18:          if (discovered-order( $w$ ) = -1) then
              // w has been discovered
19:            order-counter := order-counter + 1
20:            discovered-order( $w$ ) := order-counter
21:          end-if
22:        end-if
23:      end-for
24:    end-if
25:  end-for
26:  if (explored-order( $u$ ) = -1) then
      // u has been explored
27:    order-counter := order-counter + 1
28:    explored-order( $u$ ) := order-counter
29:  end-if
30: end-while

```

Fig. 2.14 BFS visit function for directed hypergraphs.

Fig. 2.15 Examples of directed hypergraphs for testing algorithms

$\in \{-1, 1, \dots, n\}$, where $n = 2 \times |\mathcal{V}|$, stating the position of node v in the order in which the nodes of $\mathcal{H}$ are discovered. Value -1 means that the node has not been discovered yet. $\dashv$

Definition 2.51 (Explored-order). Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed graph. Let explored-order: $\mathcal{V} \rightarrow \mathbb{Z}$ be a total function assigning to every node $v \in \mathcal{V}$ an integer value $\text{explored-order}(v) \in \{-1, 1, \dots, n\}$, where $n = 2 \times |\mathcal{V}|$, stating the position of node v in the order in which the nodes of $\mathcal{H}$ are explored. Value -1 means that the node has not been explored yet. $\dashv$

There might be several BFS-graphs of a directed hypergraph depending on the order in which the nodes are traversed by the BFS algorithm. The traversal order is stored in the node labels of a BFS-graph. We define a BFS-graph computed by BFS on directed hypergraphs as follows.

Definition 2.52 (BFS-graph). Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph. Let $\text{main}(\cdot)$ be the function depicted in Figure 2.13. A breadth-first-search graph (BFS-graph) of $\mathcal{H}$ is a directed labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}', \mathcal{L})$ with $\mathcal{G} = \text{main}(\mathcal{H})$. $\dashv$

During the design of the functions $\text{main}(\cdot)$ and $\text{visit}(\cdot)$, we have created several directed hypergraphs (Figures 2.15) that help us to test and improve these functions. Thanks to these examples, we realized that to preserve most of the specification of the function $\text{Bvisit}(\cdot, \cdot)$, the implementation of the BFS algorithm should allow that visited nodes in the input hypergraph might be re-visited several times.

Example 2.53. Let $\mathcal{H}_3 = (\mathcal{V}_3, \mathcal{E}_3)$ be the directed hypergraph depicted in Figure 2.15 (c) such that $\mathcal{V}_3 = \{v_1, v_2, v_3, v_4\}$ and $\mathcal{E}_3 = \{e_1 = (\{v_1\}, \{v_2\}), e_2 = (\{v_2\}, \{v_3\}), e_3 = (\{v_1, v_3\}, \{v_4\})\}$.

Suppose that v_3 is the first non-visited node considered by the function $\text{main}(\mathcal{H}_3)$ (for-loop in Lines 8–14). During the execution of $\text{visit}(v_3)$, no other node can be visited. Then, $\text{discovered-order}(v_3) = 1$, $\text{explored-order}(v_3) = 2$ and $\text{visited}(v_3) = \text{true}$. Let

us assume that the next non-visited node considered by the function $\text{main}(\mathcal{H}_3)$ is v_1 : $\text{discovered-order}(v_1) = 3$ and $\text{visited}(v_1) = \text{true}$. During the execution of $\text{visit}(v_1)$, this function iterates over each hyperedge that has v_1 in its tails. These hyperedges are e_3 and e_1 . To reach the head node of e_3 , $\text{visited-tail-node}(e_3) = 2$. This would only happen if v_3 is visited again. So, from v_1 only v_2 can be reached which is a node that it has not been visited before: $\text{discovered-order}(v_2) = 4$ and $\text{visited}(v_2) = \text{true}$. Therefore, v_2 is enqueued in Q (Line 16 in function $\text{visit}(\cdot)$) and no other node from v_1 can be visited: $\text{explored-order}(v_1) = 5$. From v_2 , only v_3 can be reached. v_3 was visited before ($\text{discovered-order}(v_3) = 1$) but it can be visited again because $\text{visited}(v_3) = \text{false}$ (the value was changed before the function $\text{visit}(v_1)$ was called from $\text{main}(\mathcal{H}_3)$). The value of the label $\text{discovered-order}(v_3)$ will not change and $\text{explored-order}(v_2) = 6$. The only hyperedge where v_3 is in the tail is e_3 . The value of $\text{visited-tail-node}(e_3)$ coincides now with the size of the tail of e_3 . Therefore, the node v_4 can be visited and from this node no other nodes can be visited: $\text{discovered-order}(v_4) = 7$, $\text{explored-order}(v_4) = 8$ and $\text{visited}(v_4) = \text{true}$. This concludes the execution of the BFS algorithm. As the example shows, v_3 must be revisited to reach v_4 using the hyperedge e_3 . $\triangleleft$

Notice that if we do not re-initialize the values of the labels $\text{visited}(v)$ and $\text{visited-tail-node}(e)$, for all $v \in \mathcal{V}$ and $e \in \mathcal{E}$ in a directed hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, before the function $\text{visit}(\cdot)$ is called then B -connectivity is not preserved.

Example 2.54. Let $\mathcal{H}_2 = (\mathcal{V}_2, \mathcal{E}_2)$ be the directed hypergraph depicted in Figure 2.15 (b) such that $\mathcal{V}_2 = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $\mathcal{E}_2 = \{e_1 = (\{v_1\}, \{v_2\}), e_2 = (\{v_3\}, \{v_4\}), e_3 = (\{v_3\}, \{v_5\}), e_4 = (\{v_5\}, \{v_4\}), e_5 = (\{v_2, v_4\}, \{v_6\})\}$.

Suppose that the first node visited in $\mathcal{H}_2$ is v_1 . From this node, only v_2 can be reached and from v_2 no other node can be reached. When the execution of the function $\text{visit}(v_1)$ finishes after exploring the node v_2 , $\text{visited-tail-node}(e_5) = 1$. Let's suppose that the next node visited is v_3 and the labels $\text{visited}(v)$ and $\text{visited-tail-node}(e)$, for all $v \in \mathcal{V}_2$ and $e \in \mathcal{E}_2$, are not initialized (Lines 10&11 in function $\text{main}(\cdot)$). Then from node v_3 , the node v_4 can be reached. This implies that the hyperedge e_5 is considered again and $\text{visited-tail-node}(e_5) = 2$. Therefore, the node v_6 can be reached from v_4 which is not correct because v_6 is not B -connected from v_4 . $\triangleleft$

The next proposition can readily be seen.

Proposition 2.55 (Correctness of BFS for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and $s \in \mathcal{V}$ be a node. Let $\text{main}(\mathcal{H})$ and $\text{visit}(s)$ be the functions depicted in Figures 2.13&2.14. Then:*

- (i) *The function $\text{main}(\mathcal{H})$ terminates.*

(ii) The function $\text{main}(\mathcal{H})$ traverses $\mathcal{H}$ completely.

(iii) The function $\text{visit}(s)$ visits all nodes in $\mathcal{H}$ that are B -connected from s . $\dashv$

Proposition 2.48 shows that the function $\text{Bvisit}(\cdot, \cdot)$ terminates and returns the B -connected component of the input node in the input directed hypergraph. The function $\text{visit}(\cdot)$ is very similar to $\text{Bvisit}(\cdot, \cdot)$ and the small changes introduced do not alter the conclusions of Proposition 2.48. Knowing that function $\text{visit}(\cdot)$ terminates implies that function $\text{main}(\cdot)$ also terminates. Moreover, function $\text{main}(\cdot)$ ensures that each node of the input hypergraph is visited at least one time (for-loop in Lines 8–14).

We show that the time complexity of BFS is polynomial with respect to the size of the input hypergraph.

Proposition 2.56 (Time complexity of BFS for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and $s \in \mathcal{V}$ be a node. Let $\text{main}(\cdot)$ be the function depicted in Figure 2.13.*

Then: the time complexity of function $\text{main}(\cdot)$ is $O(|\mathcal{V}| \times |\mathcal{H}|)$. $\dashv$

The time complexity of the function $\text{visit}(\cdot)$ is the same as the function $\text{Bvisit}(\cdot, \cdot)$. This is $O(|\mathcal{H}|)$. In the worst case, the function $\text{main}(\cdot)$ calls the function $\text{visit}(\cdot)$ $|\mathcal{V}|$ times. As a result the time complexity of the function $\text{main}(\cdot)$ is $O(|\mathcal{V}| \times |\mathcal{H}|)$.

Example 2.57. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph such that $\mathcal{V} = \{v_1, v_2, \dots, v_n\}$ and $\mathcal{E} = e_1, e_2, \dots, e_n$ with $n \geq 3$. Each hyperedge $e_i \in \mathcal{E}$ with $i \in \{1, \dots, n\}$ is defined as follows: $T(e_i) = \mathcal{V} \setminus \{v_i\}$ and $H(e_i) = \{v_i\}$. This implies that $|\mathcal{V}| = |\mathcal{E}|$. For all $v_i \in \mathcal{V}$, the function $\text{main}(\cdot)$ calls the function $\text{visit}(v_i)$ (Line 12) and in each call this function iterates over all hyperedges (Lines 9–25) except e_i without reaching any head node. This means that the function $\text{main}(\cdot)$ calls the function $\text{visit}(\cdot)$ $|\mathcal{V}|$ times and $\text{visit}(\cdot)$ iterates over $|\mathcal{E}| - 1$ hyperedges.* $\triangleleft$

We show that node to node B -connectivity in the directed hypergraph is not preserved in the BFS-graph returned by the function $\text{main}(\cdot)$.

Proposition 2.58 (Node to node connectivity in BFS-graphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and $\mathcal{G} = (\mathcal{V}', \mathcal{E}', \mathcal{L})$ be the BFS-graph produced by function $\text{main}(\cdot)$ depicted in Figure 2.13.*

Then: $\geq_{B(\mathcal{H})} \neq \geq_{\mathcal{G}}$, where $\geq_{\mathcal{G}}$ is the connectivity relation in $\mathcal{G}$. $\dashv$

Example 2.59. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be the B -hypergraph depicted in Figure 2.16 (a), where $\mathcal{V} = \{v_1, \dots, v_5\}$ and $\mathcal{E} = \{e_1 = (\{v_1\}, v_2), e_2 = (\{v_2\}, v_3), e_3 = (\{v_2, v_3\}, v_4), e_4 = (\{v_3, v_4\}, v_5), e_5 = (\{v_4, v_5\}, v_1)\}$.*

Fig. 2.16 (Example 2.59): node to node connectivity in BFS-graphs

Suppose that the first node considered during the execution of the function $\text{main}(\mathcal{H})$, depicted in Figure 2.13, is v_1 . This node has not been visited and the function $\text{visit}(v_1)$, depicted in Figure 2.14, is called. This implies that the following labels are modified: $\text{discovered-order}(v_1) = 1$ and $\text{visited}(v_1) = \text{true}$. Moreover, v_1 is stored in $\mathcal{V}'$. From v_1 , the only node that can be reached is v_2 . Then, v_2 is enqueued in Q and the following labels are modified: $\text{discovered-order}(v_2) = 2$ and $\text{visited}(v_2) = \text{true}$. v_2 is added into $\mathcal{V}'$ and (v_1, v_2) into $\mathcal{E}'$. The exploration of the node v_1 concludes, $\text{explored-order}(v_1) = 3$ and the function $\text{visit}(v_1)$ continues with the exploration of the node v_2 which is dequeued from Q . v_2 is in $T(e_3)$ and as a consequence, $\text{visited-tail-node}(e_3) = 1$. v_2 is also in $T(e_2)$ and therefore, v_3 is reached: $\text{discovered-order}(v_3) = 4$, $\text{visited}(v_3) = \text{true}$ and $v_3 \in Q$. As a result, v_3 is stored in $\mathcal{V}'$ and (v_2, v_3) in $\mathcal{E}'$. After the exploration of v_2 finishes, $\text{explored-order}(v_2) = 5$ and the node v_3 is dequeued from Q . Because $v_3 \in T(e_4)$ then $\text{visited-tail-node}(e_4) = 1$. Similarly $v_3 \in T(e_3)$, this implies that $\text{visited-tail-node}(e_3) = 2$ and node v_4 becomes reachable: $\text{discovered-order}(v_4) = 6$, $\text{visited}(v_4) = \text{true}$ and $v_4 \in Q$. Then, v_4 is stored in $\mathcal{V}'$ and (v_3, v_4) in $\mathcal{E}'$. v_3 is not in more tails and the exploration of successors of v_3 concludes and the value of the label $\text{explored-order}(v_3)$ is updated with the value 7. Next, the node v_4 is dequeued from Q . This node is in $T(e_5)$ and therefore, $\text{visited-tail-node}(e_5) = 1$. v_4 is also in $T(e_4)$. Because $\text{visited-tail-node}(e_4) = 2$, node v_5 becomes reachable: $\text{discovered-order}(v_5) = 8$, $\text{visited}(v_5) = \text{true}$ and $v_5 \in Q$. Therefore, v_5 is stored in $\mathcal{V}'$ and (v_4, v_5) in $\mathcal{E}'$. v_4 is not in more tails and the exploration of successors of v_4 concludes and the value of the label $\text{explored-order}(v_4)$ is updated with the value 9. The execution of the function $\text{visit}(v_1)$ continues and v_5 is dequeued from Q . v_5 is in $T(e_5)$ and therefore, v_1 becomes reachable because $\text{visited-tail-node}(e_5) = 2$. $\text{visited}(v_5) = \text{true}$ and v_1 is not enqueued again in Q . However, (v_5, v_1) is added into $\mathcal{E}'$. This concludes the

Fig. 2.17 Node to node connectivity in new BFS-graphs

exploration of node v_5 and the execution of the functions $\text{visit}(v_1)$ and $\text{main}(\mathcal{H})$. The latter returns the BFS-graph $\mathcal{G} = (\mathcal{V}', \mathcal{E}', \mathcal{L})$ created during the execution of these two functions which is depicted in Figure 2.16 (b).

In $\mathcal{G}$ is possible to reach v_4 from v_3 and v_1 from v_5 . This is not possible in the hypergraph $\mathcal{H}$. ◁

Suppose that we change Line 13 in the function $\text{visit}(\cdot)$ as follows:

15: $\mathcal{E}' := \mathcal{E}' \cup \{(v, w)\}^+$

This means that the function $\text{visit}(\cdot)$ adds a directed edge from v to any node in the connected component of v , identified by w . Recall that v is the input node of the function $\text{visit}(\cdot)$ and w is one of its successors in the directed hypergraph $\mathcal{H}$. It can readily be seen that using this approach, node-to-node B -connectivity in the directed hypergraph is also not preserved in the BFS-graph. To achieve this, we have to also modify the function $\text{main}(\cdot)$ as follows:

```

8: for every  $v \in \mathcal{V}$  do
    // Visit the node  $v$ 
9:   for every  $w \in \mathcal{V}$  do:  $\text{visited}(w) := \text{false}$ 
10:  for every  $e \in \mathcal{E}$  do:  $\text{visited-tail-node}(e) := 0$ 
11:    $\text{visit}(v)$ 
12: end-for
```

We deleted the if-condition that restricts the execution of the function $\text{visit}(\cdot)$ for nodes that have not been discovered before. As a consequence, this new version of the function

$\text{main}(\cdot)$ might call the function $\text{visit}(\cdot)$ more times than the original version of $\text{main}(\cdot)$. Figure 2.17 illustrates the impact of these changes on the generation of a BFS-graph in Example 2.59. Note that the labels of each node indicate that we started traversing the hypergraph from node v_1 and we follow the same order as in Example 2.59. Later calls of the function $\text{visit}(\cdot)$ for the rest of nodes of the hypergraph are not reflected in the labels.

2.5.2 Depth-first Search Algorithm

We could not find in the literature a specification of depth-first search for directed hypergraphs. In this section, we introduce a version of depth-first search for directed hypergraphs based on the notion of B -connectivity. Like the breadth-first search algorithm that we described in the previous section, our proposal is also inspired by the design of the function $\text{Bvisit}(\cdot, \cdot)$ (Figure 2.12).

The depth-first search algorithm for directed hypergraphs is defined using two functions: function $\text{main}(\cdot)$ depicted in Figure 2.18 and function $\text{visit}(\cdot)$ described in Figure 2.19. Function $\text{main}(\cdot)$ is identical to the function of the same name defined for BFS and depicted in Figure 2.13. On the contrary, function $\text{visit}(\cdot)$ is different to the equivalent function defined for BFS. The main difference is that $\text{visit}(\cdot)$ for DFS is a recursive function. When the head node of a hyperedge can be visited (because each of the tail nodes of the hyperedge has been visited before), the function $\text{visit}(\cdot)$ is recursively called with this head node as input parameter and the function continues from this node traversing the entire hypergraph. This means that the function $\text{visit}(\cdot)$ searches “deeper” for nodes in the hypergraph whenever possible.

Like the BFS algorithm for directed hypergraphs, lines in red and labelled with the symbol ‘+’ have been added to explicitly store the *DFS-graph*, a directed labelled graph that indicates the order in which the nodes of the input directed hypergraph have been traversed by DFS (Definition 2.60). Like in the case of BFS-graphs, every node of a DFS-graph is labelled with two timestamps: one timestamp indicating when the node was *discovered* (i.e., when it was visited for the first time), and a second timestamp stating when the node has been *explored* (i.e., when all its successors have been discovered). We use $\text{discovered-order}(v)$ and $\text{explored-order}(v)$ to refer to the timestamps that indicates when the node v was discovered and explored, respectively.

There might be several DFS-graphs for a given directed hypergraph. This depends on the order in which the nodes of the hypergraph are traversed by the DFS algorithm. This order is recorded in the labels of the nodes of a DFS-graph. We define a DFS-graph as follows.

```

function main( $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ )
1:  $\mathcal{V}' := \{\}$  +
2:  $\mathcal{E}' := \{\}$  +
3: for every  $v \in \mathcal{V}$  do
4:   discovered-order( $v$ ) := -1
5:   explored-order( $v$ ) := -1
6: end-for
7: order-counter := 0
8: for every  $v \in \mathcal{V}$  do
9:   if (discovered-order( $v$ ) = -1) then
      // Visit the node  $v$ 
10:   for every  $w \in \mathcal{V}$  do: visited( $w$ ) := false
11:   for every  $e \in \mathcal{E}$  do: visited-tail-node( $e$ ) := 0
12:   visit( $v$ )
13:   end-if
14: end-for
15:  $\mathcal{L} := \{(\text{discovered-order}(v), \text{explored-order}(v)) \mid v \in \mathcal{V}\}$  +
16: return  $\mathcal{G} = (\mathcal{V}', \mathcal{E}', \mathcal{L})$  +

```

Fig. 2.18 DFS main function for directed hypergraphs.

```

function visit( $v$ )
1: if (discovered-order( $v$ ) = -1) then
    //  $v$  has been discovered
2:   order-counter := order-counter + 1
3:   discovered-order( $v$ ) := order-counter
4: end-if
5: visited( $v$ ) := true
6:  $\mathcal{V}' := \mathcal{V}' \cup \{v\}$  +
    // Identify  $B$ -connected nodes from  $v$ 
7: for every  $e \in \mathcal{E}(v)$  do
8:   visited-tail-node( $e$ ) := visited-tail-node( $e$ ) + 1
9:   if (visited-tail-node( $e$ ) =  $|T(e)|$ ) then
    // All nodes in  $H(e)$  are  $B$ -connected from  $v$ 
10:    for every  $w \in H(e)$  do
11:       $\mathcal{E}' := \mathcal{E}' \cup \{(v, w)\}$  +
12:      if (visited( $w$ ) = false) then
13:        visit( $w$ )
14:      end-if
15:    end-for
16:  end-if
17: end-for
18: if (explored-order( $v$ ) = -1) then
    //  $v$  has been explored
19:   order-counter := order-counter + 1
20:   explored-order( $v$ ) := order-counter
21: end-if

```

Fig. 2.19 DFS visit function for directed hypergraphs.

Definition 2.60 (DFS-graph). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph. Let $\text{main}(\cdot)$ be the function depicted in Figure 2.18. A depth-first search graph (DFS-graph) of $\mathcal{H}$ is a directed labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}', \mathcal{L})$ with $\mathcal{G} = \text{main}(\mathcal{H})$. $\dashv$*

The next proposition shows the correctness of the depth-first search algorithm for directed hypergraphs.

Proposition 2.61 (Correctness of DFS for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and $s \in \mathcal{V}$ be a node. Let $\text{main}(\mathcal{H})$ and $\text{visit}(s)$ be the functions depicted in Figures 2.18&2.19. Then:*

- (i) *The function $\text{main}(\mathcal{H})$ terminates.*
- (ii) *The function $\text{main}(\mathcal{H})$ traverses $\mathcal{H}$ completely.*
- (iii) *The function $\text{visit}(s)$ visits all nodes in $\mathcal{H}$ that are B-connected from s . $\dashv$*

Proof. We start the proof of Proposition 2.61 by showing item (i). We have to distinguish between a call of the function $\text{visit}(\cdot)$ from the function $\text{main}(\cdot)$ and recursive call from the function $\text{visit}(\cdot)$ itself. In the first case, the label $\text{visited}(\cdot)$ associated to each node of the hypergraph and the label $\text{visited-tail-node}(\cdot)$ associated to each hyperedge are initialized before $\text{visit}(\cdot)$ is called. This does not happen in the second case.

It can readily be seen that after the function $\text{visit}(\cdot)$ is called from the function $\text{main}(\cdot)$, each node of the directed hypergraph can be visited at most one time. During the execution of $\text{visit}(\cdot)$, this function iterates over each of the hyperedges that has the input node in its tail (for-loop in Lines 7–17). If the head node of any of these hyperedges can be reached then it is verified that the head node has not been visited (if-condition in Line 12) in a previous recursive call after the function $\text{visit}(\cdot)$ was called from $\text{main}(\cdot)$. Because the number of nodes of the hypergraph is finite, $\text{visit}(\cdot)$ terminates.

Termination of function $\text{main}(\cdot)$ depends of the execution of this function and the execution of the function $\text{visit}(\cdot)$ that it is called from $\text{main}(\cdot)$. The execution of function $\text{main}(\cdot)$ is determined by two for-loops. The first for-loop (Lines 3–6) iterates over all nodes of the hypergraph to initialize two of their labels that indicate when a node is discovered and explored. The second for-loop (Lines 8–14) also iterates over all nodes to ensure that all nodes are visited at least one time. Because function $\text{visit}(\cdot)$ terminates then function $\text{main}(\cdot)$ also terminates. Hence item (i) holds.

Consider item (ii). The function $\text{main}(\cdot)$ ensures that all nodes of the hypergraph are visited at least one time (for-loop in Lines 8–14). The function $\text{visit}(\cdot)$ iterates over all hyperedges that have the input node in their tails. Because $\text{visit}(\cdot)$ is called for each node at

least one time, all hyperedges of the input hypergraph are considered. Therefore item (ii) holds.

Consider item (iii). The function $\text{visit}(\cdot)$ uses the same mechanism as the function $\text{Bvisit}(\cdot, \cdot)$ to determine when a head node of a hyperedge can be visited. Every hyperedge $e \in \mathcal{E}$ has a label $(\text{visited-tail-node}(e))$ that indicates how many nodes of its tail has been visited. Only when all tail nodes of a hyperedge has been visited, the nodes in the head can be reached (if-condition in Line 9). If there are head nodes that have not been visited before then $\text{visit}(\cdot)$ continue traversing the hypergraph from one of these nodes. This ensures that for a given input node, the function $\text{visit}(\cdot)$ finds all nodes of the hypergraph that are B -connected from this node and they have not been visited before. Hence item (iii) holds. $\square$

We show that the time complexity of DFS is polynomial with respect to the size of the input hypergraph.

Proposition 2.62 (Time complexity of DFS for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and $s \in \mathcal{V}$ be a node. Let $\text{main}(\cdot)$ be the function depicted in Figure 2.18.*

Then: the time complexity of function $\text{main}(\cdot)$ is $O(|\mathcal{V}| \times |\mathcal{H}|)$. $\dashv$

Proof. Let $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ be the functions depicted in Figures 2.18&2.19. The function $\text{main}(\cdot)$ has two for-loops (Lines 3–6 and 9–14). Both for-loops iterate over the nodes of the hypergraph. In the first for-loop, the value of two labels of each node are modified. This can be done in constant time. Then, this for-loop can be completed in $|\mathcal{V}|$ steps.

In the second for-loop, the function $\text{visit}(\cdot)$ is called for each node not discovered yet. Before the function $\text{visit}(\cdot)$ is called, the value of a label for each node and hyperedge is changed (Lines 10&11) in constant time. In the worst case, the update of these labels might require $|\mathcal{V}| + |\mathcal{E}|$ steps for each iteration. This is $|\mathcal{V}| \times (|\mathcal{V}| + |\mathcal{E}|)$ steps.

To complete the analysis of function $\text{main}(\cdot)$, we need to study the time complexity of function $\text{visit}(\cdot)$.

Function $\text{visit}(\cdot)$ is recursive and a recursive call can only happen if the input node has not been visited before in a previous recursive call (if-condition in Line 12). This implies that after the function $\text{visit}(\cdot)$ is called from the function $\text{main}(\cdot)$, $\text{visit}(\cdot)$ cannot be called more than $|\mathcal{V}|$ times. Let's assume that the current visiting node is v . During the execution of function $\text{visit}(v)$, there is a for-loop (Lines 7–17) that iterates over each hyperedge that contains v in its tail ($\mathcal{E}(v)$). Let's assume that there is a data-structure that allows to identify each member of $\mathcal{E}(v)$ in constant time. Let's also assume that the hyperedge selected in the current iteration of the for-loop is e . Inside of the for-loop, the value of $\text{visited-tail-node}(e)$ is

Fig. 2.20 (Example 2.64): node to node connectivity in DFS-graphs

incremented (Line 8). Because $\text{visit}(\cdot)$ can recursively call itself no more than $|\mathcal{V}| - 1$ times, one time for each node of hypergraph, the value of $\text{visited-tail-node}(e)$ can be updated at most $|T(e)|$ times after the $|\mathcal{V}| - 1$ recursive calls of $\text{visit}(\cdot)$ are completed and the execution of $\text{main}(\cdot)$ continues. Only when $\text{visited-tail-node}(e) = |T(e)|$, it is possible to iterate over all nodes in the head of e (for-loop in Lines 10–15) and check if any of these nodes has not been visited. For each head node that has not been visited before, $\text{visit}(\cdot)$ is recursively called (Line 13) with this node as input node. Notice that if the hyperedge e is a B-hyperedge ($|H(e)| = 1$) then $\text{visited-tail-node}(e)$ will be updated at most $|T(e)|$ times and only one node in $H(e)$ will be considered. Similarly, if e is a F-hyperedge then $\text{visited-tail-node}(e)$ will be updated at most once and $|H(e)|$ nodes will be considered for being visited. Assuming that updating the labels of nodes and hyperedges can be done in constant time, we can summarize the execution of the for-loop in Lines 7–17 in terms of the number of times the label of each hyperedges is updated and the number of head nodes that are examined. In the worst case, $\text{visit}(\cdot)$ will be recursively called $|\mathcal{V}| - 1$. This means that except for the input node of $\text{visit}(\cdot)$ when it is called from $\text{main}(\cdot)$, there might be one recursive call for each of the rest of nodes of the hypergraph. Therefore, the execution of these $|\mathcal{V}|$ calls of function $\text{visit}(\cdot)$ might require $\sum_{e \in \mathcal{E}} (|T(e)| + |H(e)|)$ steps. Assuming that this amount is much larger than $|\mathcal{V}|$, we can conclude that the time complexity of the function $\text{visit}(\cdot)$ is $O(|\mathcal{H}|)$.

We expect that $|\mathcal{H}| \gg |\mathcal{V}|$ and $|\mathcal{H}| \gg |\mathcal{E}|$. Then, the time complexity of the function $\text{main}(\cdot)$ is $O(|\mathcal{V}| \times |\mathcal{H}|)$. $\square$

We show that node to node B-connectivity in the directed hypergraph is not preserved in the DFS-graph returned by the function $\text{main}(\cdot)$.

Proposition 2.63 (Node to node connectivity in DFS-graphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and $\mathcal{G} = (\mathcal{V}', \mathcal{E}', \mathcal{L})$ be the DFS-graph produced by function $\text{main}(\cdot)$ depicted in Figure 2.18.*

Then: $\geq_{B(\mathcal{H})} \neq \geq_{\mathcal{G}}$, where $\geq_{\mathcal{G}}$ is the connectivity relation in $\mathcal{G}$. ⊥

Example 2.64. *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be the B-hypergraph depicted in Figure 2.20 (a), where $\mathcal{V} = \{v_1, \dots, v_5\}$ and $\mathcal{E} = \{e_1 = (\{v_1\}, v_2), e_2 = (\{v_2\}, v_3), e_3 = (\{v_2, v_3\}, v_4), e_4 = (\{v_3, v_4\}, v_5), e_5 = (\{v_4, v_5\}, v_1)\}$.*

Suppose that the first node considered during the execution of the function $\text{main}(\mathcal{H})$, depicted in Figure 2.18, is v_1 . This node has not been visited and the function $\text{visit}(v_1)$, depicted in Figure 2.19, is called. This implies that the following labels are modified: $\text{discovered-order}(v_1) = 1$ (Line 3) and $\text{visited}(v_1) = \text{true}$ (Line 5). Moreover, v_1 is stored in $\mathcal{V}'$ (Line 6).

From v_1 , the only node that can be reached is v_2 that it has not been visited before. Then, $\text{visit}(v_2)$ (Line 13) is executed and the following labels are modified: $\text{discovered-order}(v_2) = 2$ and $\text{visited}(v_2) = \text{true}$. v_2 is added into $\mathcal{V}'$ and (v_1, v_2) into $\mathcal{E}'$ (Line 11). $v_2 \in T(e_3)$ and as a consequence, $\text{visited-tail-node}(e_3) = 1$ (Line 8). Moreover, $v_2 \in T(e_2)$ and therefore, $\text{visited-tail-node}(e_2) = 1$. Because $\text{visited-tail-node}(e_2) = |T(e_2)|$ and $\text{visited}(v_3) = \text{false}$, $\text{visit}(v_3)$ is executed and the following labels are modified: $\text{discovered-order}(v_3) = 3$, $\text{visited}(v_3) = \text{true}$. Moreover, v_3 is stored in $\mathcal{V}'$ and (v_2, v_3) in $\mathcal{E}'$. Because $v_3 \in T(e_4)$ then $\text{visited-tail-node}(e_4) = 1$. Similarly $v_3 \in T(e_3)$, this implies that $\text{visited-tail-node}(e_3) = 2$ and the head node of e_3 , v_4 , becomes reachable. Because $\text{visited}(v_4) = \text{false}$, $\text{visit}(v_4)$ is executed and the following labels are modified: $\text{discovered-order}(v_4) = 4$, $\text{visited}(v_4) = \text{true}$. In addition, v_4 is stored in $\mathcal{V}'$ and (v_3, v_4) in $\mathcal{E}'$. $v_4 \in T(e_5)$ and therefore, $\text{visited-tail-node}(e_5) = 1$. Similarly $v_4 \in T(e_4)$ and as a result, $\text{visited-tail-node}(e_4) = 2$. Because $\text{visited-tail-node}(e_4) = |T(e_4)|$ and the head node of e_4 has not being visited before ($\text{visited}(v_5) = \text{false}$), $\text{visit}(v_5)$ is executed and the following labels are modified: $\text{discovered-order}(v_5) = 5$, $\text{visited}(v_5) = \text{true}$. Then, v_5 is stored in $\mathcal{V}'$ and (v_4, v_5) in $\mathcal{E}'$. v_5 is in $T(e_5)$ and therefore, $\text{visited-tail-node}(e_5) = 2$. Because $\text{visited-tail-node}(e_5) = |T(e_5)|$, the head node v_1 becomes reachable. However, v_1 has been visited before ($\text{visited}(v_1) = \text{true}$) and $\text{visit}(v_1)$ cannot be executed (if-condition in Line 12).

The execution of $\text{visit}(v_5)$ concludes, $\text{explored-order}(v_5) = 6$ (Line 20), and it backtracks to $\text{visit}(v_4)$. The same happens with $\text{visit}(v_4)$, $\text{explored-order}(v_4) = 7$, and it backtracks to $\text{visit}(v_3)$. The execution of $\text{visit}(v_3)$ also finishes, $\text{explored-order}(v_3) = 8$, and it backtracks to $\text{visit}(v_2)$. Similarly, the execution of $\text{visit}(v_2)$ ends, $\text{explored-order}(v_2) = 9$, and it backtracks to $\text{visit}(v_1)$. The same again happens with $\text{visit}(v_1)$, $\text{explored-order}(v_1) = 10$, and the execution of the algorithm continues in function $\text{main}(\cdot)$. All nodes have been discovered

and the execution of function $\text{main}(\cdot)$ terminates. The DFS-graph $\mathcal{G}$ produced by function $\text{main}(\cdot)$ is depicted in Figure 2.20 (b). In $\mathcal{G}$ is possible to reach v_4 from v_3 and v_1 from v_5 . This is not possible in the hypergraph $\mathcal{H}$. $\triangleleft$

As we did with BFS for directed hypergraphs, we can modify this version of DFS to ensure that node-to-node connectivity is preserved in the DFS-graph.

2.6 Strong Connectivity in Directed Hypergraphs

To the best of our knowledge, there is not an algorithm that can compute the strongly-connected components of a directed hypergraph.⁹ The only relevant reference that we found so far includes a study of the complexity of the problem of identifying the strongly-connected components of a directed hypergraph for the notion of B -connectivity [2]. The author of this work proved that the lower bound of this problem is $O(|\mathcal{H}|^2/\log^2(|\mathcal{H}|))$, where $|\mathcal{H}|$ is the size of the input hypergraph. This implies that it might not be possible to find an algorithm that can compute the strongly-connected components of a directed hypergraph in linear time (with respect to the size of the input hypergraph) like in the case of directed graphs.¹⁰

In [2], two algorithms are introduced to compute the *terminal* strongly-connected components of a directed hypergraph. These algorithms have been defined based on the following two observations [2]:

- (i) every terminal strongly-connected component of $\mathcal{G}(\mathcal{H})$ that only has one node (singleton) is also a terminal strongly-connected component in $\mathcal{H}$, where $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a directed hypergraph and $\mathcal{G}(\mathcal{H})$ is the *graph fragment* of $\mathcal{H}$ (Definition 2.31), and
- (ii) collapsing a strongly-connected component of a directed hypergraph into a single node do not alter the B -connectivity relation between the nodes of the hypergraph that do not belong to the collapsed strongly-connected component.

The first algorithm is depicted in Figure 2.21. The second algorithm, not included in this section, improved the performances of the previous one. According to the author, the time complexity of the second algorithm is almost linear with respect to the size of the input hypergraph [2].

⁹The Naïve algorithm for computing the strongly-connected components of a directed hypergraph, introduced in Chapter 3, was applied in [27] first. A similar version of this algorithm is studied in [28].

¹⁰The computational complexity of the problem of computing the strongly-connected components of a directed hypergraph is even worst if we take into account that the number of hyperedges of a directed hypergraph can be exponential with respect to the number of nodes whereas in a directed graph is at most quadratic.

```

function compute-terminal-sccs ( $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ )
1:  $\mathcal{H}_{pc} := \mathcal{H}$ 
2:  $\text{finish} := \text{false}$ 
3: while  $\text{finish} = \text{false}$  do
4:    $\text{finish} := \text{true}$ 
5:    $\mathcal{S}_T := \text{find-terminal-sccs}(\mathcal{G}(\mathcal{H}_{pc}))$ 
6:   for every  $scc \in \mathcal{S}_T$  do
7:     if  $(|scc| > 1)$  then
       // Collapse scc if it is not singleton
8:        $\text{finish} := \text{false}$ 
9:        $\mathcal{H}_{pc} := \text{collapse-scc}(scc, \mathcal{H}_{pc})$ 
10:    end-if
11:  end-for
12: end-while
13: return  $\mathcal{S}_T$ 

```

Fig. 2.21 Algorithm for computing terminal SCCs in directed hypergraphs [2].

In Figure 2.21, $\mathcal{H}_{pc}$ represents the *partially condensed* directed hypergraph of the input directed hypergraph $\mathcal{H}$. It is not fully condensed because only the strongly-connected components of the graph fragment $(\mathcal{G}(\mathcal{H}))$ are computed (Line 5) and collapsed (Line 9). Notice that the notion of a partially condensed directed hypergraph can be defined as the quotient hypergraph $\mathcal{H} / \simeq_{\mathcal{G}} = (\mathcal{V} / \simeq_{\mathcal{G}}, \mathcal{E}_{\simeq_{\mathcal{G}}})$, where $\simeq_{\mathcal{G}}$ is the strong-connectivity relation over $\mathcal{V}$ in $\mathcal{G}(\mathcal{H})$.¹¹

Function $\text{find-terminal-sccs}(\cdot)$ can be implemented using a standard algorithm for computing the strongly-connected components of a directed graph like Tarjan [39]. This implies that the time complexity of this function might be linear with respect to the size of the input graph.

In the case of function $\text{collapse-scc}(\cdot, \cdot)$, it can be implemented by iterating over all hyperedges of the partially condensed hypergraph $(\mathcal{H}_{pc})$ and for every hyperedge, the function changes any tail or head node that it is in a strongly-connected component by the new node that represents this strongly-connected component after it was collapsed.

The following example from [2] shows how the function $\text{compute-terminal-sccs}(\cdot)$, depicted in Figure 2.21, works in practice.

Example 2.65. Let $\text{compute-terminal-sccs}(\cdot)$ be the function depicted in Figure 2.21. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be the directed hypergraph depicted in Figure 2.22 (a), where $\mathcal{V} = \{v_1, v_2, v_3, v_4, v_5, v_6\}$

¹¹The notion of *partially condensed* directed hypergraph is not explicitly mentioned in [2]. For more details on this notion, please refer to Chapter 3 (Section 3.3.3).

Fig. 2.22 (Example 2.65): computing terminal SCCs [2].

and $\mathcal{E} = \{e_1 = (\{v_1\}, \{v_2\}), e_2 = (\{v_2\}, \{v_3\}), e_3 = (\{v_3\}, \{v_1\}), e_4 = (\{v_2, v_3\}, \{v_4, v_5\}), e_5 = (\{v_3, v_5\}, \{v_6\})\}$.

Inside of the while-loop (Lines 3–12), it is computed the terminal strongly-connected components of the graph fragment of $\mathcal{H}$, $\mathcal{G}(\mathcal{H})$, by the function $\text{find-terminal-sccs}(\mathcal{G}(\mathcal{H}))$. According to the example, this function will identify the following terminal strongly-connected components (Line 5): $\mathcal{S}_T = \{\{v_1, v_2, v_3\}, \{v_4\}, \{v_5\}, \{v_6\}\}$. The terminal strongly-connected component $\{v_1, v_2, v_3\}$ has more than one node. Therefore, it is collapsed into a new node by the function $\text{collapse-scc}(\{v_1, v_2, v_3\}, \mathcal{H}_{pc})$ (Line 9). The new node is represented by the equivalent class $[v_1]_{\simeq}$. Collapsing the strongly-connected component $\{v_1, v_2, v_3\}$ implies that some hyperedges must be modified. For instance, the hyperedges e_1 , e_2 and e_3 are deleted from $\mathcal{H}_{pc}$ because their tail and head nodes do not exist any more. The hyperedge e_4 is substituted by the hyperedges $e_6 = (\{[v_1]_{\simeq}\}, \{[v_4]_{\simeq}\})$ and $e_7 = (\{[v_1]_{\simeq}\}, \{[v_5]_{\simeq}\})$. Similarly, the hyperedge e_5 is replaced by the hyperedge $e_8 = (\{[v_1]_{\simeq}, [v_5]_{\simeq}\}, \{[v_6]_{\simeq}\})$. The while-loop does not end after the current iteration is complete because the boolean variable *finish* is set to false (Line 8).

In the next iteration, it is computed the terminal strongly-connected components of the graph fragment of the $\mathcal{H}_{pc}$ again. In this case, only three terminal strongly-connected components are identified: $\mathcal{S}_T = \{\{[v_4]_{\simeq}\}, \{[v_5]_{\simeq}\}, \{[v_6]_{\simeq}\}\}$. The node that represents the equivalent class $[v_1]_{\simeq}$ is a strongly-connected component but it is not "terminal" because the out-degree of this node in $\mathcal{H}_{pc}$ is not zero. In fact, this node is in the tails of the hyperedges e_6 , e_7 and e_8 .

The terminal strongly-connected components $\{[v_4]_{\simeq}\}$, $\{[v_5]_{\simeq}\}$ and $\{[v_6]_{\simeq}\}$ are singleton. This implies that the the boolean variable *finish* does not change and the execution of the while-loop ends. Therefore, the function $\text{find-terminal-sccs}(\mathcal{G}(\mathcal{H}))$ finishes and returns $\mathcal{S}_T$.

◁

2.7 Summary

In this chapter, we have reviewed basic notions on graph theory related to directed graphs and hypergraphs. In particular, we study the notions of connectivity and strong connectivity in both representations and how to traverse a directed graph or hypergraph and compute their strongly-connected components.

In the case of directed graphs, there are efficient algorithms that can complete any of these tasks in linear time. Moreover, there are compact representations of directed graphs suitable for these algorithms.

Unfortunately, the same is not true in directed hypergraphs. The notion of connectivity in directed hypergraphs is more complex and not unique. There are several notions of connectivity that we can use depending of the context and the problem we aim to solve. Because of the different nature of connectivity in directed hypergraphs, it might be difficult (or even not possible) to adapt existent algorithms for directed graphs to work for directed hypergraphs. When it is possible, we usually find out that the complexity of these algorithms are significantly worst than their counterparts for directed graphs. Nevertheless, directed hypergraphs proved to be useful in problems related to relational databases, propositional logic, biological networks and ontologies that require the representation of n -ary relations [18, 31, 22, 38, 27].

To our surprise, we could not find algorithms to traverse completely a directed hypergraph or to compute its strongly-connected components. Similarly, there are not many representations to store a directed hypergraph. Given that the number of hyperedges can be exponential with respect to the number of nodes of a directed hypergraph, we might need to find a succinct representation of hypergraphs for practical applications [40, 27].

Chapter 3

Strong Connectivity in Directed Hypergraphs

We start this chapter by introducing the notion of approximating graph and some relevant properties. An approximating graph is a directed graph that preserves node-to-node connectivity of a directed hypergraph. Based on this notion, we define and implement a new Depth-first search algorithm for traversing directed hypergraphs, and furthermore, we explain how it can also be used to compute approximating graphs. In addition, we show that this algorithm can also identify the strongly-connected components of a directed hypergraph similar to the algorithm by Tarjan [39] for directed graphs.

To compare the complexity and performances of the new Depth-first search algorithm, we design and implement a new simple algorithm that computes the strongly-connected components of a directed hypergraph. Besides the study of complexity, we also show the correctness of both algorithms.

To speed up the performances of the simple algorithm for computing the strongly-connected components of a directed hypergraph, we discuss an optimization that collapses the strongly-connected components of the graph fragment of the hypergraph. To evaluate both algorithms, we generate a collection of artificial and random hypergraphs. The random hypergraphs are based on Erdős-Rényi [15, 16, 19] and Newman-Watts-Strogatz[44, 30] models for random graphs.

Chapter 3 is organized as follows. The notion of approximating graph is introduced in Section 3.1. A new version of DFS for directed hypergraphs is presented in Section 3.2. It is shown in Section 3.3 that this new algorithm can easily be extended to compute strongly-*B*-connected components. Moreover, a simple algorithm for the same purpose is also presented in this section. Both algorithms are evaluated in Section 3.4. A short summary of this chapter is provided in Section 3.5.

3.1 Directed Hypergraph Approximation

The B -connectivity relation $\geq_{B(\mathcal{H})}$ of a directed hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ relates sets of nodes with nodes (i.e., $\geq_{B(\mathcal{H})}$ is of the type $\geq_{B(\mathcal{H})}: 2^{\mathcal{V}} \times \mathcal{V}$). We denote with $\geq_{B(\mathcal{H})}^d$ the part of $\geq_{B(\mathcal{H})}$ that relates nodes with nodes, where $\geq_{B(\mathcal{H})}^d = \{(x, y) \in \mathcal{V} \times \mathcal{V} \mid \{x\} \geq_{B(\mathcal{H})} y\}$.

An approximating graph of a directed hypergraph is a directed graph that preserves node-to-node B -connectivity. This is made precise in the following definition.

Definition 3.1 (Approximating Graph). *Let $\mathcal{H}$ be a directed hypergraph, and let $\geq_{B(\mathcal{H})}$ be the B -connectivity relation of $\mathcal{H}$. A directed graph $\mathcal{G}$ is an approximating graph of $\mathcal{H}$ iff $\geq_{B(\mathcal{H})}^d = \geq_{\mathcal{G}}$, where $\geq_{B(\mathcal{H})}^d = \{(x, y) \in \mathcal{V} \times \mathcal{V} \mid \{x\} \geq_{B(\mathcal{H})} y\}$ and $\geq_{\mathcal{G}}$ is the connectivity relation of $\mathcal{G}$.* $\dashv$

There may be several approximating graphs of a directed hypergraph. This will become clear when we discuss how to produce an approximating graph from a directed hypergraph.

Because the connectivity relation between any pair of nodes is preserved in an approximating graph, we can reduce the problem of finding the strongly-connected components of a directed hypergraph to the problem of finding strongly-connected components in its approximating graph.

Proposition 3.2. *Let $\mathcal{H}$ be a directed hypergraph, and let $\mathcal{G}$ be an approximating graph of $\mathcal{H}$. Then: The strongly-connected components of $\mathcal{H}$ and $\mathcal{G}$ coincide.* $\dashv$

Proof. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, and let $\mathcal{G} = (\mathcal{V}, \mathcal{E}')$ be an approximating graph of $\mathcal{H}$. Let $\text{SCCs}(\mathcal{H})$ ($\text{SCCs}(\mathcal{G})$) be the strongly-connected components of $\mathcal{H}$ ($\mathcal{G}$, respectively). Let $\geq_{B(\mathcal{H})}$ be the B -connectivity relation of $\mathcal{H}$, and let $\geq_{\mathcal{G}}$ be the connectivity relation of $\mathcal{G}$. We have to show that $\text{SCCs}(\mathcal{H}) = \text{SCCs}(\mathcal{G})$.

‘ $\subseteq$ ’: Let $S \in \text{SCCs}(\mathcal{H})$, and let $x, y \in S$. Then $x \geq_{B(\mathcal{H})}^d y$ and $y \geq_{B(\mathcal{H})}^d x$ by definition of $\text{SCCs}(\cdot)$. By Definition 3.1, it holds that $x \geq_{\mathcal{G}} y$ and $y \geq_{\mathcal{G}} x$. Thus, $S \subseteq S'$ for some $S' \in \text{SCCs}(\mathcal{G})$. It remains to show that $S' \subseteq S$. Let $z \in S'$. Recall that $x \in S'$. Then $x \geq_{\mathcal{G}} z$ and $z \geq_{\mathcal{G}} x$ by definition of $\text{SCCs}(\cdot)$. By Definition 3.1, we have that $x \geq_{B(\mathcal{H})}^d z$ and $z \geq_{B(\mathcal{H})}^d x$. Thus, $z \in S$. We obtain that $S = S'$. Hence, $S \in \text{SCCs}(\mathcal{G})$. ‘ $\supseteq$ ’: This direction can be shown similarly. $\square$

We want to formally characterise the edges in an approximating graph. To this end, we introduce the notion of a hyperedge approximation. A directed edge e approximates a hyperedge e' if e connects a common ancestor of the tail nodes of e' with a head node of e' . We consider the notion of common ancestors in (possibly cyclic) directed hypergraphs.

Definition 3.3 (Common Ancestor). Let $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ be a directed hypergraph, and let $\geq_{B(\mathcal{H})}$ be the B -connectivity relation in $\mathcal{H}$. Let $\mathcal{V}' \subseteq \mathcal{V}_{\mathcal{H}}$. The set of common ancestors in $\mathcal{H}$ of the nodes in $\mathcal{V}'$ is the set $\text{CA}_{\mathcal{H}}(\mathcal{V}') = \{v \in \mathcal{V}_{\mathcal{H}} \mid \forall v' \in \mathcal{V}' : \{v\} \geq_{B(\mathcal{H})} v'\}$. $\dashv$

Definition 3.4 (Approximating Edge). Let $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ be a directed hypergraph. An edge $e' = (x, y)$ is an approximating edge of a hyperedge $e \in \mathcal{E}_{\mathcal{H}}$ iff $x \in \text{CA}_{\mathcal{H}}(T(e))$ and $y \in H(e)$. We denote with $\mathcal{E}_{\mathcal{H}}^{\sim}(e)$ the set of all approximating edges of e , where $\mathcal{E}_{\mathcal{H}}^{\sim}(e) = \{(x, y) \in \mathcal{V}_{\mathcal{H}} \times \mathcal{V}_{\mathcal{H}} \mid x \in \text{CA}_{\mathcal{H}}(T(e)) \wedge y \in H(e)\}$. We denote with $\mathcal{E}_{\mathcal{H}}^{\sim}$ the set of all approximating edges of $\mathcal{H}$, where $\mathcal{E}_{\mathcal{H}}^{\sim} = \bigcup \{\mathcal{E}_{\mathcal{H}}^{\sim}(e) \mid e \in \mathcal{E}_{\mathcal{H}}\}$. $\dashv$

We have that $e \in \mathcal{E}_{\mathcal{H}}^{\sim}(e)$ for a directed edge e , i.e., e is an approximating edge of itself. To see this, note that e has only one tail node (and one head node) and that every node is a common ancestor of itself.

The set $\mathcal{E}_{\mathcal{H}}^{\sim}$ consists of the pairs of nodes that are B -connected in $\mathcal{H}$. In this sense, $\mathcal{E}_{\mathcal{H}}^{\sim}$ can be seen as the transitive closure of the hyperedges in $\mathcal{H}$. This is made precise in the following proposition.

Proposition 3.5. Let $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ be a directed hypergraph, and let $\geq_{B(\mathcal{H})}$ be the B -connectivity relation of $\mathcal{H}$. Then: $\{x\} \geq_{B(\mathcal{H})} y$ iff $(x, y) \in \mathcal{E}_{\mathcal{H}}^{\sim}$, for all $x, y \in \mathcal{V}_{\mathcal{H}}$ with $x \neq y$. $\dashv$

Proof. ‘ $\Rightarrow$ ’: Suppose $x, y \in \mathcal{V}_{\mathcal{H}}$ with $x \neq y$ and $\{x\} \geq_{B(\mathcal{H})} y$. It follows by Condition (ii) in the definition of $\geq_{B(\mathcal{H})}$ that there exists a hyperedge $e \in \mathcal{E}_{\mathcal{H}}$ such that $\{x\} \geq_{B(\mathcal{H})} T(e)$ and $y \in H(e)$. Then $x \in \text{CA}_{\mathcal{H}}(T(e))$ by Definition 3.3. We obtain $(x, y) \in \mathcal{E}_{\mathcal{H}}^{\sim}(e)$, and $(x, y) \in \mathcal{E}_{\mathcal{H}}^{\sim}$ by Definition 3.4.

‘ $\Leftarrow$ ’: Suppose $x, y \in \mathcal{V}_{\mathcal{H}}$ with $x \neq y$ and $(x, y) \in \mathcal{E}_{\mathcal{H}}^{\sim}$. By Definition 3.4, it follows that $(x, y) \in \mathcal{E}_{\mathcal{H}}^{\sim}(e)$ for some hyperedge $e \in \mathcal{E}_{\mathcal{H}}$, and that $x \in \text{CA}_{\mathcal{H}}(T(e))$ and $y \in H(e)$. Then $\{x\} \geq_{B(\mathcal{H})} T(e)$ by Definition 3.3. Hence, $\{x\} \geq_{B(\mathcal{H})} y$ by Condition (ii) in the definition of $\geq_{B(\mathcal{H})}$. $\square$

The following proposition shows that the set $\mathcal{E}_{\mathcal{H}}^{\sim}$ yields an approximating graph, and that $\mathcal{E}_{\mathcal{H}}^{\sim}$ is indeed maximal with that property.

Proposition 3.6. Let $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ be a directed hypergraph, and let $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}})$ be a directed graph, where $\mathcal{V}_{\mathcal{G}} = \mathcal{V}_{\mathcal{H}}$ and $\mathcal{E}_{\mathcal{G}} = \mathcal{E}_{\mathcal{H}}^{\sim}$. Then: $\mathcal{G}$ is an approximating graph of $\mathcal{H}$. Obtain $\mathcal{G}'$ by adding a directed edge $e = (x, y) \in \mathcal{V}_{\mathcal{G}} \times \mathcal{V}_{\mathcal{G}} \setminus \mathcal{E}_{\mathcal{H}}^{\sim}$ with $x \neq y$. Then: $\mathcal{G}'$ is not an approximating graph of $\mathcal{H}$. $\dashv$

Proof. We start by showing that $\mathcal{G}$ is an approximating graph of $\mathcal{H}$. By Definition 3.1, we have to show that $\geq_{B(\mathcal{H})}^d = \geq_{\mathcal{G}}$, where $\geq_{B(\mathcal{H})}$ is the B -connectivity relation of $\mathcal{H}$ and $\geq_{\mathcal{G}}$ the connectivity relation of $\mathcal{G}$. It follows from Proposition 3.5 that $\geq_{B(\mathcal{H})}^d$ is equivalent to the reflexive closure of $\mathcal{E}_{\mathcal{H}}^{\approx}$ which in turn is equivalent to $\geq_{\mathcal{G}}$.

We now show that $\mathcal{G}'$ is not an approximating graph of $\mathcal{H}$. By Definition 3.1, we show that $\geq_{\mathcal{G}'} \neq \geq_{B(\mathcal{H})}^d$, where $\geq_{\mathcal{G}'}$ is the connectivity relation of $\mathcal{G}'$. It holds that $e \in \geq_{\mathcal{G}'}$. It is sufficient to show that $e \notin \geq_{B(\mathcal{H})}^d$. This follows from Proposition 3.5. $\square$

Not all edges in $\mathcal{E}_{\mathcal{H}}^{\approx}$ are required to be present in an approximating graph of a hypergraph. In the following example, we show that node-to-node connectivity is preserved if not all possible approximating edges are included in the approximating graph.

Example 3.7. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a B -hypergraph, where $\mathcal{V} = \{v_1, \dots, v_7\}$ and $\mathcal{E} = \{e_1 = (\{v_1\}, v_2), e_2 = (\{v_1\}, v_3), e_3 = (\{v_1\}, v_4), e_4 = (\{v_2\}, v_3), e_5 = (\{v_2\}, v_5), e_6 = (\{v_2\}, v_6), e_7 = (\{v_3\}, v_2), e_8 = (\{v_3\}, v_5), e_9 = (\{v_3\}, v_6), e_{10} = (\{v_4\}, v_5), e_{11} = (\{v_4\}, v_6), e_{12} = (\{v_5, v_6\}, v_7)\}$.

The set of common ancestors of the tail nodes of the B -hyperedge e_{10} is $\text{CA}_{\mathcal{H}}(\{v_5, v_6\}) = \{v_1, v_2, v_3, v_4\}$. Therefore, the directed hyperedge e_{10} can be approximating using the following directed edges: $e_{12} = (\{v_1\}, v_7)$, $e_{13} = (\{v_2\}, v_7)$, $e_{14} = (\{v_3\}, v_7)$ and $e_{15} = (\{v_4\}, v_7)$. To ensure node-to-node connectivity, it would be sufficient to replace the directed hyperedge e_{10} with one of the following pairs of edges: $\{e_{13}, e_{15}\}$ or $\{e_{14}, e_{15}\}$. Notice that the nodes v_2, v_3 and v_4 represent the set of lowest common ancestors of the tail nodes $\{v_5, v_6\}$. $\triangleleft$

We use the notions of common ancestor and lowest common ancestor in possibly cyclic directed hypergraphs. We write $\{v\} \geq_{B(\mathcal{H})} \mathcal{V}'$ if $\{v\} \geq_{B(\mathcal{H})} v'$ for all $v' \in \mathcal{V}'$.

Definition 3.8 (Lowest Common Ancestor). Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, and let $\geq_{B(\mathcal{H})}$ be the B -connectivity relation in $\mathcal{H}$. Let $S \subseteq \mathcal{V}$. We define $\text{LCA}_{\mathcal{H}}(S) = \{v \in \mathcal{V} \mid v \in \text{CA}_{\mathcal{H}}(S) \wedge \forall v' \in \text{CA}_{\mathcal{H}}(S) : \{v\} \geq_{B(\mathcal{H})} v' \rightarrow \{v'\} \geq_{B(\mathcal{H})} v\}$ be the set of lowest common ancestors in $\mathcal{H}$ of the nodes in S . $\dashv$

It holds that all nodes in $\text{CA}_{\mathcal{H}}(\mathcal{V}')$, for some set $\mathcal{V}'$, are B -connected to some node in $\text{LCA}_{\mathcal{H}}(\mathcal{V}')$. Moreover, if a node v is B -connected to a node v' such that v and v' are in $\text{LCA}_{\mathcal{H}}(\mathcal{V}')$ then v' is also B -connected to v . Note that $\text{LCA}_{\mathcal{H}}(\mathcal{V}') = \mathcal{V}'$ if $|\mathcal{V}'| = 1$. However, it turns out that for sets $\mathcal{V}'$ of nodes with $|\mathcal{V}'| > 1$, the set $\text{LCA}_{\mathcal{H}}(\mathcal{V}')$ is not minimal with that property as it was illustrated by Example 3.7.

Adding an approximating edge for a hyperedge e to a hypergraph does not change its B -connectivity relation. However, the approximating edge may turn out to be essential for maintaining the B -connectivity relation after e has been removed from the hypergraph. This is made precise in the following definition.

Definition 3.9 (Essential Approximating Edge). *Let $\mathcal{H}_1$ be a directed hypergraph, and let e be a hyperedge in $\mathcal{H}_1$. Obtain $\mathcal{H}_2$ from $\mathcal{H}_1$ by removing e . Let $\geq_{B(\mathcal{H}_1)}$ and $\geq_{B(\mathcal{H}_2)}$ be the B -connectivity relations of $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively. An approximating edge $e' = (x, y)$ of e in $\mathcal{H}_1$ is essential iff $\{x\} \geq_{B(\mathcal{H}_1)} y$ implies $\{x\} \not\geq_{B(\mathcal{H}_2)} y$. $\dashv$*

In other words, an approximating edge (x, y) for a hyperedge e is essential if removing e from the approximating graph breaks all paths from x to y . We leave for future work the precise characterization of the notions of minimal sets of approximating edges and minimal approximating graph.

3.2 Depth-first Search Algorithm for Directed Hypergraphs

We discuss a new version of DFS such that node-to-node connectivity in a directed hypergraph is preserved in the DFS-graph. This is possible because the DFS-graph produced by this new version of DFS is in fact, an approximating graph.

To save space, we have integrated three algorithms in Figures 3.1 & 3.2. We have also restricted the input hypergraph to B -hypergraphs but as we will show later in this section, it is very simple to extend the algorithm to deal with any directed hypergraph. DFS for B -hypergraphs corresponds to the lines in black. DFS-approximation represented by the black and red lines (lines labelled with the symbol ‘+’) extends DFS by storing in the set $\mathcal{E}'$ the approximating edges. Tarjan for B -hypergraphs is represented by black and blue lines (lines labelled with the symbol ‘*’).

As in the previous DFS algorithm for directed hypergraphs (Section 2.5.2 and Figures 2.18 & 2.19), DFS for B -hypergraphs is implemented using two functions: $\text{main}(\cdot)$ and $\text{visit}(\cdot)$. The function $\text{main}(\cdot)$ initializes the timestamps $\text{discovered-order}(\cdot)$ and $\text{explored-order}(\cdot)$ for every node of the input B -hypergraph. $\text{main}(\cdot)$ also ensures that all nodes of the B -hypergraph will be explored at least one time by the function $\text{visit}(\cdot)$.

The function $\text{visit}(\cdot)$ is a recursive function that explores a given node of a B -hypergraph. Let $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ be the input hypergraph given to $\text{main}(\cdot)$ and let $v \in \mathcal{V}_{\mathcal{H}}$ be the input node given to $\text{visit}(\cdot)$. Then, $\text{successors}(v)$ represents the list of nodes of $\mathcal{H}$ that are B -connected from v . When v is explored (all its successors has been visited), $\text{successors}(v)$ is equivalent to the B -connected component of v ($\geq_{B(\mathcal{H})}(\{v\})$). Unlike the previous version of DFS for directed hypergraphs where only hyperedges with v in their tail ($\mathcal{E}(v)$) are considered, the function $\text{visit}(\cdot)$ must also consider all complex B -hyperedges even if v is not in their tails (Line 6). If v is a LCA of the tail nodes of a complex hyperedge e , then the head node of the hyperedge e will be visited if the following conditions are satisfied: (a) the head

```

function main( $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ )
1:  $\mathcal{E}' := \{\}$  +
2: list-of-sccs :=  $\{\{\}\}$  *
3: stack :=  $\square$  *
4: for every  $v \in \mathcal{V}$  do
5:   discovered-order( $v$ ) := -1
6:   explored-order( $v$ ) := -1
7:   successors( $v$ ) :=  $\{\}$ 
8: end-for
9: order-counter := 0
10: for every  $v \in \mathcal{V}$  do
11:   if (successors( $v$ ) =  $\{\}$ ) then
12:     visit( $v$ )
13:   end-if
14: end-for
15:  $\mathcal{L} := \{(\text{discovered-order}(v), \text{explored-order}(v)) \mid v \in \mathcal{V}\}$  +
16: return  $\mathcal{G} = (\mathcal{V}, \mathcal{E}', \mathcal{L})$  +
    list-of-sccs *

```

Fig. 3.1 DFS/Approximation/Tarjan main function for B -hypergraphs.

node of e is not already in successors(v) (if-condition in Line 12), and (b) the head node of e has not been discovered before (if-condition in Line 14).

Condition (a) prevents the exploration of transitive hyperedges which does not affect to the correctness of the algorithm and might improve its performance.

Example 3.10. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a B -hypergraph, where $\mathcal{V} = \{v_1, \dots, v_5\}$ and $\mathcal{E} = \{e_1 = (\{v_1\}, v_2), e_2 = (\{v_2\}, v_3), e_3 = (\{v_2, v_3\}, v_4), e_4 = (\{v_3, v_4\}, v_5), e_5 = (\{v_4, v_5\}, v_1)\}$.

DFS does not follow any specific order when it traverses a B -hypergraph. In the case of nodes v_3, v_4 and v_5 , visit($\cdot$) cannot visit any other node from any of them. For simplicity, we assume that DFS will visit v_1 first. Then, visit(v_1) is called from main($\mathcal{H}$) and the following variables are updated: discovered-order(v_1) := 1 (Line 2), successors(v_1) := $\{v_1\}$ (Line 4) and $\mathcal{E}_{\text{check}} := \{e_1, e_3, e_4, e_5\}$ (Line 6). The order in which the edges in $\mathcal{E}_{\text{check}}$ are explored is not fixed. Suppose that the first edge considered is e_1 . The head node v_2 is not in successors(v_1) and successors(v_2) = $\{\}$. Therefore, visit(v_2) is called (Line 15) and the following variables are updated: discovered-order(v_2) := 2 (Line 2), successors(v_2) := $\{v_2\}$ (Line 4) and $\mathcal{E}_{\text{check}} := \{e_2, e_3, e_4, e_5\}$ (Line 6). It can readily be seen that during the execution of the main loop in visit(v_2), the head nodes of all B -hyperedges in $\mathcal{E}_{\text{check}}$ will be explored. This is expected because $\text{CA}_{\mathcal{H}}(\{v_2, v_3\}) = \text{CA}_{\mathcal{H}}(\{v_3, v_4\}) = \text{CA}_{\mathcal{H}}(\{v_4, v_5\}) = \{v_1, v_2\}$.

```

function visit( $v$ )
1: order-counter := order-counter + 1
2: discovered-order( $v$ ) := order-counter
3: min-order( $v$ ) := order-counter
4: successors( $v$ ) :=  $\{v\}$ 
5: push(stack,  $v$ )
6:  $\mathcal{E}_{\text{check}} := \{e = (\mathcal{V}_e, v_e) \in \mathcal{E} \mid v \in \mathcal{V}_e \vee |\mathcal{V}_e| > 1\}$ 
7: new-successors := true
8: while (new-successors) do
9:   new-successors := false
10:   $\mathcal{E}_{\text{re-check}} := \{\}$ 
11:  for every  $e = (\mathcal{V}_e, v_e) \in \mathcal{E}_{\text{check}}$  do
12:    if ( $v_e \notin \text{successors}(v)$ ) then
13:      if ( $\mathcal{V}_e \subseteq \text{successors}(v)$ ) then
14:        if ( $\text{successors}(v_e) = \{\}$ ) then
15:          visit( $v_e$ ) // recursive call
16:          min-order( $v$ ) := min(min-order( $v$ ), min-order( $v_e$ ))
17:        else
18:          if stack contains  $v_e$  then
19:            min-order( $v$ ) := min(min-order( $v$ ), discovered-order( $v_e$ ))
20:          end-if
21:        end-if
22:        successors( $v$ ) := successors( $v$ )  $\cup$  successors( $v_e$ )
23:        new-successors := true
24:         $\mathcal{E}' := \mathcal{E}' \cup \{(v, v_e)\}$  // store approximating edge +
25:      else
26:         $\mathcal{E}_{\text{re-check}} := \mathcal{E}_{\text{re-check}} \cup \{e\}$  //  $e$  must be reconsidered
27:      end-if
28:    end-if
29:  end-for
30:   $\mathcal{E}_{\text{check}} := \mathcal{E}_{\text{re-check}}$ 
31: end-while
32: order-counter := order-counter + 1
33: explored-order( $v$ ) := order-counter
34: // if  $v$  is a root-node, then update successor lists (and compute the SCC)
35: if (min-order( $v$ ) = discovered-order( $v$ )) then
36:    $\text{scc} := \{\}$  *
37:   repeat
38:      $s := \text{pop}(\text{stack})$ 
39:     successors( $s$ ) := successors( $v$ )
40:     add(scc,  $s$ ) *
41:   until ( $s = v$ )
42:   add(list-of-sccs, scc) *
43: end-if

```

Fig. 3.2 DFS/Approximation/Tarjan visit function for B -hypergraphs.

Fig. 3.3 Example 3.10: DFS for B -hypergraphs

Therefore, $\text{successors}(v_2) = \mathcal{V}$ and when $\text{visit}(v_2)$ backtracks, $\text{successors}(v_1) = \mathcal{V}$ also. The execution of $\text{visit}(v_1)$ continues but no more edges in $\mathcal{E}_{\text{check}}$ will be considered because their head nodes are in $\text{successors}(v_1)$. This concludes the execution of $\text{visit}(v_1)$ and $\text{main}(\mathcal{H})$.

Notice that during the execution of $\text{visit}(v_2)$, the B -hyperedges e_4 and e_5 were evaluated even if v_2 is not one of their tail nodes. This ensures that the function $\text{visit}(v_2)$ would consider visiting all the nodes of the hypergraph that are B -connected from v_2 possible connected nodes. In this example, it would be possible to determine that the nodes v_1 and v_2 are strongly B -connected. This is, they form a strongly-connected component.

The result of the execution of $\text{main}(\mathcal{H})$ can be represented using the following directed labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}', \mathcal{L})$, where $\mathcal{V}$ is the same than in the B -hypergraph, $\mathcal{E}' = \{e_1 = (v_1, v_2), e_2 = (v_2, v_3), e_3 = (v_2, v_4), e_4 = (v_2, v_5), e_5 = (v_2, v_1)\}$ and $\mathcal{L} = \{l(v_1) = (1, 10), l(v_2) = (2, 9), l(v_3) = (3, 4), l(v_4) = (5, 6), l(v_5) = (7, 8)\}$. Notice that $\mathcal{G}$ preserves node-to-node connectivity of the original B -hypergraph $\mathcal{H}$. Both graphs, $\mathcal{H}$ and $\mathcal{G}$, are depicted in Figure 3.3. $\triangleleft$

We define the DFS-graph computed by DFS (DFS-approximation) as follows.

Definition 3.11 (DFS-Graph). Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a B -hypergraph. The directed labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}', \mathcal{L})$ that is returned by the function $\text{main}(\cdot)$ is called depth-first-search graph (DFS-graph) of $\mathcal{H}$. $\dashv$

In Proposition 3.16, we show correctness of $\text{main}(\cdot)$ for DFS only. To this end, we show that a DFS-graph of a B -hypergraph $\mathcal{H}$ is an approximating graph of $\mathcal{H}$. We start by introducing some auxiliary notions and a lemma.

Definition 3.12 ($\geq$ -admissible hyperpath). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a B -hypergraph, and let $x, y \in \mathcal{V}$ such that $\{x\} \geq_{B(\mathcal{H})} y$. Let $\mathcal{H}_{B(\{x\} \rightarrow y)} = (\mathcal{V}_{B(\{x\} \rightarrow y)}, \mathcal{E}_{B(\{x\} \rightarrow y)})$ be a B -hyperpath from x to y .*

A $\geq$ -admissible hyperpath from x to y is a sequence $\pi_{x \rightarrow y}$ of the form $\pi_{x \rightarrow y} = (v_1, e_1, v_2, e_2, \dots, e_{n-1}, v_n)$, $n > 0$, such that

- $v_i \in \mathcal{V}_{B(\{x\} \rightarrow y)}$, for all $i \in \{1, \dots, n\}$,
- $e_j \in \mathcal{E}_{B(\{x\} \rightarrow y)}$, for all $j \in \{1, \dots, n-1\}$,
- $v_1 = x$ and $v_1 \in T(e_1)$,
- $v_n = y$ and $v_n \in H(e_{n-1})$, and
- $v_i \in H(e_{i-1}) \cap T(e_i)$, for all $i \in \{2, \dots, n-1\}$.

A $\geq$ -admissible hyperpath $\pi_{x \rightarrow y}$ from x to y in $\mathcal{H}_{B(\{x\} \rightarrow y)}$ is a directed path (Definition 2.33) from x to y in $\mathcal{H}_{B(\{x\} \rightarrow y)}$.

A $\geq$ -admissible hyperpath $\pi_{x \rightarrow y}$ from x to y in $\mathcal{H}_{B(\{x\} \rightarrow y)}$ is acyclic if $x \neq y$.

A $\geq$ -admissible hyperpath $\pi_{x \rightarrow y}$ from x to y in $\mathcal{H}_{B(\{x\} \rightarrow y)}$ is simple if $v_i \neq v_j$ for all $i, j \in \{1, \dots, n-1\}$ with $i \neq j$. $\dashv$

We denote with $\text{nodes}(\pi)$ the set of nodes in a $\geq$ -admissible hyperpath $\pi = (v_1, e_1, \dots, e_{n-1}, v_n)$, i.e., $\text{nodes}(\pi) = \{v_1, \dots, v_n\}$. We refer to the length of a $\geq$ -admissible hyperpath π as the number $|\text{nodes}(\pi)|$ of its nodes. The following proposition can readily be seen.

Proposition 3.13 ($\geq$ -admissible hyperpath). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a B -hypergraph, and let $x, y \in \mathcal{V}$. Let $\geq_{B(\mathcal{H})}$ be the B -connectivity relation in $\mathcal{H}$. Then: $\{x\} \geq_{B(\mathcal{H})} y$ iff there exists a simple $\geq$ -admissible hyperpath from x to y in $\mathcal{H}$. $\dashv$*

Lemma 3.14 (Properties of DFS for B -hypergraphs). *Let $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ be a B -hypergraph. Let $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}}, \mathcal{L}_{\mathcal{G}})$ be a directed labelled graph with $\mathcal{G} = \text{main}(\mathcal{H})$. Let $\text{main}(\cdot)$ be the function depicted in Figure 3.1 and let $\text{visit}(\cdot)$ be the function depicted in Figure 3.2. Let $\text{successors} : \mathcal{V}_{\mathcal{H}} \rightarrow 2^{\mathcal{V}_{\mathcal{H}}}$ be the node labelling function that is computed during the execution of $\text{main}(\mathcal{H})$. Then the following holds:*

- (i) *for all $v \in \mathcal{V}_{\mathcal{H}}$: $\text{visit}(v)$ is called exactly once;*
- (ii) *for all $x, y \in \mathcal{V}_{\mathcal{H}}$: $\{x\} \geq_{B(\mathcal{H})} y$ iff $y \subseteq \text{successors}(x)$; and*
- (iii) *for all $x, y \in \mathcal{V}_{\mathcal{G}}$: $\{x\} \geq_{\mathcal{G}} y$ iff $y \subseteq \text{successors}(x)$. $\dashv$*

We show Lemma 3.14 by showing the following lemma instead.

Lemma 3.15 (Properties of DFS for B -hypergraphs). *Let $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ be a B -hypergraph. Let $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}}, \mathcal{L}_{\mathcal{G}})$ be a directed labelled graph with $\mathcal{G} = \text{main}(\mathcal{H})$. Let $\text{main}(\cdot)$ be the function depicted in Figure 3.1 and let $\text{visit}(\cdot)$ be the function depicted in Figure 3.2. Let $\text{successors} : \mathcal{V}_{\mathcal{H}} \rightarrow 2^{\mathcal{V}_{\mathcal{H}}}$ be the node labelling function that is computed during the execution of $\text{main}(\mathcal{H})$. Then the following holds:*

- (i) $\mathcal{V}_{\mathcal{H}} = \mathcal{V}_{\mathcal{G}}$;
- (ii) for all $v \in \mathcal{V}_{\mathcal{H}}$: $\text{visit}(v)$ is called exactly once;
- (iii) for all $x, y \in \mathcal{V}_{\mathcal{H}}$: $\{x\} \geq_{B(\mathcal{H})} y$ iff $\text{successors}(y) \subseteq \text{successors}(x)$;
- (iv) for all $x, y \in \mathcal{V}_{\mathcal{H}}$: $\{x\} \geq_{B(\mathcal{H})} y$ iff $y \subseteq \text{successors}(x)$;
- (v) for all $x, y \in \mathcal{V}_{\mathcal{G}}$: $\{x\} \geq_{\mathcal{G}} y$ iff $y \subseteq \text{successors}(x)$. ⊥

Proof. In this proof we refer to the algorithms $\text{main}(\cdot)$ in Figure 3.1 and $\text{visit}(\cdot)$ in Figure 3.2.

We start the proof of Lemma 3.14 by showing Item (i). It holds that $\mathcal{V}_{\mathcal{G}} = \mathcal{V}_{\mathcal{H}}$ by Line 16 of $\text{main}(\cdot)$.

Consider Item (ii). Every call of $\text{visit}(\cdot)$ (in Line 12 of $\text{main}(\cdot)$ and in Line 15 of $\text{visit}(\cdot)$) is guarded by an if-statement (in Line 11 of $\text{main}(\cdot)$ and in Line 14 of $\text{visit}(\cdot)$). This ensures that the call $\text{visit}(v)$, for $v \in \mathcal{V}_{\mathcal{H}}$, is executed only if $\text{successors}(v) = \emptyset$. In Line 4 of $\text{visit}(v)$, $\text{successors}(v)$ is set to a value different from the empty set. This ensures that $\text{visit}(v)$ will not be called again. The for-loop from Line 10 to Line 14 in $\text{main}(\cdot)$ ensures that $\text{visit}(v)$ is called for all nodes $v \in \mathcal{V}_{\mathcal{H}}$.

Consider Item (iii). We first show the direction ‘ $\Rightarrow$ ’ by induction that (IH) for all $x, y \in \mathcal{V}_{\mathcal{H}}$: $\{x\} \geq_{B(\mathcal{H})} y$ implies $\text{successors}(y) \subseteq \text{successors}(x)$. The induction proceeds on the length n of the longest simple $\geq$ -admissible hyperpath $\pi_{x \rightarrow y}$ from a node x to a node y in $\mathcal{H}$. Such simple paths of maximal length exist as $\mathcal{H}$ is finite (however, these paths are not necessarily unique). In the base case, we have that $n = 1$ and, thus, $x = y$. By Line 4 in function $\text{visit}(\cdot)$, $\text{successors}(x) = \{x\}$. We obtain $\text{successors}(x) \subseteq \text{successors}(y)$ by reflexivity of $\subseteq$.

Assume that we have shown the IH for n . For the induction step, we have to show that the IH holds for $n + 1$. Let $x, y \in \mathcal{V}_{\mathcal{H}}$ such that a longest simple $\geq$ -admissible from x to y is of length at most $n + 1$. Let $\pi_{x \rightarrow y} = (v_1, e_1, \dots, v_k, e_k, v_{k+1})$ be a longest such path, i.e. $|\text{nodes}(\pi_{x \rightarrow y})| \leq n + 1$. Let $e_n = (v_{e_k}, v_{e_k}) \in \mathcal{E}_{\mathcal{H}}$ be the last hyperedge of $\pi_{x \rightarrow y}$. Let $z \in \mathcal{V}_{e_k}$ be a tail node of e_k . As $\pi_{x \rightarrow y}$ is $\geq$ -admissible, it holds by Definition 3.12 that $\{x\} \geq_{B(\mathcal{H})} z$.

By Proposition 3.13 there exists a simple $\geq$ -admissible hyperpath from x to z in $\mathcal{H}$. Let $\pi_{x \rightarrow z}$ be such a hyperpath. The length of $\pi_{x \rightarrow z}$ is at most n . Suppose not, i.e. $\pi_{x \rightarrow z}$ is of length greater than n . Then $\pi_{x \rightarrow y}$ extended with the hyperedge e_k and the node y is a simple $\geq$ -admissible acyclic B -hyperpath from x to y in $\mathcal{H}$ that is longer than $n + 1$, which contradicts the fact that $\pi_{x \rightarrow y}$ is a longest such hyperpath. Therefore the IH applies to z , and we obtain that $\text{successors}(z) \subseteq \text{successors}(x)$ by the IH. By Line 4 of $\text{visit}(\cdot)$, it holds that $z \in \text{successors}(z)$. Thus, $\mathcal{V}_{e_k} \subseteq \text{successors}(x)$. We distinguish two cases depending on where $\mathcal{V}_{e_k} \subseteq \text{successors}(x)$ is established during the execution of $\text{main}(\mathcal{H})$: (a) in Line 22 of $\text{visit}(x)$, and (b) in Line 38 of $\text{visit}(v)$, for some node $v \in \mathcal{V}_{\mathcal{H}}$.

Consider Case (a). In Line 23 of $\text{visit}(x)$ the flag `new-successors` is set to true. Consequently the while-loop from Line 8 to Line 31 is entered again. We distinguish two cases for the for-loop from Line 11 to Line 29 of $\text{visit}(x)$: (a1) $e_k \in \mathcal{E}_{\text{check}}$ and (a2) $e_k \notin \mathcal{E}_{\text{check}}$. In Case (a1), the for-loop will be entered with e_k selected. We further distinguish two cases for the if-statement in Line 12: (a1)(i) $y \in \text{successors}(x)$, and (a1)(ii) $y \notin \text{successors}(x)$.

Consider Case (a1)(i). From $y \in \text{successors}(x)$ it follows by Line 22 of $\text{visit}(\cdot)$ that there exists a sequence $v_1 v_2 \dots v_k$, $k > 0$, of nodes such that $v_1 = y$, $v_k = x$, $v_i \in \mathcal{V}_{\mathcal{H}}$ for all $i \in \{1, \dots, k\}$, and $\text{successors}(v_i) \subseteq \text{successors}(v_{i+1})$, for all $i \in \{1, \dots, k-1\}$. Thus, $\text{successors}(y) \subseteq \text{successors}(x)$ by transitivity of $\subseteq$.

Consider Case (a1)(ii). The condition in the if-statement of Line 12 of $\text{visit}(x)$ is satisfied and we proceed to Line 13. Since we have that $\mathcal{V}_{e_k} \subseteq \text{successors}(x)$, the condition of the if-statement in Line 13 is satisfied. We proceed to Line 22, which ensures that $\text{successors}(y) \subseteq \text{successors}(x)$.

Consider Case (a2). The hyperedge e_k has been removed from $\mathcal{E}_{\text{check}}$ in a previous execution of the for-loop from Line 11 to Line 29 of $\text{visit}(x)$. Then the condition in the if-statement of Line 12 was not satisfied, i.e., $y \notin \text{successors}(x)$. Otherwise, e_k could have only been removed from $\mathcal{E}_{\text{check}}$ if the condition in the if-statement of Line 13 was satisfied, which cannot have been the case as $\mathcal{V}_{e_k} \subseteq \text{successors}(x)$ did not hold at that point. The remainder of this case proceeds similarly to Case (a1)(i) above.

Consider Case (b). Let $v \in \mathcal{V}_{\mathcal{H}}$ such that $\mathcal{V}_{e_k} \subseteq \text{successors}(x)$ is established in Line 38 of $\text{visit}(v)$. As $\text{successors}(x) = \text{successors}(v)$ by Line 38 of $\text{visit}(v)$, we have that $\mathcal{V}_{e_k} \subseteq \text{successors}(v)$. We now show that $\mathcal{V}_{e_k} \subseteq \text{successors}(v)$ has been established in Line 22 and not in Line 38 of $\text{visit}(v)$. Then, for showing Case (b), we proceed as in Case (a) above with node v taking the place of node x . Suppose by contradiction that the set $\text{successors}(v)$ is modified in Line 38 of $\text{visit}(v')$ of some node $v' \in \mathcal{V}_{\mathcal{H}}$ with $v \neq v'$.

The node v is removed from the stack by the `pop`-statement in Line 37 of $\text{visit}(v')$. This means that v is stored above/after node v' on the stack after v' has been declared as being

explored in Line 33 of $\text{visit}(v')$. It can readily be checked that, at this point during the execution of $\text{visit}(v')$, the call $\text{visit}(v)$ was finished before (i.e., node v has been explored). By assumption of Case (b), Line 38 of $\text{visit}(v)$ has been accessed. This means that the condition of the if-statement in Line 34 of $\text{visit}(v)$ must have been satisfied. Therefore, it holds that $\text{min-order}(v) = \text{discovered-order}(v)$.

But then v has been removed from the stack by the pop-statement in Line 37 during the execution of the repeat-until-loop from Line 36 to Line 40 of $\text{visit}(v)$. As node v is added to the stack only in Line 5 of $\text{visit}(v)$ and $\text{visit}(v)$ is executed once by Item (ii), v has not been added again to the stack. Therefore, v cannot have been removed from the stack in Line 37 of $\text{visit}(v')$; a contradiction. This finishes the direction ' $\Rightarrow$ ' of Item (iii).

Consider the direction ' $\Leftarrow$ ' of Item (iii). We show by induction on the value $n = \text{discovered-order}(\cdot)$ that (IH) for all $x, y \in \mathcal{V}_{\mathcal{H}}$ with $\text{discovered-order}(x) \leq \text{discovered-order}(y) \leq n$: $\text{successors}(y) \subseteq \text{successors}(x)$ implies $\{x\} \geq_{B(\mathcal{H})} y$. It can readily be checked that the direction ' $\Leftarrow$ ' of Item (iii) follows. In the base case, we have that $n = 1$ and $x = y$. We obtain $\{x\} \geq_{B(\mathcal{H})} y$ by reflexivity of $\geq$. Assume that we have shown the IH for n . For the induction step, we have to show that the IH holds for $n + 1$. Suppose that there is a node v with $\text{discovered-order}(v) = n + 1$; otherwise we can skip $n + 1$ and proceed to $n + 2$. Let $x, y \in \mathcal{V}_{\mathcal{H}}$ with $\text{discovered-order}(x) \leq \text{discovered-order}(y) \leq n + 1$. Suppose $\text{successors}(y) \subseteq \text{successors}(x)$. By Line 22 of $\text{visit}(\cdot)$, there exists an *inclusion sequence* $v_1 v_2 \dots v_k$, $k > 0$, of nodes such that $v_1 = y$, $v_k = x$, $v_i \in \mathcal{V}_{\mathcal{H}}$ for all $i \in \{1, \dots, k\}$, and $\text{successors}(v_i) \subseteq \text{successors}(v_{i+1})$, for all $i \in \{1, \dots, k - 1\}$. Let $e = (\mathcal{V}_e, v_e) \in \mathcal{E}_{\mathcal{H}}$ be the hyperedge that was selected in the for-loop from Line 11 to Line 29 of $\text{visit}(\cdot)$ during the execution of $\text{visit}(v_2)$. We have that $v_e = y$ and $\mathcal{V}_e \subseteq \text{successors}(v_2)$. From the IH it follows that $\{x\} \geq_{B(\mathcal{H})} v$ for all $v \in \text{successors}(v_2)$. Hence, $\{x\} \geq_{B(\mathcal{H})} y$. This finishes the direction ' $\Leftarrow$ ' and the proof of Item (iii).

Consider Item (iv). This item follows from Item (iii) together with the fact that $v \in \text{successors}(v)$ for all $v \in \mathcal{V}_{\mathcal{H}}$ by Line 4 of $\text{visit}(\cdot)$.

Item (v) can be shown along the lines of Item (iii). Note that for every hyperedge $e = (\mathcal{V}_e, v_e)$ with $|\mathcal{V}_e| > 1$ in $\mathcal{H}$ an approximating edge e' is added that connects a common ancestor of $\mathcal{V}_e$ with v_e . □

In the following Proposition, we prove that the DFS algorithm terminates and a DFS-graph of a B -hypergraph $\mathcal{H}$ is an approximating graph of $\mathcal{H}$.

Proposition 3.16 (Correctness of DFS for B -hypergraphs). *Let $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ be a B -hypergraph. Let $\text{main}(\cdot)$ be the function depicted in Figure 3.1. Let $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}}, \mathcal{L}_{\mathcal{G}})$ be a directed labelled graph such that $\mathcal{G} = \text{main}(\mathcal{H})$ is the DFS-graph of $\mathcal{H}$. Then:*

(i) The function $\text{main}(\cdot)$ for DFS terminates.

(ii) $\mathcal{G}' = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}})$ is an approximating graph of $\mathcal{H}$. ⊢

Proof. Consider item (i). By item (i) of Lemma 3.14, $\text{visit}(v)$ is called exactly once, for all $v \in \mathcal{V}_{\mathcal{H}}$. By Line 12 of $\text{visit}(v)$, the B -hyperedge in $\mathcal{E}_{\text{check}}$ which head node is in $\text{successors}(v)$ will not be considered. All B -hyperedge in $\mathcal{E}_{\text{check}}$ that fulfil the if-condition of Line 13 of $\text{visit}(v)$, will be considered only one time. The B -hyperedges stored in $\mathcal{E}_{\text{re-check}}$ will only be re-evaluated if new successors are included in $\text{successors}(v)$ (Lines 22-23).

Consider item (ii). By Definition 3.1, we have to show that for all $v \in \mathcal{V}_{\mathcal{H}}$,

$$\{v' \in \mathcal{V}_{\mathcal{H}} \mid \{v\} \geq_{B(\mathcal{H})} v'\} = \{v' \in \mathcal{V}_{\mathcal{G}} \mid \{v\} \geq_{\mathcal{G}} v'\}.$$

This follows by Item (iv) and Item (v) of Lemma 3.14. □

Proposition 3.17 (Time Complexity of DFS for B -hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a B -hypergraph. Let $\text{main}(\cdot)$ be the function depicted in Figure 3.1. The time complexity of the function $\text{main}(\cdot)$ for DFS is $O(|\mathcal{V}| \times |\mathcal{E}|^2)$.* ⊢

Proof. By item (i) of Lemma 3.14, $\text{visit}(v)$ is called exactly once, for all $v \in \mathcal{V}$. Therefore, we focus on the study of the time complexity of the function $\text{visit}(\cdot)$. By Line 6, $\mathcal{E}_{\text{check}}$ contains at most $|\mathcal{E}|$ hyperedges. Then, the while-loop (Lines 8-31) completes $|\mathcal{E}| \times (|\mathcal{E}| + 1)/2$ iterations in the worst case. This expression corresponds to $(|\mathcal{E}| + (|\mathcal{E}| - 1) + (|\mathcal{E}| - 2) + \dots + 2 + 1)$ many iterations. Using a boolean array that indicates which nodes has been stored in the stack, it is possible to verify the if-condition of Line 18 in constant time (this was also suggested in [39]). A boolean array can be also used to represent $\text{successors}(v)$, for all $v \in \mathcal{V}$. This decision implies that the if-condition in Line 12 can be completed in constant time and Lines 13&22 require $|\mathcal{V}|$ many steps. Therefore, each call to $\text{visit}(\cdot)$ takes at most $2 \times |\mathcal{V}| \times |\mathcal{E}|^2$ many steps (assuming $|\mathcal{E}|$ is sufficiently large). We can conclude that the time complexity of function $\text{main}(\cdot)$ is $O(|\mathcal{V}| \times |\mathcal{E}|^2)$. □

It is very simple to extend the function $\text{visit}(\cdot)$ depicted in Figure 3.2 to traverse any directed hypergraph and not only B -hypergraphs. Figure 3.4 shows the new version of the function $\text{visit}(\cdot)$ for directed hypergraphs. Notice that the function $\text{main}(\cdot)$ is the same as for B -hypergraphs. A directed hyperedge can have more than one node in the head. Therefore, the function $\text{visit}(\cdot)$ for B -hypergraphs has been modified to iterate over all possible head nodes of a hyperedge that become reachable (for-loop in Lines 13–25). Correctness of this new version of the DFS algorithm function $\text{visit}(\cdot)$ can readily be seen and the time complexity is the same as in the version for B -hypergraphs.

```

function visit( $v$ )
1: order-counter := order-counter + 1
2: discovered-order( $v$ ) := order-counter
3: min-order( $v$ ) := order-counter
4: successors( $v$ ) :=  $\{v\}$ 
5: push(stack,  $v$ )
6:  $\mathcal{E}_{\text{check}} := \{e \in \mathcal{E} \mid v \in H(e) \vee |H(e)| > 1\}$ 
7: new-successors := true
8: while (new-successors) do
9:   new-successors := false
10:   $\mathcal{E}_{\text{re-check}} := \{\}$ 
11:  for every  $e \in \mathcal{E}_{\text{check}}$  do
12:    if ( $H(e) \subseteq \text{successors}(v)$ ) then
13:      for every  $s \in H(e)$  do
14:        if ( $\text{successors}(s) = \{\}$ ) then
15:          visit( $s$ ) // recursive call
16:          min-order( $v$ ) := min(min-order( $v$ ), min-order( $s$ ))
17:        else
18:          if stack contains  $s$  then
19:            min-order( $v$ ) := min(min-order( $v$ ), discovered-order( $s$ ))
20:          end-if
21:        end-if
22:        successors( $v$ ) := successors( $v$ )  $\cup$  successors( $s$ )
23:        new-successors := true
24:         $\mathcal{E}' := \mathcal{E}' \cup \{(v, s)\}$  // store approximating edge +
25:      end-for
26:    else
27:       $\mathcal{E}_{\text{re-check}} := \mathcal{E}_{\text{re-check}} \cup \{e\}$  //  $e$  must be reconsidered
28:    end-if
29:  end-for
30:   $\mathcal{E}_{\text{check}} := \mathcal{E}_{\text{re-check}}$ 
31: end-while
32: order-counter := order-counter + 1
33: explored-order( $v$ ) := order-counter
34: // if  $v$  is a root-node, then update successor lists (and compute the SCC)
35: if (min-order( $v$ ) = discovered-order( $v$ )) then
36:    $\text{scc} := \{\}$  *
37:   repeat
38:      $s := \text{pop}(\text{stack})$ 
39:     successors( $s$ ) := successors( $v$ )
40:     add(scc,  $s$ ) *
41:   until ( $s = v$ )
42:   add(list-of-sccs, scc) *
43: end-if

```

Fig. 3.4 DFS/Approximation/Tarjan visit function for directed hypergraphs.

3.3 Computing Strongly- B -Connected Components in Directed Hypergraphs

3.3.1 Adapting Naïve Algorithm

The algorithm depicted in Figure 3.5 computes the strongly-connected components of a directed hypergraph. In short, the algorithm computes the B -connected component of each node of the hypergraph and identifies which of these B -connected components are the same. The nodes with identical B -connected components are in the same strongly-connected component.

The algorithm has been defined using two functions: $\text{main}(\cdot)$ and $\text{visit}(\cdot)$, where the former is very similar to the function $\text{main}(\cdot)$ of the naïve algorithm that computes the strongly-connected components of a directed graph (Figure 2.7, Section 2.3.1). The latter is nearly equivalent to the function $\text{visit}(\cdot)$ of the BFS algorithm (Figure 2.14, Section 2.5.1) which is also very similar to the function $\text{Bvisit}(\cdot, \cdot)$ depicted in Figure 2.12 (Section 2.5.1).

The function $\text{main}(\cdot)$ starts by fixing a total order $<$ on the set of nodes in $\mathcal{H}$ (Line 1). The order is used to represent a set $\mathcal{S}_v$ of all nodes connected from a node v as a sequence π_v of nodes (Line 7). The sequence π_v for every node v in $\mathcal{H}$ is then introduced as a branch in a labelled tree T (Lines 8–11). Note that each prefix of π_v is considered as a node in T . If the node is not already contained in T , then the node is labelled with the empty set (Line 9). Then the node v is added to the final node of the branch π_v (Line 11). Two different sequences of ordered nodes are introduced as different branches in T . A sequence of ordered nodes that is contained in another sequence of nodes already included in the tree might be part of the same branch of the tree (this depends of the total order $<$ considered). The function $\text{main}(\cdot)$ returns the labels of the tree T that are not empty (Line 14). Each non-empty label corresponds to a strongly-connected component of $\mathcal{H}$.

The following proposition can readily be seen.

Proposition 3.18 (Correctness of the naïve algorithm for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, and let $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ be the functions depicted in Figure 3.5. Then:*

(i) *for all $v \in \mathcal{V}$: $\text{visit}(v) = \geq_{B(\mathcal{H})}(v)$,*

(ii) *$\text{main}(\mathcal{H})$ terminates,*

(iii) *$\text{main}(\mathcal{H}) = \text{SCCs}(\mathcal{H})$.*

◻

```

function main( $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ )
1: fix total order  $<$  on  $\mathcal{V}$ 
2:  $T := \emptyset$ 
3: for every  $v \in \mathcal{V}$  do
4:   for every  $w \in \mathcal{V}$  do: visited( $w$ ) := false
5:   for every  $e \in \mathcal{E}$  do: visited-tail-node( $e$ ) := 0
6:    $\mathcal{S}_v := \text{visit}(v)$  // collect all successors of  $v$ 
7:    $\pi_v := \text{sort}_{<}(\mathcal{S}_v)$ 
   // store all successors of  $v$  in  $T$ 
8:   for every  $\pi' \in \text{prefix}(\pi_v)$  do
9:     if  $\pi' \notin \text{dom}(T)$  then  $T := T \cup \{(\pi', \emptyset)\}$ 
10:   end-for
11:    $T(\pi_v) := T(\pi_v) \cup \{v\}$ 
12: end-for
   // return SCCs( $\mathcal{H}$ )
13: return  $\{T(\pi) \mid \pi \in \text{dom}(T) \text{ and } T(\pi) \neq \emptyset\}$ 

function visit( $v$ )
1:  $Q := \{\}$ 
2:  $\mathcal{S} := \{\}$ 
3: enqueue( $Q, v$ )
4: visited( $v$ ) := true
   // Identify B-connected nodes from  $v$ 
5: while  $Q \neq \{\}$  do
6:    $u := \text{dequeue}(Q)$ 
7:    $\mathcal{S} := \mathcal{S} \cup \{u\}$ 
8:   for every  $e \in \mathcal{E}(u)$  do
9:     visited-tail-node( $e$ ) := visited-tail-node( $e$ ) + 1
10:    if (visited-tail-node( $e$ ) =  $|T(e)|$ ) then
      // All nodes in  $H(e)$  are B-connected from  $v$ 
11:      for every  $w \in H(e)$  do
12:        if (visited( $w$ ) = false) then
13:          visited( $w$ ) := true
14:          enqueue( $Q, w$ )
15:        end-if
16:      end-for
17:    end-if
18:  end-for
19: end-while
   // Return the B-connected component of  $v$ 
20: return  $\mathcal{S}$ 

```

Fig. 3.5 Naïve algorithm that computes the SCCs of a directed hypergraph.

Consider item (i) of Proposition 3.21. Proposition 2.48 shows that the function $\text{Bvisit}(\cdot, \cdot)$ terminates and returns the B -connected component of the input node in the input directed hypergraph. Since this function is similar to the function $\text{visit}(\cdot)$ depicted in Figure 3.5, item (i) of Proposition 3.21 holds. Regarding item (ii) of Proposition 3.21, the function $\text{visit}(\cdot)$ terminates because it is very similar to function $\text{Bvisit}(\cdot, \cdot)$. By Proposition 2.23, it has been proved that function $\text{main}(\cdot)$ of the naïve algorithm that computes the strongly-connected components of a directed graph terminates. Given how similar is this function to function $\text{main}(\cdot)$ depicted in Figure 3.5, item (ii) of Proposition 3.21 also holds. Finally, item (iii) of Proposition 3.21 also follows from Proposition 2.23 and item (i) of Proposition 3.21.

We show that the time complexity of the naïve algorithm that computes the strongly-connected components of a directed hypergraph is not linear with respect to the size of the hypergraph.

Proposition 3.19 (Time complexity of the naïve algorithm for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph. Let $\text{main}(\cdot)$ be the function depicted in Figure 3.5. Then: the time complexity of function $\text{main}(\cdot)$ is $O(|\mathcal{V}| \times |\mathcal{H}|)$, where $|\mathcal{H}| = |\mathcal{V}| + \sum_{e \in \mathcal{E}} (|T(e)| + |H(e)|)$. $\dashv$*

Proof. We start the proof by analysing the time complexity of the function $\text{visit}(\cdot)$. This function is very similar to the function $\text{Bvisit}(\cdot, \cdot)$ depicted in Figure 2.12. Therefore, the time complexity of function $\text{visit}(\cdot)$ is $O(|\mathcal{H}|)$.

We continue the proof by analysing the time complexity of the function $\text{main}(\cdot)$. In Line 1, a total order on $\mathcal{V}$ is fixed. This can be done in linear time with respect to the number of nodes of the hypergraph. After this task is executed, a for-loop over each of the nodes of the hypergraph must be completed (Lines 3&12). Most of the processing done by $\text{main}(\cdot)$ occurs inside of this for-loop which requires $|\mathcal{V}|$ iterations. The first task to be achieved inside of the loop is to set all nodes of the hypergraph as unvisited (Line 4) and all the counters associated to each hyperedge with the value zero (Line 5). Both tasks can be achieved in $|\mathcal{V}| + |\mathcal{E}|$ steps. The next task is to compute the B -connected component of the node currently selected by the main for-loop and store this B -connected component in $\mathcal{S}_v$ (Line 6). This is done by the function $\text{visit}(\cdot)$ in $|\mathcal{H}|$ steps in the worst-case. Next, the nodes of the connected component are ordered based on the total order that has been defined (Line 7). Depending on the algorithm implemented by the function $\text{sort}_{<}(\cdot)$, this can be done in $|\mathcal{V}| \times \log |\mathcal{V}|$ steps in the worst-case. The task that comes after aims to identify if the connected component computed earlier already exist (Lines 8–11). This is done by extracting each node of the connected component and by adding the node into a labelled tree which represents all connected components that has been computed so far. The order of

the nodes of each connected component is preserved in the labelled tree. For this kind of trees, the operations `insert()` and `search()` can be executed in $\log |\mathcal{V}|$ steps [10]. Therefore, Line 9 can be completed in $2 \times \log |\mathcal{V}|$ steps in the worst-case. Consequently, the for-loop in Lines 8–10 can be completed in $|\mathcal{V}| \times 2 \times \log |\mathcal{V}|$ steps. The last task in the main for-loop is to add a node in the label of the last node of the connected component that has been included in T (Line 11). This requires to search for the last node added in T . As it was explained earlier, this operation can be done in $\log |\mathcal{V}|$ steps.

As a summary, the number of steps to be completed inside of the main for-loop (Lines 3&12) is the following: $|\mathcal{V}| + |\mathcal{E}| + |\mathcal{H}| + |\mathcal{V}| \times \log |\mathcal{V}| + |\mathcal{V}| \times 2 \times \log |\mathcal{V}| + \log |\mathcal{V}|$. This means that the number of steps to complete the execution of the function `main()` is: $|\mathcal{V}| + |\mathcal{V}| \times (|\mathcal{V}| + |\mathcal{E}| + |\mathcal{H}| + |\mathcal{V}| \times \log |\mathcal{V}| + |\mathcal{V}| \times 2 \times \log |\mathcal{V}| + \log |\mathcal{V}|)$. Assuming that $|\mathcal{H}|$ is much larger than $|\mathcal{V}|$ and $|\mathcal{E}|$, we can conclude that the time complexity of function `main()` is $O(|\mathcal{V}| \times |\mathcal{H}|)$. $\square$

Example 3.20 shows how the naïve algorithm computes the strongly-connected components of a directed hypergraph. In particular, the example illustrates how the nodes of each strongly-connected component are stored in the labelled tree T .

Example 3.20. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a B -hypergraph, where $\mathcal{V} = \{v_1, \dots, v_5\}$ and $\mathcal{E} = \{e_1 = (\{v_1\}, v_2), e_2 = (\{v_2\}, v_3), e_3 = (\{v_2, v_3\}, v_4), e_4 = (\{v_3, v_4\}, v_5), e_5 = (\{v_4, v_5\}, v_1)\}$.

Line 1 in function `main()` fix a total order $<$ on $\mathcal{V}$. In this example, let $<$ be a total order such that $v_1 < v_2 < v_3 < v_4 < v_5$. Notice that $<$ is equal to the transitive closure of the relation $\{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5)\}$.

For each node of the hypergraph, the function `visit()` returns the B -connected component (Line 6). These are the connected components computed in each iteration of the for-loop (Lines 3&12): $CC(v_1) = CC(v_2) = \mathcal{V}$, $CC(v_3) = \{v_3\}$, $CC(v_4) = \{v_4\}$, and $CC(v_5) = \{v_5\}$.

Each connected component will be stored in a labelled tree T to identify which of them are the same connected component. Given the fixed order of this example, the tree consists of the following branches b_1 , b_2 , b_3 and b_4 , where

$$\begin{aligned} b_1 : & (v_1, \{\}) \longrightarrow (v_1 v_2, \{\}) \longrightarrow (v_1 v_2 v_3, \{\}) \longrightarrow (v_1 v_2 v_3 v_4, \{\}) \longrightarrow \\ & (v_1 v_2 v_3 v_4 v_5, \{v_1, v_2\}) \\ b_2 : & (v_3, \{v_3\}) \\ b_3 : & (v_4, \{v_4\}) \\ b_4 : & (v_5, \{v_5\}) \end{aligned}$$

The function `main()` returns the following strongly-connected components (one for each non-empty label in T): $\{\{v_1, v_2\}, \{v_3\}, \{v_4\}, \{v_5\}\}$. $\triangleleft$

```

function naïve( $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ )
1:  $\mathcal{V}' = \mathcal{V}$ 
2:  $\mathcal{E}' = \{\}$ 
3: for every  $v \in \mathcal{V}$  do
    // compute the  $B$ -connected component of  $v$ 
4:    $\mathcal{S}_v := \text{Bvisit}(v, \mathcal{H})$ 
    // compute approximating edges
5:   for every  $w \in \mathcal{S}_v \setminus \{v\}$  do
6:      $\mathcal{E}' = \mathcal{E}' \cup \{(v, w)\}$ 
7:   end-for
8: end-for
    // return  $\text{SCCs}(\mathcal{G})$  using Tarjan for directed graphs
9: return  $\text{main}(\mathcal{G} = (\mathcal{V}', \mathcal{E}'))$ 

```

Fig. 3.6 Naïve algorithm that computes the SCCs of a directed hypergraph [28].

Fig. 3.7 Approximating graph using the new version of the naïve algorithm

It has been suggested an alternative specification of the naïve algorithm that computes strongly-connected components of a directed hypergraph [28]. This new version of the naïve algorithm is depicted in Figure 3.6. The advantages of this new specification are that it is more compact and easier to understand, and it produces an approximating graph of the input directed hypergraph. The approximating graph generated is of maximal size because it contains a directed edge between any pair of nodes that are B -connected in the original hypergraph. Figure 3.7 shows an example of approximating graph produced by the naïve algorithm depicted in Figure 3.6. Notice that this approximating graph contains more edges than the approximating graph in Figure 3.3 produced by the depth-first-search algorithm for directed hypergraphs that we have discussed in Section 3.2.

The following propositions can readily be seen.

Proposition 3.21 (Correctness of the new version of the naïve algorithm for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, and let $\text{naïve}(\cdot)$ be the function depicted in Figure 3.6. Then:*

(i) $\text{naïve}(\mathcal{H})$ terminates,

(ii) $\text{naïve}(\mathcal{H}) = \text{SCCs}(\mathcal{H})$. ⊢

Proposition 3.22 (Time complexity of the naïve algorithm for directed hypergraphs). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph, and let $\text{naïve}(\cdot)$ be the function depicted in Figure 3.6. Then: the time complexity of function $\text{naïve}(\cdot)$ is $O(|\mathcal{V}| \times |\mathcal{H}|)$, where $|\mathcal{H}| = |\mathcal{V}| + \sum_{e \in \mathcal{E}} (|T(e)| + |H(e)|)$.* ⊢

3.3.2 Adapting Tarjan's Algorithm

By item (ii) of Proposition 3.16 (Section 3.2), it has been shown that the DFS-graph produced by the function $\text{main}(\cdot)$ depicted in Figure 3.1 is an approximating graph. That is, node-to-node connectivity in a B -hypergraph is preserved in the DFS-graph. Therefore, it would be possible to compute the strongly-connected components of a B -hypergraph in any of its DFS-graphs using one of the existent algorithms for directed graphs, e.g., the algorithm by Tarjan. This is similar to the approach followed by the second version of the naïve algorithm that we have discussed in Section 3.3.1 and it is depicted in Figure 3.6.

In this section, we discuss how DFS for B -hypergraphs can be extended to compute strongly-connected components similarly to algorithm by Tarjan [39]. As for directed graphs, the labels computed for each node of the DFS-graph are crucial in this version of Tarjan for B -hypergraphs. We first define the notion of a root-node [39] which corresponds to the first node of an strongly-connected component that is discovered during the execution of the function $\text{visit}(\cdot)$ depicted in Figure 3.2 (Section 3.2).

Definition 3.23 (Root-node of a strongly-connected component). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a B -hypergraph. Let $\text{main}(\cdot)$ be the function depicted in Figure 3.1. Let $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}}, \mathcal{L}_{\mathcal{G}})$ be a directed labelled graph such that $\mathcal{G} = \text{main}(\mathcal{H})$ is the DFS-graph of $\mathcal{H}$. Let $\mathcal{S} \subseteq \mathcal{V}_{\mathcal{G}}$ be a strongly-connected component of $\mathcal{G}$.*

A node $v \in \mathcal{V}_{\mathcal{G}}$ is the root-node of $\mathcal{S}$ iff for all $w \in \mathcal{S} \setminus \{v\}$, $\text{discovered-order}(v) < \text{discovered-order}(w)$. ⊢

The problem of finding the strongly-connected components of a B -hypergraph can be reduced to the problem of finding the root-nodes of these strongly-connected components in

a DFS-graph. Therefore, we need to indicate how a root-node can easily be identified. This can be done by using the function $\text{min-order}(\cdot)$ (also known as $\text{lowlink}(\cdot)$ [39]). The function $\text{min-order}(v)$ maps a node v to the value $\text{discovered-order}(w)$ of some node w that has been discovered before v (i.e., it holds that $\text{discovered-order}(w) \leq \text{discovered-order}(v)$) such that v and w are contained in the same strongly-connected component.

Given a DFS-graph computed from a B -hypergraph by $\text{main}(\cdot)$, we can now use the function $\text{min-order}(\cdot)$ to identify the root-node of each strongly-connected component as follows.

Proposition 3.24 (Characterization of a root-node). *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a B -hypergraph. Let $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}}, \mathcal{L}_{\mathcal{G}})$ be a directed labelled graph such that $\mathcal{G} = \text{main}(\mathcal{H})$ is the DFS-graph of $\mathcal{H}$. Let $\mathcal{S} \subseteq \mathcal{V}_{\mathcal{G}}$ be a strongly-connected component such that $v \in \mathcal{S}$.*

Then: v is the root-node of $\mathcal{S}$ iff $\text{discovered-order}(v) = \text{min-order}(v)$. $\dashv$

The proposition can be shown along the lines of the proof for the algorithm by Tarjan for directed graphs [39].

Figures 3.1&3.2 describe how DFS can be extended to compute the strongly-connected components of a B -hypergraph using a similar approach as Tarjan for directed graphs. The new lines that are needed for Tarjan are written in blue (lines labelled with the symbol ‘*’). As it can be observed, the new lines introduce very little changes. In fact, these lines only store in a list the different strongly-connected components that has been discovered during the execution of the function $\text{visit}(\cdot)$.

After a node is explored, the function $\text{visit}(\cdot)$ checks if the node is a root-node (Line 34). If it is the case, then all nodes in the stack that have been stored after the root-node are part of the same strongly-connected component. Tarjan will remove these nodes (including the root-node) from the stack (Lines 35-39) and they will be stored in the set of strongly-connected components (Line 40) that the function $\text{main}(\cdot)$ will return as a result.

The proofs for termination and time complexity of DFS for B -hypergraphs are also valid for this version of Tarjan. Notice that Tarjan for B -hypergraphs has worst time complexity than the naïve algorithms for directed hypergraphs that we have discussed in Section 3.3.1.

To show that Tarjan for B -hypergraphs is correct, we have to prove that the strongly connected components stored in list-of-sccs are correct.

Theorem 3.25 (Correctness of Tarjan for B -hypergraphs). *The extended version of DFS depicted in Figures 3.1 & 3.2 for computing strongly-connected components works correctly on any B -hypergraph. $\dashv$*

Proof. In this proof we refer to the algorithms $\text{main}(\cdot)$ in Figure 3.1 and $\text{visit}(\cdot)$ in Figure 3.2. Let $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ be a B -hypergraph. By Item (i) of Proposition 3.16, $\text{main}(\mathcal{H})$

Fig. 3.8 Example 3.8: Tarjan for B -hypergraphs

terminates. Let $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}}, \mathcal{L}_{\mathcal{G}})$ be a directed labelled graph with $\mathcal{G} = \text{main}(\mathcal{H})$. By Item (ii) of Proposition 3.16, $\mathcal{G}$ is an approximating graph of $\mathcal{H}$. By Proposition 3.17, the strongly-connected components of $\mathcal{H}$ and $\mathcal{G}$ coincide, i.e. $\text{SCCs}(\mathcal{H}) = \text{SCCs}(\mathcal{G})$.

We reduce the problem of determining the strongly-connected components of $\mathcal{H}$ to the problem of determining the root-nodes (cf. Definition 3.23) of these strongly-connected components in $\mathcal{G}$. Root-nodes are characterised in terms of the function $\text{min-order}(\cdot)$ by Proposition 3.24. We can show that the function $\text{min-order}(\cdot)$ is computed correctly by $\text{main}(\mathcal{H})$ and the subsequent calls to $\text{visit}(\cdot)$ similarly to Theorem 2.29 for directed graphs. $\square$

Notice that if we use the function $\text{visit}(\cdot)$ depicted in Figure 3.4 instead of the function depicted in Figure 3.2, the algorithm can compute the strongly-connected components of any directed hypergraph.

The following example illustrates how the algorithm discussed in this section can compute the strongly-connected components of a B -hypergraph.

Example 3.26. Let $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$ be a B -hypergraph, where $\mathcal{V}_{\mathcal{H}} = \{v_1, v_2, v_3, v_4, v_5\}$ and $\mathcal{E}_{\mathcal{H}} = \{e_1, e_2, e_3, e_4, e_5, e_6\} \cup \{e_7 = (\{v_1, v_5\}, v_4)\}$ with $e_1 = (v_1, v_2)$, $e_2 = (v_2, v_3)$, $e_3 = (v_3, v_2)$, $e_4 = (v_2, v_1)$, $e_5 = (v_4, v_5)$ and $e_6 = (v_5, v_3)$.

Tarjan for directed hypergraphs does not follow any specific order when it traverses a B -hypergraph. Let us assume that Tarjan visits v_1 first. Then, $\text{visit}(v_1)$ is called from $\text{main}(\mathcal{H})$ and the following labels and data-structures are updated: $\text{discovered-order}(v_1) := 1$ (Line 2), $\text{min-order}(v_1) := 1$ (Line 3), $\text{successors}(v_1) := \{v_1\}$ (Line 4), $\text{stack} := \{v_1\}$ (Line 5) and $\mathcal{E}_{\text{check}} := \{e_1, e_7\}$ (Line 6).

The order in which the edges in $\mathcal{E}_{\text{check}}$ are explored is not fixed. Suppose that the first hyperedge considered is e_7 , where $T(e_7) = \{v_1, v_5\}$. Because $v_5 \notin \text{successors}(v_1)$, the

head node of e_7 , v_4 , cannot be visited. The hyperedge e_1 is also in $\mathcal{E}_{check}$ and $T(e_1) \in \text{successors}(v_1)$. Then v_2 , the head node of e_1 , can be visited and $\text{visit}(v_2)$ is called (Line 15) and the following labels and data-structures are updated: $\text{discovered-order}(v_2) := 2$ (Line 2), $\text{min-order}(v_2) := 2$ (Line 3), $\text{successors}(v_2) := \{v_2\}$ (Line 4), $\text{stack} := \{v_2, v_1\}$ (Line 5) and $\mathcal{E}_{check} := \{e_2, e_4, e_7\}$ (Line 6).

$T(e_7) \notin \text{successors}(v_2)$ but $T(e_2) \in \text{successors}(v_2)$ and $T(e_4) \in \text{successors}(v_2)$. Let us assume that e_2 is considered before e_4 . Then v_3 , the head node of e_2 , can be visited and $\text{visit}(v_3)$ is called (Line 15) and the following labels and data-structures are updated: $\text{discovered-order}(v_3) := 3$ (Line 2), $\text{min-order}(v_3) := 3$ (Line 3), $\text{successors}(v_3) := \{v_3\}$ (Line 4), $\text{stack} := \{v_3, v_2, v_1\}$ (Line 5) and $\mathcal{E}_{check} := \{e_3, e_7\}$ (Line 6).

$T(e_7) \notin \text{successors}(v_3)$ but $T(e_3) \in \text{successors}(v_3)$. Then v_2 , the head node of e_3 , can be visited. However, v_2 has been visited before and also $v_2 \in \text{stack}$. Therefore, $\text{min-order}(v_3) := 2$ (Line 19) and $\text{successors}(v_3)$ is extended with $\text{successors}(v_2)$: $\text{successors}(v_3) := \{v_2, v_3\}$ (Line 22).

From v_3 , no other node in the hypergraph can be visited and the node is labelled as explored: $\text{explored-order}(v_3) := 4$. Because $\text{discovered-order}(v_3) \neq \text{min-order}(v_3)$, v_3 is not the root-node of a strongly-connected component and $\text{visit}(v_3)$ backtracks to $\text{visit}(v_2)$. As a result, the value in $\text{min-order}(v_2)$ is recomputed (Line 16) but it does not change because $\text{min-order}(v_2) = \text{min-order}(v_3)$. $T(e_4) \in \text{successors}(v_2)$ and v_1 , the head node of e_4 , can be visited. However, v_1 has been visited before and also $v_1 \in \text{stack}$. Therefore, $\text{min-order}(v_2) := 1$ (Line 19) and $\text{successors}(v_2)$ is extended with $\text{successors}(v_1)$: $\text{successors}(v_2) := \{v_1, v_2, v_3\}$ (Line 22).

From v_2 , no other node in the hypergraph can be visited and the node is labelled as explored: $\text{explored-order}(v_2) := 5$. Because $\text{discovered-order}(v_2) \neq \text{min-order}(v_2)$, v_2 is not the root-node of a strongly-connected component and $\text{visit}(v_2)$ backtracks to $\text{visit}(v_1)$. As a result, the value in $\text{min-order}(v_1)$ is recomputed (Line 16) but it does not change because $\text{min-order}(v_1) = \text{min-order}(v_2)$. Moreover, $\text{successors}(v_1)$ is extended with $\text{successors}(v_2)$: $\text{successors}(v_1) := \{v_1, v_2, v_3\}$ (Line 22).

The hyperedge e_7 is reconsidered again but $T(e_7) \notin \text{successors}(v_1)$. Then, no more nodes can be visited from v_1 and the node is labelled as explored: $\text{explored-order}(v_1) := 6$. v_1 is the root-node of a strongly-connected component because $\text{discovered-order}(v_1) = \text{min-order}(v_1)$. The nodes in the stack, $\{v_1, v_2, v_3\}$, are identified as a strongly-connected component and the stack becomes empty: $\text{stack} = \{\}$. Moreover, the list of successors of the three nodes become the same: $\text{successors}(v_1) := \text{successors}(v_2) := \text{successors}(v_3) := \{v_1, v_2, v_3\}$ (Line 38). The execution of $\text{visit}(v_1)$ ends and the execution of $\text{main}(\mathcal{H})$ continues.

Let us assume that v_4 is the next node to be visited. Then, $\text{visit}(v_4)$ is called from $\text{main}(\mathcal{H})$ and the following labels and data-structures are updated: $\text{discovered-order}(v_4) := 7$ (Line 2), $\text{min-order}(v_4) := 7$ (Line 3), $\text{successors}(v_4) := \{v_4\}$ (Line 4), $\text{stack} := \{v_4\}$ (Line 5) and $\mathcal{E}_{\text{check}} := \{e_5, e_7\}$ (Line 6).

Suppose that the first hyperedge considered in $\mathcal{E}_{\text{check}}$ is e_7 . Because $T(e_7) \notin \text{successors}(v_4)$, the head node of e_7 , v_4 , cannot be visited. The hyperedge e_5 is also in $\mathcal{E}_{\text{check}}$ and $T(e_5) \in \text{successors}(v_4)$. Then v_5 , the head node of e_5 , can be visited and $\text{visit}(v_5)$ is called (Line 15) and the following labels and data-structures are updated: $\text{discovered-order}(v_5) := 8$ (Line 2), $\text{min-order}(v_5) := 8$ (Line 3), $\text{successors}(v_5) := \{v_5\}$ (Line 4), $\text{stack} := \{v_5, v_4\}$ (Line 5) and $\mathcal{E}_{\text{check}} := \{e_6, e_7\}$ (Line 6).

$T(e_7) \notin \text{successors}(v_5)$ but $T(e_6) \in \text{successors}(v_5)$. Then v_3 , the head node of e_6 , can be visited. However, v_3 has been visited before but $v_3 \notin \text{stack}$. Therefore, $\text{min-order}(v_5)$ is not updated but $\text{successors}(v_5)$ is extended with $\text{successors}(v_3)$: $\text{successors}(v_5) := \{v_1, v_2, v_3, v_5\}$ (Line 22).

Extending $\text{successors}(v_5)$ produces that $T(e_7) \in \text{successors}(v_5)$. Then v_4 , the head node of e_7 , can be visited. However, v_4 has been visited before and also $v_4 \in \text{stack}$. Therefore, $\text{min-order}(v_5) := 7$ (Line 19) and $\text{successors}(v_5)$ is extended with $\text{successors}(v_4)$: $\text{successors}(v_5) := \{v_1, v_2, v_3, v_4, v_5\}$ (Line 22).

From v_5 , no other node in the hypergraph can be visited and the node is labelled as explored: $\text{explored-order}(v_5) := 9$. Because $\text{discovered-order}(v_5) \neq \text{min-order}(v_5)$, v_5 is not the root-node of a strongly-connected component and $\text{visit}(v_5)$ backtracks to $\text{visit}(v_4)$. As a result, the value in $\text{min-order}(v_4)$ is recomputed (Line 16) but it does not change because $\text{min-order}(v_4) = \text{min-order}(v_5)$. Moreover, $\text{successors}(v_4)$ is extended with $\text{successors}(v_5)$: $\text{successors}(v_4) := \{v_1, v_2, v_3, v_4, v_5\}$ (Line 22).

The hyperedge e_7 is reconsidered again but $H(e_7) \in \text{successors}(v_4)$ and by the if-condition in Line 12, e_7 is discarded. Then, no more nodes can be visited from v_4 and the node is labelled as explored: $\text{explored-order}(v_4) := 10$. v_4 is the root-node of a strongly-connected component because $\text{discovered-order}(v_4) = \text{min-order}(v_4)$. The nodes in the stack, $\{v_4, v_5\}$, are identified as a strongly-connected component and the stack becomes empty: $\text{stack} = \{\}$. Moreover, the list of successors of the three nodes become the same: $\text{successors}(v_4) := \text{successors}(v_5) := \{v_1, v_2, v_3, v_4, v_5\}$ (Line 38). The execution of $\text{visit}(v_4)$ ends and the execution of $\text{main}(\mathcal{H})$ also finishes because all nodes in the hypergraph has been explored.

Recall that when the first strongly-connected component was computed, $\{v_1, v_2, v_3\}$, the list of successors of the nodes v_1 , v_2 and v_3 becomes the same (Line 38). This is relevant for traversing the entire hypergraph and for computing its strongly-connected components. Notice that without Line 38, it might not be possible to reach v_1 from v_4 and the algorithm

might not identify that v_4 and v_5 are strongly-connected. In this example, v_5 can reach v_1 and later v_4 because $\text{successors}(v_3) = \{v_1, v_2, v_3\}$.

As a result of the execution of the function $\text{main}(\mathcal{H})$, it holds that $\text{SCCs}(\mathcal{H}) = \{\{v_1, v_2, v_3\}, \{v_4, v_5\}\}$. Let $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}}, \mathcal{L}_{\mathcal{G}})$ be a DFS-graph of $\mathcal{H}$, i.e., $\mathcal{G} = \text{main}(\mathcal{H})$. It holds that $\mathcal{V}_{\mathcal{G}} = \mathcal{V}_{\mathcal{H}}$ and $\mathcal{E}_{\mathcal{G}} = \{e_1, e_2, e_3, e_4, e_5, e_6\} \cup \{e'_7\}$, where $e'_7 = (v_5, v_4)$ is the approximating edge of e_7 . The values for the labelling function $\mathcal{L}_{\mathcal{G}}$ are the following: $\mathcal{L}_{\mathcal{G}}(v_1) = (1, 6)$, $\mathcal{L}_{\mathcal{G}}(v_2) = (2, 5)$, $\mathcal{L}_{\mathcal{G}}(v_3) = (3, 4)$, $\mathcal{L}_{\mathcal{G}}(v_4) = (7, 10)$ and $\mathcal{L}_{\mathcal{G}}(v_5) = (8, 9)$. Recall that $\mathcal{L}_{\mathcal{G}}^1$ corresponds to the function discovered-order and $\mathcal{L}_{\mathcal{G}}^2$ to explored-order, respectively. It can readily be checked that $\text{SCCs}(\mathcal{G}) = \text{SCCs}(\mathcal{H})$. For an illustration of $\mathcal{H}$ and $\mathcal{G}$, see Figure 3.8.

3.3.3 Condensing the Graph Fragment of a Directed Hypergraph

The condensed directed hypergraph (Definition 2.42) is the result of collapsing the strongly-connected components of a directed hypergraph into single nodes and turning hyperedges between nodes into hyperedges between the newly formed nodes.

Similarly, it is also possible to compute the *partially condensed* directed hypergraph of a directed hypergraph. The idea is to identify and collapse the strongly-connected components of the graph fragment (Definition 2.31) of a directed hypergraph and re-calculate the hyperedges of the original hypergraph to consider the newly formed nodes.

Definition 3.27 (Partially Condensed Directed Hypergraph). Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and let $\mathcal{G}(\mathcal{H}) = (\mathcal{V}, \mathcal{E}')$ be the graph fragment of $\mathcal{H}$. Let $\simeq_{\mathcal{G}}$ be the strong-connectivity relation over $\mathcal{V}$ in $\mathcal{G}(\mathcal{H})$. The partially condensed directed hypergraph of $\mathcal{H}$ is defined as the quotient hypergraph of $\mathcal{H}$ under $\simeq_{\mathcal{G}}$, written $\mathcal{H} / \simeq_{\mathcal{G}}$, is the hypergraph $\mathcal{H} / \simeq_{\mathcal{G}} = (\mathcal{V} / \simeq_{\mathcal{G}}, \mathcal{E}_{\simeq_{\mathcal{G}}})$, where $\mathcal{V} / \simeq_{\mathcal{G}} = \{[x]_{\simeq_{\mathcal{G}}} \mid x \in \mathcal{V}\}$ is the set of all strongly-connected components of $\mathcal{G}(\mathcal{H})$. $\dashv$

The following proposition establishes the relation between B -connectivity in a directed hypergraph $\mathcal{H}$ and B -connectivity in the partially condensed directed hypergraph $\mathcal{H} / \simeq_{\mathcal{G}}$. This is done by showing that a B -connected component in $\mathcal{H}$ is equivalent to a B -connected component in $\mathcal{H} / \simeq_{\mathcal{G}}$ after the nodes of the B -connected component in $\mathcal{H} / \simeq_{\mathcal{G}}$ are expanded to retrieve the original nodes that were defined in $\mathcal{H}$.

Proposition 3.28. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and let $\mathcal{V}' \subseteq \mathcal{V}$ be a set of nodes. Let $\mathcal{G}(\mathcal{H}) = (\mathcal{V}, \mathcal{E}')$ be the graph fragment of $\mathcal{H}$ and let $\simeq_{\mathcal{G}}$ be the strong-connectivity relation over $\mathcal{V}$ in $\mathcal{G}(\mathcal{H})$. Let $\mathcal{H} / \simeq_{\mathcal{G}} = (\mathcal{V} / \simeq_{\mathcal{G}}, \mathcal{E}_{\simeq_{\mathcal{G}}})$ be the partially condensed directed hypergraph of $\mathcal{H}$ under $\simeq_{\mathcal{G}}$.

Then: $\geq_{B(\mathcal{H})}(\mathcal{V}') = \bigcup \geq_{\mathcal{H} / \simeq_{\mathcal{G}}}([\mathcal{V}']_{\simeq_{\mathcal{G}}})$, where $[\mathcal{V}']_{\simeq_{\mathcal{G}}} = \{[v]_{\simeq_{\mathcal{G}}} \mid v \in \mathcal{V}'\}$. $\dashv$

```

function compute-pchypergraph ( $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ )
1:  $\mathcal{V}_{pc} := \{\}$ 
2:  $\mathcal{E}_{pc} := \{\}$ 
3:  $T_{scc} := \{\}$ 
   // compute the SCCs of  $\mathcal{G}(\mathcal{H})$  using Tarjan (Section 2.3.2, Figure 2.8)
4:  $\mathcal{S}_{scc} := \text{main}(\mathcal{G}(\mathcal{H}))$ 
   // identify each SCC by one of its nodes
5: for every  $\mathcal{S} \in \mathcal{S}_{scc}$  do
6:    $v := \text{select}(\mathcal{S})$ 
7:    $\mathcal{V}_{pc} := \mathcal{V}_{pc} \cup \{v\}$ 
   // create a look-up table that relates each node of a SCC with its representative node
8:   for every  $w \in \mathcal{S}$  do:  $\text{add}(T_{scc}, (w, v))$ 
9: end-for
   // re-compute the original hyperedges based on the nodes in  $\mathcal{V}_{pc}$ 
10: for every  $e \in \mathcal{E}$  do
11:    $T(e') := \{\}$ 
12:    $H(e') := \{\}$ 
13:   for every  $s \in T(e)$  do:  $T(e') := T(e') \cup T_{scc}(s)$ 
14:   for every  $s \in H(e)$  do:  $H(e') := H(e') \cup T_{scc}(s)$ 
15:   if  $(T(e') \neq H(e'))$  then
16:      $\mathcal{E}_{pc} := \mathcal{E}_{pc} \cup e'$ 
17:   end-if
18: end-for
19: return  $\mathcal{H}_{pc} = (\mathcal{V}_{pc}, \mathcal{E}_{pc})$ 

```

Fig. 3.9 Algorithm that computes the partially condensed directed hypergraph.

Proof. “ $\subseteq$ ”: We show the contrapositive. Suppose $[v]_{\simeq_{\mathcal{G}}} \notin \bigcup_{\mathcal{H}/\simeq_{\mathcal{G}}} ([\mathcal{V}']_{\simeq_{\mathcal{G}}})$. Then there is no B -hyperpath in $\mathcal{H}/\simeq_{\mathcal{G}}$ from $[\mathcal{V}']_{\simeq_{\mathcal{G}}}$ to $[v]_{\simeq_{\mathcal{G}}}$. This implies that there is no B -hyperpath in $\mathcal{H}$ from $\mathcal{V}'$ to v , i.e., $v \notin \bigcup_{\mathcal{H}} (\mathcal{V}')$.

“ $\supseteq$ ”: We show the contrapositive. Suppose $v \notin \bigcup_{\mathcal{H}} (\mathcal{V}')$. Then there is no B -hyperpath in $\mathcal{H}$ from $\mathcal{V}'$ to v . Then there is no B -hyperpath in $\mathcal{H}/\simeq_{\mathcal{G}}$ from $[\mathcal{V}']_{\simeq_{\mathcal{G}}}$ to $[v]_{\simeq_{\mathcal{G}}}$, i.e., $[v]_{\simeq_{\mathcal{G}}} \notin \bigcup_{\mathcal{H}/\simeq_{\mathcal{G}}} ([\mathcal{V}']_{\simeq_{\mathcal{G}}})$. $\square$

Expanding a collapsed strongly-connected components can be done using set-union because each collapsed strongly-connected component represents a set of nodes. This is why in Proposition 3.28, we refer to the B -connected component of $[\mathcal{V}']_{\simeq_{\mathcal{G}}}$ after it has been expanded using the expression: $\bigcup_{\mathcal{H}/\simeq_{\mathcal{G}}} ([\mathcal{V}']_{\simeq_{\mathcal{G}}})$. Note that $\mathcal{V}' \subseteq \bigcup_{\mathcal{H}/\simeq_{\mathcal{G}}} ([\mathcal{V}']_{\simeq_{\mathcal{G}}})$, where $[\mathcal{V}']_{\simeq_{\mathcal{G}}} = \{[v]_{\simeq_{\mathcal{G}}} \mid v \in \mathcal{V}'\}$.

The algorithm depicted in Figure 3.9 computes a partially condensed directed hypergraph $\mathcal{H}_{pc} = (\mathcal{V}_{pc}, \mathcal{E}_{pc})$. In Lines 1–2, the data-structures that store the new nodes ($\mathcal{V}_{pc}$) and hyperedges ($\mathcal{E}_{pc}$) of the partially condensed directed hypergraph are initialized. The algorithm also uses a *look-up table*, T_{scc} , that it is also initialized in Line 3. T_{scc} relates for each node in $\mathcal{V}_{pc}$ the set of nodes that has been collapsed in the original hypergraph. Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a directed hypergraph and let $v, w \in \mathcal{V}$ be two nodes such that v and w are in the same strongly-connected component and $v \in \mathcal{V}_{pc}$ is also a node in the partially condensed directed hypergraph $\mathcal{H}_{pc}$. Then, $T_{scc}(w)$ returns v .

In Line 4, the strongly-connected components of the graph fragment of the original hypergraph, $\mathcal{G}(\mathcal{H})$ are computed using Tarjan (Section 2.3.2, Figure 2.8) and stored in $\mathcal{S}_{scc}$. In Lines 5–9, a node for each strongly-connected component is randomly selected by the function $\text{select}(\cdot)$ (Line 6) and this node becomes a node in the partially condensed directed hypergraph (Line 7). The look-up table $T_{scc}(w)$ is also filled with pair of nodes (Line 8), where one node of the pair is a node of a strongly-connected component and the other node of the pair is the representative node of the strongly-connected component that was selected by the function $\text{select}(\cdot)$.

The last task of the algorithm is to re-calculate the hyperedges of the original hypergraph $\mathcal{H}$ to consider the newly formed nodes in $\mathcal{V}_{pc}$. This is done using a for-loop (Lines 10–18) that iterates over all the hyperedges of $\mathcal{H}$. For each hyperedge $e \in \mathcal{E}$, the algorithm substitutes each node in the tail (head, respectively) by the representative node of the strongly-connected component where this node belongs (Lines 13 & 14). If the tail and head of the new hyperedge are different then the hyperedge is added into the set of hyperedges of the partially condensed directed hypergraph $\mathcal{H}_{pc}$ (Line 16). The result of the execution of the function $\text{compute-pchypergraph}(\cdot)$ is the partially condensed directed hypergraph of the input hypergraph.

It can readily be seen that the algorithm is correct and its time complexity is $O(|\mathcal{H}|)$. The for-loop in Lines 5–9 includes a nested for-loop (Line 8). Assuming that Lines 6–7 are executed in constant time, the combination of both for-loops iterate over the nodes of the original hypergraph which represent $|\mathcal{V}|$ iterations. The for-loop in Lines 10–18 includes two other for-loops (Lines 13 & 14) that iterates over the nodes of the tail and the head of each hyperedge of the original hypergraph. This means $\sum_{e \in \mathcal{E}} (|T(e)| + |H(e)|)$ iterations. Therefore, the time complexity of the function $\text{compute-pchypergraph}(\cdot)$ for a given hypergraph $\mathcal{H}$ is $O(|\mathcal{H}|)$.

By computing the partially condensed directed hypergraph of a hypergraph, we might significantly reduce the amount of nodes and hyperedges of the original hypergraph. This approach can improve the performances of the algorithms that we have discussed in this chapter.

Moreover, there might be hypergraphs that are in fact, directed graphs. Applying Tarjan for directed graphs first might result on computing all the strongly-connected components of these graphs in linear time with respect to their size.

3.4 Evaluation

We have implemented the Naïve and Tarjan algorithms (cf. Figures 3.5 and Figures 3.1& 3.2, respectively) for directed hypergraphs using Java. To evaluate them, we have also developed two generators of artificial hypergraphs and two generators of random hypergraphs. An artificial hypergraph GSC is a B -hypergraph with one or more large strongly- B -connected components. The nodes of a GSC hypergraph are partitioned in sets of nodes. Each of these sets are connected by hyperedges to only one of the other sets and it is not possible that two sets are connected to the same sets of nodes. We extended a GSC generator to produce artificial hypergraphs that follows a “*bow-tie*” configuration [9]. A bow-tie hypergraph extends a GSC hypergraph with two group of nodes: an in-component and an out-component. The *in-component* is a set of nodes connected by directed edges to the nodes of the GSC hypergraph. The nodes of the GSC hypergraph are connected by directed edges to the nodes of the *out-component*.

The generators of random hypergraphs are inspired by Erdős-Rényi [15, 16, 19] and Newman-Watts-Strogatz[44, 30] random graph models. In a random graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, every pair of nodes $x, y \in \mathcal{V}$ is connected by a probability p . The model by Erdős-Rényi is also known as $\mathcal{G}(|\mathcal{V}|, L)$ model, where $|\mathcal{V}|$ nodes are connected by L randomly selected edges. For computing random graphs is more intuitive the $\mathcal{G}(|\mathcal{V}|, p)$ model [19], where each pair of nodes in $\mathcal{V}$ are connected with probability p by an edge. One problem of these random graphs models is that they do not have the small-World property, i.e., the average distance between two nodes depends logarithmically on $|\mathcal{V}|$. The random graph model by Newman-Watts-Strogatz[44, 30] produces graphs with the small-World property. From a ring of connected nodes, a new edge connecting two randomly chosen nodes of the ring is added with probability p . Adding this new edge might imply that an edge from the original ring is deleted. This edge connects one of the two nodes randomly chosen.

To extend the $\mathcal{G}(|\mathcal{V}|, p)$ model for B -hypergraphs, we created $\mathcal{G}(|\mathcal{V}|, t, p)$ model, where t represents the maximum number of nodes in a tail. Each possible tail of nodes of size t to 1 is connected to a single node in $\mathcal{V}$ with probability p . We also modify the model by Newman-Watts-Strogatz as follows. Like the original model, we also create a ring of nodes connected by directed edges. Instead of adding directed edges to connect randomly chosen nodes of the ring, we added B -hyperedges. Each hyperedge has a tail of size t or less. The

nodes of the tail are randomly chosen from the nodes of the ring. We do the same with the head node of the hyperedge. An original edge of the ring that connects a tail node or a head node might be also deleted.

Table 3.1 contains, for each ontology, the number of nodes, the number of hyperedges, the number of strongly- B -connected components and the largest number of nodes that a tail of a hyperedge might have.

Table 3.2 lists the time needed for each algorithm to compute the strongly- B -connected components of the hypergraphs. The time values are the average of 10 executions. Each algorithm has two implementations. One implementation coincides with the original algorithm and the second implementation slightly differs from the original. The goal is to show the impact of the implementation details on the performances of the algorithms. The tail counter is a counter associated to each hyperedge of the hypergraph. When a tail node is visited, the counter is incremented. This is how the BFS algorithm for directed hypergraph was defined. Instead of using a tail counter, the algorithm for traversing the hypergraph can iterate over all tails of the hypergraph until the algorithm finds a tail such that all its nodes have been visited. This means that all the head nodes with this tail becomes candidates to being visited. An adaptive approach change the strategy of iterating over tails depending on the number of nodes that have been visited so far.

As Table 3.2 shows, none of the two implementations of the Naïve algorithm consistently outperform the other implementation. In the case of Tarjan, the implementation that adapts the execution of the algorithm seems to perform better than the other implementations of the Tarjan and Naïve algorithms. However, a broad evaluation that includes also real data might be needed to complete this analysis.

Table 3.1 Selected hypergraphs for the evaluation Naïve and Tarjan algorithms.

$\mathcal{H}$	Properties of $\mathcal{H}$			
	#nodes	#hyperedges	#SCCs	max. tail
GSC-true	10 000	37 980	1	10
GSC-false	10 000	27 990	1 000	10
BT-true	1 300	401 790	801	5
BT-false	1 300	401 295	900	5
ER-0.01p	1 000	4993 197	1	2
ER-0.05p	1 000	24978 067	1	2
ER-0.3p	50	3767563	1	4
WS-1.0p	10 000	160 000	10 000	8
NWS-1.0p	10 000	320 000	1	8

Table 3.2 Evaluation of Naïve and Tarjan algorithms.

$\mathcal{H}$	Naïve algorithm		Tarjan	
	tail-iterator	tail-counter	tail-iterator	adaptive
GSC-true	1 136.6 s	12.7 s	1.1 s	1.3 s
GSC-false	34.4 s	6.0 s	0.9 s	1.3 s
BT-true	18.7 s	5.6 s	2.6 s	2.6 s
BT-false	5.1 s	4.4 s	2.5 s	2.7 s
ER-0.01p	49.7 s	451.0 s	345.0 s	7.5 s
ER-0.05p	74.2 s	2026.2 s	1 703.5 s	48.8 s
ER-0.3p	4.9 s	44.5 s	7.9 s	5.0 s
WS-1.0p	60.3 s	1.1 s	1.0 s	1.1 s
NWS-1.0p	117.7 s	192.4 s	1 412.5 s	11.4 s

3.5 Summary

In this chapter, we introduced the notion of approximating graph which is a directed graph that preserves node-to-node connectivity of a directed hypergraph. Based on this notion, we define and implement a new depth-first search algorithm for traversing directed hypergraphs, and furthermore, we explain how it can also be used to compute approximating graphs. In addition, we show that this algorithm can also identify the strongly-connected components of a directed hypergraph similar to the algorithm by Tarjan [39] for directed graphs.

To compare the complexity and performances of the new depth-first search algorithm, we design and implement a Naïve algorithm that also computes the strongly-connected components of a directed hypergraph. Despite the fact that the time complexity of Tarjan for directed hypergraphs is worst than the other algorithm discussed in this chapter, the evaluation result does not confirm it. With the right optimizations, Tarjan can be even faster.

Chapter 4

Preliminaries II: Syntactic Locality and Atomic Decomposition

In this thesis we consider ontologies formulated in the expressive description logic *SROIQ* [21] which underlies the Web Ontology Language OWL 2.¹ For the evaluation of our algorithms, we consider prominent biomedical ontologies formulated in the light-weight description logic *EL*⁺⁺ [6], which is at the core of the OWL 2 EL profile.² We refer to [7] for a detailed introduction to description logics.

Chapter 4 is organized as follows. Section 4.1 provides a brief overview on syntactic locality-based modules. Basic notions on atomic decomposition are presented in Section 4.2. We conclude the chapter with an overview on state-of-the-art implementation for module extraction and atomic decomposition in Section 4.3.

4.1 Syntactic Locality-based Modules

A module for a signature is a subset of an ontology that preserves all entailments formulated using symbols from the signature only.

Definition 4.1 (Module). *Let $\mathcal{M}$ be a subset of an ontology $\mathcal{O}$, and Σ a signature. $\mathcal{M}$ is a module of $\mathcal{O}$ wrt. Σ if for all entailments α with $\text{sig}(\alpha) \subseteq \Sigma$: $\mathcal{M} \models \alpha$ iff $\mathcal{O} \models \alpha$, where $\text{sig}(\alpha)$ represents the signature of axiom α . $\dashv$*

The notion of a module can also be defined in terms of conservative extensions: $\mathcal{M}$ is a module of $\mathcal{O}$ wrt. Σ if $\mathcal{O}$ is a deductive conservative extension of $\mathcal{M}$ wrt. Σ . Clearly, several subsets of $\mathcal{O}$ can be modules wrt. Σ .

¹<http://www.w3.org/TR/owl2-overview/>

²http://www.w3.org/TR/owl2-profiles/OWL_2_EL

Computing a minimal module is expensive for expressive fragments of OWL2 [12]. The notion of *locality* has been suggested as a practical approach to computing approximations of minimal modules [12]. Locality refers to axioms, i.e., axioms are either local or non-local, and it depends on a signature. A module wrt. Σ of an ontology consists of the axioms that are non-local wrt. Σ . The locality notion comes in two flavours: *semantic* and *syntactic* locality. Intuitively, an axiom is semantically local wrt. Σ if it does not say anything about the terms in Σ . In other words, given a signature Σ , the axiom can always be satisfied independently of the interpretation of the Σ -terms. Typically only two versions of semantic locality are considered: $\emptyset$ -locality and Δ -locality. Here $\emptyset$ stands for the empty interpretation of each concept and role name that are not in Σ , i.e., the concept name is false everywhere and the role contains no successors. On the other hand, Δ stands for the entire domain of an interpretation, i.e., all non- Σ concept names are true everywhere and each non- Σ role connects every point with every point (the role name is interpreted as the Cartesian product $\Delta \times \Delta$). The notion of $\emptyset$ -locality is defined as follows (Δ -locality is defined similarly) [12].

Definition 4.2 (Semantic $\emptyset$ -locality of an axiom). *An axiom α is $\emptyset$ -local wrt. a signature Σ if for every interpretation $\mathcal{I}$, there is an interpretation $\mathcal{J}$ such that the following holds:*

- (i) $\mathcal{I}$ and $\mathcal{J}$ coincide on the interpretation of concept names and roles in Σ ;
- (ii) $\mathcal{J}$ interprets all concept names and role names that occur in α but not in Σ as empty set; and
- (iii) $\mathcal{J}$ is a model of α . ⊥

The following proposition shows the relationship between semantic $\emptyset$ -locality and conservative extensions.

Proposition 4.3 ([12]). *Let $\mathcal{M}$ be a subset of an ontology $\mathcal{O}$. Let Σ be a signature. If all axioms in $\mathcal{O} \setminus \mathcal{M}$ are semantic $\emptyset$ -local wrt. $\Sigma \cup \text{sig}(\mathcal{M})$, then $\mathcal{O}$ is a deductive conservative extension of $\mathcal{M}$ wrt. Σ . The converse does not hold.* ⊥

In other words, “extending $\mathcal{M}$ to $\mathcal{O}$ ” or “importing $\mathcal{M}$ into $\mathcal{O} \setminus \mathcal{M}$ ” does not change the meaning of the symbols in Σ . In the literature the term ‘safety’ is used, i.e., $\mathcal{O} \setminus \mathcal{M}$ is safe for signature Σ . We could also say that $\mathcal{O} \setminus \mathcal{M}$ can safely be extended with $\mathcal{M}$, or $\mathcal{O} \setminus \mathcal{M}$ can safely import $\mathcal{M}$, where ‘safe’ means not changing the meaning of terms in the signature Σ . Formally, $\mathcal{O} \setminus \mathcal{M}$ is safe for Σ if $\mathcal{O} \setminus \mathcal{M}$ is logically equivalent wrt. Σ to the empty ontology (i.e. $\mathcal{O} \setminus \mathcal{M}$ does not say anything about symbols in Σ).

Checking whether an axiom is semantically local can be computationally expensive as it requires reasoning. Recall that reasoning with OWL2 is intractable in general (it is

2-NExpTime-complete in the worst case).³ The notion of *syntactic locality* was introduced to allow for efficient computation [12]. Checking for syntactic locality involves checking that an axiom is of a certain form (syntax), no reasoning is needed. On the downside, syntactic locality is merely an approximation of semantic locality: all syntactically local axioms are also semantically local, but not necessarily *vice versa*. In general, as less axioms are local syntactically than semantically, fewer axioms are not contained in the corresponding module. That is, modules based on syntactic locality contain the modules based on semantic locality, and possibly more axioms.

The syntactic locality-based notions that are typically considered are $\perp$ -locality, $\top$ -locality and $\perp\top^*$ -locality. Recall that $\perp$ and $\top$ are atomic formulas representing falsehood and truth, respectively. Intuitively, a $\perp$ -locality-based module of an ontology $\mathcal{O}$ wrt. a signature Σ contains a set of axioms from $\mathcal{O}$ which represent the semantic relationships between the terms in Σ together with the terms that are *less specific* according to the concept hierarchy of $\mathcal{O}$ (these terms are located above the Σ -terms in the concept hierarchy). Analogously, a $\top$ -locality-based module of $\mathcal{O}$ wrt. Σ contains a set of axioms which represent the relations between the terms in Σ together with the terms that are *more specific* according to the concept hierarchy of $\mathcal{O}$. $\perp\top^*$ -locality modules can be computed by nesting the extraction of $\perp$ - and $\top$ -modules to produce modules of smaller size [36].

The notion of syntactic $\perp$ -locality for *SHOIQ* axioms is defined as follows; the notion of syntactic $\top$ -locality can be defined similarly [12]. Let N_C and N_R be countable and mutually disjoint sets of concept names and role names, respectively. A *signature* is a finite subset of N_C and N_R .

Definition 4.4 (Syntactic $\perp$ -locality of an axiom). *Let Σ be a signature. The concepts $C_\Sigma^\perp$ that are $\perp$ -local wrt. Σ and the concepts $C_\Sigma^\top$ that are $\top$ -local wrt. Σ are defined simultaneously by the following grammar:*

$$\begin{aligned} C_\Sigma^\perp &::= \perp \mid A \mid \neg C_\Sigma^\top \mid C \sqcap C_\Sigma^\perp \mid C_\Sigma^\perp \sqcap C \mid C_\Sigma^\perp \sqcup C_\Sigma^\perp \mid \exists r.C \mid \exists R.C_\Sigma^\perp \mid \\ &\quad (\geq n \, r.C) \mid (\geq n \, R.C_\Sigma^\perp) \\ C_\Sigma^\top &::= \top \mid \neg C_\Sigma^\perp \mid C_\Sigma^\top \sqcap C_\Sigma^\top \mid C \sqcup C_\Sigma^\top \mid C_\Sigma^\top \sqcup C \mid \forall r.C \mid \forall R.C_\Sigma^\top \mid \\ &\quad < n \, r.C \mid < n \, R.C_\Sigma^\top \mid (\geq 0 \, R.C) \end{aligned}$$

where $A \in N_C \setminus \Sigma$ and $r \in N_R \setminus \Sigma$. The set of concepts $C_\Sigma^\perp$ ($C_\Sigma^\top$) is denoted with $\mathcal{L}_\Sigma^\perp$ ($\mathcal{L}_\Sigma^\top$).

³But even reasoning with tractable logics as the light-weight description logic $\mathcal{EL}$ can be seen as too difficult in some cases, depending on the application requirements.

```

function  $\text{Mod}^x(\Sigma, \mathcal{O})$  returns  $x$ -local module of  $\mathcal{O}$  wrt.  $\Sigma$ 
1:  $m := 0$ 
2:  $\mathcal{M}_m := \emptyset$ 
3: do
4:    $m := m + 1$ 
5:    $\mathcal{M}_m := \{\alpha \in \mathcal{O} \mid \alpha \text{ is not } x\text{-local wrt. } \Sigma \cup \text{sig}(\mathcal{M}_{m-1})\}$ 
6: until  $\mathcal{M}_m = \mathcal{M}_{m-1}$ 
7: return  $\mathcal{M}_m$ 

```

Fig. 4.1 Module Extraction Algorithm [11]

An axiom α is syntactically $\perp$ -local wrt. Σ if α is of the form $C_\Sigma^\perp \sqsubseteq C$, $C \sqsubseteq C_\Sigma^\top$, $C_\Sigma^\perp \equiv C_\Sigma^\perp$, $C_\Sigma^\top \equiv C_\Sigma^\top$, $r \sqsubseteq R$ or $\text{trans}(r)$, where C is an arbitrary concept, $r \in \mathbf{N}_R \setminus \Sigma$ and R is a role name or an inverse role. $\dashv$

Note that, for any signature Σ , the sets $\mathcal{L}_\Sigma^\perp$ and $\mathcal{L}_\Sigma^\top$ are disjoint, i.e., no concept is both, $\perp$ -local wrt. Σ and $\perp$ -local wrt. Σ .

4.1.1 Extracting modules based on locality

Figure 4.1 presents the module extraction algorithm from [11], where x stands for any of the semantic and syntactic locality notions, i.e. $x \in \{\emptyset, \Delta, \perp, \top\}$.

The algorithm takes an ontology $\mathcal{O}$ and a signature Σ as input and returns a subset $\mathcal{M}$ of $\mathcal{O}$. The computed set $\mathcal{M}$ consists of axioms that are not x -local wrt. $\Sigma \cup \text{sig}(\mathcal{M})$, or equivalently, $\mathcal{O} \setminus \mathcal{M}$ consists of axioms that are local wrt. $\Sigma \cup \text{sig}(\mathcal{M})$. The algorithm produces a finite sequence $\mathcal{M}_0, \dots, \mathcal{M}_n$, $n \geq 0$, of subsets of the ontology $\mathcal{O}$, where $\mathcal{M}_n$ is the $\perp$ -local module of $\mathcal{O}$ wrt. $\Sigma \cup \text{sig}(\mathcal{M}_n)$ with Σ being the initial signature. This sequence induces a relation on $\mathcal{O}$ representing the successive inclusion of the axioms into the module. This is illustrate in the following example.

Example 4.5. Let $\mathcal{O} = \{\alpha_1, \dots, \alpha_4\}$ and $\Sigma = \{A, r\}$, where

$$\begin{aligned}
\alpha_0 &:= A \sqsubseteq B_1 \sqcap B_2 \sqcap X_3 \\
\alpha_1 &:= \exists r.B_1 \sqsubseteq X_1 \\
\alpha_2 &:= B_2 \sqsubseteq X_2 \\
\alpha_3 &:= X_1 \sqcap X_2 \sqcap X_3 \sqsubseteq Y
\end{aligned}$$

The following picture shows the graph $G_{\text{mod}^\perp(\mathcal{O}, \Sigma)} = (V, E)$ induced by the module extraction algorithm in Figure 4.1 on the input $\mathcal{O}$ and Σ , as well as the sequence $\mathcal{M}_0, \dots, \mathcal{M}_3, \mathcal{M}_4$

```

function  $\text{Mod}^*(\Sigma, \mathcal{O})$  returns  $\perp\top^*$ -local module of  $\mathcal{O}$  wrt.  $\Sigma$ 
1:  $\mathcal{M}^\top := \mathcal{O}$ 
2: do
3:    $\mathcal{M}^\perp := \text{Mod}^\perp(\Sigma, \mathcal{M}^\top)$ 
4:    $\mathcal{M}^\top := \text{Mod}^\top(\Sigma, \mathcal{M}^\perp)$ 
5: until  $\mathcal{M}^\perp = \mathcal{M}^\top$ 
6: return  $\mathcal{M}^\perp$ 

```

Fig. 4.2 Module Extraction Algorithm for $\perp\top^*$ -local modules [36]

produced by the algorithm. Again we indicate with additional tilted arrows the symbols that are provided by the initial signature.

◁

An algorithm that computes $\perp\top^*$ -local modules is depicted in Figure 4.2. The $\perp\top^*$ -local module for a given ontology is computed by nesting the extraction of $\perp$ - and $\top$ -local modules until a fix-point is reached, where the $\perp$ - and $\top$ -module is the same. It has been proved that the number of steps until a fix-point is reached cannot be bigger than the number of axioms of the ontology [36].

The following example illustrates how the algorithm that computes $\perp\top^*$ -local modules works.

Example 4.6. Let $\mathcal{O} = \{ \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5 \}$ be an ontology such that $\alpha_1 = X \sqsubseteq A \sqcap B$, $\alpha_2 = B \sqsubseteq C$, $\alpha_3 = B \sqcap C \sqsubseteq A \sqcap B \sqcap D$, $\alpha_4 = D \sqsubseteq A \sqcup E \sqcup F$ and $\alpha_5 = E \sqsubseteq D$. Let $\Sigma = \{A, B\}$ be the signature of the $\perp\top^*$ -local module that we want to extract. The algorithm depicted in Figure 4.2 computes the following nested modules:

$$\begin{aligned}
\mathcal{M}^\perp &= \{ \alpha_2, \alpha_3, \alpha_4, \alpha_5 \} \\
\mathcal{M}^\top &= \{ \alpha_2, \alpha_3, \alpha_5 \} \\
\mathcal{M}^\perp &= \{ \alpha_2, \alpha_3 \} \\
\mathcal{M}^\top &= \{ \alpha_2, \alpha_3 \}
\end{aligned}$$

The algorithm stops when $\mathcal{M}^\perp = \mathcal{M}^\top$ (Line 5) and returns the module $\{ \alpha_2, \alpha_3 \}$ (Line 6). ◁

4.1.2 Signature Containment for Module Extraction

The state of non-locality of an axiom can also be checked in terms of signature containment [38]. To this end, we introduce the notion of minimal non-locality signature for *SROIQ* axioms.

Definition 4.7 (Minimal non-Locality Signature). *Let $x \in \{\perp, \top\}$ denote a locality notion. A Minimal non- x -Locality Signature for an axiom α is a signature $\Sigma \subseteq \text{sig}(\alpha)$ such that α is not x -local wrt. Σ , and Σ is minimal (wrt. set inclusion) with this property. The set of minimal non- x -locality signatures is denoted by $MLS^x(\alpha)$.* $\dashv$

The notion of minimal non-locality signature turns out to be equivalent to the notion of minimal globalising signatures, which were introduced specifically for computing modules from an atomic decomposition [13].

The following example shows that there can be exponentially many minimal non-locality signatures for an axiom using merely conjunction and disjunction as logical operators.

Example 4.8. *Let $\alpha = (X_{11} \sqcup X_{12} \sqcup \dots \sqcup X_{1m}) \sqcap \dots \sqcap (X_{n1} \sqcup X_{n2} \sqcup \dots \sqcup X_{nm}) \sqsubseteq Y$ be an axiom. The minimal non- $\perp$ -locality signature $MLS(\alpha)$ of α is as follows:*

$$MLS^\perp(\alpha) = \{ \{X_{1i_1}, X_{2i_2}, \dots, X_{ni_n}\} \mid i_1, i_2, \dots, i_n \in \{1, \dots, m\} \}$$

Then: $|MLS^\perp(\alpha)| = m^n$. $\triangleleft$

However, exponentially many minimal non-locality signatures can be avoided if the axiom is normalised.

An ontology $\mathcal{O}$ (that is formulated in the description logic *SROIQ*) is normalised by applying the normalisation rules presented in [34], which are an extension of the normalisation for *EL* ontologies [38]. Axioms of a normalised ontology have one of the following forms, where $A_i \in \mathbf{N}_C \cup \{\top\}$, $B_i \in \mathbf{N}_C \cup \{\perp\}$, $R_i \in \mathbf{N}_R \cup \text{inv}(\mathbf{N}_R)$, $X, Y \in \{\exists R.B, (\geq n R.B), \exists R.Self \mid B \in \mathbf{N}_C, R \in \mathbf{N}_R \cup \text{inv}(\mathbf{N}_R), n \geq 0\}$ and $\ell, m \geq 0$:

$$\begin{array}{ll} \alpha_1 : A_1 \sqcap \dots \sqcap A_\ell \sqsubseteq B_1 \sqcup \dots \sqcup B_m & \alpha_5 : X \sqsubseteq Y \\ \alpha_2 : X \sqsubseteq B_1 \sqcup \dots \sqcup B_m & \alpha_6 : R_1 \sqsubseteq R_2 \\ \alpha_3 : A_1 \sqcap \dots \sqcap A_\ell \sqsubseteq Y & \alpha_7 : Dis(R_1, R_2) \\ \alpha_4 : R_1 \circ \dots \circ R_\ell \sqsubseteq R_{\ell+1} & \end{array}$$

where $\text{inv}(\mathbf{N}_R)$ is the set of inverse roles r^- , for $r \in \mathbf{N}_R$, and $\exists R.Self$ expresses the local reflexivity of R . The normalisation of an ontology $\mathcal{O}$ runs in linear time in the size of $\mathcal{O}$.

According to [34], the normalised ontology preserves Σ -entailments of $\mathcal{O}$.⁴ Notice that the normalisation rules can be applied backwards over normalised axioms to compute the original axioms of the ontology. However, denormalisation requires a careful application of the normalisation rules to ensure that we obtain the original axioms [34].

There are at most two minimal non-locality signatures for a normalised axiom.

Proposition 4.9. *Let α be a normalised axiom. Then: $|MLS^\perp(\alpha)| = 1$ and $|MLS^\top(\alpha)| \leq 2$.* ⊢

We can apply additional normalisation rules to reduce the number of symbols on the left- and right-hand side of normalised axioms [25]. Bounding the number of symbols in an axiom results in bounding the size of the minimal non-locality signatures of the axiom.

We now give simple conditions under which normalised axioms are not syntactic local. Similar non-locality conditions are presented in the notions of $\perp$ - and $\top$ -reachability in [34].

Proposition 4.10 (Non-locality via Signature Containment). *Let α be a normalised axiom. Let Σ be a signature. Then: α is not $\perp$ -local wrt. Σ iff one of the following holds:*

- $\text{sig}(\text{LHS}(\alpha)) \subseteq \Sigma$ if α is of the form $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6$;
- $\text{sig}(\alpha) \subseteq \Sigma$ if α is of the form α_7 ;

Then: α is not $\top$ -local wrt. Σ iff α is of the form α_7 or one of the following holds:

- $\text{sig}(\text{RHS}(\alpha)) \cap \Sigma \neq \emptyset$ if α is of the form $\alpha_3, \alpha_4, \alpha_5, \alpha_6$;
 - $\text{sig}(\text{RHS}(\alpha)) \subseteq \Sigma$ if α is of the form α_1, α_2 .
- ⊢

4.2 Atomic Decomposition

An ontology might contain exponentially many different modules. The *atomic decomposition* of an ontology provides a polynomial size representation of all possible modules of the ontology. This representation can be modelled as a graph where the nodes are atoms and the edges represent dependencies between these atoms. An *atom* is a set of highly related axioms of an ontology in the sense that they always co-occur in modules [14].

⁴The normalisation in [34] can straightforwardly be extended to *SROIQ*-ontologies. Then a normalised axiom can be of the forms as described, where A_i and B_i additionally range over nominals. However, nominals are not contained in any minimal non-locality signature of a normalised axiom.

Definition 4.11 (Atom). *An atom α is a maximal set of axioms of an ontology $\mathcal{O}$ such that for every module $\mathcal{M}$ of $\mathcal{O}$ either $\alpha \cap \mathcal{M} = \alpha$ or $\alpha \cap \mathcal{M} = \emptyset$.* $\dashv$

Consequently, we have that two axioms α and β are contained in an atom α iff $\text{Mod}_{\mathcal{O}}^x(\text{sig}(\alpha)) = \text{Mod}_{\mathcal{O}}^x(\text{sig}(\beta))$, where $\text{sig}(\alpha)$ ($\text{sig}(\beta)$) is the signature of the axiom α (β). We denote with $\text{Atoms}_{\mathcal{O}}^x$ the set of all atoms of $\mathcal{O}$ wrt. syntactic x -locality modules, for $x \in \{\perp, \top\}$. The atoms of an ontology partition the ontology into pairwise disjoint subsets. All axioms of the ontology are distributed over atoms such that every axiom occurs in exactly one atom. A dependency relation between atoms can be established as follows [14].

Definition 4.12 (Dependency relation between atoms). *An atom α_2 depends on an atom α_1 in an ontology $\mathcal{O}$ (written $\alpha_1 \succ_{\mathcal{O}} \alpha_2$) if α_2 occurs in every module of $\mathcal{O}$ containing α_1 . The binary relation $\succ_{\mathcal{O}}$ is a partial order.* $\dashv$

In other words, an atom α_2 depends on an atom α_1 in an ontology $\mathcal{O}$ if the module $\text{Mod}_{\mathcal{O}}^x(\text{sig}(\beta))$ is contained in the module $\text{Mod}_{\mathcal{O}}^x(\text{sig}(\alpha))$, for some α, β with $\alpha \in \alpha_1$ and $\beta \in \alpha_2$. For a given ontology $\mathcal{O}$, the poset $\langle \text{Atoms}_{\mathcal{O}}^x, \succ_{\mathcal{O}} \rangle$ was introduced as the *Atomic Decomposition (AD)* of $\mathcal{O}$, and it represents the modular structure of the ontology [14].

From the information provided by the AD is not possible to determine the unique set of atoms that is required to build the locality-based module for a given signature. To overcome this limitation, it has been suggested to enrich the AD of an ontology with a suitable set of labels that expose how atoms and signatures are related. Thus, the labels will determine which atoms become relevant for a given signature. The *Minimal Globalizing Signature (MGS)* of an axiom α [13] is the smallest set $\text{MGS}^x(\alpha)$ of signatures for which the α is not x -local.

The *Minimal Globalizing Signature* of an atom α , $\text{MGS}(\alpha)$, is the union and minimisation wrt. set inclusion of the MGS of each axiom in α : $\text{MGS}(\alpha) = \cup_{\alpha \in \alpha}^* \text{MGS}(\alpha)$.⁵ To decrease the amount of labels associated to each atom after all MGSs have been computed, an algorithm to calculate the *Minimal Seed Signature (MSS)* for a set of atoms was defined [13]. Notice that the size of $\text{MGS}^x(\alpha)$ can be exponential in the size of α . Moreover, it has been determined that the worst-case runtime and memory consumption of the algorithm for computing all MSSs of an ontology $\mathcal{O}$ is exponential in the size of $\mathcal{O}$ [13]. For the practical application of this algorithm, it is possible to define a time-out to interrupt the computation of the MSSs of an ontology when it is taken too long. The atoms that do not have all MSSs computed will be included in any locality-based module even if they are irrelevant for the given signature. It has been suggested to use MGSs instead of MSSs to compute $\perp$ - or $\top$ -locality-based modules. Preliminary results of this approach can be found in [43].

⁵This operator, $\cup_{\alpha \in \alpha}^*$, will not allow any proper superset of a minimal signature of an axiom α already included in the MGS of an atom α .

4.3 Reference Implementations

We now introduce some of the most prominent tools that have been optimized to compute syntactic locality-based modules and the atomic decomposition of OWL ontologies.

FaCT++⁶ is an OWL 2 reasoner implemented in C++. The latest version, FaCT++ v1.6.3, from May 2014 supports OWLAPI v4.0.1. FaCT++ implements optimized algorithms for computing syntactic locality-based modules and the atomic decomposition of OWL 2 ontologies [41].

OWLAPI-tools⁷ is a collection of Java libraries that extend the functionality of the OWLAPI. The latest version, OWLAPI-tools v4.0.1, was released in November 2014 together with the OWLAPI v4.0.1.⁸ The collection include a specific library for computing syntactic locality-based modules and the atomic decomposition of OWL 2 ontologies. This library implements the same algorithms as FaCT++ and a similar Java interface [42]. We refer to this library as OWLAPI-AD.

It is also worth mentioning the system CEL⁹, a reasoner implemented in LISP for the description logic EL^{++} [6]. CEL is also capable of computing syntactic locality-based modules for OWL EL ontologies. CEL implicitly represents an OWL EL ontology as a directed hypergraph where each symbol of the ontology is a node in the hypergraph and each axiom is represented as a directed hyperedge. Connected components in the directed hypergraph correspond to syntactic locality-based modules [38]. The latest version of CEL v1.1.2 was released in August 2009, and the most recent version of the plug-in supports OWLAPI v3.2.4. A version of CEL implemented in Java with the name of jcel¹⁰ is also available. However, jcel does not include the algorithms for computing syntactic locality-based modules.

⁶<http://code.google.com/p/factplusplus/>

⁷<https://github.com/owlcs/owlapitools/>

⁸<http://sourceforge.net/projects/owlapi/files/>

⁹<https://code.google.com/p/cel/>

¹⁰<http://jcel.sourceforge.net>

Chapter 5

Module Extraction and Atomic Decomposition of OWL Ontologies

In this chapter, we introduce the notion of an axiom dependency hypergraph which is a generalization of the atomic decomposition of an ontology. A module in the ontology corresponds to a connected component in the hypergraph, and the atoms of the ontology to the strongly-connected components.

Chapter 5 is organized as follows. Section 5.1 provides a definition of the notion of axiom dependency hypergraph. Section 5.2 shows how the B -connectivity relation in an axiom dependency hypergraph can be used to extract syntactic locality-based modules. The computation of $\perp\top^*$ -local modules using axiom dependency hypergraphs is also discussed in detail. In Section 5.3, we prove that an atom of an ontology corresponds to a strongly- B -connected component in the axiom dependency hypergraph. We also introduce in the same section the notions of (partially) condensed axiom dependency hypergraphs. A summary of the chapter is included in Section 5.4.

5.1 Axiom Dependency Hypergraphs

In this section, we will show how directed hypergraphs can be used to explicitly represent the locality-based modules of an ontology. To this end, we define the notion of an *axiom dependency hypergraph (ADH)* as follows.

Definition 5.1 (Axiom Dependency Hypergraph). *Let $\mathcal{O}$ be an ontology. Let $x \in \{\perp, \top\}$ denote a locality notion. The Axiom Dependency Hypergraph $\mathcal{H}_{\mathcal{O}}^x$ for $\mathcal{O}$ wrt. x -locality (x -ADH) is defined as the directed hypergraph $\mathcal{H}_{\mathcal{O}}^x = (\mathcal{V}^x, \mathcal{E}^x)$, where*

- $\mathcal{V}^x = \mathcal{O}$; and

- $e = (T(e), H(e)) \in \mathcal{E}^x$ iff $T(e) \subseteq \mathcal{V}^x$ and $H(e) = \{\beta\}$, for some $\beta \in \mathcal{V}^x$, such that:

(i) $\beta \notin T(e)$, and

(ii) β is not x -local wrt. $\text{sig}(T(e))$. ⊣

The nodes of the ADH are the axioms in the ontology. Hyperedges are directed and they connect one or more tail nodes with one head node, where the tail and the head are disjoint. This implies that an ADH is in fact a B -hypergraph. Intuitively, the tail nodes of an hyperedge e correspond to axioms that provide the signature symbols required by the axiom represented by the head node of e to be non-local. We can think on computing a B -connected component in an ADH as how the module extraction algorithm computes a module by successively including axioms into the module (cf. Figure 4.1, Section 4.1.1).

The notion of ADH depends on the notion of syntactic locality. Using Prop. 4.10, we can similarly define this notion using minimal non-locality signatures by replacing Item (ii) of Def. 5.1 with:

(iib) $\Sigma \subseteq \text{sig}(T(e))$, for some $\Sigma \in \text{MLS}(\beta)$.

An ADH $\mathcal{H}_{\mathcal{O}}^x$ contains all locality-based dependencies between different axioms of the ontology $\mathcal{O}$. These dependencies are represented by the hyperedges in $\mathcal{H}_{\mathcal{O}}^x$. Note that $\mathcal{H}_{\mathcal{O}}^x$ may contain exponentially many hyperedges, many of which can be considered redundant in the following sense.

Definition 5.2. A hyperedge e in a B -hypergraph $\mathcal{H}$ is called *redundant* if there is a hyperedge e' in $\mathcal{H}$ such that $H(e) = H(e')$ and $T(e') \subsetneq T(e)$. ⊣

A *compact* version of a B -hypergraph $\mathcal{H}$ is obtained from $\mathcal{H}$ by removing all redundant hyperedges while the B -connectivity relation between axioms is preserved.

Proposition 5.3. Let $\mathcal{H}_1$ be a directed B -hypergraph and $\mathcal{H}_2$ its compacted version. Let $\geq_B^1$ and $\geq_B^2$ be the B -connectivity relations in $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively. Then: $\geq_B^1 = \geq_B^2$. ⊣

Proof. We show that $\geq_B^1 \subseteq \geq_B^2$; the other direction is trivial. As $\mathcal{H}_2$ is the compacted version of $\mathcal{H}_1$, both graphs share the same set of nodes. It is sufficient to show for all subsets $\mathcal{V}'$ of nodes and all nodes v that $\mathcal{V}' \geq_B^1 v$ implies $\mathcal{V}' \geq_B^2 v$ (IH). We show this by induction on the maximal length $n = \text{dist}_{\mathcal{H}}(\mathcal{V}', v)$ of an acyclic hyperpath from a node v' in $\mathcal{V}'$ to v . For the induction base, $n = 1$. We have that $v' = v$. Suppose that $\mathcal{V}' \geq_B^1 v$. We obtain that $v \in \mathcal{V}'$ and, thus, $\mathcal{V}' \geq_B^2 v$ by reflexivity of $\geq_B^2$ (Item (i) in the definition of B -connectivity). Assume (IH) was shown for n . Consider the induction step where $n \rightarrow n + 1$. Suppose that $\mathcal{V}' \geq_B^1 v$. Then by definition of B -connectivity we have that either (a) $v \in \mathcal{V}'$, or (b) there is a hyperedge

e_1 in $\mathcal{H}_1$ with $H(e_1) = \{v\}$ and $(\dagger) \mathcal{V}' \geq_B^1 v'$ for all $v' \in T(e_1)$. For (a), we have already established $\mathcal{V}' \geq_B^2 v$ for $n = 1$. Consider (b). Suppose e_1 is a redundant hyperedge; otherwise we immediately obtain $\mathcal{V}' \geq_B^2 v$ as e_1 is a hyperedge in $\mathcal{H}_2$. By redundancy of e_1 and Def. 5.2, there is a hyperedge e_2 in $\mathcal{H}_2$ such that $H(e_2) = v$ and $T(e_2) \subsetneq T(e_1)$. Together with $(\dagger)$ it follows that $\mathcal{V}' \geq_B^1 v'$ for all $v' \in T(e_2)$. Hence, $\mathcal{V}' \geq_B^2 v$ by Item (ii) in the definition of B -connectivity. This concludes the induction and the proof. $\square$

In the remainder of the thesis, we consider ADHs that are compact. Notice that compact ADHs are unique and they may still contain exponentially many hyperedges.

Proposition 5.4. *Let $\mathcal{H}_{\mathcal{O}}^x$ be an x -ADH, where $x \in \{\perp, \top\}$ is a syntactic locality notion. There exists ontologies $\mathcal{O}$ such that $\mathcal{H}_{\mathcal{O}}^x$ contains exponentially many hyperedges in the number of axioms of $\mathcal{O}$.* $\dashv$

We show the proposition with the following example.

Example 5.5. *Let $\mathcal{O}_{n,k} = \{\alpha\} \cup \{\gamma_j^i \mid j \in \{1, \dots, n\}, i \in \{1, \dots, k\}\}$ be an ontology, where $n, k > 0$ and $\alpha = X_1 \sqcap \dots \sqcap X_j \sqsubseteq Y$ and $\gamma_j^i = A_j^i \sqsubseteq X_j$. The axioms γ_j^i state that each concept name X_j subsumes k many concept names $A_j^1, \dots, A_j^k$, and axiom α states that each X_j is subsumed by concept name Y . The corresponding axiom dependency graph $\mathcal{H}_{\mathcal{O}_{n,k}}^\perp$ is the tuple $\mathcal{H}_{\mathcal{O}_{n,k}}^\perp = (\mathcal{V}^\perp, \mathcal{E}^\perp)$, where $\mathcal{V}^\perp = \mathcal{O}_{n,k}$ and*

$$\mathcal{E}^\perp = \{(\{\beta_1, \dots, \beta_n\}, \{\alpha\}) \mid \beta_j \in \{\gamma_j^1, \dots, \gamma_j^k\}, j \in \{1, \dots, n\}\}.$$

It can readily be seen that $\mathcal{H}_{\mathcal{O}_{n,k}}^\perp$ contains k^n many hyperedges, where n, k are each bound by the number of axioms in $\mathcal{O}_{n,k}$. Note that every strongly-connected component of $\mathcal{H}_{\mathcal{O}_{n,k}}^\perp$ (as well as every atom of $\mathcal{O}_{n,k}$) is a singleton set. This means the hyperedges in $\mathcal{H}_{\mathcal{O}_{n,k}}^\perp$ are not required for determining the strongly-connected components of $\mathcal{H}_{\mathcal{O}_{n,k}}^\perp$. The hyperedges that are needed are the ones that preserve the mutual B -connectivity relation. $\triangleleft$

Next, we characterise modules and atoms together with their dependencies in terms of ADHs for which B -connectivity is crucial.

5.2 Locality-based Modules

B -connectivity in an ADH can be used to identify locality-based modules in the corresponding ontology. A locality-based module of an ontology $\mathcal{O}$ for the signature of an axiom α (or a subset of axioms $\mathcal{O}' \subseteq \mathcal{O}$) corresponds to the B -connected component in the ADH for $\mathcal{O}$ from α (or $\mathcal{O}'$) [25].

Proposition 5.6. *Let $\mathcal{O}$ be an ontology, $\mathcal{O}' \subseteq \mathcal{O}$ and $\Sigma = \text{sig}(\mathcal{O}')$. Let $\geq_B$ be the B-connectivity relation of the x -ADH for $\mathcal{O}$ and let $\text{Mod}^x(\Sigma, \mathcal{O})$ be the function depicted in Figure 4.1, where $x \in \{\perp, \top\}$. Then: $\text{Mod}^x(\Sigma, \mathcal{O}) = \geq_B(\mathcal{O}')$. $\dashv$*

Proof. We show the proposition by showing both directions of the set containment in $\text{Mod}^x(\Sigma) = \geq_B(\mathcal{O}')$. Let $\mathcal{H}_{\mathcal{O}}^x$ be the x -ADH for $\mathcal{O}$ and let $\geq_B$ be the B-connectivity relation in $\mathcal{H}_{\mathcal{O}}^x$.

“ $\subseteq$ ”: Let $\beta \in \mathcal{M}$, where $\mathcal{M} = \text{Mod}^x(\Sigma, \mathcal{O})$. The function $\text{Mod}^x(\Sigma, \mathcal{O})$ computes a sequence $\mathcal{M}_0, \dots, \mathcal{M}_n$ such that $\mathcal{M}_0 = \emptyset$, $\mathcal{M}_i \subseteq \mathcal{M}_{i+1}$, for $i \in \{0, \dots, n-1\}$, and $\mathcal{M}_n = \mathcal{M}$. We show by induction on $n > 0$ the hypothesis (IH): $\mathcal{M}_1 \geq_B \alpha$, for every axiom $\alpha \in \mathcal{M}_n$. For the induction base, $n = 1$. We have that $\mathcal{M}_1 \geq_B \alpha$, for every axiom $\alpha \in \mathcal{M}_1$ by reflexivity of $\geq_B$ (Item (i) in the definition of B-connectivity). Suppose we have shown (IH) for n . Consider the induction step where $n \rightarrow n+1$. Let $\alpha \in \mathcal{M}_{n+1}$. We have to show that $\mathcal{M}_1 \geq_B \alpha$. By Line 5 of the function $\text{Mod}^x(\Sigma, \mathcal{O})$, α is not x -local wrt. $\Sigma \cup \text{sig}(\mathcal{M}_n)$. Since $\Sigma \subseteq \text{sig}(\mathcal{M}_1)$, we have that α is not x -local wrt. $\text{sig}(\mathcal{M}_n)$. Then there exists a hyperedge e in $\mathcal{H}_{\mathcal{O}}^x$ with $H(e) = \{\alpha\}$ and $T(e) \subseteq \mathcal{M}_n$ by Item (ii) of Def. 5.1. This implies that $\mathcal{M}_n \geq_B \alpha$. By (IH) for n , we have that $\mathcal{M}_1 \geq_B \beta$, for all $\beta \in \mathcal{M}_n$. We obtain that $\mathcal{M}_1 \geq_B \beta$ by transitivity of $\geq_B$ (Item (ii) in the definition of B-connectivity). It remains to show that $\mathcal{O}' \geq_B \alpha$, for all $\alpha \in \mathcal{M}_1$. Then we obtain $\mathcal{O}' \geq_B \beta$, which concludes this direction of the proof. Let $\alpha \in \mathcal{M}_1$. By Line 5 of the function $\text{Mod}^x(\Sigma, \mathcal{O})$, α is not x -local wrt. Σ . From the fact that $\Sigma = \text{sig}(\mathcal{O}')$, it follows from Item (ii) of Def. 5.1 that there exists a hyperedge e in $\mathcal{H}_{\mathcal{O}}^x$ with $H(e) = \{\alpha\}$ and $T(e) \subseteq \mathcal{O}'$. Hence, $\mathcal{O}' \geq_B \alpha$.

$\geq$ -admissible hyperpath".

“ $\supseteq$ ”: Let $\mathcal{M}$ be the module computed by $\text{Mod}^x(\Sigma, \mathcal{O})$. Let $\beta \in \mathcal{O}$ such that $\mathcal{O}' \geq_B \beta$. We show that (IH) $\mathcal{O}' \geq_B \beta$ implies $\beta \in \mathcal{M}$ by induction on the maximal length $n = \text{dist}_{\mathcal{H}}(\mathcal{O}', \beta)$ of a simple $\geq$ -admissible hyperpath from an axiom α in $\mathcal{O}'$ to β . For the induction base, $n = 1$. We have that $\beta \in \mathcal{O}'$. By definition of locality, every axiom in $\mathcal{O}'$ is not x -local wrt. $\text{sig}(\mathcal{O}')$. Together with the fact that $\Sigma = \text{sig}(\mathcal{O}')$ we obtain by Line 5 of the function $\text{Mod}^x(\Sigma, \mathcal{O})$ that $\beta \in \mathcal{M}$. Suppose we have shown (IH) for n . Consider the induction step where $n \rightarrow n+1$. By definition of B-connectivity we have that either (a) $\beta \in \mathcal{O}'$, or (b) there is a hyperedge e in $\mathcal{H}_{\mathcal{O}}^x$ with $H(e) = \{\beta\}$ and $\mathcal{O}' \geq_B \alpha$, for all $\alpha \in T(e)$. For (a), we have already established that $\beta \in \mathcal{M}$ for $n = 1$. Consider (b). As e is a hyperedge in $\mathcal{H}_{\mathcal{O}}^x$, it follows by Item (ii) of Def. 5.1 that β is not x -local wrt. $\text{sig}(T(e))$. By (IH) we have that $\alpha \in \mathcal{M}$, for all $\alpha \in T(e)$. There is a set $\mathcal{M}_i$ computed by the function $\text{Mod}^x(\Sigma, \mathcal{O})$ such that $T(e) \subseteq \mathcal{M}_i$, where $\mathcal{M}_i \subseteq \mathcal{M}$. This implies that β is not x -local wrt. $\text{sig}(\mathcal{M}_i)$, and $\beta \in \mathcal{M}_{i+1}$ by Line 5 of the function $\text{Mod}^x(\Sigma, \mathcal{O})$. Hence, $\beta \in \mathcal{M}$. $\square$

Fig. 5.1 (Example 5.7): Σ -extension of a $\perp$ -ADH.

It can readily be seen that ADHs do not contain sufficient information to compute a module for *any* signature as the following simple example shows.

Example 5.7. Let $\mathcal{O} = \{\alpha_1, \alpha_2, \alpha_3\}$ be an ontology such that $\alpha_1 = A \sqsubseteq C$, $\alpha_2 = C \sqcap B \sqsubseteq D$ and $\alpha_3 = D \sqsubseteq A$. Let $\Sigma = \{A, B\}$ be a signature. Let $\mathcal{H}_O^\perp = (\mathcal{V}^\perp, \mathcal{E}^\perp)$ be the ADH of $\mathcal{O}$ for the notion of syntactic $\perp$ -locality, where $\mathcal{V}^\perp = \{\alpha_1, \alpha_2, \alpha_3\}$ and $\mathcal{E}^\perp = \{e_1 = (\{\alpha_2\}, \alpha_3), e_2 = (\{\alpha_3\}, \alpha_1)\}$. The syntactic $\perp$ -local module of the ontology $\mathcal{O}$ for the signature Σ is $\mathcal{M} = \{\alpha_1, \alpha_2, \alpha_3\}$. The $\perp$ -ADH for $\mathcal{O}$ contains no hyperedge e with $H(e) = \{\alpha_2\}$ and, consequently, α_2 cannot be reached via a hyperedge. $\triangleleft$

The problem can be solved by incorporating the signature Σ into the ADH. The Σ -extension $\mathcal{H}_{O,\Sigma}^x$ of an x -ADH $\mathcal{H}_O^x$ for an ontology $\mathcal{O}$ wrt. x -locality, $x \in \{\perp, \top\}$, is defined as the ADH according to Def. 5.1 but with Item (ii) replaced with:

(iii) β is not x -local wrt. $\Sigma \cup \text{sig}(T(e))$.

Intuitively, no symbol in Σ contributes to the dependencies between axioms. Consequently, less axioms in the tail are needed to provide the signature for non-locality of β . Note that non-redundant hyperedges in the original ADH may become redundant in the Σ -extended ADH. The remaining hyperedges represent the dependencies between axioms *modulo* Σ .

Example 5.8. Let $\mathcal{O}$, Σ and $\mathcal{H}_O^\perp$ as in Example 5.7. The Σ -extension of $\mathcal{H}_O^\perp$ has the same nodes and hyperedges but also one more hyperedge: $e_3 = (\{\alpha_1\}, \alpha_2)$. Hence, α_2 can now be reached via the hyperedge e_3 . Axiom α_1 is the only axiom that is not- $\perp$ local wrt. Σ . The axioms that correspond to the nodes B -connected from α_1 coincide with the axioms of the syntactic $\perp$ -local module computed by the function $\text{Mod}^\perp(\Sigma, \mathcal{O})$ depicted in Figure 4.1. For an illustration of this example, see Figure 5.1. $\triangleleft$

Given the Σ -extension of an ADH for an ontology, B -connectivity can be used to determine the axioms that are not local wrt. to Σ and to compute the corresponding locality-based module.

Proposition 5.9. Let $\mathcal{O}$ be an ontology, Σ a signature and $x \in \{\perp, \top\}$. Let $\mathcal{O}'$ be the set of axioms from $\mathcal{O}$ that are not x -local wrt. Σ . Let $\geq_B$ be the B -connectivity relation of the Σ -extension of the x -ADH for $\mathcal{O}$. Then: $\text{Mod}^x(\Sigma, \mathcal{O}) = \geq_B(\mathcal{O}')$. $\dashv$

Proof. The proof is similar to the proof of Prop. 5.6. Let $\mathcal{H}_{\mathcal{O},\Sigma}^x$ be the Σ -extension of the ADH for $\mathcal{O}$ as given in Def. 5.1 with Item (ii) replaced by Item (iii) above. Let $\geq_B$ be the B-connectivity relation in $\mathcal{H}_{\mathcal{O},\Sigma}^x$.

“ $\subseteq$ ”: Let $\mathcal{M} = \text{Mod}^x(\Sigma, \mathcal{O})$ and let $\beta \in \mathcal{M}$. The function $\text{Mod}^x(\Sigma, \mathcal{O})$ computes a sequence $\mathcal{M}_0, \dots, \mathcal{M}_n$ such that $\mathcal{M}_0 = \emptyset$, $\mathcal{M}_i \subseteq \mathcal{M}_{i+1}$, for $i \in \{0, \dots, n-1\}$, and $\mathcal{M}_n = \mathcal{M}$. We show by induction on $n > 0$ the hypothesis (IH): $\mathcal{M}_1 \geq_B \alpha$, for every axiom $\alpha \in \mathcal{M}_n$. For the induction base, $n = 1$. We have that $\mathcal{M}_1 \geq_B \alpha$, for every axiom $\alpha \in \mathcal{M}_1$ by reflexivity of $\geq_B$ (Item (i) in the definition of B-connectivity). Suppose we have shown (IH) for n . Consider the induction step where $n \rightarrow n+1$. Let $\alpha \in \mathcal{M}_{n+1}$. We have to show that $\mathcal{M}_1 \geq_B \alpha$. By Line 5 of the algorithm in Figure 4.1, α is not x -local wrt. $\Sigma \cup \text{sig}(\mathcal{M}_n)$. Note that $\Sigma \subseteq \text{sig}(\mathcal{M}_n)$ does not hold in general, different to the proof of Prop. 5.6. There exists a hyperedge e in the Σ -extension $\mathcal{H}_{\mathcal{O},\Sigma}^x$ of the ADH for $\mathcal{O}$ with $H(e) = \{\alpha\}$ and $T(e) \subseteq \mathcal{M}_n$ by Item (iii) of Def. 5.1. This implies that $\mathcal{M}_n \geq_B \alpha$. By (IH) for n , we have that $\mathcal{M}_1 \geq_B \beta$, for all $\beta \in \mathcal{M}_n$. We obtain that $\mathcal{M}_1 \geq_B \beta$ by transitivity of $\geq_B$ (Item (ii) in the definition of B-connectivity). Since $\mathcal{O}' = \mathcal{M}_1$, we obtain $\mathcal{O}' \geq_B \beta$, which concludes this direction of the proof. Note that $\mathcal{O}'$ is the set of axioms from $\mathcal{O}$ that are not x -local wrt. Σ .

“ $\supseteq$ ”: Let $\mathcal{O}'$ be the set of axioms from $\mathcal{O}$ that are not x -local wrt. Σ . Let $\beta \in \mathcal{O}$ such that $\mathcal{O}' \geq_B \beta$. Let $\mathcal{M} = \text{Mod}^x(\Sigma, \mathcal{O})$. We show that (IH) $\mathcal{O}' \geq_B \beta$ implies $\beta \in \mathcal{M}$ by induction on the maximal length $n = \text{dist}_{\mathcal{H}}(\mathcal{O}', \beta)$ of an simple $\geq$ -admissible hyperpath from an axiom α in $\mathcal{O}'$ to β . For the induction base, $n = 1$. We have that $\beta \in \mathcal{O}'$. By the fact that $\mathcal{O}'$ consists of axioms that are not x -local wrt. Σ , the function $\text{Mod}^x(\Sigma, \mathcal{O})$ computes $\mathcal{M}_1 = \mathcal{O}'$. Thus, $\beta \in \mathcal{M}$. Suppose we have shown (IH) for n . Consider the induction step where $n \rightarrow n+1$. By definition of B-connectivity we have that either (a) $\beta \in \mathcal{O}'$, or (b) there is a hyperedge e in the Σ -extension $\mathcal{H}_{\mathcal{O},\Sigma}^x$ of the ADH for $\mathcal{O}$ with $H(e) = \{\beta\}$ and $\mathcal{O}' \geq_B \alpha$, for all $\alpha \in T(e)$. For (a), we have already established that $\beta \in \mathcal{M}$ for $n = 1$. Consider (b). As e is a hyperedge in $\mathcal{H}_{\mathcal{O},\Sigma}^x$, it follows by Item (iii) of Def. 5.1 that β is not x -local wrt. $\Sigma \cup \text{sig}(T(e))$. By (IH) we have that $\alpha \in \mathcal{M}$, for all $\alpha \in T(e)$. There is a set $\mathcal{M}_i$ computed by the function $\text{Mod}^x(\Sigma, \mathcal{O})$ such that $T(e) \subseteq \mathcal{M}_i$. This implies that β is not x -local wrt. $\Sigma \cup \text{sig}(\mathcal{M}_i)$, and $\beta \in \mathcal{M}_{i+1}$ by Line 5 of the algorithm. Hence, $\beta \in \mathcal{M}$. $\square$

Build a completely new ADH each time that we need to compute the module of a given signature might be very expensive. There are alternative approaches to compute syntactic x -local modules in an ADH without building a Σ -extension ADH. One of these approaches is to add to the ADH an *auxiliary* node. The node represents a new axiom with only one MLS and this MLS is a singleton set with just one fresh symbol, a symbol that it was not in the ontology before. This implies that no other node of the ADH can reach the auxiliary node. The signature of this auxiliary node extends the signature of the module that we want to

compute with the fresh symbol of its MLS. New hyperedges with the auxiliary node in their tails are added into the ADH. This is how we extend the original ADH with the signature of the module that we want to compute. The B -connected component of the auxiliary node (excluding the auxiliary node) corresponds to the syntactic x -local module for the given signature. After the module is computed, the auxiliary node and hyperedges are removed from the ADH. This approach is similar to one suggested in [34], where an auxiliary node and the appropriate edges are added to the hypergraph.¹ Note that the number of auxiliary hyperedges that we might add to the original ADH can be exponential with respect to the number of nodes of the ADH. Therefore, extending the original ADH with a new node and the required hyperedges might be also expensive.

Proposition 5.10. *Let $\mathcal{O}$ be an ontology, Σ a signature and $x \in \{\perp, \top\}$. Let α be an axiom such that $\alpha \notin \mathcal{O}$, $MLS^x(\alpha) = \{\{x\}\}$ and $\text{sig}(\alpha) = \Sigma \cup \{x\}$, where x is a symbol and $x \notin \text{sig}(\mathcal{O})$. Let $\mathcal{O}'$ be an ontology such that $\mathcal{O}' = \mathcal{O} \cup \{\alpha\}$. Let $\geq_B$ be the B -connectivity relation of the x -ADH for $\mathcal{O}'$. Then: $\text{Mod}^x(\Sigma, \mathcal{O}) = (\geq_B(\{\alpha\})) \setminus \{\alpha\}$. $\dashv$*

The proof of this proposition is similar to the proof of Proposition 5.9.

The following example illustrates how we can compute a syntactic x -local module of an ontology by adding an auxiliary node to the ADH.

Example 5.11. *Let $\mathcal{O}$, Σ and $\mathcal{H}_{\mathcal{O}}^{\perp}$ as in Example 5.7. Let $\mathcal{O}' = \mathcal{O} \cup \{\alpha_4\}$, where $\alpha_4 = X \sqsubseteq A \sqcap B$, $X \notin \text{sig}(\mathcal{O})$ and $\Sigma = \{A, B\}$. Let $\mathcal{H}_{\mathcal{O}'}^{\perp} = (\mathcal{V}^{\perp}, \mathcal{E}^{\perp})$ be the ADH of $\mathcal{O}'$ for the notion of syntactic $\perp$ -locality, where $\mathcal{V}^{\perp} = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ and $\mathcal{E}^{\perp} = \{e_1 = (\{\alpha_2\}, \alpha_3), e_2 = (\{\alpha_3\}, \alpha_1), e_3 = (\{\alpha_4\}, \alpha_1), e_4 = (\{\alpha_1, \alpha_4\}, \alpha_2)\}$.*

The syntactic $\perp$ -local module of the ontology $\mathcal{O}$ for the signature Σ is $\mathcal{M} = \{\alpha_1, \alpha_2, \alpha_3\}$. Let $\geq_B$ be the B -connectivity relation of $\mathcal{H}_{\mathcal{O}'}^{\perp}$. Then: $\geq_B(\{\alpha_4\}) = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$. For an illustration of this example, see Figure 5.2. $\triangleleft$

Another approach to compute syntactic x -local modules without building a Σ -extension ADH is to ignore the ADH and adapt the algorithm for extracting locality-based modules [11], cf. Figure 4.1 to check locality via signature containment. The idea is to identify which axioms have an MLS that it is covered by the signature of the module that we want to compute. Let $\mathcal{O}$ be an ontology and Σ the signature of the module that we want to extract. Each time that a new axiom in $\mathcal{O}$ is identified as non-local, Σ is extended with the signature of this axiom. Using this approach, we do not have to build a specific ADH to compute the module; we just check MLSs of axioms. This is also the approach suggested to compute

¹Note that, different to ADHs, in the hypergraphs defined in [34] each node represents a symbol of the signature of the ontology and each hyperedge represents an axiom.

Fig. 5.2 (Example 5.7): An alternative representation of the Σ -extension of a $\perp$ -ADH.

modules from a *labelled* atomic decomposition using so-called *Minimal Seed Signatures* [14]. Note that the MLSs and signatures of the axioms encode exponentially many hyperedges of the ADH. We can use the MLSs and signatures of the axioms of an ontology as a *succinct* representation of the original ADH. This compact representation is defined and studied in Section 6.1.

Finally, it is worth mentioning that B -connectivity in $\perp$ -ADH coincides with the notion of $\perp$ -reachability in [34], whereas B -connectivity in $\top$ -ADH is not the same as $\top$ -reachability. The following example taken from [34] illustrates this difference.

Example 5.12. Let $\mathcal{O} = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ be an ontology in normal form according to [34], where $\alpha_1 = A \sqsubseteq \exists r.D_1$, $\alpha_2 = B \sqsubseteq (\geq nr.D_2)$, for $n \geq 0$, $\alpha_3 = \exists r.\top \sqsubseteq C$ and $\alpha_4 = D_1 \sqsubseteq D_2$. To facilitate the understanding of how modules are computed under the notions of $\top$ -locality and $\top$ -reachability, we include in the example the minimal non- $\top$ -locality signatures of the axioms of $\mathcal{O}$ and also their signatures. We distinguish between the signature of an axiom and the signature of its left-hand side (LHS). Note that the axioms of $\mathcal{O}$ are concept inclusion axioms in normal form.

$$\begin{array}{lll}
 \text{MLS}^\top(\alpha_1) = \{\{r\}, \{D_1\}\} & \text{sig}(\alpha_1) = \{r, A, D_1\} & \text{sig}(\text{LHS}(\alpha_1)) = \{A\} \\
 \text{MLS}^\top(\alpha_2) = \{\{r\}, \{D_1\}\} & \text{sig}(\alpha_2) = \{r, B, D_2\} & \text{sig}(\text{LHS}(\alpha_2)) = \{B\} \\
 \text{MLS}^\top(\alpha_3) = \{\{C\}\} & \text{sig}(\alpha_3) = \{r, C\} & \text{sig}(\text{LHS}(\alpha_3)) = \{r\} \\
 \text{MLS}^\top(\alpha_4) = \{\{D_2\}\} & \text{sig}(\alpha_4) = \{D_1, D_2\} & \text{sig}(\text{LHS}(\alpha_4)) = \{D_1\}
 \end{array}$$

Let $\Sigma = \{C\}$ be a signature. Using the notion of $\top$ -locality which corresponds to B -connectivity in $\top$ -ADH, $\text{Mod}^\top(\Sigma, \mathcal{O}) = \mathcal{O}$. However, computing the same module using the notion of $\top$ -reachability [34] produces a different result: $\text{Mod}^\Delta(\Sigma, \mathcal{O}) = \{\alpha_1, \alpha_2, \alpha_3\}$. This result is a consequence of how $\top$ -reachability extends the signature Σ when a new axiom is included into a module. In the case of concept inclusion axioms in normal form, only the

```

function  $\text{extract}^*(\Sigma, \mathcal{O})$  returns  $\perp\top^*$ -local module of  $\mathcal{O}$  wrt.  $\Sigma$ 
1:  $\mathcal{H}^\perp := \text{computeADH}^\perp(\Sigma, \mathcal{O})$ 
2:  $\mathcal{H}^\top := \text{computeADH}^\top(\Sigma, \mathcal{O})$ 
3: do
4:    $\mathcal{M}^\perp := \text{extract}(\Sigma, \mathcal{H}^\perp)$ 
5:    $\mathcal{H}^\top := \text{deactivate}(\mathcal{V}^\top \setminus \mathcal{M}^\perp, \mathcal{H}^\top)$ 
6:    $\mathcal{M}^\top := \text{extract}(\Sigma, \mathcal{H}^\top)$ 
7:    $\mathcal{H}^\perp := \text{deactivate}(\mathcal{V}^\perp \setminus \mathcal{M}^\top, \mathcal{H}^\perp)$ 
8: until  $\mathcal{M}^\perp = \mathcal{M}^\top$ 
9: return  $\mathcal{M}^\perp$ 

```

Fig. 5.3 Module Extraction Algorithm for $\perp\top^*$ -local modules.

symbols of the left-hand side of an axiom that is $\top$ -reachable are added to Σ . Starting with $\Sigma = \{C\}$, the only axiom that is $\top$ -reachable from Σ is α_3 . This axiom is added into the module and $\Sigma = \Sigma \cup \{r\}$, where $\text{sig}(\text{LHS}(\alpha_3)) = \{r\}$. Similarly, α_2 and α_1 are added into the module and $\Sigma = \Sigma \cup \{B\} \cup \{A\}$. $D_2 \in \text{sig}(\alpha_2)$ but D_2 is not in the left-hand side of α_2 . Therefore, D_2 is not added in Σ and α_4 cannot be included into the module. $\triangleleft$

5.2.1 Syntactic $\perp\top^*$ -local modules

Characterising syntactic $\perp\top^*$ -local modules in terms of the B -connectivity relation in an ADH might not be feasible. However, it is possible to compute syntactic $\perp\top^*$ -local modules using ADHs similar to how it is done by the function $\text{Mod}^*(\Sigma, \mathcal{O})$ depicted in Figure 4.2. Recall that this function computes a syntactic $\perp\top^*$ -local module by nesting the extraction of $\perp$ - and $\top$ -local modules until a fix-point is reached, where the $\perp$ - and $\top$ -module are the same.

The function $\text{extract}^*(\Sigma, \mathcal{O})$ depicted in Figure 5.3 shows how to compute syntactic $\perp\top^*$ -local modules of an ontology $\mathcal{O}$ using ADHs. In Lines 1&2, the Σ -extensions of two ADHs for $\mathcal{O}$ are computed. One ADH is created for the notion of syntactic $\perp$ -locality and the other ADH for the notion of syntactic $\top$ -locality. Inside of a do-until loop (Lines 3–8), the function $\text{extract}(\Sigma, \mathcal{H}^\perp)$ stores in $\mathcal{M}^\perp$ the syntactic $\perp$ -local module for the signature in Σ in the ADH $\mathcal{H}^\perp$ (Line 4). To do this, the function $\text{extract}(\Sigma, \mathcal{H}^\perp)$ identifies which axioms are non-local with respect to a signature Σ and select the nodes that correspond to these non-local axioms. For these nodes, the function $\text{extract}(\Sigma, \mathcal{H}^\perp)$ computes the B -connected component in $\mathcal{H}^\perp$ and returns the set of axioms that correspond to the nodes of the B -connected component. By Proposition 5.9, the nodes of the B -connected component correspond to the syntactic $\perp$ -local module of the ontology $\mathcal{O}$ for the signature Σ .

After $\mathcal{M}^\perp$ is computed, all the nodes in the ADH $\mathcal{H}^\top$ that do not correspond to an axiom in $\mathcal{M}^\perp$ are deactivated (or removed) by the function $\text{deactivate}(\mathcal{M}^\perp, \mathcal{H}^\top)$ (Line 5). This implies that the hyperedges that have in their tails or heads any of these nodes are also deactivated (or removed) in $\mathcal{H}^\top$.

From the new version of $\mathcal{H}^\top$ it is computed the syntactic $\top$ -local module for the signature Σ which is stored in $\mathcal{M}^\top$ (Line 6). Each node in $\mathcal{H}^\perp$ that does not correspond to an axiom in $\mathcal{M}^\top$ is deactivated (or removed). The hyperedges that have in their tails or heads any of these nodes are also deactivated (or removed) in $\mathcal{H}^\perp$ (Line 7).

If $\mathcal{M}^\perp$ is not equal to $\mathcal{M}^\top$, then a new iteration starts where $\mathcal{M}^\perp$, $\mathcal{M}^\top$, $\mathcal{H}^\perp$ and $\mathcal{H}^\top$ are re-calculated. When $\mathcal{M}^\perp$ is equal to $\mathcal{M}^\top$, the execution of function $\text{extract}^*(\Sigma, \mathcal{O})$ concludes and it returns $\mathcal{M}^\perp$ as a result of the execution.

Proposition 5.13. *Let $\mathcal{O}$ be an ontology and Σ a signature. Let $\text{Mod}^*(\Sigma, \mathcal{O})$ be the function depicted in Figure 4.2. Let $\text{extract}^*(\Sigma, \mathcal{O})$ be the function depicted in Figure 5.3. Then: $\text{Mod}^*(\Sigma, \mathcal{O}) = \text{extract}^*(\Sigma, \mathcal{O})$. $\dashv$*

Proof. We sketch the proof. Let $\mathcal{O}$ be an ontology, Σ a signature and $x \in \{\perp, \top\}$. Let $\mathcal{O}'$ be the set of axioms from $\mathcal{O}$ that are not x -local wrt. Σ . Let $\geq_B$ be the B -connectivity relation of the Σ -extension of the x -ADH for $\mathcal{O}$. By Proposition 5.9, $\text{Mod}^x(\Sigma, \mathcal{O}) = \geq_B(\mathcal{O}')$.

The function $\text{Mod}^*(\Sigma, \mathcal{O})$ computes $\perp\top^*$ -locality modules by nesting the extraction of $\perp$ - and $\top$ -modules until a fixpoint is reached and the $\perp$ - and $\top$ -modules become the same. This is precisely how $\text{extract}^*_{\mathcal{O}}(\Sigma)$ has been defined. So, the statement of Prop. 5.13 can be readily be checked. $\square$

The following example illustrates how we can compute a syntactic $\perp\top^*$ -local module of an ontology $\mathcal{O}$ for a signature Σ using the function $\text{extract}^*(\Sigma, \mathcal{O})$.

Example 5.14. *Let $\mathcal{O} = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$ be an ontology such that $\alpha_1 = X \sqsubseteq A \sqcap B \sqcap D$, $\alpha_2 = B \sqsubseteq C$, $\alpha_3 = B \sqcap C \sqsubseteq A \sqcap B \sqcap D$, $\alpha_4 = D \sqsubseteq A \sqcup E \sqcup F$ and $\alpha_5 = E \sqsubseteq D$. Let $\Sigma = \{A, B\}$ be a signature. The following MLSs for syntactic $\perp$ - and $\top$ -locality can be identified:*

$$\left| \begin{array}{l} \text{MLS}^\perp(\alpha_1) = \{\{X\}\} \\ \text{MLS}^\perp(\alpha_2) = \{\{B\}\} \\ \text{MLS}^\perp(\alpha_3) = \{\{B, C\}\} \\ \text{MLS}^\perp(\alpha_4) = \{\{D\}\} \\ \text{MLS}^\perp(\alpha_5) = \{\{E\}\} \end{array} \right| \left| \begin{array}{l} \text{MLS}^\top(\alpha_1) = \{\{A\}, \{B\}, \{D\}\} \\ \text{MLS}^\top(\alpha_2) = \{\{C\}\} \\ \text{MLS}^\top(\alpha_3) = \{\{A\}, \{B\}, \{D\}\} \\ \text{MLS}^\top(\alpha_4) = \{\{A, E, F\}\} \\ \text{MLS}^\top(\alpha_5) = \{\{D\}\} \end{array} \right| \left| \begin{array}{l} \text{sig}(\alpha_1) = \{A, B, D, X\} \\ \text{sig}(\alpha_2) = \{B, C\} \\ \text{sig}(\alpha_3) = \{A, B, C, D\} \\ \text{sig}(\alpha_4) = \{A, D, E, F\} \\ \text{sig}(\alpha_5) = \{D, E\} \end{array} \right|$$

The function $\text{extract}^*(\Sigma, \mathcal{O})$ (depicted in Figure 5.3) computes the Σ -extension of two ADHs: one for syntactic $\perp$ -locality and one for syntactic $\top$ -locality (Lines 1&2). In this example,

Fig. 5.4 (Example 5.14): Computing a $\perp^*$ -local module using the function $\text{extract}^*(\Sigma, \mathcal{O})$.

we use an auxiliary node to build the Σ -extension of an ADH. This auxiliary node corresponds to the axiom α_6 that it is not in $\mathcal{O}$. This new axiom must satisfy two requirements: (1) $MLS^\perp(\alpha_6) = MLS^\top(\alpha_6) = \{\{Y\}\}$ and (2) $\text{sig}(\alpha_6) = \Sigma \cup \{Y\}$, where $Y \notin \text{sig}(\mathcal{O})$ is a fresh symbol. Note that α_6 is not explicitly defined because it is not easy to satisfy both requirements using just one axiom. For computing the $\perp\top^*$ -local module, the function $\text{extract}^*(\Sigma, \mathcal{O})$ does not require that α_6 is defined explicitly. It is enough to assume that such axiom exists and satisfies requirements (1) and (2).

Let $\mathcal{H}^\perp = (\mathcal{V}^\perp, \mathcal{E}^\perp)$ be the Σ -extension of the $\perp$ -ADH computed by the function $\text{extract}^*(\Sigma, \mathcal{O})$, where $\mathcal{V}^\perp = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}$ and $\mathcal{E}^\perp = \{e_1 = (\{\alpha_1\}, \alpha_2), e_2 = (\{\alpha_1\}, \alpha_4), e_3 = (\{\alpha_2\}, \alpha_3), e_4 = (\{\alpha_3\}, \alpha_2), e_5 = (\{\alpha_3\}, \alpha_4), e_6 = (\{\alpha_4\}, \alpha_5), e_7 = (\{\alpha_5\}, \alpha_4), e_8 = (\{\alpha_6\}, \alpha_2)\}$. $\mathcal{H}^\perp$ is depicted in Figure 5.4 (b).

Let $\mathcal{H}^\top = (\mathcal{V}^\top, \mathcal{E}^\top)$ be the Σ -extension of the $\top$ -ADH computed by the function $\text{extract}^*(\Sigma, \mathcal{O})$, where $\mathcal{V}^\top = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}$ and $\mathcal{E}^\top = \{e_1 = (\{\alpha_1\}, \alpha_3), e_2 = (\{\alpha_1\}, \alpha_5), e_3 = (\{\alpha_2\}, \alpha_1), e_4 = (\{\alpha_2\}, \alpha_3), e_5 = (\{\alpha_3\}, \alpha_1), e_6 = (\{\alpha_3\}, \alpha_2), e_7 = (\{\alpha_3\}, \alpha_5), e_8 = (\{\alpha_4\}, \alpha_1), e_9 = (\{\alpha_4\}, \alpha_3), e_{10} = (\{\alpha_4\}, \alpha_5), e_{11} = (\{\alpha_5\}, \alpha_1), e_{12} = (\{\alpha_5\}, \alpha_3), e_{13} = (\{\alpha_6\}, \alpha_3)\}$. $\mathcal{H}^\top$ is depicted in Figure 5.4 (a).

After the Σ -extension of the ADHs for $\perp$ - and $\top$ -locality are computed, $\text{extract}^*(\Sigma, \mathcal{O})$ computes a syntactic $\perp\top^*$ -local module by nesting the extraction of $\perp$ - and $\top$ -local modules until a fix-point is reached, where the $\perp$ - and $\top$ -local modules are the same. This is done inside the do-until loop (Lines 3–8).

The first step in the loop is to calculate the $\perp$ -local module for Σ using $\mathcal{H}^\perp$. This module is stored in $\mathcal{M}^\perp$ (Line 4), where $\mathcal{M}^\perp = \{\alpha_2, \alpha_3, \alpha_4, \alpha_5\}$. Then, the nodes that correspond to the axioms that are not in $\mathcal{M}^\perp$ are deactivated (or removed) from $\mathcal{H}^\perp$ (Line 5) which implies also that the related hyperedges are also deactivated (or removed). In particular, the node that corresponds to axiom α_1 and the hyperedges e_1, e_2, e_3, e_5, e_8 and e_{11} are deactivated (or removed). This is illustrated in Figure 5.4 (c), where the node and the hyperedges that have been deactivated (or removed) are in gray color.

In the new version of $\mathcal{H}^\top$, it is computed the $\top$ -local module for Σ . This module is stored in $\mathcal{M}^\top$ (Line 6), where $\mathcal{M}^\top = \{\alpha_2, \alpha_3, \alpha_5\}$. The function $\text{extract}^*(\Sigma, \mathcal{O})$ deactivates (or removes) the nodes in $\mathcal{H}^\perp$ that are not in $\mathcal{M}^\top$ and the related hyperedges (Line 7). In particular, the nodes that correspond to the axioms α_1 and α_4 and the hyperedges e_1, e_2, e_5, e_6 and e_7 are deactivated (or removed). This is depicted in Figure 5.4 (d).

The current iteration of the do-until loop ends and a new iteration starts because $\mathcal{M}^\perp \neq \mathcal{M}^\top$. $\mathcal{M}^\perp$ is computed again using the new version of $\mathcal{H}^\perp$ and new nodes in $\mathcal{H}^\top$ are deactivated (or removed), where $\mathcal{M}^\perp = \{\alpha_2, \alpha_3\}$. This is illustrated in Figure 5.4 (e).

$\mathcal{M}^\top$ is also computed again using the new version of $\mathcal{H}^\top$ and new nodes in $\mathcal{H}^\perp$ are deactivated (or removed), where $\mathcal{M}^\top = \{\alpha_2, \alpha_3\}$. This is shown in Figure 5.4 (f).

The do-until loop ends because $\mathcal{M}^\perp = \mathcal{M}^\top$ and the function $\text{extract}^*(\Sigma, \mathcal{O})$ returns $\mathcal{M}^\perp$ as the syntactic $\perp^\top$ -local module in $\mathcal{O}$ for the signature Σ . $\triangleleft$

5.3 Atomic Decomposition

In the previous section, we have established that locality-based modules of an ontology $\mathcal{O}$ correspond to sets of B -connected nodes in the axiom dependency hypergraph for $\mathcal{O}$. An atom of $\mathcal{O}$ consists of axioms α that share the same modules wrt. the signature of α . It holds that for every x -local atom $\mathfrak{a} \subseteq \mathcal{O}$ with $x \in \{\perp, \top\}$: $\alpha, \beta \in \mathfrak{a}$ if, and only if, $\text{Mod}^x(\text{sig}(\alpha), \mathcal{O}) = \text{Mod}^x(\text{sig}(\beta), \mathcal{O})$ [14]. Together with Proposition 5.6, we can now characterise the notion of an atom with a corresponding notion in ADHs. We have that two nodes in an ADH correspond to two axioms that are contained in the same atom if, and only if, the nodes agree on the set of nodes that are B -connected from them. Formally: $\alpha, \beta \in \mathfrak{a}$ if, and only if, $\geq_B(\alpha) = \geq_B(\beta)$, where $\geq_B$ be the B -connectivity relation of the ADH $\mathcal{H}_\mathcal{O}^x$ for $\mathcal{O}$ and $\alpha, \beta \in \mathcal{O}$. It follows that all axioms of an atom correspond to a set of nodes that are strongly- B -connected in $\mathcal{H}_\mathcal{O}^x$. Nodes that are strongly- B -connected constitute strongly- B -connected components of $\mathcal{H}_\mathcal{O}^x$. Consequently, the set of atoms for an ontology $\mathcal{O}$ corresponds to the set of strongly- B -connected components in the ADH for $\mathcal{O}$. Let $\text{SCCs}(\mathcal{H}_\mathcal{O}^x)$ be the set of strongly- B -connected components of the ADH $\mathcal{H}_\mathcal{O}^x$, where $x \in \{\perp, \top\}$.

Proposition 5.15. *Let $\mathcal{O}$ be an ontology and let $x \in \{\perp, \top\}$ denote a locality notion. Let $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ be the x -ADH for $\mathcal{O}$. Then: $\text{Atoms}_\mathcal{O}^x = \text{SCCs}(\mathcal{H}_\mathcal{O}^x)$.* $\dashv$

To show Proposition 5.15, we show the following proposition instead.

Proposition 5.16. *Let $\simeq_B$ be the strong- B -connectivity relation in the x -ADH $\mathcal{H}_\mathcal{O}^x$ for the ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathfrak{a} \in \text{Atoms}_\mathcal{O}^x$. Then: $\mathfrak{a} = [\alpha]_{\simeq_B}$, for every $\alpha \in \mathfrak{a}$.* $\dashv$

Proof. Let $\geq_B$ the B -connectivity relation in $\mathcal{H}_\mathcal{O}^x$. Let $\alpha \in \mathfrak{a}$. We have that $\beta, \beta' \in \mathfrak{a}$ iff $\text{Mod}^x(\text{sig}(\beta), \mathcal{O}) = \text{Mod}^x(\text{sig}(\beta'), \mathcal{O})$, cf. [14]. By Prop. 5.6, this is equivalent to $\geq_B(\beta) = \geq_B(\beta')$. This is equivalent to $\beta \simeq_B \beta'$. It follow that $\beta, \beta' \in [\alpha]_{\simeq_B}$. Hence $\mathfrak{a} = [\alpha]_{\simeq_B}$. $\square$

The *condensed ADH* (*cADH*) of an ontology $\mathcal{O}$ is formed by collapsing the strongly- B -connected components into single nodes and turning hyperedges between nodes that correspond to axioms of $\mathcal{O}$ into hyperedges between the newly formed nodes that correspond to atoms of $\mathcal{O}$. The cADH is the quotient hypergraph (Definition 2.41 in Section 2.4) $\mathcal{H}_\mathcal{O}^x / \simeq_B$

of $\mathcal{H}_\mathcal{O}^x$ under the strong- B -connectivity relation $\simeq_B$ in $\mathcal{H}_\mathcal{O}$. The $\simeq_B$ -equivalence classes are the strongly- B -connected components of $\mathcal{H}_\mathcal{O}^x$.

Definition 5.17 (Condensed Axiom Dependency Hypergraph). *Let $\mathcal{H}_\mathcal{O}^x$ be the x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\simeq_B$ be the strong- B -connectivity relation in $\mathcal{H}_\mathcal{O}^x$.*

The condensed axiom dependency hypergraph for $\mathcal{O}$ wrt. x -locality (x -cADH) is defined as the quotient $\mathcal{H}_\mathcal{O}^x / \simeq_B$ of $\mathcal{H}_\mathcal{O}^x$ under $\simeq_B$. $\dashv$

Similarly, it is also possible to compute the *partially condensed* ADH (pcADH) of an ADH. The idea is to identify and collapse the strongly-connected components of the graph fragment (Definition 2.31 in Section 2.4) of the ADH such that only simple B -hyperedges are considered ($|T(e)| = 1$). The hyperedges of the ADH are re-calculated to consider the newly formed nodes.

Definition 5.18 (Partially Condensed Axiom Dependency Hypergraph). *Let $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ be the x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G}_\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$ be a directed graph such that $\mathcal{V}_\mathcal{H} = \mathcal{V}_\mathcal{O}^x$ and $\mathcal{E}_\mathcal{H} = \{(T(e), H(e)) \in \mathcal{E}_\mathcal{O}^x \mid |T(e)| = 1\}$. Let $\simeq_\mathcal{G}$ be the strong-connectivity relation in $\mathcal{G}_\mathcal{H}$.*

The partially condensed axiom dependency hypergraph for $\mathcal{O}$ wrt. x -locality (x -pcADH) is defined as the quotient $\mathcal{H}_\mathcal{O}^x / \simeq_\mathcal{G}$ of $\mathcal{H}_\mathcal{O}^x$ under $\simeq_\mathcal{G}$. $\dashv$

The following proposition establishes the relation between B -connectivity in an ADH and B -connectivity in the cADH. This is done by showing that a B -connected component in an ADH is equivalent to a B -connected component in the cADH after the nodes of the B -connected component in the cADH are expanded to retrieve the original nodes that were defined in the ADH.

Proposition 5.19. *Let $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ be the x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\geq_B$ be the B -connectivity relation in $\mathcal{H}_\mathcal{O}^x$ and let $\simeq_B$ be the strong- B -connectivity relation in $\mathcal{H}_\mathcal{O}^x$.*

Then for $\mathcal{V}' \subseteq \mathcal{V}_\mathcal{O}^x$: $\geq_B(\mathcal{V}') = \bigcup_{\mathcal{H} / \simeq_B} ([\mathcal{V}']_{\simeq_B})$, where $[\mathcal{V}']_{\simeq_B} = \{[v]_{\simeq_B} \mid v \in \mathcal{V}'\}$. $\dashv$

The proof is similar to the proof of Proposition 2.43.

The following proposition establishes the relation between B -connectivity in an hypergraph and B -connectivity in the pcADH. This is done like in the case of cADHs.

Proposition 5.20. *Let $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ be the x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\geq_B$ be the B -connectivity relation in $\mathcal{H}_\mathcal{O}^x$ and let $\simeq_B$ be the strong- B -connectivity relation in $\mathcal{H}_\mathcal{O}^x$. Let $\mathcal{G}_\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$ be a directed graph such that $\mathcal{V}_\mathcal{H} = \mathcal{V}_\mathcal{O}^x$ and*

$\mathcal{E}_{\mathcal{H}} = \{(T(e), H(e)) \in \mathcal{E}_{\mathcal{O}}^x \mid |T(e)| = 1\}$. Let $\mathcal{H}_{\mathcal{O}}^x / \simeq_{\mathcal{G}} = (\mathcal{V}_{\simeq_{\mathcal{G}}}, \mathcal{E}_{\simeq_{\mathcal{G}}})$ be the pcADH of $\mathcal{H}_{\mathcal{O}}^x$, where $\simeq_{\mathcal{G}}$ is the strong-connectivity relation in $\mathcal{G}_{\mathcal{H}}$.

Then for $\mathcal{V}' \subseteq \mathcal{V}_{\mathcal{O}}^x$: $\geq_B(\mathcal{V}') = \bigcup \geq_{\mathcal{H}} / \simeq_{\mathcal{G}} ([\mathcal{V}']_{\simeq_{\mathcal{G}}})$, where $[\mathcal{V}']_{\simeq_{\mathcal{G}}} = \{[v]_{\simeq_{\mathcal{G}}} \mid v \in \mathcal{V}'\}$. $\dashv$

The proof is similar to the proof of Proposition 3.28.

As a corollary we generalise Proposition 5.9 by establishing that a module corresponds to the connected component in the (partially) condensed ADH for the ontology.

Corollary 5.21. *Let $\mathcal{O}$ be an ontology, Σ a signature and $x \in \{\perp, \top\}$. Let $\mathcal{O}' \subseteq \mathcal{O}$ be the set of axioms from $\mathcal{O}$ that are not x -local wrt. Σ . Let $\mathcal{H}_{\mathcal{O}}^x$ be the x -ADH for $\mathcal{O}$ and let $\geq_B$ and $\simeq_B$ be the B -connectivity and strong- B -connectivity relation, respectively, of $\mathcal{H}_{\mathcal{O}}^x$. Let $\simeq_{\mathcal{G}}$ be the strong-connectivity relation in the graph fragment of $\mathcal{H}_{\mathcal{O}}^x$. Let $\geq_{B(\mathcal{H}_{\simeq_B})}$ and $\geq_{B(\mathcal{H}_{\simeq_{\mathcal{G}}})}$ be the B -connectivity relation, respectively, of the Σ -extension of the x -cADH and x -pcADH for $\mathcal{H}_{\mathcal{O}}^x$. Let $\text{Mod}^x(\Sigma, \mathcal{O})$ be the function depicted in Figure 4.1.*

Then:

- (i) $\text{Mod}^x(\Sigma, \mathcal{O}) = \bigcup \geq_{B(\mathcal{H}_{\simeq_B})} ([\mathcal{O}']_{\simeq_B})$, where $[\mathcal{O}']_{\simeq_B} = \{[\alpha]_{\simeq_B} \mid \alpha \in \mathcal{O}'\}$.
- (ii) $\text{Mod}^x(\Sigma, \mathcal{O}) = \bigcup \geq_{B(\mathcal{H}_{\simeq_{\mathcal{G}}})} ([\mathcal{O}']_{\simeq_{\mathcal{G}}})$, where $[\mathcal{O}']_{\simeq_{\mathcal{G}}} = \{[\alpha]_{\simeq_{\mathcal{G}}} \mid \alpha \in \mathcal{O}'\}$. $\dashv$

Proof. To show item (i), it is sufficient to show that $\alpha \in \text{Mod}^x(\Sigma, \mathcal{O})$ iff $[\alpha]_{\simeq_B} \in \geq_{B(\mathcal{H}_{\simeq_B})}(\mathcal{V})$, where $\mathcal{V} = \{[\beta]_{\simeq_B} \mid \beta \in \mathcal{O}'\}$.

“ $\subseteq$ ”: We show the contrapositive. Suppose that $[\alpha]_{\simeq_B} \notin \geq_{B(\mathcal{H}_{\simeq_B})}(\mathcal{V})$. It follows by the definition of quotient hypergraph (Definition 2.41) that $\alpha \notin \geq_B(\mathcal{O}')$. By Proposition 5.6, $\alpha \notin \text{Mod}^x(\Sigma, \mathcal{O})$.

“ $\supseteq$ ”: Suppose $[\alpha]_{\simeq_B} \in \geq_{B(\mathcal{H}_{\simeq_B})}(\mathcal{V})$. It follows by Definition 2.41 that $\alpha \in \geq_B(\mathcal{O}')$. By Proposition 5.6, $\alpha \in \text{Mod}^x(\Sigma, \mathcal{O})$.

Item (ii) can be shown similarly. $\square$

The dependency relation $\succ_{\mathcal{O}}^x$ between x -local atoms of $\mathcal{O}$, for $x \in \{\perp, \top\}$, is defined as follows [14]. For atoms $\mathfrak{a}, \mathfrak{b} \in \text{Atoms}_{\mathcal{O}}^x$ and axioms $\alpha \in \mathfrak{a}$ and $\beta \in \mathfrak{b}$: $\mathfrak{a} \succ_{\mathcal{O}}^x \mathfrak{b}$ if, and only if, $\mathfrak{b} \subseteq \text{Mod}^x(\text{sig}(\alpha), \mathcal{O})$ if, and only if, $\text{Mod}^x(\text{sig}(\beta), \mathcal{O}) \subseteq \text{Mod}^x(\text{sig}(\alpha), \mathcal{O})$.

The following proposition establishes a correspondence between the notions of dependency between atoms and B -connectivity from *individual nodes* in the cADH.

Proposition 5.22. *Let $\mathcal{O}$ be an ontology with $\alpha, \beta \in \mathcal{O}$. Let $\mathfrak{a}, \mathfrak{b} \in \text{Atoms}_{\mathcal{O}}^x$ such that $\alpha \in \mathfrak{a}$ and $\beta \in \mathfrak{b}$, where $x \in \{\perp, \top\}$. Let $\mathcal{H}_{\mathcal{O}}^x$ be the x -ADH for $\mathcal{O}$ and let $\geq_B$ and $\simeq_B$ be the B -connectivity and strong- B -connectivity relation, respectively, of $\mathcal{H}_{\mathcal{O}}^x$. Let $\geq_{B(\mathcal{H}_{\simeq_B})}$ be the B -connectivity relation of the x -cADH for $\mathcal{H}_{\mathcal{O}}^x$.*

Then: $\mathfrak{a} \succ_{\mathcal{O}}^x \mathfrak{b}$ iff $[\alpha]_{\simeq_B} \geq_{B(\mathcal{H}_{\simeq_B})} [\beta]_{\simeq_B}$. $\dashv$

Proof. We first show the left-to-right direction. Let $\alpha \succ_{\mathcal{O}}^x \beta$, i.e., β depends on α in $\mathcal{O}$. It holds that $\alpha \subseteq \text{Mod}^x(\text{sig}(\alpha), \mathcal{O})$; cf. [14]. It follows from $\alpha \succ_{\mathcal{O}}^x \beta$ that $\beta \subseteq \text{Mod}^x(\text{sig}(\alpha), \mathcal{O})$. By Proposition 5.6, it follows that $\beta \subseteq \geq_B(\alpha)$. In particular, we have that $\beta \in \geq_B(\alpha)$. We have to show that $[\alpha]_{\simeq_B} \geq_{B(\mathcal{H}_{\simeq_B})} [\beta]_{\simeq_B}$. Suppose it does not hold. It follows from the definition of quotient hypergraph (Definition 2.41) that β is not B -connected from α in $\mathcal{H}_{\mathcal{O}}^x$, i.e. $\beta \notin \geq_B(\alpha)$; a contradiction.

Consider the direction from right to left. Suppose $[\alpha]_{\simeq_B} \geq_{B(\mathcal{H}_{\simeq_B})} [\beta]_{\simeq_B}$. By Definition 2.41, we have that $\beta \in \geq_B(\alpha)$. We have to show that $\alpha \succ_{\mathcal{O}}^x \beta$. Suppose not, i.e. β does not depend on α . Then there is an axiom $\gamma \in \mathcal{O}$ such that $\alpha \subseteq \text{Mod}^x(\text{sig}(\gamma), \mathcal{O})$ but $\beta \not\subseteq \text{Mod}^x(\text{sig}(\gamma), \mathcal{O})$; cf. [14]. It follows by Proposition 5.6 that $\alpha \in \geq_B(\gamma)$ and $\beta \notin \geq_B(\gamma)$. But from $\beta \in \geq_B(\alpha)$ and $\alpha \in \geq_B(\gamma)$ it follows by transitivity of $\geq_B$ that $\beta \in \geq_B(\gamma)$; a contradiction. This concludes the proof. $\square$

Recall that an approximating graph of a directed hypergraph (Definition 3.1) is a directed graph that preserves node-to-node B -connectivity in the directed hypergraph. By Proposition 6.49, we showed that the dependencies between pairs of atoms in the AD of an ontology are preserved in the B -connectivity relation of the x -cADH and vice-versa. We can conclude that the AD of an ontology correspond to the approximating graph of the x -cADH.

Corollary 5.23. *Let $\mathcal{O}$ be an ontology and let $\langle \text{Atoms}_{\mathcal{O}}^x, \succ_{\mathcal{O}} \rangle$ be the AD of $\mathcal{O}$. Let $\mathcal{H}_{\mathcal{O}}^x$ be the x -ADH for $\mathcal{O}$ and let $\simeq_B$ be the strong- B -connectivity relation of $\mathcal{H}_{\mathcal{O}}^x$. Let $\mathcal{H}_{\mathcal{O}}^x / \simeq_B$ be the x -cADH of $\mathcal{H}_{\mathcal{O}}^x$.*

Then: $\langle \text{Atoms}_{\mathcal{O}}^x, \succ_{\mathcal{O}} \rangle$ corresponds to the approximating graph of $\mathcal{H}_{\mathcal{O}}^x / \simeq_B$. $\dashv$

Proof. This corollary follows from Proposition 6.49 and Definition 3.1. $\square$

Example 5.24. *Let $\mathcal{O} = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$ be an ontology, where $\alpha_1 = A \sqsubseteq B$, $\alpha_2 = B \sqcap C \sqcap D \sqsubseteq E$, $\alpha_3 = E \sqsubseteq A \sqcap C \sqcap D$, $\alpha_4 = A \sqsubseteq X$ and $\alpha_5 = X \sqsubseteq A$. The signatures and the minimal non- $\perp$ -locality signatures of the axioms of $\mathcal{O}$ are the following:*

$$\begin{array}{ll} \text{MLS}^{\perp}(\alpha_1) = \{\{A\}\} & \text{sig}(\alpha_1) = \{A, B\} \\ \text{MLS}^{\perp}(\alpha_2) = \{\{B, C, D\}\} & \text{sig}(\alpha_2) = \{B, C, D, E\} \\ \text{MLS}^{\perp}(\alpha_3) = \{\{E\}\} & \text{sig}(\alpha_3) = \{A, C, D, E\} \\ \text{MLS}^{\perp}(\alpha_4) = \{\{A\}\} & \text{sig}(\alpha_4) = \{A, X\} \\ \text{MLS}^{\perp}(\alpha_5) = \{\{X\}\} & \text{sig}(\alpha_5) = \{A, X\} \end{array}$$

The $\perp$ -ADH $\mathcal{H}_\mathcal{O}^\perp$ contains the following hyperedges:

$$\begin{aligned} e_1 &= (\{\alpha_1, \alpha_3\}, \{\alpha_2\}) & e_2 &= (\{\alpha_1\}, \{\alpha_4\}) & e_3 &= (\{\alpha_2\}, \{\alpha_3\}) & e_4 &= (\{\alpha_3\}, \{\alpha_1\}) \\ e_5 &= (\{\alpha_3\}, \{\alpha_4\}) & e_6 &= (\{\alpha_4\}, \{\alpha_1\}) & e_7 &= (\{\alpha_4\}, \{\alpha_5\}) & e_8 &= (\{\alpha_5\}, \{\alpha_1\}) \\ e_9 &= (\{\alpha_5\}, \{\alpha_4\}) \end{aligned}$$

We obtain the following $\perp$ -local modules for the signature of each axiom:

$$\begin{aligned} \text{Mod}^\perp(\text{sig}(\alpha_1), \mathcal{O}) &= \{\alpha_1, \alpha_4, \alpha_5\} & \text{Mod}^\perp(\text{sig}(\alpha_4), \mathcal{O}) &= \{\alpha_1, \alpha_4, \alpha_5\} \\ \text{Mod}^\perp(\text{sig}(\alpha_2), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\} & \text{Mod}^\perp(\text{sig}(\alpha_5), \mathcal{O}) &= \{\alpha_1, \alpha_4, \alpha_5\} \\ \text{Mod}^\perp(\text{sig}(\alpha_3), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\} \end{aligned}$$

The resulting atoms in $\text{Atoms}_\mathcal{O}^\perp$ are $\mathbf{a}_1 = \{\alpha_2, \alpha_3\}$ and $\mathbf{a}_2 = \{\alpha_1, \alpha_4, \alpha_5\}$, where $\mathbf{a}_1 \succ_\mathcal{O}^\perp \mathbf{a}_2$, i.e. $\mathbf{a}_2$ depends on $\mathbf{a}_1$. The ADH $\mathcal{H}_\mathcal{O}^\perp$ with the SCCs and the condensed ADH $\mathcal{H}_\mathcal{O}^\perp / \simeq_B$ is depicted in Figure 5.5.

Fig. 5.5 Example 5.24: From the $\perp$ -ADH to the condensed $\perp$ -ADH

Consider the strongly-connected components of $\mathcal{H}_\mathcal{O}^\perp$. Axiom α_1 is B -connected with the axioms α_4 and α_5 , α_4 is B -connected with α_1 and α_5 , and α_5 is B -connected with α_1 and α_4 . Axiom α_2 is B -connected with α_3 and vice versa. Axioms α_2, α_3 are each B -connected with α_1, α_4 and α_5 , but not vice versa. Hence, $\{\alpha_1, \alpha_4, \alpha_5\}$ and $\{\alpha_2, \alpha_3\}$ are the strongly-connected components of $\mathcal{H}_\mathcal{O}^\perp$. Moreover, we say that the former component depends on the latter as any two axioms contained in them are unilaterally and not mutually B -connected. Note that the atoms $\mathbf{a}_1$ and $\mathbf{a}_2$ of $\mathcal{O}$ and their dependency coincide with the strongly-connected components of $\mathcal{H}_\mathcal{O}^\perp$. $\triangleleft$

In Example 5.12, we observed the notions of $\top$ -locality and $\top$ -reachability might produce different modules using the same ontology and signature. According to [34], $\top$ -reachability is

Fig. 5.6 Example 5.25: $\top$ -locality vs. $\top$ -reachability

equivalent to the F -connectivity relation in a directed hypergraph. In the following example, we show that $\top$ -reachability is not transitive and the atomic decomposition computed under this notion might not coincide with the atomic decomposition using the notion of $\top$ -locality.

Example 5.25. Let $\mathcal{O} = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ be the ontology of Example 5.12, where $\alpha_1 = A \sqsubseteq \exists r.D_1$, $\alpha_2 = B \sqsubseteq (\geq nr.D_2)$, for $n \geq 0$, $\alpha_3 = \exists r.\top \sqsubseteq C$ and $\alpha_4 = D_1 \sqsubseteq D_2$. Using the notion of $\top$ -locality which corresponds to B -connectivity in a $\top$ -ADH, we obtain the following modules for the signature of each axiom, and from them the following two atoms $\mathfrak{a}_1$ and $\mathfrak{a}_2$:

$$\begin{aligned}
 \text{Mod}^\top(\text{sig}(\alpha_1), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_4\} & \mathfrak{a}_1 &= \{\alpha_1, \alpha_2, \alpha_4\} \\
 \text{Mod}^\top(\text{sig}(\alpha_2), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_4\} & \mathfrak{a}_2 &= \{\alpha_3\} \\
 \text{Mod}^\top(\text{sig}(\alpha_3), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\} \\
 \text{Mod}^\top(\text{sig}(\alpha_4), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_4\}
 \end{aligned}$$

The AD of $\mathcal{O}$ is a poset $\langle \text{Atoms}_\mathcal{O}^\top, \succ_\mathcal{O}^\top \rangle$, where $\text{Atoms}_\mathcal{O}^\top = \{\mathfrak{a}_1, \mathfrak{a}_2\}$ represents the set of atoms of $\mathcal{O}$ and $\mathfrak{a}_2 \succ_\mathcal{O}^\top \mathfrak{a}_1$ indicates that atom $\mathfrak{a}_1$ depends on atom $\mathfrak{a}_2$ (cf. Definition 4.12).

The $\top$ -ADH of ontology $\mathcal{O}$ with the SCCs marked in red is depicted in Figure 5.6 (a). We can observe that the nodes α_1 , α_2 and α_4 are strongly- B -connected and therefore, they are in the same strongly- B -connected component. The strongly- B -connected component $[\alpha_1]_{\simeq B}$ corresponds to the atom $\mathfrak{a}_1$ and $[\alpha_3]_{\simeq B}$ corresponds to the atom $\mathfrak{a}_2$.

Using the notion of $\top$ -reachability instead of $\top$ -locality, we obtain the following modules for the signature of each axiom, and from them the following three atoms α'_1 , α'_2 and α'_3 :

$$\begin{aligned} \text{Mod}^\Delta(\text{sig}(\alpha_1), \mathcal{O}) &= \{\alpha_1, \alpha_2\} & \alpha'_1 &= \{\alpha_1\} \\ \text{Mod}^\Delta(\text{sig}(\alpha_2), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_4\} & \alpha'_2 &= \{\alpha_2, \alpha_4\} \\ \text{Mod}^\Delta(\text{sig}(\alpha_3), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_3\} & \alpha'_3 &= \{\alpha_3\} \\ \text{Mod}^\Delta(\text{sig}(\alpha_4), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_4\} \end{aligned}$$

We can observe that two modules are different when the notion of $\top$ -reachability is used. These are the modules for the signature of α_1 and α_3 . Because the modules are different, the atoms and their dependencies are also different: $\text{Atoms}_{\mathcal{O}}^\Delta = \{\alpha'_1, \alpha'_2, \alpha'_3\}$, $\alpha'_2 \succ_{\mathcal{O}}^\Delta \alpha'_1$ and $\alpha'_3 \succ_{\mathcal{O}}^\Delta \alpha'_1$.

Note that by $\top$ -reachability, it is possible to reach α_2 from α_1 or α_3 . Moreover, α_4 is $\top$ -reachable from α_2 . However, α_4 is not $\top$ -reachable from α_1 or α_3 . So, it seems that transitivity is not preserved for the notion of $\top$ -reachability. To illustrate this problem, we show in Figure 5.6 (b) the $\top$ -ADH of ontology $\mathcal{O}$ with the SCCs computed according to $\top$ -reachability. The SCC $[\alpha_1]_{\sim_B}$ corresponds to the atom α'_1 , $[\alpha_2]_{\sim_B}$ to α'_2 and $[\alpha_3]_{\sim_B}$ to α'_3 . $\triangleleft$

5.4 Summary

In this chapter, we introduce the notion of axiom dependency hypergraph which is a directed hypergraph that generalizes the (labelled) atomic decomposition of an ontology. A module in the ontology corresponds to a connected component in the ADH, and the atoms of the ontology to the strongly- B -connected components. The AD of an ontology also includes the dependency relations between pair of atoms of an ontology. These dependency relations correspond to the approximating graph of a condensed ADH.

The ADH of an ontology does not provide enough information to compute the locality-based module for any signature Σ . To overcome this limitation, we incorporate Σ to the ADH of an ontology. This produces the Σ -extension of the ADH where the locality-based module for Σ can now be computed.

In this chapter, we also compare the notions of $\top$ -reachability [34] and $\top$ -locality. Using an ADH, we show that $\top$ -reachability is not transitive and the atomic decomposition computed under this notion might not coincide with the atomic decomposition using the notion of $\top$ -locality.

From the implementation point of view, it might not be feasible to represent an ADH as a directed hypergraph. By Proposition 5.4, we have shown that an ADH might contain expo-

nentially many hyperedges with respect to the number of axioms of an ontology. Instead of using a directed hypergraph to represent an ADH, we can use the MLSs and signatures of the axioms of an ontology as a succinct representation of the ADH. This compact representation is defined and studied in Chapter 6.

Chapter 6

HyS

In this chapter, we introduce a labelled graph representation of an axiom dependency hypergraph [27]. This representation is more compact and facilitates the implementation of applications for axiom dependency hypergraphs. We also show that existent normalization rules [33, 34] for OWL 2 ontologies might produce incorrect results when the atomic decomposition of the normalized ontology and their modules are computed. Based on the algorithms for traversing directed hypergraphs that we discussed in Chapters 2 & 3, we define two algorithms for traversing axiom dependency hypergraphs encoded in directed labelled graphs. One algorithm traverses a labelled graph in a Breadth-first search fashion and the other is very similar to the Depth-first search algorithm presented in Chapter 3. We show how these algorithms can be also used to compute the strongly-connected components of an axiom dependency hypergraph encoded in a labelled graph. For all the algorithms presented in this chapter, we show their correctness and complexity. To test the suitability of these algorithms in real ontologies, we have implemented the application HyS for computing the atomic decomposition and the modules of ontologies. We provide an experimental evaluation of HyS with a selection of large and prominent biomedical ontologies, most of which are available in the NCBO Bioportal.

Chapter 6 is organized as follows. Section 6.1 introduces the notion of a directed labelled graph representation of an axiom dependency hypergraph. It is shown that the connectivity in the labelled graph corresponds to B -connectivity in the axiom dependency hypergraph. Based on this result, it is discussed how to compute (strongly-) B -connected components in the labelled graph. Section 6.2 discusses if existent normalization rules [33, 34] for OWL 2 ontologies might alter B -connectivity of the axiom dependency hypergraph after the ontology is normalized. In Section 6.3, we introduces two algorithms for traversing directed labelled graphs. Based on these algorithms, we show how to compute syntactic local modules and the atomic decomposition of an ontology in Sections 6.4 & 6.5. We describe

in Section 6.6 the application *HyS* for extracting locality-based modules and compute the atomic decomposition of OWL-EL ontologies. In Section 6.7, we evaluate *HyS* against state-of-the-art implementations for the computation of modules and the atomic decomposition of OWL 2 ontologies. A short summary of this chapter is provided in Section 6.8.

6.1 Directed Labelled Graph Representation

The number of hyperedges of an ADH may be exponential in the size of the input ontology [25] (cf. Proposition 5.4), which makes it impractical to represent the entire ADH explicitly. We implement an ADH $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ of an ontology $\mathcal{O}$ for the syntactic locality notion $x \in \{\perp, \top\}$, as a directed labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ containing the simple hyperedges ($|T(e)| = 1$) of $\mathcal{H}_\mathcal{O}^x$ and encoding the complex hyperedges ($|T(e)| > 1$) in the node labels as follows. A node v_α in $\mathcal{G}$ for an axiom α is labelled with the pair $\mathcal{L}(v_\alpha) = (\text{MLS}^x(\alpha), \text{sig}(\alpha))$ consisting of the minimal non- x -locality signatures of α and the signature of α . In fact, not all symbols of $\text{sig}(\alpha)$ are needed in the second component, only those symbols that occur in the minimal non-locality signature of some axiom in the ontology. Condensed (or partially condensed) axiom dependency hypergraphs are implemented in a similar way with the difference that nodes represent sets of axioms. A node v_S for a set S of axioms is labelled with the pair $\mathcal{L}(v_S) = (\text{MLS}^x(S), \text{sig}(S))$, where $\text{MLS}^x(S) = \bigcup_{\alpha \in S} \text{MLS}^x(\alpha)$ and $\text{sig}(S) = \bigcup_{\alpha \in S} \text{sig}(\alpha)$.

We now introduce the notion of a labelled graph representation of a possibly (partially) condensed axiom dependency hypergraph.

Definition 6.1 (Labelled graph-encoding of an ADH). *Let $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ be an ADH, pcADH or cADH of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$ is a syntactic locality notion. The graph representation $\mathcal{G}$ of $\mathcal{H}_\mathcal{O}^x$ encodes $\mathcal{H}_\mathcal{O}^x$ in a directed labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$, where*

- $\mathcal{V} := \mathcal{V}_\mathcal{O}^x$;
- $\mathcal{E} := \{(v, v') \mid \Sigma_{v'} \subseteq \text{sig}(v), \text{ for some } \Sigma_{v'} \in \text{MLS}^x(v')\}$;
- $\mathcal{L}(v) := (\text{MLS}^x(v), \text{sig}(v))$, for every $v \in \mathcal{V}$. ⊣

To define the labelled graph-encoding $\mathcal{G}$ of an ADH $\mathcal{H}_\mathcal{O}^x$, we assume that every node v in $\mathcal{H}_\mathcal{O}^x$ is associated with a set $\text{MLS}^x(v)$ of minimal non-locality signatures, and a set $\text{sig}(v)$ of signature symbols. Intuitively, every node in an ADH can be thought of requiring a signature in order to be reachable, and of providing a signature to enable reachability of other nodes.

Note that a node in $\mathcal{H}_{\mathcal{O}}^x$ represents an axiom if $\mathcal{H}_{\mathcal{O}}^x$ is an ADH, and a set of axioms if $\mathcal{H}_{\mathcal{O}}^x$ is a pcADH or cADH.

Proposition 6.2. *Let $\mathcal{H}_{\mathcal{O}}^x = (\mathcal{V}_{\mathcal{O}}^x, \mathcal{E}_{\mathcal{O}}^x)$ be the x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of $\mathcal{H}_{\mathcal{O}}^x$. Then: $e = (T(e), H(e)) \in \mathcal{E}_{\mathcal{O}}^x$ with $H(e) = \{v_h\}$ iff $\Sigma_{v_h} \subseteq \bigcup_{v \in T(e)} \text{sig}(v)$, for some $\Sigma_{v_h} \in \text{MLS}^x(v_h)$. $\dashv$*

Proof. This follows from Def. 5.1, Prop. 4.10 and Def. 6.1. $\square$

The following example illustrates the notion of directed labelled graph for a $\perp$ -ADH.

Example 6.3. *Let $\mathcal{O} = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$ be an ontology, where $\alpha_1 = F \sqsubseteq H$, $\alpha_2 = B \sqsubseteq A \sqcap F \sqcap G$, $\alpha_3 = A \sqsubseteq B \sqcap C \sqcap E \sqcap F$, $\alpha_4 = C \sqcap G \sqsubseteq D$ and $\alpha_5 = D \sqcap E \sqsubseteq A$. The signatures and minimal non- $\perp$ -locality signatures of the axioms of $\mathcal{O}$ are the following.*

$$\begin{aligned} \text{MLS}^{\perp}(\alpha_1) &= \{\{F\}\} & \text{sig}(\alpha_1) &= \{F, H\} \\ \text{MLS}^{\perp}(\alpha_2) &= \{\{B\}\} & \text{sig}(\alpha_2) &= \{A, B, F, G\} \\ \text{MLS}^{\perp}(\alpha_3) &= \{\{A\}\} & \text{sig}(\alpha_3) &= \{A, B, C, E, F\} \\ \text{MLS}^{\perp}(\alpha_4) &= \{\{C, G\}\} & \text{sig}(\alpha_4) &= \{C, D, G\} \\ \text{MLS}^{\perp}(\alpha_5) &= \{\{D, E\}\} & \text{sig}(\alpha_5) &= \{A, D, E\} \end{aligned}$$

$\mathcal{H}_{\mathcal{O}}^{\perp} = (\mathcal{V}_{\mathcal{O}}^{\perp}, \mathcal{E}_{\mathcal{O}}^{\perp})$ is the $\perp$ -ADH of $\mathcal{O}$, where $\mathcal{V}_{\mathcal{O}}^{\perp} = \mathcal{O}$ and $\mathcal{E}_{\mathcal{O}}^{\perp}$ has the following hyperedges:

$$\begin{aligned} e_1 &= (\{\alpha_2\}, \alpha_1) & e_2 &= (\{\alpha_2\}, \alpha_3) \\ e_3 &= (\{\alpha_3\}, \alpha_1) & e_4 &= (\{\alpha_3\}, \alpha_2) \\ e_5 &= (\{\alpha_4\}, \alpha_2) & e_6 &= (\{\alpha_5\}, \alpha_3) \\ e_7 &= (\{\alpha_2, \alpha_3\}, \alpha_4) & e_8 &= (\{\alpha_3, \alpha_4\}, \alpha_5) \end{aligned}$$

$\mathcal{H}_{\mathcal{O}}^{\perp}$ is depicted in Figure 6.1 (a) and its labelled graph representation, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$, in Figure 6.1 (b). We can observe that $\mathcal{V} = \mathcal{V}_{\mathcal{O}}^{\perp}$ and $\mathcal{E} = \{e \in \mathcal{E}_{\mathcal{O}}^{\perp} \mid |T(e)| = 1\}$. Each node $v \in \mathcal{V}$ corresponds to an axiom of $\mathcal{O}$ and it is labelled with the signature and minimal non- $\perp$ -locality signatures of the related axiom. $\dashv$

We now define the notion of Σ -connectivity that inductively establishes when a node is Σ -connected from a node or a set of nodes in a labelled graph.

Definition 6.4. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of an ADH, pcADH or a cADH. Let Σ be a signature. The Σ -connectivity relation $\geq_{\Sigma(\mathcal{G})}: 2^{\mathcal{V}} \times 2^{\mathcal{V}}$ in $\mathcal{G}$ wrt. Σ is defined inductively as follows: For $\mathcal{V}_0 \subseteq \mathcal{V}$,*

- (i) $v \in \geq_{\Sigma(\mathcal{G})}(\mathcal{V}_0)$, for all $v \in \mathcal{V}_0$; and

Fig. 6.1 (Example 6.10): Labelled graph encoding of $\mathcal{H}_{\mathcal{O}}^{\perp}$.

(ii) $v \in \geq_{\Sigma(\mathcal{G})}(\mathcal{V}_0)$ if $\Sigma_v \subseteq \Sigma \cup \bigcup_{v' \in \geq_{\Sigma(\mathcal{G})}(\mathcal{V}_0)} \text{sig}(v')$ for some $\Sigma_v \in \text{MLS}^x(v)$. $\dashv$

The set inclusion

$$\Sigma_v \subseteq \Sigma \cup \bigcup_{s \in \mathcal{S}} \text{sig}(s) \text{ for some } \Sigma_v \in \text{MLS}^x(v)$$

represents the state of node v being B -connected from the set $\mathcal{S}$ of nodes in the labelled graph representation of an ADH. A B -connected component can be computed without having an explicit representation of the hyperedges of the ADH. It is sufficient to check for signature containment.

Note that the signature Σ is also part of item (ii) of Definition 6.4. This implies that if Σ is not empty, we can compute in the labelled graph that encodes an ADH a B -connected component of the Σ -extension of the ADH without modifying the labelled graph (cf. Proposition 6.7).

The following proposition establishes the relation between B -connectivity in an ADH and Σ -connectivity in the labelled graph-encoding of the ADH.

Proposition 6.5. *Let $\mathcal{H}_{\mathcal{O}}^x = (\mathcal{V}_{\mathcal{O}}^x, \mathcal{E}_{\mathcal{O}}^x)$ be the x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\geq_B$ be the B -connectivity relation in $\mathcal{H}_{\mathcal{O}}^x$. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of $\mathcal{H}_{\mathcal{O}}^x$ and let $\geq_{\Sigma}$ be the Σ -connectivity relation in $\mathcal{G}$ wrt. Σ , where $\Sigma = \emptyset$. Then for $\mathcal{V}_0 \subseteq \mathcal{V}_{\mathcal{O}}^x$: $\geq_B(\mathcal{V}_0) = \geq_{\Sigma}(\mathcal{V}_0)$. $\dashv$*

Proof. We show this proposition by induction using Definition 6.4 and Proposition 6.2.

“ $\subseteq$ ”: Let $\beta \in \mathcal{O}$ such that $\mathcal{V}_0 \geq_B \beta$. We show that (IH) $\mathcal{V}_0 \geq_B \beta$ implies $\beta \in \geq_{\Sigma}(\mathcal{V}_0)$ by induction on the maximal length $n = \text{dist}_{\mathcal{H}}(\mathcal{V}_0, \beta)$ of a simple $\geq$ -admissible hyperpath (cf. Definition 3.12) from an axiom α in $\mathcal{V}_0$ to β . For the induction base, $n = 1$. We have that

$\beta \in \mathcal{V}_0$. By Definition 6.4, $\beta \in \geq_\Sigma(\mathcal{V}_0)$. Suppose we have shown (IH) for n . Consider the induction step where $n \rightarrow n+1$. By definition of B -connectivity (cf. Definition 2.32) we have that either (a) $\beta \in \mathcal{V}_0$, or (b) there is a hyperedge e in $\mathcal{H}_\mathcal{O}^x$ with $H(e) = \{\beta\}$ and $\mathcal{V}_0 \geq_B \alpha$, for all $\alpha \in T(e)$. For (a), we have already established that $\beta \in \geq_\Sigma(\mathcal{V}_0)$ for $n=1$. Consider (b). By (IH) we have that $\mathcal{V}_0 \geq_\Sigma \alpha$, for all $\alpha \in T(e)$. By Proposition 6.2, there is a $\Sigma_\beta \in \text{MLS}^x(\beta)$ such that $\Sigma_\beta \subseteq \bigcup_{\alpha \in T(e)} \text{sig}(\alpha)$. By Definition 6.4, $\beta \in \geq_\Sigma(\mathcal{V}_0)$.

The other direction can be shown similarly using induction. $\square$

Given Proposition 6.5, we obtain the following proposition showing that hyperedges are faithfully encoded in a labelled graph as to preserve node-to-node B -connectivity.

Proposition 6.6. *Let $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ be the x -ADH of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph representation of $\mathcal{H}_\mathcal{O}^x$. Let $\geq_B$ be the B -connectivity relation of $\mathcal{H}_\mathcal{O}^x$ and let $\geq_\Sigma$ be the Σ -connectivity relation in $\mathcal{G}$ wrt. Σ , where $\Sigma = \emptyset$.*

Then: $\{v\} \geq_B v'$ iff $\{v\} \geq_\Sigma v'$, for all $v, v' \in \mathcal{V}_\mathcal{O}^x$. $\dashv$

The following proposition shows that B -connectivity in the Σ -extension of an ADH equals Σ -connectivity in the labelled graph representation of the ADH. This means that we do not have to modify the labelled graph representation of an ADH for an ontology $\mathcal{O}$ to compute the B -connected component for any possible signature Σ .

Proposition 6.7. *Let $\mathcal{H}_\mathcal{O}^x$ be the x -ADH of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let Σ be a signature and let $\mathcal{H}_{\mathcal{O}, \Sigma}^x$ be the Σ -extension of $\mathcal{H}_\mathcal{O}^x$. Let $\mathcal{G}$ be the labelled graph representation of $\mathcal{H}_\mathcal{O}^x$. Let $\geq_B$ be the B -connectivity relation in $\mathcal{H}_{\mathcal{O}, \Sigma}^x$ and let $\geq_\Sigma$ be the Σ -connectivity relation in $\mathcal{G}$ wrt. Σ . Then: $\geq_B = \geq_\Sigma$.* $\dashv$

Proof. The proposition can readily be seen using Def. 6.4 and the definition of Σ -extension of an ADH (cf. Def. 5.1 but with Item (ii) replaced with Item (iii)). $\square$

We introduce the notion of (partial) condensation of a graph representation of an ADH.

Definition 6.8. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph representation of the x -ADH $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ be the graph fragment of $\mathcal{H}_\mathcal{O}^x$, a directed graph where $\mathcal{V}' = \mathcal{V}_\mathcal{O}^x$ and $\mathcal{E}' = \{e \in \mathcal{E}_\mathcal{O}^x \mid |T(e)| = 1\}$. Let $\simeq$ be the strong- B -connectivity relation in $\mathcal{H}_\mathcal{O}^x$ or the strong-connectivity relation in $\mathcal{G}'$. The quotient of $\mathcal{G}$ under $\simeq$, written $\mathcal{G}/\simeq$, is the labelled graph $\mathcal{G}/\simeq = (\mathcal{V}_\simeq, \mathcal{E}_\simeq, \mathcal{L}_\simeq)$, where*

- $\mathcal{V}_\simeq := \{[v]_\simeq \mid v \in \mathcal{V}\};$
- $\mathcal{E}_\simeq := \{([v]_\simeq, [v']_\simeq) \in \mathcal{V}_\simeq \times \mathcal{V}_\simeq \mid v \not\simeq v' \text{ and } \Sigma_{[v]_\simeq} \subseteq \text{sig}([v]_\simeq), \text{ for some } \Sigma_{[v]_\simeq} \in \text{MLS}^\perp([v']_\simeq)\};$

- $\mathcal{L}_\simeq([v]_\simeq) := (\bigcup_{v' \in [v]_\simeq} \text{MLS}^x(v'), \bigcup_{v' \in [v]_\simeq} \text{sig}(v')), \text{ for all } v \in \mathcal{V}.$ $\dashv$

The following proposition states that the graph representation of a (partially) condensed ADH is equivalent to the (partially) condensed graph representation of the ADH.

Proposition 6.9. *Let $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ be the x -ADH of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ be the graph fragment of $\mathcal{H}_\mathcal{O}^x$, a directed graph where $\mathcal{V}' = \mathcal{V}_\mathcal{O}^x$ and $\mathcal{E}' = \{e \in \mathcal{E}_\mathcal{O}^x \mid |T(e)| = 1\}$. Let $\simeq$ be the strong-B-connectivity relation in $\mathcal{H}_\mathcal{O}^x$ or the strong-connectivity relation in $\mathcal{G}'$.*

Let $\mathcal{G}$ be the labelled graph representation of $\mathcal{H}_\mathcal{O}^x$, and $\mathcal{G}_{\mathcal{H}/\simeq}$ be the labelled graph representation of $\mathcal{H}_\mathcal{O}^x/\simeq$. Then: $\mathcal{G}_{\mathcal{H}/\simeq} = \mathcal{G}/\simeq$. $\dashv$

Proof. It can readily be checked that the set of nodes are equivalent in $\mathcal{G}_{\mathcal{H}/\simeq}$ and $\mathcal{G}/\simeq$ using Def. 6.8, Def. 6.1 and Def. 2.41.

Let $v, v' \in \mathcal{V}_\mathcal{O}^x$. Suppose $([v]_\simeq, [v']_\simeq)$ is an edge in $\mathcal{G}_{\mathcal{H}/\simeq}$. By Def. 6.1, $e = (T(e), H(e))$ is a simple hyperedge in $\mathcal{H}_\mathcal{O}^x/\simeq$ such that $T(e) = \{[v]_\simeq\}$ and $H(e) = \{[v']_\simeq\}$. By Def. 5.17 and 2.41, there is a hyperedge $e' = (T(e'), H(e'))$ in $\mathcal{H}_\mathcal{O}^x$ such that $T(e) = \{[x]_\simeq \mid x \in T(e')\}$ and $H(e) = \{[x]_\simeq \mid x \in H(e')\}$, and $T(e) \cap H(e) = \emptyset$. Then, $[v]_\simeq \neq [v']_\simeq$ and $H(e') = \{v'\}$. By Def. 5.1 using Condition (iib) (cf. Section 5.1), $\Sigma_{v'} \subseteq \text{sig}(T(e'))$, for some $\Sigma_{v'} \in \text{MLS}^x(v')$. We show that there exists a signature $\Sigma' \in \text{MLS}^x([v']_\simeq)$ such that $\Sigma' \subseteq \text{sig}([v]_\simeq)$. Then $([v]_\simeq, [v']_\simeq)$ is an edge in $\mathcal{G}/\simeq$ by Def. 6.8, which concludes this direction. Set $\Sigma' = \Sigma_{v'}$. We have that $\Sigma' \in \text{MLS}^x([v']_\simeq)$ since $\Sigma_{v'} \in \text{MLS}^x(v')$ and $\text{MLS}^x(v') \subseteq \text{MLS}^x([v']_\simeq)$. It holds that $\Sigma' \subseteq \text{sig}([v]_\simeq)$ since $\Sigma_{v'} \subseteq \text{sig}(T(e'))$, $\text{sig}(T(e')) \subseteq \text{sig}([v]_\simeq)$ as $T(e') \subseteq [v]_\simeq$.

Consider the other direction. Suppose $([v]_\simeq, [v']_\simeq)$ is an edge in $\mathcal{G}/\simeq$. By Def. 6.8, we have that $\Sigma' \subseteq \text{sig}([v]_\simeq)$, for some $\Sigma' \in \text{MLS}^x([v']_\simeq)$. Take such a signature Σ' . This implies that there exists a $v'' \in [v']_\simeq$ such that $\Sigma' \in \text{MLS}^x(v'')$. Note that $v'' \notin [v]_\simeq$ as $v'' \in [v']_\simeq$ and $[v]_\simeq \neq [v']_\simeq$ by Def. 6.8. By Def. 5.1 using Condition (iib) (cf. Section 5.1), we have that there is a hyperedge $e = (T(e), H(e))$ in $\mathcal{H}_\mathcal{O}^x$ such that $T(e) = [v]_\simeq$ and $H(e) = \{v''\}$. By Def. 5.17 and 2.41, there is a hyperedge $e' = (T(e'), H(e'))$ in $\mathcal{H}_\mathcal{O}^x/\simeq$ such that $T(e') = \{[v]_\simeq\}$ and $H(e') = \{[v']_\simeq\}$. Note that $[v']_\simeq = [v']_\simeq$ as $v'' \in [v']_\simeq$. Thus, we have that $H(e') = \{[v']_\simeq\}$. It follows by Def. 6.1 that $([v]_\simeq, [v']_\simeq)$ is an edge in $\mathcal{G}_{\mathcal{H}/\simeq}$.

It remains to show that $\mathcal{L}_1 = \mathcal{L}_2$, where $\mathcal{L}_1$ is the labeling function of $\mathcal{G}_{\mathcal{H}/\simeq}$ and $\mathcal{L}_2$ is the labelling function of $\mathcal{G}/\simeq$. Suppose v is a node in $\mathcal{H}_\mathcal{O}^x$. Then $[v]_\simeq$ is a node in both graphs, $\mathcal{G}_{\mathcal{H}/\simeq}$ and $\mathcal{G}/\simeq$. Then $\mathcal{L}_1([v]_\simeq) = (\text{MLS}^x([v]_\simeq), \text{sig}([v]_\simeq))$ by Def. 6.1. By Def. 6.8 this is equivalent to $\mathcal{L}_2([v]_\simeq) = (\bigcup_{v' \in [v]_\simeq} \text{MLS}^x(v'), \bigcup_{v' \in [v]_\simeq} \text{sig}(v'))$. $\square$

The following example illustrates the notion of condensed directed labelled graph for a $\perp$ -ADH.

Example 6.10. Let $\mathcal{O}$ be the ontology of Example 6.10. The signatures and minimal non- $\perp$ -locality signatures of the axioms of $\mathcal{O}$ are the following.

$$\begin{aligned} MLS^\perp(\alpha_1) &= \{\{F\}\} & \text{sig}(\alpha_1) &= \{F, H\} \\ MLS^\perp(\alpha_2) &= \{\{B\}\} & \text{sig}(\alpha_2) &= \{A, B, F, G\} \\ MLS^\perp(\alpha_3) &= \{\{A\}\} & \text{sig}(\alpha_3) &= \{A, B, C, E, F\} \\ MLS^\perp(\alpha_4) &= \{\{C, G\}\} & \text{sig}(\alpha_4) &= \{C, D, G\} \\ MLS^\perp(\alpha_5) &= \{\{D, E\}\} & \text{sig}(\alpha_5) &= \{A, D, E\} \end{aligned}$$

From the signature of each axiom of $\mathcal{O}$, we obtain the following $\perp$ -local modules:

$$\begin{aligned} \text{Mod}^\perp(\text{sig}(\alpha_1), \mathcal{O}) &= \{\alpha_1\} & \text{Mod}^\perp(\text{sig}(\alpha_2), \mathcal{O}) &= \mathcal{O} \\ \text{Mod}^\perp(\text{sig}(\alpha_3), \mathcal{O}) &= \mathcal{O} & \text{Mod}^\perp(\text{sig}(\alpha_4), \mathcal{O}) &= \mathcal{O} \\ \text{Mod}^\perp(\text{sig}(\alpha_5), \mathcal{O}) &= \mathcal{O} \end{aligned}$$

From these modules, we can compute the following two atoms: $\mathbf{a}_1 = \{\alpha_1\}$ and $\mathbf{a}_2 = \mathcal{O} \setminus \{\alpha_1\}$, where $\mathbf{a}_2 \succ_{\mathcal{O}}^\perp \mathbf{a}_1$.

Let $\mathcal{H}_{\mathcal{O}}^\perp = (\mathcal{V}_{\mathcal{O}}^\perp, \mathcal{E}_{\mathcal{O}}^\perp)$ be the $\perp$ -ADH of $\mathcal{O}$ depicted Figure 6.1 (b) and $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph representation of $\mathcal{H}_{\mathcal{O}}^\perp$ depicted in Figure 6.1 (b).

Like the atoms of $\mathcal{O}$, the SCCs of $\mathcal{H}_{\mathcal{O}}^\perp$ are two: $[\alpha_1]_{\simeq}$ which corresponds to $\mathbf{a}_1$ and $[\alpha_2]_{\simeq}$ which corresponds to $\mathbf{a}_2$. Condensing $\mathcal{H}_{\mathcal{O}}^\perp$ produces the cADH $\mathcal{H}_{\mathcal{O}}^\perp / \simeq = (\mathcal{V}_{\mathcal{O}}^\perp / \simeq, \mathcal{E}_{\mathcal{O}}^\perp / \simeq)$, where $\mathcal{V}_{\mathcal{O}}^\perp / \simeq = \{[\alpha_1]_{\simeq}, [\alpha_2]_{\simeq}\}$ and $\mathcal{E}_{\mathcal{O}}^\perp / \simeq = \{e_{10} = (\{[\alpha_2]_{\simeq}\}, [\alpha_1]_{\simeq})\}$.

Condensing $\mathcal{G}$ produces the following condensed labelled graph: $\mathcal{G}_{\simeq} = (\mathcal{V}_{\simeq}, \mathcal{E}_{\simeq}, \mathcal{L}_{\simeq})$, where $\mathcal{V}_{\simeq} = \{[\alpha_1]_{\simeq}, [\alpha_2]_{\simeq}\}$, $\mathcal{E}_{\simeq} = \{e_{10} = (\{[\alpha_2]_{\simeq}\}, [\alpha_1]_{\simeq})\}$ and $\mathcal{L}_{\simeq} = \{\mathcal{L}([\alpha_1]_{\simeq}), \mathcal{L}([\alpha_2]_{\simeq})\}$. These are the labels of each node in $\mathcal{G}_{\simeq}$:

$$\begin{aligned} \mathcal{L}_1([\alpha_1]_{\simeq}) &= \{\{F\}\} & \mathcal{L}_2([\alpha_1]_{\simeq}) &= \{F, H\} \\ \mathcal{L}_1([\alpha_2]_{\simeq}) &= \{\{A\}, \{B\}, \{C, G\}, \{D, E\}\} & \mathcal{L}_2([\alpha_2]_{\simeq}) &= \{A, B, C, D, E, F, G\} \end{aligned}$$

It can readily be seen that the labelled graph representation of $\mathcal{H}_{\mathcal{O}}^\perp / \simeq$ is the same as $\mathcal{G}_{\simeq}$. $\dashv$

6.2 Normalization and Denormalization of OWL Ontologies

Example 4.8 in Section 4.1.2 shows that it is possible to create an axiom with exponentially many minimal non-locality signatures using merely conjunction and disjunction as logical operators. According to Proposition 4.9, an axiom in normal form does not have more than

two minimal non-locality signatures. To normalize any *SROIQ* axiom, [34] suggested twenty one normalization rules. Using these rules, the normalisation of an ontology runs in linear time in the size of the ontology.

We define the functions normalization and denormalization as follows.

Definition 6.11 (Normalization function). *Let S be the set of all *SROIQ*-axioms. Let $norm : S \rightarrow 2^S$ be a partial function that maps a *SROIQ*-axiom α to a set $norm(\alpha)$ of axioms in normal form according to [34].* $\dashv$

Definition 6.12 (Denormalization function). *Let S be the set of all *SROIQ*-axioms. Let $denorm : 2^S \rightarrow S$ be a partial function that maps a set of *SROIQ*-axioms $S = \{\alpha_1, \dots, \alpha_m\}$ in normal form to a *SROIQ*-axiom $denorm(S)$, where a *SROIQ*-axiom is in normal form if it is in one of the forms defined by [34].* $\dashv$

In [27], we suggested the possibility of using the rules defined in [34] to normalize the axioms of any *SROIQ* ontology and build from them axiom dependency hypergraphs. With this approach, we avoid having axioms with exponentially many minimal non-locality signatures. In this section, we analyse if B -connectivity of the ADH of an ontology is preserved when the ADH is built after the axioms of the ontology has been normalized. Recall that when an axiom is normalized, the original signature and MLSs might have been modified. As Definition 6.1 and Proposition 6.6 have shown, the MLSs and signatures of the axioms of an ontology faithfully encoded the hyperedges of the ADH of the ontology and therefore, they also preserve node-to-node B -connectivity. Consequently, it is possible that B -connectivity of the original ADH is altered in the new ADH formed by axioms in normal form. To test the validity of this hypothesis, we start the analysis with the following example.

Example 6.13. *Let $\mathcal{O} = \{\alpha_1, \alpha_2, \alpha_3\}$ be an ontology, where $\alpha_1 = \exists r.(A \sqcap B \sqcap C) \sqsubseteq D$, $\alpha_2 = D \sqsubseteq \exists r.(A \sqcup B \sqcup C)$ and $\alpha_3 = B \sqsubseteq C$.*

The signatures and the minimal non- $\perp$ -locality signatures of the axioms of $\mathcal{O}$ are the following:

$$\begin{aligned} MLS^\perp(\alpha_1) &= \{\{r, A, B, C\}\} & sig(\alpha_1) &= \{r, A, B, C, D\} \\ MLS^\perp(\alpha_2) &= \{\{D\}\} & sig(\alpha_2) &= \{r, A, B, C, D\} \\ MLS^\perp(\alpha_3) &= \{\{B\}\} & sig(\alpha_3) &= \{B, C\} \end{aligned}$$

We obtain the following $\perp$ -local modules for the signature of each axiom, and from them the following two atoms α_1 and α_2 :

$$\begin{aligned} Mod^\perp(sig(\alpha_1), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_3\} & \alpha_1 &= \{\alpha_1, \alpha_2\} \\ Mod^\perp(sig(\alpha_2), \mathcal{O}) &= \{\alpha_1, \alpha_2, \alpha_3\} & \alpha_2 &= \{\alpha_3\} \\ Mod^\perp(sig(\alpha_3), \mathcal{O}) &= \{\alpha_3\} \end{aligned}$$

Fig. 6.2 (Examples 6.13 & 6.14): Normalization and B -connectivity.

The AD of $\mathcal{O}$ is a poset $\langle \text{Atoms}_{\mathcal{O}}^{\perp}, \succ_{\mathcal{O}}^{\perp} \rangle$, where $\text{Atoms}_{\mathcal{O}}^{\perp} = \{\alpha_1, \alpha_2\}$ represents the set of atoms of $\mathcal{O}$ and $\alpha_1 \succ_{\mathcal{O}}^{\perp} \alpha_2$ indicates that atom α_2 depends on atom α_1 (cf. Definition 4.12).

Let us consider the computation of the $\perp$ -local module for the signature $\Sigma = \{C\}$. This is $\text{Mod}^{\perp}(\{C\}, \mathcal{O}) = \{\}$.

The $\perp$ -ADH $\mathcal{H}_{\mathcal{O}}^{\perp}$ with the SCCs marked in red is depicted in Figure 6.2 (a). As we can see, the SCCs of $\mathcal{H}_{\mathcal{O}}^{\perp}$ coincide with the atoms of $\mathcal{O}$, $\text{Atoms}_{\mathcal{O}}^{\perp}$. After $\mathcal{H}_{\mathcal{O}}^{\perp}$ is condensed into $\mathcal{H}_{\mathcal{O}}^{\perp} / \simeq_B$, the directed edge that connects $[\alpha_1]_{\simeq_B}$ with $[\alpha_2]_{\simeq_B}$ corresponds to the dependency between atoms α_1 and α_2 . $\mathcal{H}_{\mathcal{O}}^{\perp} / \simeq_B$ is depicted in Figure 6.2 (b).

Let $\Sigma = \{C\}$ be a signature. None of the axioms that correspond to the nodes in the Σ -extension of $\mathcal{H}_{\mathcal{O}}^{\perp}$ are $\perp$ -local wrt. the signature Σ . $\triangleleft$

We now compute the AD and the $\perp$ -local module for signature $\Sigma = \{C\}$ of the normalized version of the ontology of Example 6.13.

Example 6.14. Let $\mathcal{O} = \{\alpha_1, \alpha_2, \alpha_3\}$ be the ontology of Example 6.13, where $\alpha_1 = \exists r.(A \sqcap B \sqcap C) \sqsubseteq D$, $\alpha_2 = D \sqsubseteq \exists r.(A \sqcup B \sqcup C)$ and $\alpha_3 = B \sqsubseteq C$.

Let $\mathcal{O}' = \text{norm}(\mathcal{O})$ be the ontology produced after the axioms of $\mathcal{O}$ have been normalized. Note that α_3 is already in normal form but α_1 and α_2 are not. To normalize axiom α_1 , we apply rule NR8 [34] that has been defined to simplify existential restrictions with complex concepts: $\dots \exists r.C_1 \dots \xrightarrow{\text{NR8}} \dots \exists r.X \dots$, $X \sqsubseteq C_1$ and $C_1 \sqsubseteq X$, where X is a new concept name ($X \notin \text{sig}(\mathcal{O})$), r is a role name and C_1 is a concept expression. As a result, $\text{NR8}(\alpha_1)$ produced the following axioms in normal form: $\alpha_{11} = \exists r.X \sqsubseteq D$, $\alpha_{12} = X \sqsubseteq A \sqcap B \sqcap C$ and $\alpha_{13} = A \sqcap B \sqcap C \sqsubseteq X$, where X is a fresh concept name.

To normalize axiom α_2 , we also apply rule NR8 that produces the following axioms: $\alpha_{21} = D \sqsubseteq \exists r.Y$, $\alpha_{22} = Y \sqsubseteq A \sqcup B \sqcup C$ and $\alpha_{23} = A \sqcup B \sqcup C \sqsubseteq Y$, where Y is a fresh concept name. Note that α_{23} is not in normal form yet. We apply rule NR6 [34] on axiom α_{23} . This normalization rule eliminates disjunctions from the left-hand side (LNS) of a concept inclusion axiom: $C_1 \sqcup C_2 \sqsubseteq C_3 \xrightarrow{\text{NR6}} C_1 \sqsubseteq C_3, C_2 \sqsubseteq C_3$, where C_1, C_2 and C_3 are possibly concept expressions. In particular, $\text{NR6}(\alpha_{23})$ generates the following axioms in normal form: $\alpha_{231} = A \sqsubseteq Y$, $\alpha_{232} = B \sqsubseteq Y$ and $\alpha_{233} = C \sqsubseteq Y$.

As a result of normalizing axioms α_1 and α_2 , the ontology $\mathcal{O}' = \{\alpha_{11}, \alpha_{12}, \alpha_{13}, \alpha_{21}, \alpha_{22}, \alpha_{231}, \alpha_{232}, \alpha_{233}, \alpha_3\}$, where $\alpha_{11} = \exists r.X \sqsubseteq D$, $\alpha_{12} = X \sqsubseteq A \sqcap B \sqcap C$, $\alpha_{13} = A \sqcap B \sqcap C \sqsubseteq X$, $\alpha_{21} = D \sqsubseteq \exists r.Y$, $\alpha_{22} = Y \sqsubseteq A \sqcup B \sqcup C$, $\alpha_{231} = A \sqsubseteq Y$, $\alpha_{232} = B \sqsubseteq Y$, $\alpha_{233} = C \sqsubseteq Y$ and $\alpha_3 = B \sqsubseteq C$.

The signatures and the minimal non- $\perp$ -locality signatures of the axioms of $\mathcal{O}'$ are the following:

$$\begin{aligned}
 MLS^\perp(\alpha_{11}) &= \{\{r, X\}\} & \text{sig}(\alpha_{11}) &= \{r, D, X\} \\
 MLS^\perp(\alpha_{12}) &= \{\{X\}\} & \text{sig}(\alpha_{12}) &= \{A, B, C, X\} \\
 MLS^\perp(\alpha_{13}) &= \{\{A, B, C\}\} & \text{sig}(\alpha_{13}) &= \{A, B, C, X\} \\
 MLS^\perp(\alpha_{21}) &= \{\{D\}\} & \text{sig}(\alpha_{21}) &= \{r, D, Y\} \\
 MLS^\perp(\alpha_{22}) &= \{\{Y\}\} & \text{sig}(\alpha_{22}) &= \{A, B, C, Y\} \\
 MLS^\perp(\alpha_{231}) &= \{\{A\}\} & \text{sig}(\alpha_{231}) &= \{A, Y\} \\
 MLS^\perp(\alpha_{232}) &= \{\{B\}\} & \text{sig}(\alpha_{232}) &= \{B, Y\} \\
 MLS^\perp(\alpha_{233}) &= \{\{C\}\} & \text{sig}(\alpha_{233}) &= \{C, Y\} \\
 MLS^\perp(\alpha_3) &= \{\{B\}\} & \text{sig}(\alpha_3) &= \{B, C\}
 \end{aligned}$$

We obtain the following $\perp$ -local modules for the signature of each axiom:

$$\begin{aligned}
 \text{Mod}^\perp(\text{sig}(\alpha_{11}), \mathcal{O}') &= \mathcal{O}' \\
 \text{Mod}^\perp(\text{sig}(\alpha_{21}), \mathcal{O}') &= \mathcal{O}' \\
 \text{Mod}^\perp(\text{sig}(\alpha_{12}), \mathcal{O}') &= \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\} \\
 \text{Mod}^\perp(\text{sig}(\alpha_{22}), \mathcal{O}') &= \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\} \\
 \text{Mod}^\perp(\text{sig}(\alpha_{13}), \mathcal{O}') &= \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\} \\
 \text{Mod}^\perp(\text{sig}(\alpha_{231}), \mathcal{O}') &= \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\} \\
 \text{Mod}^\perp(\text{sig}(\alpha_3), \mathcal{O}') &= \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\} \\
 \text{Mod}^\perp(\text{sig}(\alpha_{232}), \mathcal{O}') &= \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\} \\
 \text{Mod}^\perp(\text{sig}(\alpha_{233}), \mathcal{O}') &= \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\}
 \end{aligned}$$

From these modules, we can compute the following two atoms: $\alpha'_1 = \{\alpha_{11}, \alpha_{21}\}$ and $\alpha'_2 = \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\}$. The AD of $\mathcal{O}'$ is a poset $\langle \text{Atoms}_{\mathcal{O}'}^\perp, \succ_{\mathcal{O}'}^\perp \rangle$ such that $\text{Atoms}_{\mathcal{O}'}^\perp = \{\alpha'_1, \alpha'_2\}$ and $\alpha'_1 \succ_{\mathcal{O}'}^\perp \alpha'_2$. The AD of $\mathcal{O}$ and $\mathcal{O}'$ might coincide apparently. However, note that not all the axioms that were produced after the normalization of α_1 and α_2 are in atom α'_1 but in α'_2 . Moreover, $\text{Mod}^\perp(\{C\}, \mathcal{O}') = \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\}$. Recall that in $\mathcal{O}$, $\text{Mod}^\perp(\{C\}, \mathcal{O})$ was empty. This means that the computation of syntactic locality-based modules and the AD might be different when the axioms of an ontology are normalized.

We can represent the modules and the AD of $\mathcal{O}'$ using a $\perp$ -ADH $\mathcal{H}_{\mathcal{O}'}^\perp = (\mathcal{V}_{\mathcal{O}'}^\perp, \mathcal{E}_{\mathcal{O}'}^\perp)$, where $\mathcal{V}_{\mathcal{O}'}^\perp = \mathcal{O}'$ and $\mathcal{E}_{\mathcal{O}'}^\perp = \{e_1, \dots, e_{27}\}$. The ADH $\mathcal{H}_{\mathcal{O}'}^\perp$ with the SCCs marked in red is depicted in Figure 6.2 (c). To facilitate the identification of the different hyperedges, we use different colours and line patterns. Simple hyperedges ($|T(e)| = 1$) starting from the nodes on the left (α_{11} , α_{12} , α_{13} and α_3) are black solid edges. Simple hyperedges starting from any other node are blue dashed edges. In the case of complex hyperedges, they are purple dotted hyperedges. This is the complete list of hyperedges of $\mathcal{H}_{\mathcal{O}'}^\perp$:

$$\begin{array}{lll}
e_1 = (\{\alpha_{11}\}, \alpha_{12}) & e_2 = (\{\alpha_{11}\}, \alpha_{21}) & e_3 = (\{\alpha_{12}\}, \alpha_{13}) \\
e_4 = (\{\alpha_{12}\}, \alpha_{231}) & e_5 = (\{\alpha_{12}\}, \alpha_{232}) & e_6 = (\{\alpha_{12}\}, \alpha_{233}) \\
e_7 = (\{\alpha_{12}\}, \alpha_3) & e_8 = (\{\alpha_{13}\}, \alpha_{12}) & e_9 = (\{\alpha_{13}\}, \alpha_{231}) \\
e_{10} = (\{\alpha_{13}\}, \alpha_{232}) & e_{11} = (\{\alpha_{13}\}, \alpha_{233}) & e_{12} = (\{\alpha_{13}\}, \alpha_3) \\
e_{13} = (\{\alpha_{21}\}, \alpha_{22}) & e_{14} = (\{\alpha_{22}\}, \alpha_{13}) & e_{15} = (\{\alpha_{22}\}, \alpha_{231}) \\
e_{16} = (\{\alpha_{22}\}, \alpha_{232}) & e_{17} = (\{\alpha_{22}\}, \alpha_{233}) & e_{18} = (\{\alpha_{22}\}, \alpha_3) \\
e_{19} = (\{\alpha_{231}\}, \alpha_{22}) & e_{20} = (\{\alpha_{232}\}, \alpha_{22}) & e_{21} = (\{\alpha_{232}\}, \alpha_3) \\
e_{22} = (\{\alpha_{233}\}, \alpha_{22}) & e_{23} = (\{\alpha_3\}, \alpha_{232}) & e_{24} = (\{\alpha_3\}, \alpha_{233}) \\
e_{25} = (\{\alpha_{12}, \alpha_{21}\}, \alpha_{11}) & e_{26} = (\{\alpha_{13}, \alpha_{21}\}, \alpha_{11}) & e_{27} = (\{\alpha_{231}, \alpha_{232}, \alpha_{233}\}, \alpha_{13}) \\
e_{28} = (\{\alpha_{231}, \alpha_3\}, \alpha_{13}) & &
\end{array}$$

As we can see, the SCCs of $\mathcal{H}_{\mathcal{O}'}^\perp$ coincide apparently with the SCCs of $\mathcal{H}_{\mathcal{O}}^\perp$. However, note that the SCC that contains the node that corresponds to axiom α_3 also contains nodes that correspond to axioms that were produced during the normalization of α_1 and α_2 . Figure 6.2 (d) depicted how $\mathcal{H}_{\mathcal{O}'}^\perp$ is condensed into $\mathcal{H}_{\mathcal{O}'}^\perp / \simeq_B$.

Let $\Sigma = \{C\}$ be a signature. The node that corresponds to the axiom α_{233} is not $\perp$ -local with respect to Σ . The B -connected component of α_{233} includes all nodes of the Σ -extension of $\mathcal{H}_{\mathcal{O}'}^\perp$ with the exception of the nodes that correspond to the axioms α_{11} and α_{21} . Recall that in $\mathcal{H}_{\mathcal{O}}^\perp$, there is not an axiom that is not $\perp$ -local wrt. Σ . Therefore, the B -connected component for this signature is the empty set. $\triangleleft$

In Example 6.14, we observe that $\text{denorm}(\text{Mod}^\perp(\{C\}, \mathcal{O}'))$ is not the same as $\text{Mod}^\perp(\{C\}, \mathcal{O})$ in Example 6.13. In particular, $\text{Mod}^\perp(\{C\}, \mathcal{O}') = \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\}$, whereas $\text{Mod}^\perp(\{C\}, \mathcal{O})$ returns an empty set of axioms.

Based on this result, we obtain the following proposition.

Proposition 6.15. *There exists an ontology $\mathcal{O}$ and a signature $\Sigma \subseteq \text{sig}(\mathcal{O})$ such that $\text{Mod}^x(\Sigma, \mathcal{O}) \neq \text{denorm}(\text{Mod}^x(\Sigma, \mathcal{O}'))$, where $\mathcal{O}' = \text{norm}(\mathcal{O})$ and $\text{Mod}^x(\Sigma, \mathcal{O})$ is the function depicted in Figure 4.1.* $\dashv$

In Examples 6.14 & 6.14, we note that it is not possible to obtain the same B -connected component from the $\perp$ -ADH of the ontology $\mathcal{O}$ and from the $\perp$ -ADH of the ontology $\mathcal{O}'$. This is made more precise in the following proposition.

Proposition 6.16. *There exists an ontology $\mathcal{O}$ and a set of axioms $S \subseteq \mathcal{O}$ such that $\geq_{B(\mathcal{H})}(S) \neq \text{denorm}(\geq_{B(\mathcal{H}')} (S'))$, where $\mathcal{O}' = \text{norm}(\mathcal{O})$, $S' = \text{norm}(S)$, $\mathcal{H}$ is the x -ADH for $\mathcal{O}$, $\mathcal{H}'$ is the x -ADH for $\mathcal{O}'$, $\geq_B(\mathcal{H})$ ($\geq_B(\mathcal{H}')$) is the B -connectivity relation in $\mathcal{H}$ ($\mathcal{H}'$, respectively) and $x \in \{\perp, \top\}$.* $\dashv$

The analysis of Examples 6.14 & 6.14 also indicates that B -connectivity in the Σ -extension of the $\perp$ -ADH of the ontology $\mathcal{O}$ does not coincide with B -connectivity in the Σ -extension of the $\perp$ -ADH of the ontology $\mathcal{O}'$ as it is established by the following proposition.

Proposition 6.17. *There exists an ontology $\mathcal{O}$ and a signature $\Sigma \subseteq \text{sig}(\mathcal{O})$ such that $\geq_B(\mathcal{H}) (S) \neq \text{denorm}(\geq_B(\mathcal{H}') (S'))$, where $\mathcal{O}' = \text{norm}(\mathcal{O})$, $\mathcal{H}$ is the Σ -extension of the x -ADH for $\mathcal{O}$, $\mathcal{H}'$ is the Σ -extension of the x -ADH for $\mathcal{O}'$, $x \in \{\perp, \top\}$, $S(S')$ is the set of axioms that are not x -local in $\mathcal{O}$ ($\mathcal{O}'$, respectively) wrt. Σ and $\geq_B(\mathcal{H})$ ($\geq_B(\mathcal{H}')$) is the B -connectivity relation in $\mathcal{H}$ ($\mathcal{H}'$, respectively). $\dashv$*

In Example 6.14, we also noticed that when an axiom is normalized, some of the axioms in normal form are more relevant than others in the following sense: they better preserve the B -connectivity relation of the original ADH. For instance in the $\perp$ -ADH of ontology $\mathcal{O}$ in Example 6.13, $\mathcal{H}_{\mathcal{O}}^{\perp}$, the axioms α_1 and α_2 are strongly- B -connected. In the $\perp$ -ADH of ontology $\mathcal{O}'$ in Example 6.14, $\mathcal{H}_{\mathcal{O}'}^{\perp}$, only the axioms α_{11} and α_{21} preserve the original strong- B -connectivity relation between α_1 and α_2 . The rest of axioms that were produced after α_1 and α_2 were normalized are, in fact, strongly- B -connected with α_3 . This is not what is happening in $\mathcal{H}_{\mathcal{O}}^{\perp}$ where α_3 is only strongly- B -connected with itself.

The axioms α_{11} and α_{21} have a particularity: each of them are B -connected to the other axioms that were produced after α_1 or α_2 were normalized. Therefore, it seems that the axioms α_{11} , α_{21} and α_3 are the axioms that better preserve B -connectivity in $\mathcal{H}_{\mathcal{O}}^{\perp}$. We name these axioms as *primary axioms* and we denote the rest of axioms in normal form as *auxiliary axioms*.

Definition 6.18 (Primary and auxiliary axioms). *Let $\mathcal{O}$ be an ontology and let $\mathcal{O}' = \text{norm}(\mathcal{O})$ be the ontology $\mathcal{O}$ in normal form according to [34]. Let $\alpha \in \mathcal{O}$ be an axiom and let $\text{norm}(\alpha)$ be a set of axioms in normal form. Let $\mathcal{H}_{\mathcal{O}'}^x$ be the x -ADH of $\mathcal{O}'$ and let $\geq_B$ be the B -connectivity relation in $\mathcal{H}_{\mathcal{O}'}^x$, where $x \in \{\perp, \top\}$.*

An axiom $\beta \in \text{norm}(\alpha)$ is a primary axiom of $\text{norm}(\alpha)$ wrt. $\mathcal{O}$ iff $\text{norm}(\alpha) \subseteq \geq_B(\beta)$. If β is not a primary axiom, then β is an auxiliary axiom. $\dashv$

Note that every axiom α in $\mathcal{O}$ that is in normal form (i.e. $\text{norm}(\alpha) = \alpha$) is a primary axiom in $\mathcal{O}'$.

Let's consider Example 6.14 again. The primary axioms of $\mathcal{O}'$ are α_{11} , α_{21} and α_3 . If we only take into account primary axioms for computing modules and atoms (or B -connected components and strongly- B -connected components), then the AD and the $\perp$ -local module for the signature $\Sigma = \{C\}$ coincide in $\mathcal{O}$ and $\mathcal{O}'$.

Recall that $\text{Mod}^{\perp}(\text{sig}(\alpha_{11}), \mathcal{O}') = \text{Mod}^{\perp}(\text{sig}(\alpha_{21}), \mathcal{O}') = \mathcal{O}'$ and $\text{Mod}^{\perp}(\text{sig}(\alpha_3), \mathcal{O}') = \mathcal{O}' \setminus \{\alpha_{11}, \alpha_{21}\}$. If we only consider primary axioms to compute the atoms of $\mathcal{O}'$, then we have

the following two atoms: $\alpha'_1 = \{\alpha_{11}, \alpha_{21}\}$ and $\alpha'_2 = \{\alpha_3\}$. Because $\text{Mod}^\perp(\text{sig}(\alpha_3), \mathcal{O}') \subseteq \text{Mod}^\perp(\text{sig}(\alpha_{11}), \mathcal{O}')$, then $\alpha'_1 \succ_{\mathcal{O}'}^\perp \alpha'_2$.

To denormalize α'_1 and α'_2 , we only consider primary axioms as follows. Let α be the axiom before normalization and let $\beta \in \text{norm}(\alpha)$ be a primary axiom. If β is in an atom α' , then $\alpha \in \text{denorm}(\alpha')$. This means that $\alpha_1 = \text{denorm}(\alpha'_1)$ and $\alpha_2 = \text{denorm}(\alpha'_2)$, where α_1 and α_2 are the atoms of ontology $\mathcal{O}$ in Example 6.13. Note that two or more primary axioms $\beta, \gamma \in \text{norm}(\alpha)$ are always in the same atom because $\text{Mod}^x(\text{sig}(\beta), \mathcal{O}') = \text{Mod}^x(\text{sig}(\gamma), \mathcal{O}')$ (cf. Definition 6.18).

Recall also that $\text{Mod}^\perp(\{C\}, \mathcal{O}) = \{\}$. To compute $\text{Mod}^\perp(\{C\}, \mathcal{O}')$, we try to identify which primary axioms in $\mathcal{O}'$ are not $\perp$ -local with respect to the signature $\{C\}$. None of them are not $\perp$ -local and $\text{Mod}^\perp(\{C\}, \mathcal{O}') = \{\}$.

Although the approach of only considering primary axioms when a module or an atom is computed seems to work well in Example 6.14, the next example shows that this is not always the case. In particular, the example includes an axiom that after being normalized, none of its axioms in normal form are primary axioms.

Example 6.19. Let $\mathcal{O} = \{\alpha_1, \alpha_2\}$ be an ontology, where $\alpha_1 = A \sqcap B \sqcap C \sqsubseteq D$, $\alpha_2 = (A \sqcap B) \sqcup (C \sqcap D) \sqsubseteq E$.

The signatures and the minimal non- $\perp$ -locality signatures of the axioms of $\mathcal{O}$ are the following:

$$\begin{aligned} \text{MLS}^\perp(\alpha_1) &= \{\{A, B, C\}\} & \text{sig}(\alpha_1) &= \{A, B, C, D\} \\ \text{MLS}^\perp(\alpha_2) &= \{\{A, B\}, \{C, D\}\} & \text{sig}(\alpha_2) &= \{A, B, C, D, E\} \end{aligned}$$

We obtain the following $\perp$ -local modules for the signature of each axiom and from them atom α_1 :

$$\begin{aligned} \text{Mod}^\perp(\text{sig}(\alpha_1), \mathcal{O}) &= \{\alpha_1, \alpha_2\} & \alpha_1 &= \{\alpha_1, \alpha_2\} \\ \text{Mod}^\perp(\text{sig}(\alpha_2), \mathcal{O}) &= \{\alpha_1, \alpha_2\} \end{aligned}$$

The AD of $\mathcal{O}$ is a poset $\langle \text{Atoms}_{\mathcal{O}}^\perp, \succ_{\mathcal{O}}^\perp \rangle$ such that $\text{Atoms}_{\mathcal{O}}^\perp = \{\alpha_1\}$ and by reflexivity of $\succ_{\mathcal{O}}^\perp$, $\alpha_1 \succ_{\mathcal{O}}^\perp \alpha_1$. Let's consider the computation of the $\perp$ -local module for the signature $\Sigma = \{C, D\}$. Then: $\text{Mod}^\perp(\{C, D\}, \mathcal{O}) = \mathcal{O}$.

The $\perp$ -ADH of the ontology $\mathcal{O}$ is $\mathcal{H}_{\mathcal{O}}^\perp = (\mathcal{V}_{\mathcal{O}}^\perp, \mathcal{E}_{\mathcal{O}}^\perp)$, where $\mathcal{V}_{\mathcal{O}}^\perp = \mathcal{O}$ and $\mathcal{E}_{\mathcal{O}}^\perp = \{e_1 = (\{\alpha_1\}, \alpha_2), e_2 = (\{\alpha_2\}, \alpha_1)\}$.

It can readily be seen that $\mathcal{H}_{\mathcal{O}}^\perp$ has only one strongly-B-connected component and it coincides with the atom of $\mathcal{O}$. Condensing $\mathcal{H}_{\mathcal{O}}^\perp$ produces an cADH with only one node and no hyperedges. This node corresponds to an equivalent class for the strongly-B-connected component of $\mathcal{H}_{\mathcal{O}}^\perp$.

Let $\Sigma = \{C, D\}$ be a signature. The B-connected component for Σ in the Σ -extension of $\mathcal{H}_{\mathcal{O}}^\perp$ includes the nodes that correspond to the axioms α_1 and α_2 of $\mathcal{O}$. $\triangleleft$

We now compute the AD and the $\perp$ -local module for signature $\Sigma = \{C, D\}$ of the normalized version of the ontology of Example 6.19.

Example 6.20. Let $\mathcal{O}$ be the ontology of Example 6.19. Let $\mathcal{O}' = \text{norm}(\mathcal{O})$ be the ontology produced after the axioms of $\mathcal{O}$ have been normalized. Note that α_1 is already in normal form but α_2 is not. To normalize axiom α_2 , we apply rule NR6 [34] to eliminate the disjunction on the left-hand side (LHS) of α_2 . As a result, NR6 produced the following two axioms that are already in normal form: $\alpha_{21} = A \sqcap B \sqsubseteq E$ and $\alpha_{22} = C \sqcap D \sqsubseteq E$.

According to Definition 6.18, α_1 is the only primary axiom. If we ignore the auxiliary axioms α_{21} and α_{22} , then there is only one atom in $\mathcal{O}'$ that contains axiom α_1 and $\text{Mod}^\perp(\{C, D\}, \mathcal{O}) = \{\}$. This means that the approach of considering only primary axioms for computing atoms and modules does not work in this example.

Let's consider all axioms in $\mathcal{O}'$. The signatures and the minimal non- $\perp$ -locality signatures of the axioms of $\mathcal{O}'$ are the following:

$$\begin{aligned} \text{MLS}^\perp(\alpha_1) &= \{\{A, B, C\}\} & \text{sig}(\alpha_1) &= \{A, B, C, D\} \\ \text{MLS}^\perp(\alpha_{21}) &= \{\{A, B\}\} & \text{sig}(\alpha_{21}) &= \{A, B, E\} \\ \text{MLS}^\perp(\alpha_{22}) &= \{\{C, D\}\} & \text{sig}(\alpha_{22}) &= \{C, D, E\} \end{aligned}$$

We obtain the following $\perp$ -local modules for the signature of each axiom and from them the atoms α'_1 , α'_2 and α'_3 :

$$\begin{aligned} \text{Mod}^\perp(\text{sig}(\alpha_1), \mathcal{O}) &= \{\alpha_1, \alpha_{21}, \alpha_{22}\} & \alpha'_1 &= \{\alpha_1\} \\ \text{Mod}^\perp(\text{sig}(\alpha_{21}), \mathcal{O}) &= \{\alpha_{21}\} & \alpha'_2 &= \{\alpha_{21}\} \\ \text{Mod}^\perp(\text{sig}(\alpha_{22}), \mathcal{O}) &= \{\alpha_{22}\} & \alpha'_3 &= \{\alpha_{22}\} \end{aligned}$$

The AD of $\mathcal{O}'$ is a poset $\langle \text{Atoms}_{\mathcal{O}'}^\perp, \succ_{\mathcal{O}'}^\perp \rangle$ such that $\text{Atoms}_{\mathcal{O}'}^\perp = \{\alpha'_1, \alpha'_2, \alpha'_3\}$ and the following dependency relations can be found: $\alpha'_1 \succ_{\mathcal{O}'}^\perp \alpha'_2$ and $\alpha'_1 \succ_{\mathcal{O}'}^\perp \alpha'_3$.

Let's consider the computation of the $\perp$ -local module for the signature $\Sigma = \{C, D\}$: $\text{Mod}^\perp(\{C, D\}, \mathcal{O}') = \{\alpha_{22}\}$.

Neither the AD of $\mathcal{O}'$ nor $\text{Mod}^\perp(\{C, D\}, \mathcal{O}')$ correspond to the AD of $\mathcal{O}$ and $\text{Mod}^\perp(\{C, D\}, \mathcal{O})$, respectively. $\triangleleft$

Example 6.20 shows that the approach of considering only primary axioms in normal form to compute modules and atoms does not work in general. Moreover, this example also points out the influence of certain normalization rules in the preservation (or not) of the B -connectivity relation in the x -ADH of a normalized ontology. In the example, we can observe that rule NR6 does not alter the original two MLSs of the axiom α_2 : $\text{MLS}^\perp(\alpha_2) = \{\{A, B\}, \{C, D\}\}$. After α_2 is normalized by NR6, two new axioms in normal form, α_{21} and

α_{22} are included in $\mathcal{O}'$. Each of these axioms has only one MLS that corresponds to one of the MLSs of α_2 : $\text{MLS}^\perp(\alpha_{21}) = \{\{A, B\}\}$ and $\text{MLS}^\perp(\alpha_{22}) = \{\{C, D\}\}$.

The reason why B -connectivity is not preserved in the $\perp$ -local ADH of ontology $\mathcal{O}'$ is because not all the symbols of the signature of α_2 are in the signatures of axioms α_{21} and α_{22} . In the original ontology $\mathcal{O}$, $\text{MLS}^\perp(\alpha_1) \subseteq \text{sig}(\alpha_2)$ and for every $S \in \text{MLS}^\perp(\alpha_2)$: $S \subseteq \text{sig}(\alpha_1)$. This is why α_1 and α_2 are strongly- B -connected. In the normalized ontology $\mathcal{O}'$, the signatures of α_{21} and α_{22} are not enough to cover the symbols of the minimal non-locality signature in $\text{MLS}^\perp(\alpha_1)$. In fact, $\text{sig}(\alpha_{21})$ does not provide enough symbols to cover the symbols of the minimal non-locality signature in $\text{MLS}^\perp(\alpha_{22})$. The same is also true for $\text{sig}(\alpha_{22})$ and $\text{MLS}^\perp(\alpha_{21})$.

One possible approach to avoid the impact of rule $NR6$ in the B -connectivity relation of the $\perp$ -local ADH of ontology $\mathcal{O}'$ is to preserve the symbols of the signature of α_2 in the signatures of α_{21} and α_{22} : $\text{sig}(\alpha_2) = \text{sig}(\alpha_{21}) = \text{sig}(\alpha_{22})$. This can be done by modifying axioms α_{21} and α_{22} as it is shown in the following example.

Example 6.21. Let $\mathcal{O}$ be the ontology defined in Example 6.19 and let $\mathcal{O}' = \text{norm}(\mathcal{O})$ be the ontology defined in Example 6.20. Let $\mathcal{O}'' = \{\alpha_1, \alpha'_{21}, \alpha'_{22}\}$ be a modified version of ontology $\mathcal{O}'$, where $\alpha_1 = A \sqcap B \sqcap C \sqsubseteq D$, $\alpha'_{21} = A \sqcap B \sqsubseteq C \sqcup D \sqcup E$ and $\alpha'_{22} = C \sqcap D \sqsubseteq A \sqcup B \sqcup E$. According to [34], the axioms of $\mathcal{O}''$ are in normal form.

The signatures and minimal non- $\perp$ -locality signatures of the new axioms of $\mathcal{O}''$ are the following:

$$\begin{aligned} \text{MLS}^\perp(\alpha_1) &= \{\{A, B, C\}\} & \text{sig}(\alpha_1) &= \{A, B, C, D\} \\ \text{MLS}^\perp(\alpha'_{21}) &= \{\{A, B\}\} & \text{sig}(\alpha'_{21}) &= \{A, B, C, D, E\} \\ \text{MLS}^\perp(\alpha'_{22}) &= \{\{C, D\}\} & \text{sig}(\alpha'_{22}) &= \{A, B, C, D, E\} \end{aligned}$$

We obtain the following $\perp$ -local modules for the signature of each axiom and from them the atom α''_1 :

$$\begin{aligned} \text{Mod}^\perp(\text{sig}(\alpha_1), \mathcal{O}'') &= \{\alpha_1, \alpha'_{21}, \alpha'_{22}\} & \alpha''_1 &= \{\alpha_1, \alpha'_{21}, \alpha'_{22}\} \\ \text{Mod}^\perp(\text{sig}(\alpha'_{21}), \mathcal{O}'') &= \{\alpha_1, \alpha'_{21}, \alpha'_{22}\} \\ \text{Mod}^\perp(\text{sig}(\alpha'_{22}), \mathcal{O}'') &= \{\alpha_1, \alpha'_{21}, \alpha'_{22}\} \end{aligned}$$

The AD of $\mathcal{O}''$ is a poset $\langle \text{Atoms}_{\mathcal{O}''}^\perp, \succ_{\mathcal{O}''}^\perp \rangle$ such that $\text{Atoms}_{\mathcal{O}''}^\perp = \{\alpha''_1\}$ and by reflexivity of $\succ_{\mathcal{O}''}^\perp$: $\alpha''_1 \succ_{\mathcal{O}''}^\perp \alpha''_1$. After denormalization, this is the same AD as the AD of $\mathcal{O}$.

Let's consider the computation of the $\perp$ -local module for the signature $\Sigma = \{C, D\}$: $\text{Mod}^\perp(\{C, D\}, \mathcal{O}'') = \{\alpha_1, \alpha'_{21}, \alpha'_{22}\}$. After denormalization, this is the same module as in $\mathcal{O}$. $\triangleleft$

In the following example (cf. Example 6.22), we show that this approach of preserving the signature of the original axioms after denormalization does not work for the ontology of

Example 6.13. In particular, there is not a correspondence between the atoms of each of the two examples. Moreover, the $\perp$ -local module for $\Sigma = \{C\}$ is also different in both examples.

Example 6.22. Let $\mathcal{O}$ be the ontology of Example 6.13. Let $\mathcal{O}' = \text{norm}(\mathcal{O})$ be the ontology of Example 6.14. Let $\mathcal{O}''$ be an ontology that is created from the axioms of $\mathcal{O}'$ as follows. Consider the axiom $\alpha_1 \in \mathcal{O}$ and the axiom $\alpha_{11} \in \text{norm}(\alpha_1)$. Then, $\alpha'_{11} \in \mathcal{O}''$ is axiom such that $MLS^\perp(\alpha'_{11}) = MLS^\perp(\alpha_{11})$ and $\text{sig}(\alpha'_{11}) = \text{sig}(\alpha_{11}) \cup \text{sig}(\alpha_1)$. To achieve this, we modify the axioms of $\mathcal{O}'$ as follows.

$$\begin{array}{ll}
\alpha_{11} = \exists r.X \sqsubseteq D & \rightarrow \alpha'_{11} = \exists r.X \sqsubseteq A \sqcup B \sqcup C \sqcup D \\
\alpha_{12} = X \sqsubseteq A \sqcap B \sqcap C & \rightarrow \alpha'_{12} = X \sqsubseteq \exists r.A \sqcap B \sqcap C \\
\alpha_{13} = A \sqcap B \sqcap C \sqsubseteq X & \rightarrow \alpha'_{13} = A \sqcap B \sqcap C \sqsubseteq \exists r.X \\
\alpha_{21} = D \sqsubseteq \exists r.Y & \rightarrow \alpha'_{21} = D \sqsubseteq \exists r.Y \sqcup A \sqcup B \sqcup C \\
\alpha_{22} = Y \sqsubseteq A \sqcup B \sqcup C & \rightarrow \alpha'_{22} = Y \sqsubseteq \exists r.A \sqcup B \sqcup C \sqcup D \\
\alpha_{231} = A \sqsubseteq Y & \rightarrow \alpha'_{231} = A \sqsubseteq \exists r.Y \sqcup B \sqcup C \sqcup D \\
\alpha_{232} = B \sqsubseteq Y & \rightarrow \alpha'_{232} = B \sqsubseteq \exists r.Y \sqcup A \sqcup C \sqcup D \\
\alpha_{233} = C \sqsubseteq Y & \rightarrow \alpha'_{233} = C \sqsubseteq \exists r.Y \sqcup A \sqcup B \sqcup D \\
\alpha_3 = B \sqsubseteq C & \rightarrow \alpha'_3 = B \sqsubseteq C
\end{array}$$

The signatures and the minimal non- $\perp$ -locality signatures of the axioms of $\mathcal{O}'$ are the following:

$$\begin{array}{ll}
MLS^\perp(\alpha'_{11}) = \{\{r, X\}\} & \text{sig}(\alpha'_{11}) = \{r, A, B, C, D, X\} \\
MLS^\perp(\alpha'_{12}) = \{\{X\}\} & \text{sig}(\alpha'_{12}) = \{r, A, B, C, D, X\} \\
MLS^\perp(\alpha'_{13}) = \{\{A, B, C\}\} & \text{sig}(\alpha'_{13}) = \{r, A, B, C, D, X\} \\
MLS^\perp(\alpha'_{21}) = \{\{D\}\} & \text{sig}(\alpha'_{21}) = \{r, A, B, C, D, Y\} \\
MLS^\perp(\alpha'_{22}) = \{\{Y\}\} & \text{sig}(\alpha'_{22}) = \{r, A, B, C, D, Y\} \\
MLS^\perp(\alpha'_{231}) = \{\{A\}\} & \text{sig}(\alpha'_{231}) = \{r, A, B, C, D, Y\} \\
MLS^\perp(\alpha'_{232}) = \{\{B\}\} & \text{sig}(\alpha'_{232}) = \{r, A, B, C, D, Y\} \\
MLS^\perp(\alpha'_{233}) = \{\{C\}\} & \text{sig}(\alpha'_{233}) = \{r, A, B, C, D, Y\} \\
MLS^\perp(\alpha'_3) = \{\{B\}\} & \text{sig}(\alpha'_3) = \{B, C\}
\end{array}$$

We obtain the following $\perp$ -local modules for the signature of each axiom:

$$\begin{array}{ll}
Mod^\perp(\text{sig}(\alpha'_{11}), \mathcal{O}'') = \mathcal{O}'' & Mod^\perp(\text{sig}(\alpha'_{12}), \mathcal{O}'') = \mathcal{O}'' \\
Mod^\perp(\text{sig}(\alpha'_{13}), \mathcal{O}'') = \mathcal{O}'' & Mod^\perp(\text{sig}(\alpha'_{21}), \mathcal{O}'') = \mathcal{O}'' \\
Mod^\perp(\text{sig}(\alpha'_{22}), \mathcal{O}'') = \mathcal{O}'' & Mod^\perp(\text{sig}(\alpha'_{231}), \mathcal{O}'') = \mathcal{O}'' \\
Mod^\perp(\text{sig}(\alpha'_{232}), \mathcal{O}'') = \mathcal{O}'' & Mod^\perp(\text{sig}(\alpha'_{233}), \mathcal{O}'') = \mathcal{O}'' \\
Mod^\perp(\text{sig}(\alpha'_3), \mathcal{O}'') = \mathcal{O}'' &
\end{array}$$

From these modules, we can compute the following atom: $\alpha_1'' = \mathcal{O}''$. The AD of $\mathcal{O}''$ is a poset $\langle \text{Atoms}_{\mathcal{O}''}^\perp, \succ_{\mathcal{O}''}^\perp \rangle$ such that $\text{Atoms}_{\mathcal{O}''}^\perp = \{\alpha_1''\}$ and by reflexivity of $\succ_{\mathcal{O}''}^\perp$: $\alpha_1'' \succ_{\mathcal{O}''}^\perp \alpha_1''$. The ADs of $\mathcal{O}$ and $\mathcal{O}''$ are different. Moreover, $\text{Mod}^\perp(\{C\}, \mathcal{O}'') = \mathcal{O}''$ which is also different from $\text{Mod}^\perp(\{C\}, \mathcal{O}) = \{\}$.

Similarly, we can show that the B -connectivity relation in the $\perp$ -ADH of ontology $\mathcal{O}$ is not preserved in the $\perp$ -ADH of ontology $\mathcal{O}''$. $\triangleleft$

It seems that none of the two approaches (primary axioms and signature preservation) that we have suggested to preserve B -connectivity work in general. In the previous examples, we could observe that there are certain normalization rules (cf. [34]) that alter the B -connectivity relation of the original x -ADH and it does not seem to be straightforward to find a general method that repairs this relation. We also notice that there are few normalization rules that do not modify the B -connectivity relation of the original x -ADH. This is the case of rule NR11 [34] that replaces a concept equivalence axiom by two concept inclusions axioms. This is shown in the following example.

Example 6.23. Let $\beta = A \sqcap B \equiv C \sqcap D$ be an axiom and let $\text{NR11}(\beta) = \{\beta_1, \beta_2\}$, where $\beta_1 = A \sqcap B \sqsubseteq C \sqcap D$ and $\beta_2 = C \sqcap D \sqsubseteq A \sqcap B$. Let's consider the notion of $\perp$ -locality. The MLSs and signatures of these axioms are the following:

$$\begin{aligned} \text{MLS}^\perp(\beta) &= \{\{A, B\}, \{C, D\}\} & \text{sig}(\beta) &= \{A, B, C, D\} \\ \text{MLS}^\perp(\beta_1) &= \{\{A, B\}\} & \text{sig}(\beta_1) &= \{A, B, C, D\} \\ \text{MLS}^\perp(\beta_2) &= \{\{C, D\}\} & \text{sig}(\beta_2) &= \{A, B, C, D\} \end{aligned}$$

$\triangleleft$

In Example 6.23, $\text{sig}(\beta) = \text{sig}(\beta_1) = \text{sig}(\beta_2)$ and $\text{MLS}^\perp(\beta) = \text{MLS}^\perp(\beta_1) \cup \text{MLS}^\perp(\beta_2)$. The MLSs and the signature of β have been preserved in the normalized axioms β_1 and β_2 and therefore, the B -connectivity relation is not affected by normalization. We denote the normalization rules that preserve the MLSs and the signature of an axiom as *safe*.

Definition 6.24 (Safe normalization rules). Let $\mathcal{O}$ be an ontology and let $\alpha \in \mathcal{O}$ be an axiom. Let $x \in \{\perp, \top\}$ be a syntactic locality notion. Let $\text{norm}_{\text{NRi}}(\alpha) = \{\alpha_1, \dots, \alpha_m\}$ be the set of axioms after α is normalised using rule NRi . NRi is a safe normalization rule for x if

(i) for every $\alpha_k \in \text{norm}_{\text{NRi}}(\alpha)$: $\text{sig}(\alpha_k) = \text{sig}(\alpha)$ and

(ii) $\text{MLS}^x(\alpha) = \bigcup_{k=1}^{k=m} \text{MLS}^x(\alpha_k)$, where $\alpha_k \in \text{norm}_{\text{NRi}}(\alpha)$. $\dashv$

The following proposition establishes which normalization rules defined in [34] are safe.

Proposition 6.25. *Let $\{NR1, \dots, NR21\}$ be the normalization rules defined in [34] and let $x \in \{\perp, \top\}$ be a syntactic locality notion. Only the normalization rules $NR1$, $NR2$ and $NR11$ are safe for x .* $\dashv$

Proof. Proposition 6.25 follows from the definition of the normalization rules (cf. [34]) and Definition 6.24. $\square$

Proposition 6.26. *Let $\mathcal{O}$ be an ontology and let $\alpha \in \mathcal{O}$ be an axiom. Let NRi be a safe normalization rule and let $x \in \{\perp, \top\}$ be a syntactic locality notion. Let $\mathcal{O}' = (\mathcal{O} \setminus \{\alpha\}) \cup \text{norm}_{NRi}(\alpha)$. Let $\geq_B(\mathcal{H})$ be the B -connectivity relation in the x -ADH $\mathcal{H}$ of the ontology $\mathcal{O}$ and let $\geq_B(\mathcal{H}')$ be the B -connectivity relation in the x -ADH $\mathcal{H}'$ of the ontology $\mathcal{O}'$.*

Then: $\geq_B(\mathcal{H}) = \geq_B(\mathcal{H}')$. $\dashv$

Proof. Proposition 6.26 follows from Definition 6.24 and Proposition 6.6. $\square$

Similarly, we can show that if NRi is a normalization rule that it is not safe, then B -connectivity of the x -ADH of an ontology might not be preserved after NRi is used to normalize the ontology.

Proposition 6.27. *Let $x \in \{\perp, \top\}$ be a syntactic locality notion.*

There exists an ontology $\mathcal{O}$, an axiom $\alpha \in \mathcal{O}$ and a not safe normalization rule NRi such that $\geq_B(\mathcal{H}) \neq \geq_B(\mathcal{H}')$, where $\geq_B(\mathcal{H})$ is the B -connectivity relation in the x -ADH $\mathcal{H}$ of the ontology $\mathcal{O}$, $\geq_B(\mathcal{H}')$ is the B -connectivity relation in the x -ADH $\mathcal{H}'$ of the ontology $\mathcal{O}' = (\mathcal{O} \setminus \{\alpha\}) \cup \text{norm}_{NRi}(\alpha)$. $\dashv$

Proof. Proposition 6.27 follows from Definition 6.24 and Examples 6.14 & 6.20. $\square$

We can extend Proposition 6.26 to show that B -connectivity in the Σ -extension of an ADH is also preserved.

Proposition 6.28. *Let $\mathcal{O}$ be an ontology and let $\alpha \in \mathcal{O}$ be an axiom. Let NRi be a safe normalization rule and let $x \in \{\perp, \top\}$ be a syntactic locality notion. Let $\mathcal{O}' = (\mathcal{O} \setminus \{\alpha\}) \cup \text{norm}_{NRi}(\alpha)$ and let $\Sigma \subseteq \text{sig}(\mathcal{O})$ be a signature. Let $\geq_B(\mathcal{H})$ be the B -connectivity relation in the Σ -extension of the x -ADH $\mathcal{H}$ for the ontology $\mathcal{O}$ and let $\geq_B(\mathcal{H}')$ be the B -connectivity relation in the Σ -extension of the x -ADH $\mathcal{H}'$ for the ontology $\mathcal{O}'$.*

Then: $\geq_B(\mathcal{H}) = \geq_B(\mathcal{H}')$. $\dashv$

Proof. Proposition 6.28 follows from Definition 6.24 and Proposition 6.7. $\square$

Similarly, we can extend Proposition 6.27 to show that B -connectivity in the Σ -extension of an ADH might not be preserved if we use normalization rules that are not safe.

Proposition 6.29. *Let $x \in \{\perp, \top\}$ be a syntactic locality notion.*

There exists an ontology $\mathcal{O}$, an axiom $\alpha \in \mathcal{O}$ and a not safe normalization rule N_{Ri} such that $\geq_B(\mathcal{H}) \neq \geq_B(\mathcal{H}')$, where $\geq_B(\mathcal{H})$ is the B -connectivity relation in the Σ -extension of the x -ADH $\mathcal{H}$ for the ontology $\mathcal{O}$, $\geq_B(\mathcal{H}')$ is the B -connectivity relation in the Σ -extension of the x -ADH $\mathcal{H}'$ for the ontology $\mathcal{O}' = (\mathcal{O} \setminus \{\alpha\}) \cup \text{norm}_{N_{Ri}}(\alpha)$. $\dashv$

Proof. Proposition 6.29 follows from Definition 6.24 and Examples 6.14 & 6.20. $\square$

By Propositions 6.27 & 6.29, we conclude that the B -connectivity relation in the x -ADH (or its Σ -extension) of an ontology might not be preserved in the x -ADH (or its Σ -extension) of the normalized ontology if this ontology has been normalized using rules that are not safe. The MLSs and signatures of the axioms of an ontology faithfully encode the hyperedges of a x -ADH (or its Σ -extension). When an axiom is normalized using normalization rules that are not safe, the MLSs and signature of the axiom are not preserved in the new axioms in normal form. This explains why the B -connectivity relation in the x -ADH for the normalized ontology might differ from the B -connectivity relation in the x -ADH for the original ontology.

Examples 6.20 & 6.23 illustrate that repairing the B -connectivity relation in a x -ADH it does not seem to be easy. However, we cannot discard that there might be alternative approaches to repair the B -connectivity relation in a x -ADH that we did not consider yet. In particular, for the x -ADH of an ontology that has been normalized using the normalization rules defined in [34]. Note that without a method that successfully repairs the B -connectivity in a x -ADH, we cannot apply the normalization rules defined in [34] to normalized *SROIQ* ontologies and create from the normalized versions of these ontologies x -ADHs (or their directed labelled graph representation).

It might be also possible to relax the conditions that an axiom in normal form must fulfil to be safe. In particular, normalization rules might introduce fresh symbols in the MLSs and signatures of the new axioms in normal form. This implies that the MLSs and signatures of these axioms are modified to incorporate the fresh symbols but this not might not alter the B -connectivity relation of the x -ADH of the original ontology. This is a possibility that the current definition of safe normalization rule (cf. Definition 6.24) does not consider.

6.3 Directed Labelled Graph Traversal

In this section, we present two new algorithms to traverse directed labelled graphs. These algorithms are based on the notion of Σ -connectivity (cf. Definition 6.4) in directed labelled graphs that encode axiom dependency hypergraphs. Note that Σ -connectivity in a directed

labelled graph corresponds to B -connectivity in the axiom dependency hypergraphs (cf. Propositions 6.5).

Both algorithms try to reproduce how breadth-first search (BFS) and depth-first search (DFS) traverse a directed graph.

6.3.1 Breadth-first Search Algorithm

The BFS algorithm for directed labelled graphs is depicted in Figures 6.3 – 6.4. There are two principal functions and three auxiliary functions. $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ are the principal functions. Like the BFS algorithm for directed hypergraphs (cf. Figures 2.13 & 2.14), lines in red and labelled with the symbol ‘+’ have been added to explicitly store the *BFS-graph*, a directed labelled graph that indicates the order in which the nodes of the labelled graph-encoding of an ADH were traversed by BFS (Definition 6.30). Every node of a BFS-graph is labelled with two timestamps: one timestamp indicating when the node was *discovered* (i.e., when it was visited for the first time), and a second timestamp stating when the node has been *explored* (i.e., when all its direct successors have been discovered). We use $\text{discovered-order}(v)$ and $\text{explored-order}(v)$ to refer to the timestamps that indicate when the node v was discovered and explored, respectively.

There might be several BFS-graphs of labelled graph-encoding of an ADH, depending on the order in which the nodes are traversed by the BFS algorithm. The traversal order is stored in the node labels of a BFS-graph. We define a BFS-graph computed by BFS on directed labelled graphs as follows.

Definition 6.30 (BFS-graph). *Let $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ be the x -ADH of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of $\mathcal{H}_\mathcal{O}^x$. Let $\text{main}(\cdot)$ be the function depicted in Figure 6.3. A breadth-first-search graph (BFS-graph) of $\mathcal{G}$ is a directed labelled graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{L}')$ with $\mathcal{G}' = \text{main}(\mathcal{G})$. $\dashv$*

The fundamental objective of function $\text{main}(\cdot)$ is to ensure that all nodes of the input directed labelled graph are visited (for-loop in Lines 8–14). This function is identical to the main function of BFS for directed hypergraphs depicted in Figure 2.13.

Given a node of a labelled graph, the function $\text{visit}(\cdot)$ identifies which other nodes of the graph are Σ -connected from this node. This function is very similar to the function $\text{visit}(\cdot)$ of BFS for directed hypergraphs (cf. Figure 2.14).

To determine which nodes are Σ -connected from a particular node, we use the MLSs and signature associated to each node of the labelled graph. Lines 10–12 in the function $\text{visit}(\cdot)$ for labelled graphs, compute the successors of the node currently visited using three auxiliary

```

function main( $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ )
1:  $\mathcal{V}' := \{\}$  +
2:  $\mathcal{E}' := \{\}$  +
3: for every  $v \in \mathcal{V}$  do
4:   discovered-order( $v$ ) := -1
5:   explored-order( $v$ ) := -1
6: end-for
7: order-counter := 0
8: for every  $v \in \mathcal{V}$  do
9:   if (discovered-order( $v$ ) = -1) then
      // Visit the node  $v$ 
10:   for every  $w \in \mathcal{V}$  do: visited( $w$ ) := false
11:   visit( $v$ )
12:   end-if
13: end-for
14:  $\mathcal{L}' := \{(\text{discovered-order}(v), \text{explored-order}(v)) \mid v \in \mathcal{V}\}$  +
15: return  $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{L}')$  +

```

Fig. 6.3 BFS main function for directed labelled graphs.

functions depicted in Figure 6.4: compute-successors-DG($\cdot, \cdot$), compute-successors-DLG($\cdot, \cdot, \cdot$) and get-signatures($\cdot$).

The auxiliary function compute-successors-DG($\cdot, \cdot$) identifies the successors of a node or set of nodes that are connected using simple hyperedges. Simple hyperedges are explicitly represented in the labelled graph. Using an adjacency-list to store these hyperedges, we can determine the successors of a particular node very efficiently.

The function compute-successors-DLG($\cdot, \cdot, \cdot$) has as input parameters, a labelled graph, a signature and a set of nodes. This function identifies which nodes of the labelled graph that are not in a particular set, have at least one MLS covered by the input signature. In other words, this function determines which nodes of the labelled graph that are not in a particular set are Σ -connected given a specific signature.

The third auxiliary function, get-signatures($\cdot$), collects the signatures of a set of nodes of a labelled graph. The following example illustrates how BFS for directed labelled graphs works.

Example 6.31. Let $\mathcal{O} = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$ be the ontology of Example 6.10. Let $\mathcal{H}_{\mathcal{O}}^{\perp} = (\mathcal{V}_{\mathcal{O}}^{\perp}, \mathcal{E}_{\mathcal{O}}^{\perp})$ be the $\perp$ -ADH of $\mathcal{O}$ and let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of $\mathcal{H}_{\mathcal{O}}^{\perp}$. Both are depicted in Figure 6.1.

```

function compute-successors-DG( $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}), S$ )
1:  $S' := \{\}$ 
2: for every  $v \in S$  do
3:   for every  $s \in \mathcal{E}(v)$  do
4:     if  $(s \notin S)$  and  $(\text{visited}(s) = \text{false})$  then
5:        $S' := S' \cup \{s\}$ 
6:     end-if
7:   end-for
8: end-for
9: return  $S'$ 

function compute-successors-DLG( $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}), \Sigma, S$ )
1:  $S' := \{\}$ 
2: for every  $v \in \mathcal{V}$  do
3:   if  $(v \notin S)$  and  $(\text{visited}(v) = \text{false})$  then
4:     for every  $\Sigma' \in \text{MLS}(v)$  do
5:       if  $(\Sigma' \subseteq \Sigma)$  then
6:          $S' := S' \cup \{v\}$ 
7:       end-if
8:     end-for
9:   end-if
10: end-for
11: return  $S'$ 

function get-signatures( $S$ )
1:  $\Sigma := \{\}$ 
2: for every  $v \in S$  do
3:    $\Sigma := \Sigma \cup \text{sig}(v)$ 
4: end-for
5: return  $\Sigma$ 

```

Fig. 6.4 BFS auxiliary functions for directed labelled graphs.

```

function visit( $v$ )
    // v has been discovered
1:  $Q := \{\}$ 
2: enqueue( $Q, v$ )
3: order-counter := order-counter + 1
4: discovered-order( $v$ ) := order-counter
5: visited( $v$ ) := true
6: sig-successors :=  $\{\}$ 
7:  $\mathcal{V}' := \mathcal{V}' \cup \{v\}^+$ 
    // Identify B-connected nodes from v
8: while  $Q \neq \{\}$  do
9:      $u := \text{dequeue}(Q)$ 
10:    successors := compute-successors-DG( $\mathcal{G}, \{u\}$ )
11:    sig-successors := sig-successors  $\cup$  get-signatures(successors)  $\cup$  sig( $u$ )
12:    successors := successors  $\cup$  compute-successors-DLG( $\mathcal{G}, \text{sig-successors}, \text{successors} \cup \{u\}$ )
13:    for every  $w \in \text{successors}$  do
14:         $\mathcal{E}' := \mathcal{E}' \cup \{(u, w)\}^+$ 
15:        if (visited( $w$ ) = false) then
16:            visited( $w$ ) := true
17:            enqueue( $Q, w$ )
18:             $\mathcal{V}' := \mathcal{V}' \cup \{w\}^+$ 
19:            if (discovered-order( $w$ ) = -1) then
                // w has been discovered
20:                order-counter := order-counter + 1
21:                discovered-order( $w$ ) := order-counter
22:            end-if
23:        end-if
24:    end-for
25:    if (explored-order( $u$ ) = -1) then
        // u has been explored
26:        order-counter := order-counter + 1
27:        explored-order( $u$ ) := order-counter
28:    end-if
29: end-while

```

Fig. 6.5 BFS visit function for directed labelled graphs.

The signatures and minimal non- $\perp$ -locality signatures of the axioms of $\mathcal{O}$ that labelled the nodes of $\mathcal{G}$ are the following:

$$\begin{aligned} MLS^\perp(\alpha_1) &= \{\{F\}\} & \text{sig}(\alpha_1) &= \{F, H\} \\ MLS^\perp(\alpha_2) &= \{\{B\}\} & \text{sig}(\alpha_2) &= \{A, B, F, G\} \\ MLS^\perp(\alpha_3) &= \{\{A\}\} & \text{sig}(\alpha_3) &= \{A, B, C, E, F\} \\ MLS^\perp(\alpha_4) &= \{\{C, G\}\} & \text{sig}(\alpha_4) &= \{C, D, G\} \\ MLS^\perp(\alpha_5) &= \{\{D, E\}\} & \text{sig}(\alpha_5) &= \{A, D, E\} \end{aligned}$$

Suppose that during the execution of the function $\text{main}(\mathcal{G})$, $\text{visit}(\alpha_1)$ is called first. As a result, the node α_1 is marked as the first node discovered in $\mathcal{G}$ and as visited: $\text{discovered-order}(\alpha_1) := 1$ (Line 4) and $\text{visited}(\alpha_1) := \text{true}$ (Line 5). Moreover, $\mathcal{V}' := \{\alpha_1\}$ (Line 7). The node α_1 is enqueued in Q (Line 2) and later dequeued in Line 9. After $\text{sig-successors} := \{F, H\}$ (Lines 11) is computed, there is not any successors of α_1 identified by the functions $\text{compute-successors-DG}(\cdot, \cdot)$ and $\text{compute-successors-DLG}(\cdot, \cdot, \cdot)$ (Lines 10&12). Therefore, α_1 is marked as the first node explored in $\mathcal{G}$: $\text{explored-order}(\alpha_1) := 2$ (Line 27). The execution of $\text{visit}(\alpha_1)$ concludes and continues the execution of $\text{main}(\mathcal{G})$.

Suppose that the next node to be visited is α_2 . The function $\text{main}(\mathcal{G})$ initialized every node of the $\mathcal{G}$ as non-visited (Line 10): $\text{visited}(\alpha_i) := \text{false}$, for every node $\alpha_i \in \mathcal{V}$. Then, $\text{visit}(\alpha_2)$ is called from $\text{main}(\mathcal{G})$.

The node α_2 is marked as discovered and visited: $\text{discovered-order}(\alpha_2) := 3$ and $\text{visited}(\alpha_2) := \text{true}$. α_2 is enqueued in Q and later dequeued from Q . As a result, Q becomes empty. The function $\text{compute-successors-DL}(\mathcal{G}, \{\alpha_2\})$ identifies the successors α_1 and α_3 that are connected from α_2 by simple hyperedges. Then, $\text{successors} := \{\alpha_1, \alpha_3\}$ and the signatures of α_1 , α_2 and α_3 are stored in sig-successors and therefore, $\text{sig-successors} := \{A, B, C, E, F, G, H\}$ (Line 11). The function $\text{compute-successors-DLG}(\mathcal{G}, \text{sig-successors}, \text{successors} \cup \{\alpha_2\})$ identifies also as a successor of α_2 , the node α_4 because $MLS^\perp(\alpha_4) \subseteq \text{sig-successors}$ and $\alpha_4 \notin \text{successors}$. Then, α_4 is added to successors (Line 12) and as result: $\text{successors} := \{\alpha_1, \alpha_2, \alpha_3\} \cup \{\alpha_4\}$. α_4 is also enqueued in Q and with this node, $Q := \{\alpha_1, \alpha_3\} \cup \{\alpha_4\}$. Except α_1 that was already in $\mathcal{V}'$, the nodes α_2 , α_3 and α_4 are also stored in $\mathcal{V}'$ and as result: $\mathcal{V}' := \{\alpha_1\} \cup \{\alpha_2, \alpha_3, \alpha_4\}$. The following edges are also added to the BFS-graph: $\mathcal{E}' := \{e_1 = (\alpha_2, \alpha_1), e_2 = (\alpha_2, \alpha_3), e_{10} = (\alpha_2, \alpha_4)\}$ (Line 14). The nodes α_3 and α_4 are marked as discovered (Line 21): $\text{discovered-order}(\alpha_3) := 4$, $\text{discovered-order}(\alpha_4) := 5$ and $\text{visited}(\alpha_3) := \text{visited}(\alpha_4) := \text{true}$ (Line 16).

In Line 27, α_2 is marked as explored ($\text{explored-order}(\alpha_2) := 6$) and α_1 is dequeued from Q (Line 9). The auxiliary function $\text{compute-successors-DL}(\mathcal{G}, \{\alpha_1\})$ returns the empty set (α_1 does not have successors) but $\text{compute-successors-DLG}(\mathcal{G}, \text{sig-successors}, \text{successors} \cup$

$\{\alpha_1\}$) returns the following set of nodes: $\text{successors} := \{\alpha_2, \alpha_3, \alpha_4\}$ (Line 12). This result might be confusing because it seems that we are computing the successors of α_1 and this node does not have successors. Recall that sig-successors currently stores the signatures of the axioms α_1 , α_2 and α_3 but not α_4 . This means that we are not computing the successors of α_1 but identifying any node Σ -connected from α_1 , α_2 and α_3 . In fact, $\text{visit}(\alpha_2)$ is computing the Σ -connected component of α_2 .

The nodes currently stored in successors have been visited before and they are not enqueued in Q again. However, we include the following edges in $\mathcal{E}'$: $\{e_{11} = (\alpha_1, \alpha_2), e_{12} = (\alpha_1, \alpha_3), e_{13} = (\alpha_1, \alpha_4)\}$ (Line 14). This concludes the “re-exploration” of α_1 and α_3 is dequeued from Q (Line 9). After $\text{compute-successors-DL}(\mathcal{G}, \{\alpha_3\})$ and $\text{compute-successors-DLG}(\mathcal{G}, \text{sig-successors}, \text{successors} \cup \{\alpha_3\})$ are executed, $\text{successors} := \{\alpha_1, \alpha_2, \alpha_4\}$. All these nodes have been visited and α_3 is marked as explored: $\text{explored-order}(\alpha_3) := 7$ (Line 27). In addition, the following edges are added in $\mathcal{E}'$: $\{e_3 = (\alpha_3, \alpha_1), e_4 = (\alpha_3, \alpha_2), e_{14} = (\alpha_3, \alpha_4)\}$ (Line 14).

The node α_4 is dequeued from Q . sig-successors is extended with the signature of α_4 : $\text{sig-successors} := \{A, B, C, E, F, G, H\} \cup \{D\}$. The auxiliary function $\text{compute-successors-DL}(\mathcal{G}, \{\alpha_4\})$ returns the empty set but the function $\text{compute-successors-DLG}(\mathcal{G}, \text{sig-successors}, \text{successors} \cup \{\alpha_4\})$ returns the following set of nodes: $\text{successors} := \{\alpha_1, \alpha_2, \alpha_3, \alpha_5\}$ (Line 12). The nodes α_1 , α_2 and α_3 have been visited before but this is not the case of α_5 . In Line 16, α_5 is marked as visited ($\text{visited}(\alpha_5) := \text{true}$) and in Line 21, it is marked as discovered ($\text{discovered-order}(\alpha_5) := 8$). Moreover, α_5 is enqueued in Q and the node is also added into $\mathcal{V}'$: $\mathcal{V}' := \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\} \cup \{\alpha_5\}$. Furthermore, the following edges are added in $\mathcal{E}'$: $\{e_{15} = (\alpha_4, \alpha_1), e_5 = (\alpha_4, \alpha_2), e_{16} = (\alpha_4, \alpha_3), e_{17} = (\alpha_4, \alpha_5)\}$. No new successors are identified from α_4 and this node is marked as explored: $\text{explored-order}(\alpha_4) := 9$.

The node α_5 is dequeued from Q . sig-successors is extended with the signature of α_5 but no new symbols are added: $\text{sig-successors} := \{A, B, C, D, E, F, G, H\} \cup \{\}$. The function $\text{compute-successors-DL}(\mathcal{G}, \{\alpha_5\})$ returns the empty set but $\text{compute-successors-DLG}(\mathcal{G}, \text{sig-successors}, \text{successors} \cup \{\alpha_5\})$ identifies the following successors: $\text{successors} := \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ (Line 12). These successors have been visited before α_5 is marked as explored: $\text{explored-order}(\alpha_5) := 10$. Before the exploration of α_5 concludes, the following edges are added into $\mathcal{E}'$: $\{e_{18} = (\alpha_5, \alpha_1), e_{19} = (\alpha_5, \alpha_2), e_6 = (\alpha_5, \alpha_3), e_{20} = (\alpha_5, \alpha_4)\}$.

The execution of $\text{visit}(\alpha_2)$ concludes and continues the execution of $\text{main}(\mathcal{G})$. All nodes of $\mathcal{G}$ have been discovered and the execution of $\text{main}(\mathcal{G})$ finishes and returns the BFS-graph $\mathcal{G}'$ which is depicted in Figure 6.6. $\dashv$

Fig. 6.6 (Example 6.31): BFS-graph for directed labelled graphs.

Example 6.31 shows that node to node Σ -connectivity in the labelled graph-encoding of an ADH is not preserved in the BFS-graph returned by the function $\text{main}(\cdot)$ depicted in Figure 6.3.

Proposition 6.32 (Node to node connectivity in BFS-graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of a x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{L}')$ be the BFS-graph produced by the function $\text{main}(\cdot)$ depicted in Figure 6.3.*

Then: $\geq_{\Sigma} \neq \geq_{\mathcal{G}'}$, where $\geq_{\Sigma}$ is the Σ -connectivity relation in $\mathcal{G}$ and $\geq_{\mathcal{G}'}$ is the connectivity relation in $\mathcal{G}'$. $\dashv$

Suppose that we change Line 14 in the function $\text{visit}(\cdot)$ as follows:

$$15 : \mathcal{E}' := \mathcal{E}' \cup \{(v, w)\}^+$$

Recall that v is the input node of the function $\text{visit}(\cdot)$ and w is one of its successors in the labelled graph $\mathcal{G}$. It can readily be seen that node-to-node Σ -connectivity in the labelled graph is also not preserved in the BFS-graph. To produce a BFS-graph that preserves node-to-node Σ -connectivity, we have to modify function $\text{main}(\cdot)$ also. In particular, we have to delete the if-condition in Line 9. By this if-condition, the function $\text{visit}(\cdot)$ can only be called if the input node has not been discovered before. Note that correctness of the algorithm is not affected by this change. However, it might execute more calls of the function $\text{visit}(\cdot)$ that are strictly needed to traverse the entire input labelled graph.

The next proposition can readily be seen and it establishes the correctness of the auxiliary functions depicted in Figure 6.4.

Proposition 6.33 (Correctness of the auxiliary functions of BFS for directed labelled graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of a x -ADH for an ontology $\mathcal{O}$, where $x \in$*

$\{\perp, \top\}$. Let $S, S', S'' \subseteq \mathcal{V}$ be three sets of nodes, where $S' = \{v \in \mathcal{V} \mid \text{visited}(v) = \text{true}\}$, $S \neq S'$ and $S'' = S \cup S'$. Let Σ be a signature. Let $\text{get-signatures}(\cdot)$, $\text{compute-successors-DL}(\cdot, \cdot)$ and $\text{compute-successors-DLG}(\cdot, \cdot, \cdot)$ be the functions depicted in Figure 6.4. Then:

- (i) $\text{get-signatures}(S) = \text{sig}(S)$.
- (ii) $\text{compute-successors-DL}(\mathcal{G}, S) = \{\mathcal{E}(v) \in (\mathcal{V} \setminus S'') \mid \text{for every } v \in S\}$.
- (iii) $\text{compute-successors-DLG}(\mathcal{G}, \Sigma, S) = \{v \in (\mathcal{V} \setminus S'') \mid \Sigma_v \subseteq \Sigma, \text{ for some } \Sigma_v \in \text{MLS}^x(v)\}$.

⊥

Item (iii) of Proposition 6.33 is similar to the definition of Σ -connectivity (cf. Definition 6.4) and how a Σ -connected component is computed. In particular, the result of the execution of the function $\text{compute-successors-DLG}(\mathcal{G}, \Sigma, S)$ is a subset of the Σ -connected component of the set of nodes S in the labelled graph $\mathcal{G}$.

To compute the entire Σ -connected component of a set of nodes S , we can iteratively execute the function $\text{compute-successors-DLG}(\mathcal{G}, \Sigma, \{S\})$, where $\Sigma = \text{sig}(S)$. In each iteration, Σ is extended with the signature of the new nodes that are found by this function. The extended Σ is used as input signature in the next iteration and so on. The iterative process should end when no more new nodes are found.

Next, we show the time complexity of the auxiliary functions of the BFS algorithm for directed labelled graphs.

Proposition 6.34 (Time complexity of the auxiliary functions of BFS for directed labelled graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of a x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let Σ be a signature. Let $S, S', S'' \subseteq \mathcal{V}$ be three sets of nodes, where $S' = \{v \in \mathcal{V} \mid \text{visited}(v) = \text{true}\}$, $S \neq S'$ and $S'' = S \cup S'$. Let $m = \max \{|\mathcal{E}(v)| \mid \text{for every } v \in \mathcal{V}\}$. Let $n = \max \{|\text{MLS}^x(v)| \mid \text{for every } v \in \mathcal{V}\}$. Let $k = \max \{|\Sigma_v| \mid \text{for every } \Sigma_v \in \text{MLS}^x(v) \text{ and for every } v \in \mathcal{V}\}$.*

Let $\text{get-signatures}(\cdot)$, $\text{compute-successors-DL}(\cdot, \cdot)$ and $\text{compute-successors-DLG}(\cdot, \cdot, \cdot)$ be the functions depicted in Figure 6.4. Then:

- (i) $O(|S|)$ is the time complexity of $\text{get-signatures}(S)$.
- (ii) $O(|S| \times m)$ is the time complexity of $\text{compute-successors-DL}(\mathcal{G}, S)$.
- (iii) $O(|\mathcal{V}| + |\mathcal{V} \setminus S''| \times n \times k)$ is the time complexity of $\text{compute-successors-DLG}(\mathcal{G}, \Sigma, S)$.

⊥

Proof. We start the proof of Proposition 6.34 by showing item (i). We assume that there is a data-structure that stores the labels of each node of the labelled graph. This includes the signature associated to each node. Therefore, the function $\text{compute-successors-DL}(\cdot, \cdot)$ only has to iterate over the input set of nodes and for each node retrieves the signature in constant time. Hence item (i) holds.

Consider item (ii). The function $\text{compute-successors-DL}(\cdot, \cdot)$ iterates over a set of input nodes. For each node, the function retrieves the successors of the node connected by directed edges. We assume that there is a data-structure that represents a adjacency-list where each successor of a particular node can be retrieved in constant time. Then, item (ii) holds.

Consider item (iii). Similar to item (i), we assume that there is a data-structure that stores the labels of each node of the labelled graph. Using this data-structure, it is possible to retrieve each MLS of a particular axiom in constant time using the function $\text{get-MLS}(\cdot, \cdot)$. We also assume that the signature given to the function $\text{compute-successors-DLG}(\cdot, \cdot, \cdot)$ is internally represented using a specific data-structure. This data-structure is able to verify the presence of a particular symbol in constant time. Then, item (iii) holds. $\square$

The next proposition can readily be seen.

Proposition 6.35 (Correctness of BFS for directed labelled graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of a x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $s \in \mathcal{V}$ be a node. Let $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ be the functions depicted in Figures 6.3 & 6.5, respectively. Then:*

(i) $\text{main}(\mathcal{G})$ terminates.

(ii) $\text{visit}(s)$ visits all nodes in $\mathcal{G}$ that are Σ -connected from s .

(iii) $\text{main}(\mathcal{G})$ traverses $\mathcal{G}$ completely. $\dashv$

Proof. We start the proof of Proposition 6.35 by showing item (i). It can readily be seen that after the function $\text{visit}(\cdot)$ is called from the function $\text{main}(\cdot)$, each node of the directed labelled graph can be visited at most one time. The function $\text{main}(\cdot)$ iterates over the nodes of the labelled graph and only when a node has not been discovered, the function $\text{visit}(\cdot)$ is called. Hence item (i) holds.

Consider item (ii). Let $s \in \mathcal{V}$ be the input node of the function $\text{visit}(\cdot)$. By Proposition 6.33, the auxiliary functions $\text{compute-successors-DL}(\cdot, \cdot, \cdot)$ and $\text{compute-successors-DLG}(\cdot, \cdot, \cdot)$ identify the successors of node s . The application of these functions on the nodes found in each iteration of the while-loop in Lines 8–29 computes the Σ -connected component for the signature of node s . Hence item (ii) holds.

Consider item (iii). By item (ii), $\text{visit}(s)$ visits all nodes in $\mathcal{G}$ that are Σ -connected from s . The function $\text{main}(\cdot)$ iterates over all nodes of $\mathcal{G}$ that has not been discovered yet. Then item (iii) holds. $\square$

We show that the time complexity of BFS for directed labelled graphs is polynomial with respect to the number of nodes of the labelled graph and the number of MLSs associated to each node. Recall that the number of MLSs of an axiom might be exponential with respect to the signature of the axiom (cf. Example 4.8).

Proposition 6.36 (Time complexity of BFS for directed labelled graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of a x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $n = \max \{|MLS^x(v)| \mid \text{for every } v \in \mathcal{V}\}$. Let $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ be the functions depicted in Figures 6.3 & 6.5, respectively.*

Then: the time complexity of function $\text{main}(\cdot)$ is $O(|\mathcal{V}|^3 \times n)$. $\dashv$

Proof. Let Σ be a signature. Let $m = \max \{|\mathcal{E}(v)| \mid \text{for every } v \in \mathcal{V}\}$. Let $k = \max \{|\Sigma_v| \mid \text{for every } \Sigma_v \in \text{MLS}^x(v) \text{ and for every } v \in \mathcal{V}\}$. Let $\text{get-signatures}(\cdot)$, $\text{compute-successors-DL}(\cdot, \cdot)$ and $\text{compute-successors-DLG}(\cdot, \cdot, \cdot)$ be the functions depicted in Figure 6.4.

We start the proof of Proposition 6.36 by studying the time complexity of the function $\text{visit}(\cdot)$ depicted in Figure 6.5. There is a nested for-loop (Lines 13–24) inside of the while-loop (Lines 8–29). Both loops iterates over the nodes of the labelled graph $\mathcal{G}$. However, if we consider all the iterations of the while-loop, the for-loop iterates only once over each new successor discovered by the auxiliary functions. Moreover, the while-loop iterates only once over the nodes in Q and Q stores only one time the new successors discovered by the auxiliary functions. Then, the complete execution of both loops requires $2 \times |\mathcal{V}|$ iterations in the worst-case. Inside the for-loop all the different steps are executed in constant time. This is not the case of the while-loop where the three auxiliary functions depicted in Figure 6.4 are executed in each iteration.

Simplifying the time complexity results of Proposition 6.34, we can assume that the three auxiliary functions execute $|\mathcal{V}| + (|\mathcal{V}| \times m) + (|\mathcal{V}| + |\mathcal{V}| \times n \times k)$ steps each of them in constant time. Let us assume that $n \gg k$ and $n \gg m$. We can simplify a bit more and conclude that the execution of the three auxiliary functions requires $|\mathcal{V}| \times n$ steps in the worst-case. Therefore, the time complexity of the function $\text{visit}(\cdot)$ for an input labelled graph $\mathcal{G}$ is $O(|\mathcal{V}|^2 \times n)$.

The function $\text{main}(\cdot)$ has two nested for-loops that iterates over all the nodes of the input labelled graph $\mathcal{G}$. This implies $|\mathcal{V}| \times |\mathcal{V}|$ iterations. However, it is only from the outer for-loop where the function $\text{visit}(\cdot)$ is called. Hence, the time complexity of the function $\text{main}(\cdot)$ is $O(|\mathcal{V}|^3 \times n)$. $\square$

```

function main( $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ )
1:  $\mathcal{V}' := \{\}; \mathcal{E}' := \{\}^+$ 
2:  $\text{list-of-sccs} := \{\{\}\}^*$ 
3:  $\text{stack} := []$ 
4: for every  $v \in \mathcal{V}$  do
5:    $\text{discovered-order}(v) := -1; \text{explored-order}(v) := -1$ 
6:    $\text{visited-successors}(v) := \{\}$ 
7:    $\text{sig-visited-successors}(v) := \{\}$ 
8: end-for
9:  $\text{order-counter} := 0$ 
10: for every  $v \in \mathcal{V}$  do
11:   if ( $\text{discovered-order}(v) = -1$ ) then
12:      $\text{visit}(v)$ 
13:   end-if
14: end-for
15:  $\mathcal{L}' := \{(\text{discovered-order}(v), \text{explored-order}(v), \text{min-order}(v)) \mid v \in \mathcal{V}\}^+$ 
16: return  $\mathcal{G} = (\mathcal{V}', \mathcal{E}', \mathcal{L}'); \text{list-of-sccs}^*$ 

```

Fig. 6.7 DFS/Approximation/Tarjan main function for directed labelled graphs.

6.3.2 Depth-first Search Algorithm

In this section, we discuss how to adapt the DFS algorithm for directed hypergraphs depicted in Figures 3.1 & 3.2 to directed labelled graphs. Recall that this version of DFS for directed hypergraphs produces a DFS-graph that preserves node-to-node B -connectivity of the input directed hypergraph. The goal is to achieve the same result using as input a directed labelled graph instead of a directed hypergraph.

We have integrated three algorithms in Figures 6.7 & 6.9. DFS for directed labelled graphs corresponds to the lines in black. DFS-approximation represented by the black and red lines (lines labelled with the symbol ‘+’) extends DFS by storing the DFS-graph in $\mathcal{G}'$. Tarjan for directed labelled graphs is represented by black and blue lines (lines labelled with the symbol ‘*’).

The function $\text{main}(\cdot)$ ensures that all nodes of the input directed labelled graph are visited (for-loop in Lines 10–14). The result of the execution of $\text{main}(\cdot)$ is a DFS-graph $\mathcal{G}'$ and the list of strongly- B -connected components of the ADH encoded by the labelled graph $\mathcal{G}$. This function is like the main function of DFS for directed hypergraphs depicted in Figure 3.1.

Given a node of a labelled graph, the recursive function $\text{visit}(\cdot)$ identifies which other nodes of the graph are Σ -connected from this node. In addition, $\text{visit}(\cdot)$ identifies which

function compute-successors-DG($\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}), S_1, S_2$)

```

1:  $Q := \{\}$ 
2: for every  $v \in S_1$  do
3:   for every  $s \in \mathcal{E}(v)$  do
4:     if ( $s \notin S_2$ ) then
5:       enqueue( $Q, s$ )
6:     end-if
7:   end-for
8: end-for
9: return  $Q$ 

```

function compute-successors-DLG($\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}), \Sigma, S$)

```

1:  $Q := \{\}$ 
2: for every  $v \in \mathcal{V}$  do
3:   if ( $v \notin S$ ) then
4:     for every  $\Sigma' \in \text{MLS}(v)$  do
5:       if ( $\Sigma' \subseteq \Sigma$ ) then
6:         enqueue( $Q, v$ )
7:       end-if
8:     end-for
9:   end-if
10: end-for
11: return  $Q$ 

```

Fig. 6.8 DFS/Approximation/Tarjan auxiliary functions for directed labelled graphs.

of these nodes are strongly- B -connected in the ADH encoded by the labelled graph. This function is similar to the function $\text{visit}(\cdot)$ of DFS for directed hypergraphs (cf. Figure 3.2).

Like BFS for directed labelled graphs, we have two auxiliary functions to identify the successors of a set of nodes in the labelled graph. Both functions are depicted in Figure 6.8. The function $\text{compute-successors-DG}(\cdot, \cdot)$ identifies the successors that are Σ -connected from a set of nodes using some of the directed edges of the labelled graph. These successors can be identified very efficiently if the edges are stored using an adjacency-list. To identify any successor Σ -connected from a set of nodes, we use the function $\text{compute-successors-DLG}(\cdot, \cdot, \cdot)$. Given an input signature that represents the signature of a set of nodes, this function identifies which of their successors are not part of a given set of nodes.

Recall that a *DFS-graph* is a directed labelled graph that stores the order in which the nodes of a directed hypergraph were traversed by DFS (cf. Definition 3.11). Every node of a DFS-graph is labelled with two timestamps: one timestamp indicating when the node was *discovered* (i.e., when it was visited for the first time), and a second timestamp stating

```

function visit( $v$ ), where  $v \in \mathcal{V}$ 
1: order-counter := order-counter + 1
2: discovered-order( $v$ ) := order-counter
3: min-order( $v$ ) := order-counter
4: visited-successors( $v$ ) :=  $\{v\}$ 
5: sig-visited-successors( $v$ ) := sig( $v$ )
6: push(stack,  $v$ )
7:  $\mathcal{V}' := \mathcal{V}' \cup \{v\}$  +
   // Identify all non-visited successors of  $v$  connected by directed edges
8: non-visited-successors := compute-successors-DG( $\mathcal{G}$ ,  $\{v\}$ , visited-successors( $v$ ))
9: while ( $|\text{non-visited-successors}| \geq 1$ ) do
10:   $s := \text{dequeue}(\text{non-visited-successors})$ 
11:  if (discovered-order( $s$ ) = -1) then
12:    visit( $s$ ) // recursive call
13:    min-order( $v$ ) := min(min-order( $v$ ), min-order( $s$ ))
14:  else
15:    if stack contains  $s$  then
16:      min-order( $v$ ) := min(min-order( $v$ ), discovered-order( $s$ ))
17:    end-if
18:  end-if
19:  visited-successors( $v$ ) := visited-successors( $v$ )  $\cup$  visited-successors( $s$ )
20:  sig-visited-successors( $v$ ) := sig-visited-successors( $v$ )  $\cup$  sig-visited-successors( $s$ )
21:   $\mathcal{E}' := \mathcal{E}' \cup \{(v, s)\}$  // store approximating edge +
22:  if ( $|\text{non-visited-successors}| < 1$ ) then
23:    non-visited-successors := compute-successors-DLG( $\mathcal{G}$ , sig-visited-successors( $v$ ),
                                                         visited-successors( $v$ ))
24:  end-if
25: end-while
26: order-counter := order-counter + 1
27: explored-order( $v$ ) := order-counter
   // if  $v$  is a root-node, then update successor lists (and compute the SCC)
28: if (min-order( $v$ ) = discovered-order( $v$ )) then
29:    $\text{scc} := \{\}$  *
30:   repeat
31:      $s := \text{pop}(\text{stack})$ 
32:     visited-successors( $s$ ) := visited-successors( $v$ )
33:     sig-visited-successors( $s$ ) := sig-visited-successors( $v$ )
34:     add(scc,  $s$ ) *
35:   until ( $s = v$ )
36:   add(list-of-sccs, scc) *
37: end-if

```

Fig. 6.9 DFS/Approximation/Tarjan visit function for directed labelled graphs.

Fig. 6.10 (Example 6.38): DFS-graph for directed labelled graphs.

when the node has been *explored* (i.e., when all its successors have been discovered). We use $\text{discovered-order}(v)$ and $\text{explored-order}(v)$ to refer to the timestamps that indicate when the node v was discovered and explored, respectively.

There might be several DFS-graphs of the labelled graph-encoding of an ADH, depending on the order in which the nodes are traversed by the DFS algorithm. The traversal order is stored in the node labels of a DFS-graph. We define the DFS-graph computed by DFS (DFS-approximation) for directed labelled graphs as follows.

Definition 6.37 (DFS-graph). Let $\mathcal{H}_{\mathcal{O}}^x = (\mathcal{V}_{\mathcal{O}}^x, \mathcal{E}_{\mathcal{O}}^x)$ be the x -ADH of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of $\mathcal{H}_{\mathcal{O}}^x$. Let $\text{main}(\cdot)$ be the function depicted in Figure 6.7. A depth-first-search graph (DFS-graph) of $\mathcal{G}$ is a directed labelled graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{L}')$ with $\mathcal{G}' = \text{main}(\mathcal{G})$. $\dashv$

The following example illustrates how DFS (DFS-approximation) for directed labelled graphs works.

Example 6.38. Let $\mathcal{O} = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\}$ be the ontology of Examples 6.10 & ex:bfs-directed-labelled-graph. Let $\mathcal{H}_{\mathcal{O}}^{\perp} = (\mathcal{V}_{\mathcal{O}}^{\perp}, \mathcal{E}_{\mathcal{O}}^{\perp})$ be the $\perp$ -ADH of $\mathcal{O}$ and let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of $\mathcal{H}_{\mathcal{O}}^{\perp}$. Both are depicted in Figure 6.1.

The signatures and minimal non- $\perp$ -locality signatures of the axioms of $\mathcal{O}$ that labelled the nodes of $\mathcal{G}$ are the following:

$$\begin{aligned} MLS^\perp(\alpha_1) &= \{\{F\}\} & \text{sig}(\alpha_1) &= \{F, H\} \\ MLS^\perp(\alpha_2) &= \{\{B\}\} & \text{sig}(\alpha_2) &= \{A, B, F, G\} \\ MLS^\perp(\alpha_3) &= \{\{A\}\} & \text{sig}(\alpha_3) &= \{A, B, C, E, F\} \\ MLS^\perp(\alpha_4) &= \{\{C, G\}\} & \text{sig}(\alpha_4) &= \{C, D, G\} \\ MLS^\perp(\alpha_5) &= \{\{D, E\}\} & \text{sig}(\alpha_5) &= \{A, D, E\} \end{aligned}$$

Suppose that during the execution of the function $\text{main}(\mathcal{G})$, $\text{visit}(\alpha_1)$ is called first. As a result, the node α_1 is marked as the first node discovered in $\mathcal{G}$: $\text{discovered-order}(\alpha_1) = 1$ (Line 2) and $\text{min-order}(\alpha_1) = 1$ (Line 3). Moreover, $\text{visited-successors}(\alpha_1) = \{\alpha_1\}$ (Line 4), $\text{sig-visited-successors}(\alpha_1) = \{F, H\}$ (Line 5), $\text{stack} = \{\alpha_1\}$ (Line 6) and $\mathcal{V}' = \{\alpha_1\}$ (Line 7). The node α_1 does not have any successor connected by a directed edge and $\text{non-visited-successors} = \{\}$ (Line 8). Therefore, α_1 is marked as the first node explored in $\mathcal{G}$: $\text{explored-order}(\alpha_1) = 2$ (Line 27). Because α_1 is the root node ($\text{discovered-order}(\alpha_1) = \text{min-order}(\alpha_1)$) and the only node of a new strongly- Σ -connected component, the stack becomes empty: $\text{stack} = \{\}$ (Line 31). The execution of $\text{visit}(\alpha_1)$ concludes and continues the execution of $\text{main}(\mathcal{G})$.

Suppose that the next node to be visited is α_2 . Then, $\text{visit}(\alpha_2)$ is called from $\text{main}(\mathcal{G})$. As a result, the node α_2 is marked as discovered in $\mathcal{G}$: $\text{discovered-order}(\alpha_2) = 3$ (Line 2) and $\text{min-order}(\alpha_1) = 3$ (Line 3). Moreover, the following variables are also updated: $\text{visited-successors}(\alpha_2) = \{\alpha_2\}$ (Line 4), $\text{sig-visited-successors}(\alpha_2) = \{A, B, F, G\}$ (Line 5), $\text{stack} = \{\alpha_2\}$ (Line 6) and $\mathcal{V}' = \{\alpha_1, \alpha_2\}$ (Line 7).

α_2 has two successors: α_1 and α_3 . Each of these successors are connected from α_2 by a directed edge. Therefore, $\text{non-visited-successors} = \{\alpha_1, \alpha_3\}$ (Line 8). α_1 has been discovered before and this node and its signature are accumulated as follows: $\text{visited-successors}(\alpha_2) = \{\alpha_2\} \cup \{\alpha_1\}$ and $\text{sig-visited-successors}(\alpha_2) = \{A, B, F, G\} \cup \{F, H\}$. α_3 has not been discovered before and a recursive call of function $\text{visit}(\alpha_3)$.

During the execution $\text{visit}(\alpha_3)$, a new recursive call is executed: $\text{visit}(\alpha_4)$. Similarly from $\text{visit}(\alpha_4)$, the function $\text{visit}(\alpha_5)$ is also called. During the execution of these three recursive calls, the algorithm will identify that the nodes α_3 , α_4 and α_5 are in the same strongly- Σ -connected as α_2 , where α_2 is the root node. As a result, $\text{min-order}(\alpha_2) = \text{min-order}(\alpha_3) = \text{min-order}(\alpha_4) = \text{min-order}(\alpha_5) = 3$. The nodes $\alpha_2, \alpha_3, \alpha_4$ and α_5 are removed from the stack during the execution of $\text{visit}(\alpha_2)$ and after the other recursive calls have finished. The execution of $\text{visit}(\alpha_2)$ concludes and continues the execution of $\text{main}(\mathcal{G})$ that also ends because all nodes of $\mathcal{G}$ have been discovered.

Note that when each call of the function $\text{visit}(\cdot)$ concludes, all nodes have been marked as explored: $\text{explored-order}(\alpha_2) = 10$, $\text{explored-order}(\alpha_3) = 9$, $\text{explored-order}(\alpha_4) = 8$ and $\text{explored-order}(\alpha_5) = 7$ (Line 27). $\dashv$

Next, we show the following properties of DFS for the labelled graph-encoding of an ADH.

Lemma 6.39. *Let $\mathcal{H}_{\mathcal{O}}^x = (\mathcal{V}_{\mathcal{O}}^x, \mathcal{E}_{\mathcal{O}}^x)$ be the x -ADH of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of $\mathcal{H}_{\mathcal{O}}^x$. Let $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ be the functions depicted in Figures 6.7 & 6.9, respectively. Let $\mathcal{G}' = \text{main}(\mathcal{G})$ be a DFS-graph, where $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{L}')$. Let $\geq_{\Sigma}$ be the Σ -connectivity relation in $\mathcal{G}$ wrt. Σ , where $\Sigma = \emptyset$. Let $\geq_{\mathcal{G}'}$ be the connectivity relation in $\mathcal{G}'$. Let $\text{visited-successors} : \mathcal{V} \rightarrow 2^{\mathcal{V}}$ be a node labelling function that is computed during the execution of $\text{main}(\mathcal{G})$. Let $\text{sig-visited-successors} : \text{sig}(\mathcal{O}) \rightarrow 2^{\text{sig}(\mathcal{O})}$ be a node labelling function that is computed during the execution of $\text{main}(\mathcal{G})$. Then the following holds:*

- (i) *for all $x \in \mathcal{V}$: $\text{visit}(x)$ is called exactly once;*
- (ii) *$\mathcal{V} = \mathcal{V}'$;*
- (iii) *for all $v, y \in \mathcal{V}$: $\text{visited-successors}(y) \subseteq \text{visited-successors}(x)$ iff $\text{sig-visited-successors}(y) \subseteq \text{sig-visited-successors}(x)$;*
- (iv) *for all $x, y \in \mathcal{V}$: $\{x\} \geq_{\Sigma} y$ iff $\text{visited-successors}(y) \subseteq \text{visited-successors}(x)$;*
- (v) *for all $x, y \in \mathcal{V}$: $\{x\} \geq_{\Sigma} y$ iff $y \subseteq \text{visited-successors}(x)$;*
- (vi) *for all $x, y \in \mathcal{V}'$: $\{x\} \geq_{\mathcal{G}'} y$ iff $y \subseteq \text{visited-successors}(x)$.* $\dashv$

Proof. We start the proof of Lemma 6.39 by showing item (i). Every call of $\text{visit}(\cdot)$ (in Line 12 of $\text{main}(\cdot)$ and in Line 15 of $\text{visit}(\cdot)$) is guarded by an if-statement (in Line 11 of $\text{main}(\cdot)$ and in Line 14 of $\text{visit}(\cdot)$). This ensures that the call $\text{visit}(x)$, for $x \in \mathcal{V}$, is executed only if x has not been discovered before. In Line 2 of $\text{visit}(x)$, $\text{discovered-order}(x)$ is set to a value different from -1 . This ensures that $\text{visit}(v)$ will not be called again. The for-loop from Line 10 to Line 14 in $\text{main}(\cdot)$ ensures that $\text{visit}(v)$ is called for all nodes $v \in \mathcal{V}_{\mathcal{H}}$. Then, item (i) holds.

Consider item (ii). By item (i), the function $\text{visit}(\cdot)$ is called for each node of $\mathcal{G}$ once. In Line 7, $\text{visit}(\cdot)$ adds the input node into $\mathcal{V}'$. Hence, item (ii) holds.

Item (iii) can readily be seen. Let $x, y \in \mathcal{V}$. When $\text{visit}(x)$ is called, $\text{visited-successors}(x) = \{x\}$ and $\text{sig-visited-successors}(x) = \text{sig}(x)$ (Lines 4 & 5). By Lines 22 & 23 and 35 & 36,

if $\text{visited-successors}(y)$ is added into $\text{visited-successors}(x)$, then $\text{sig-visited-successors}(y)$ is added to $\text{sig-visited-successors}(x)$ and vice versa.

Consider item (iv). “ $\Rightarrow$ ”: Suppose $\{x\} \geq_{\Sigma} y$, where $x, y \in \mathcal{V}$. This implies that $\geq_{\Sigma}(\{y\}) \subseteq \geq_{\Sigma}(\{x\})$. By the definition of Σ -connectivity (cf. Definition 6.4), $\bigcup_{y' \in \geq_{\Sigma}(\{y\})} \text{sig}(y') \subseteq \bigcup_{x' \in \geq_{\Sigma}(\{x\})} \text{sig}(x')$. We need to show that $\text{sig-visited-successors}(y) \subseteq \text{sig-visited-successors}(x)$. This can be done by induction along the lines of the proof of Lemma 3.15 (item (iii)). Here, it is possible to show by induction on the length n of a sequence $v_1 \dots v_n$ such that for every $i \in \{1, \dots, n-1\}$, $\Sigma_{v_i} \subseteq \bigcup_{1 \leq j < i} \text{sig}(v_j)$ for some $\Sigma_{v_i} \in \text{MLS}(v_i)$, that $\text{sig-visited-successors}(v_n) \subseteq \text{sig-visited-successors}(v_1)$. We conclude the proof by showing that $\text{visited-successors}(y) \subseteq \text{visited-successors}(x)$. This follows from item (iii).

The other direction can be shown similarly.

Consider item (v). This immediately follows from item (iv).

Consider item (vi). This follows from item (iv) and the fact that when a node y is added into $\text{visited-successors}(x)$, an edge e is added into $\mathcal{E}'$ (Line 24) such that $e = (x, y)$ or $e = (x, s)$ and $y \in \text{visited-successors}(s)$. $\square$

The next proposition follows from Lemma 6.39.

Proposition 6.40 (Correctness of DFS for the labelled graph-encoding of an ADH). *Let $\mathcal{H}_{\mathcal{O}}^x = (\mathcal{V}_{\mathcal{O}}^x, \mathcal{E}_{\mathcal{O}}^x)$ be the x -ADH of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of $\mathcal{H}_{\mathcal{O}}^x$. Let $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ be the functions depicted in Figures 6.7 & 6.9, respectively. Let $\mathcal{G}' = \text{main}(\mathcal{G})$ be a DFS-graph, where $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{L}')$. Then:*

(i) *The function $\text{main}(\cdot)$ for DFS terminates.*

(ii) *$\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ is an approximating graph of $\mathcal{H}$.* $\dashv$

Proof. Consider Item (i). By item(i) of Lemma 6.39, $\text{visit}(v)$ is called exactly once, for all $v \in \mathcal{V}$. Recall that the number of nodes and edges of $\mathcal{G}$ is finite. Similarly, the signature of each axiom and its MLSs are also finite. Therefore, the auxiliary functions can only find a finite number of new successors and the while-loop in Lines 12–28 terminates.

Consider item (ii). Let $\geq_{\Sigma}$ be the Σ -connectivity relation in $\mathcal{G}$ wrt. Σ , where $\Sigma = \emptyset$. Let $\geq_{\mathcal{G}'}$ be the connectivity relation in $\mathcal{G}'$. By Definitions 3.1 & 6.4, we have to show that for all $v \in \mathcal{V}$,

$$\{v' \in \mathcal{V} \mid \{v\} \geq_{\Sigma} v'\} = \{v' \in \mathcal{V}' \mid \{v\} \geq_{\mathcal{G}'} v'\}.$$

This follows by Item (v) and Item (vi) of Lemma 6.39. $\square$

We show that the time complexity of DFS for directed labelled graphs is polynomial with respect to the number of nodes of the labelled graph and the number of MLSs associated to each node. Recall that the number of MLSs of an axiom might be exponential with respect to the signature of the axiom (cf. Example 4.8).

Proposition 6.41 (Time complexity of DFS for directed labelled graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of a x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $n = \max \{|MLS^x(v)| \mid \text{for every } v \in \mathcal{V}\}$. Let $\text{main}(\cdot)$ and $\text{visit}(\cdot)$ be the functions depicted in Figures 6.7 & 6.9, respectively.*

Then: the time complexity of function $\text{main}(\cdot)$ is $O(|\mathcal{V}|^3 \times n)$. ⊢

Proof. By item(i) of Lemma 6.39, $\text{visit}(v)$ is called exactly once, for all $v \in \mathcal{V}$. This represents $|\mathcal{V}|$ calls including the calls from the function $\text{main}(\cdot)$ and the recursive calls from the function $\text{visit}(\cdot)$ itself.

Inside the function $\text{visit}(\cdot)$, there is only one while-loop (Lines 12–28) that might iterate $|\mathcal{V}|$ times in the worst-case. In each iteration, the auxiliary functions depicted in Figure 6.8 are called. The jointly time complexity of these functions is $O(|\mathcal{V}| \times n)$ (cf. Proposition 6.36).

Hence, the time complexity of the function $\text{main}(\cdot)$ is $O(|\mathcal{V}|^3 \times n)$. □

We conclude that the time complexity of BFS and DFS for directed labelled graphs is the same.

6.4 Locality-based Modules

Proposition 6.7 shows that B -connectivity in the Σ -extension of an ADH equals Σ -connectivity in the labelled graph representation of the ADH. This means that we do not have to modify the labelled graph representation of an ADH for an ontology $\mathcal{O}$ to compute the Σ -connected component for any possible signature Σ . Recall that this is equivalent to compute the syntactic local module for Σ and the notions of $\perp$ - and $\top$ -locality (cf. Proposition 5.9).

Figure 6.11 shows a simple algorithm to compute syntactic local modules using the labelled graph-encoding of an ADH. The algorithm uses four auxiliary functions. Three of them are similar to the auxiliary functions of BFS for directed labelled graphs (cf. Figure 6.4). However, the functions $\text{compute-successors-DG}(\cdot, \cdot, \cdot)$ and $\text{compute-successors-DLG}(\cdot, \cdot, \cdot)$ have some minor differences. For clarity, we include the new versions of these two functions in Figure 6.11.

Recall that the auxiliary functions are used in BFS to identify new connected nodes in a directed labelled graph representation given a particular signature. $\text{compute-successors-DG}(\cdot, \cdot, \cdot)$ takes advantage of the fact that all simple directed hyperedges are explicitly represented in the

directed labelled graph. An adjacency list data-structure can provide efficient access to direct successors of a given node. Therefore, the algorithm first searches for successors connected via simple directed hyperedges from the nodes that have been already identified as part of the connected component that it is being computed. Instead of two input parameters, the new version of `compute-successors-DG`($\mathcal{G}, S_1, S_2$) has three: a labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ and two sets of nodes $S_1, S_2 \subseteq \mathcal{V}$. This function computes the successors of $S_1 \setminus S_2$ that are connected by simple directed hyperedges.

`compute-successors-DLG`($\cdot, \cdot, \cdot$) explores complex directed hyperedges that are encoded in the labels of each node of the directed labelled graph. To achieve this, the function searches for nodes that are not part of the current version of the connected component and they have at least one minimal non-locality signature that is in Σ (which has been extended with the signatures of the nodes already included in the connect component). When no new nodes can be added into the connected component, the algorithm identifies which are the axioms that correspond with the nodes in the connected component. These axioms represent the locality-based module for Σ and the ontology from which the directed labelled graph was built. Note that in these two auxiliary functions, it is not required to check if a node has been visited before. This was needed in BFS for directed labelled graphs.

The function $\text{Mod}^x(\Sigma, \mathcal{G})$ works as follows. In Line 2, it determines the set S_1 of initial nodes in $\mathcal{G}$. Every initial node S_1 is associated with a minimal non-locality signature that is contained in Σ . In Line 5, it is identified the set of nodes that are connected via simple directed hyperedges that are explicitly given in $\mathcal{E}$. Note that $\mathcal{E}(v)$ denotes the set of nodes that are directly connected in $\mathcal{G}$ from the node v using simple directed hyperedges. In Line 7, the input signature is extended with the symbols that are associated to the nodes discovered so far. Using the extended signature Σ_m , the function `compute-successors-DLG`($\cdot, \cdot, \cdot$) computes the nodes that are connected via complex directed hyperedges implicitly represented by the labels $\mathcal{L}(v)$ of the nodes v in S_m . The do-until loop (Lines 3–9) iterates until a fixed point is reached and no more new nodes are found.

The following proposition shows that the functions $\text{Mod}^x(\Sigma, \mathcal{G})$ and $\text{Mod}^x(\Sigma, \mathcal{O})$ depicted in Figures 6.11 & 4.1, respectively, compute the same syntactic x -local module for a signature Σ and an ontology $\mathcal{O}$.

Proposition 6.42. *Let $\mathcal{O}$ be an ontology, Σ a signature and $x \in \{\perp, \top\}$. Let $\mathcal{H}_{\mathcal{O}, \Sigma}^x$ be the Σ -extension of the x -ADH, x -pcADH or x -cADH for $\mathcal{O}$. Let $\mathcal{G}$ be the labelled graph representation of $\mathcal{H}_{\mathcal{O}, \Sigma}^x$. Let $\text{Mod}^x(\Sigma, \mathcal{O})$ be the function depicted in Figure 4.1 and let $\text{Mod}^x(\Sigma, \mathcal{G})$ be the function depicted in Figure 6.11. Then: $\text{Mod}^x(\Sigma, \mathcal{G}) = \text{Mod}^x(\Sigma, \mathcal{O})$. $\dashv$*

Proof. It can readily be seen that the function $\text{Mod}^x(\Sigma, \mathcal{G})$ computes the Σ -connected component for S_1 (Line 2) as defined in Definition 6.4.

```

function Modx( $\Sigma, \mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ )
1:  $\Sigma_0 := \Sigma, m := 1$ 
2:  $S_0 := \emptyset, S_1 := \text{compute-successors-DLG}(\mathcal{G}, \Sigma_0, S_0)$ 
3: do
4:    $m := m + 1$ 
5:    $S' := \text{compute-successors-DG}(\mathcal{G}, S_{m-1}, S_{m-2})$ 
6:    $\Sigma_m := \Sigma_{m-1} \cup \text{get-signatures}(S')$ 
7:    $S_m := S' \cup S_{m-1}$ 
8:    $S_m := S_m \cup \text{compute-successors-DLG}(\mathcal{G}, \Sigma_m, S_m)$ 
9: until  $S_m = S_{m-1}$ 
10: return get_axioms( $S_m$ )

function compute-successors-DG( $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}), S_1, S_2$ )
1:  $S' := \{\}$ 
2: for every  $v \in S_1$  do
3:   if ( $v \notin S_2$ ) then
4:     for every  $s \in \mathcal{E}(v)$  do
5:       if ( $s \notin S_1$ ) then
6:          $S' := S' \cup \{s\}$ 
7:       end-if
8:     end-for
9:   end-if
10: end-for
11: return  $S'$ 

function compute-successors-DLG( $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}), \Sigma, S$ )
1:  $S' := \{\}$ 
2: for every  $v \in \mathcal{V}$  do
3:   if ( $v \notin S$ ) then
4:     for every  $\Sigma' \in \text{MLS}(v)$  do
5:       if ( $\Sigma' \subseteq \Sigma$ ) then
6:          $S' := S' \cup \{v\}$ 
7:       end-if
8:     end-for
9:   end-if
10: end-for
11: return  $S'$ 

```

Fig. 6.11 Module extraction algorithm for the labelled graph-encoding of an ADH

Let $\geq_B$ be the B -connectivity relation in $\mathcal{H}_{\mathcal{O}, \Sigma}^x$ and let $\geq_\Sigma$ be the Σ -connectivity relation in $\mathcal{G}$ wrt. Σ . Proposition 6.7 shows B -connectivity in the Σ -extension of an ADH equals Σ -connectivity in the labelled graph representation of the ADH. This is $\geq_B = \geq_\Sigma$.

By Proposition 5.9, $\text{Mod}^x(\Sigma, \mathcal{O}) = \geq_B(\mathcal{O}')$, where $\mathcal{O}'$ is a set of axioms from $\mathcal{O}$ that are not x -local wrt. Σ . $\square$

Next, we show that the time complexity of $\text{Mod}^x(\cdot, \cdot)$ for directed labelled graphs is polynomial with respect to the number of nodes of the labelled graph and the number of MLSs associated to each node. Recall that the number of MLSs of an axiom might be exponential with respect to the signature of the axiom (cf. Example 4.8).

Proposition 6.43 (Time complexity of the module extraction algorithm). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of the x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $n = \max \{ |MLS^x(v)| \mid \text{for every } v \in \mathcal{V} \}$. Let $\text{Mod}^x(\cdot, \cdot)$ be the function depicted in Figure 6.11. Then: the time complexity of function $\text{Mod}^x(\cdot, \cdot)$ is $O(|\mathcal{V}|^2 \times n)$. $\dashv$*

Proof. Inside the function $\text{Mod}^x(\cdot, \cdot)$, there is only one loop (Lines 12–28) that might iterate $|\mathcal{V}|$ times in the worst-case. In each iteration, the auxiliary functions depicted in Figure 6.11 are called. The jointly time complexity of these functions is $O(|\mathcal{V}| \times n)$ (cf. Proposition 6.36).

Hence, the time complexity of the function $\text{Mod}^x(\cdot, \cdot)$ is $O(|\mathcal{V}|^2 \times n)$. $\square$

6.5 Atomic Decomposition

In Section 5.3, we have established that all axioms of an atom in an ontology correspond to the sets of nodes of the ADH that are strongly- B -connected (cf. Proposition 5.15). We have also shown a correspondence between the notions of dependency between atoms and B -connectivity from individual nodes in the cADH (cf. Proposition 6.49). In the same section, we concluded that the atomic decomposition of an ontology correspond to the approximating graph of the cADH (cf. Corollary 5.23).

In this section, we discuss two algorithms to compute the strongly- B -connected components of an ADH represented by a labelled graph. One of these algorithms is based on the Naïve algorithm depicted in Figure 2.7. This algorithm computes the strongly-connected components of a directed graph. The second algorithm that we discuss in this section is the DFS algorithm depicted in Figures 6.7 & 6.9. This algorithm not only traverse a labelled graph or computes the graph approximation of the ADH encoded in the labelled graph. It is also able to identify the strongly- Σ -connected components of the labelled graph-encoding of an ADH. It is based on the DFS algorithm for directed hypergraphs depicted in Figures 3.1 & 3.2.

```

function main( $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ )
1:  fix total order  $<$  on  $\mathcal{V}$ 
2:   $T := \emptyset$ 
3:  for every  $v \in \mathcal{V}$  do
4:    for every  $v \in \mathcal{V}$  do
5:      visited( $v$ ) := false; discovered-order :=  $-1$ 
6:    end-for
7:     $S_v := \text{visit}(v)$  // compute  $\geq_\Sigma(v)$ 
8:     $\pi_v := \text{sort}_{<}(S_v)$ 
      // store  $\pi_v$  in  $T$ 
9:    for every  $\pi' \in \text{prefix}(\pi_v)$  do
10:     if  $\pi' \notin \text{dom}(T)$  then  $T := T \cup \{(\pi', \emptyset)\}$ 
11:    end-for
12:     $T(\pi_v) := T(\pi_v) \cup \{v\}$ 
13:  end-for
      // return SCCs( $\mathcal{G}$ )
14: return  $\{T(\pi) \mid \pi \in \text{dom}(T) \text{ and } T(\pi) \neq \emptyset\}$ 

```

Fig. 6.12 Naïve algorithm to compute the SCCs of a directed labelled graph.

We finally conclude this section by showing how to produce the cADH using the labelled graph encoding of an ADH.

6.5.1 Naïve Algorithm

The algorithm depicted in Figure 6.12 computes the strongly-connected components of a directed labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$. In short, the algorithm computes the connected component of each node of the graph and identifies which of these connected components are the same. The nodes with the same connected component are in the same strongly-connected component (Proposition 2.3).

The algorithm has been defined using two functions: $\text{main}(\cdot)$ and $\text{visit}(\cdot)$, where the former is identical to the function $\text{main}(\cdot)$ depicted in Figure 2.7. The latter is nearly equivalent to the function $\text{visit}(\cdot)$ of the BFS algorithm or the DFS algorithm for directed labelled graphs (cf. Figures 6.5 & 6.9, respectively). It is only required that $\text{visit}(\cdot)$ returns the nodes that has been discovered and collected. Notice that the function $\text{visit}(\cdot)$ of BFS for directed labelled graphs can compute the connected component of a node if all nodes of the graph are initialized as unvisited: $\text{visited}(v) = \text{false}$, for all $v \in \mathcal{V}$ (Lines 4–6 in function $\text{main}(\cdot)$) of the Naïve algorithm. If we used the function $\text{visit}(\cdot)$ of DFS for directed labelled

graphs, we have to initialize all nodes as undiscovered: $\text{discovered-order}(v) = -1$, for all $v \in \mathcal{V}$. This is also done in Lines 4–6 in function $\text{main}(\cdot)$ of the Naïve algorithm.

The function $\text{main}(\cdot)$ starts by fixing a total order $<$ on the set of nodes in $\mathcal{G}$ (Line 1). The order is used to represent a set S_v of all nodes connected from a node v as a sequence π_v of nodes (Line 8). The sequence π_v for every node v in $\mathcal{G}$ is then introduced as a branch in a labelled tree T (Lines 9–13). Note that each prefix of π_v is considered as a node in T . If the node is not already contained in T , then the node is labelled with the empty set (Line 10). Then the node v is added to the final node of the branch π_v (Line 12). Two different sequences of ordered nodes are introduced as different branches in T . A sequence of ordered nodes that is contained in another sequence of nodes already included in the tree might be part of the same branch of the tree (this depends of the total order $<$ considered). The function $\text{main}(\cdot)$ returns the labels of the tree T that are not empty (Line 14). Each non-empty label corresponds to a strongly-connected component of $\mathcal{G}$.

Proposition 6.44 (Correctness of the naïve algorithm for directed labelled graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of the x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\text{main}(\cdot)$ be the function depicted in Figure 2.7. Let $\text{visit}(\cdot)$ be the function depicted in Figures 6.5 & 6.9. Then:*

- (i) *for all $v \in \mathcal{V}$: $\text{visit}(v) = \text{CC}(v)$,*
- (ii) *$\text{main}(\mathcal{G})$ terminates,*
- (iii) *$\text{main}(\mathcal{G}) = \text{SCCs}(\mathcal{G})$.* ⊢

Proof. Item (i) of Proposition 6.44 follows from Propositions 6.35 & 6.40. Moreover, items (ii) & (iii) of Proposition 6.44 follows from Proposition 2.23. □

We show that the time complexity of the naïve algorithm for computing strongly-connected components in directed labelled graphs is polynomial with respect to the number of nodes of the labelled graph and the number of MLSs associated to each node. Recall that the number of MLSs of an axiom might be exponential with respect to the signature of the axiom (cf. Example 4.8).

Proposition 6.45 (Time complexity of the naïve algorithm for directed labelled graphs). *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of the x -ADH for an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $n = \max \{ |\text{MLS}^x(v)| \mid \text{for every } v \in \mathcal{V} \}$. Let $\text{main}(\cdot)$ be the function depicted in Figure 6.12. Let $\text{visit}(\cdot)$ be the function depicted in Figures 6.5 & 6.9.*

Then: the time complexity of function $\text{Mod}^x(\cdot, \cdot)$ is $O(|\mathcal{V}|^2 \times n)$. ⊢

Proof. The complexity of the function $\text{visit}(\cdot)$ for BFS or DFS is $O(|\mathcal{V}|^2 \times n)$ (cf. Propositions 6.36 & 6.41, respectively). According to Proposition 2.24, the time complexity of the function $\text{main}(\cdot)$ without considering the time needed by $\text{visit}(\cdot)$ is quadratic with respect to the number of nodes of the input labelled graph ($O(|\mathcal{V}|^2)$).

If we consider the time needed by the function $\text{visit}(\cdot)$, then the complete execution of the function $\text{main}(\cdot)$ requires $|\mathcal{V}| \times (|\mathcal{V}| + (|\mathcal{V}|^2 \times n))$ steps. This implies that the time complexity of the function $\text{main}(\cdot)$ is $O(|\mathcal{V}|^3 \times n)$. $\square$

6.5.2 Tarjan's Algorithm

By collecting the strongly- Σ -connected components computed by DFS for directed labelled graphs, we obtain an algorithm that is similar to Tarjan for directed graphs.

To show that this algorithm is correct, we have to prove that the strongly connected components stored in list-of-sccs are correct.

Theorem 6.46 (Correctness of Tarjan for the labelled graph-encoding of an ADH). *The extended version of DFS depicted in Figures 6.7 & 6.9 for computing strongly-connected components works correctly on any directed labelled graph.* $\dashv$

Proof. Like the proof of correctness of Tarjan for directed graphs [39] and B -hypergraphs (Theorem 3.25), we show that a node and the node it refers to by its $\text{min-order}(\cdot)$ value are in the same SCC. We show this by induction of the number of terminated calls of the function $\text{visit}(\cdot)$. $\square$

The proofs for termination and time complexity of DFS for directed labelled graphs are also valid for this version of Tarjan. Note that the time complexity of the Naïve algorithm and Tarjan are the same in directed labelled graphs.

6.5.3 Computing the Condensed Axiom Dependency Hypergraph

To compute the atomic decomposition of an ontology, it is not enough to compute the atoms but also the dependencies between these atoms. Recall that there is a correspondence between the notions of dependency between atoms and B -connectivity from individual nodes of a condensed axiom dependency hypergraph (cf. Proposition 6.49). In addition, B -connectivity in the ADH corresponds to Σ -connectivity in its labelled graph representation (cf. Proposition 6.5). By Proposition 6.9, we also know that it is equivalent to condensed a ADH and compute its labelled graph representation or calculate the cADH using the labelled graph representation directly.

function collapse-SCCs($\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$, $\text{SCCs} = \{s_i \subseteq \mathcal{V} \mid i \in I\}$)

- 1: $\mathcal{V}' := \{v_i \mid i \in I\}$
- 2: $\mathcal{E}' := \{(v_i, v_j) \in \mathcal{V}' \times \mathcal{V}' \mid x \in s_i, y \in s_j \text{ for some } s_i, s_j \in \text{SCCs}, (x, y) \in \mathcal{E}, \text{ and } v_i \neq v_j\}$
- 3: **for every** $v_i \in \mathcal{V}'$ **do**
- 4: $\mathcal{L}'(v_i) := (\bigcup_{v \in s_i} \text{MLS}^x(v), \bigcup_{v \in s_i} \text{sig}(v))$
- 5: **end-done**
- 6: **return** $\mathcal{G}' = (\mathcal{V}', \mathcal{E}', \mathcal{L}')$

Fig. 6.13 Function for collapsing the SCCs of an ADH encoded by a labelled graph.

In this section, we introduce two functions that work together to condensed an ADH using its labelled graph representation. In order to obtain the dependencies between atoms, we compute the graph representation of the condensed hypergraph by collapsing the strongly-connected components into single nodes. This is done by the function collapse-SCCs(.).

The function collapse-SCCs(.) depicted in Figure 6.13 takes as an input a labelled graph representation $\mathcal{G}$ of an ADH $\mathcal{H}$ and a partition SCCs of the nodes in $\mathcal{H}$ consisting of sets of nodes that are strongly- B -connected in $\mathcal{H}$. For every element in SCCs, a new node is introduced (Line 1). Then, directed edges are computed which connect the newly introduced nodes (Line 2). Two strongly- B -connected components s_1, s_2 are connected by an edge if there is an edge e in the input graph $\mathcal{G}$ that connects two nodes x, y that are contained in s_1 and s_2 , respectively. Note that no reflexive edges are introduced. Finally, the node labels are updated to represent the hyperedges between the newly introduced nodes (Line 3).

For a strongly-connected component s_i represented by a new node v_i , the set $\text{MLS}^x(v_i)$ of minimal non- x -locality signatures of v_i is the union of all minimal non- x -locality signatures $\text{MLS}^x(v)$ of each node $v \in s_i$. Similarly, the signature $\text{sig}(v_i)$ of v_i is set to the union of signatures $\text{sig}(v)$ of each node $v \in s_i$.

Proposition 6.47. *Let $\mathcal{H}_\mathcal{O}^x = (\mathcal{V}_\mathcal{O}^x, \mathcal{E}_\mathcal{O}^x)$ be the x -ADH (or x -pcADH) of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be the labelled graph-encoding of $\mathcal{H}_\mathcal{O}^x$. Let $\mathcal{G}'' = (\mathcal{V}'', \mathcal{E}'')$ be the graph fragment of $\mathcal{H}_\mathcal{O}^x$, a directed graph where $\mathcal{V}'' = \mathcal{V}_\mathcal{O}^x$ and $\mathcal{E}'' = \{e \in \mathcal{E}_\mathcal{O}^x \mid |T(e)| = 1\}$. Let $\simeq$ be the strong- B -connectivity relation in $\mathcal{H}_\mathcal{O}^x$ or the strong-connectivity relation in $\mathcal{G}''$. Let collapse-SCCs($\cdot, \cdot$) be the function depicted Figure 6.13.*

Then: $\mathcal{G}/\simeq = \text{collapse-SCCs}(\mathcal{G}, [\mathcal{V}_\mathcal{O}^x]_\simeq)$, where $[\mathcal{V}_\mathcal{O}^x]_\simeq = \{[v]_\simeq \mid v \in \mathcal{V}_\mathcal{O}^x\}$. ⊢

Proof. The algorithm for collapse-SCCs(.) diverts from Def. 6.8 as identifying all possible simple hyperedges is a quadratic process. To optimize running time, the set $\mathcal{E}'$ of edges in $\mathcal{G}'$ is computed according to $\mathcal{E}' := \{([v]_\simeq, [v']_\simeq) \mid (v, v') \in \mathcal{E} \text{ and } v \not\simeq v'\}$. Note that $\mathcal{E}' \subseteq \mathcal{E}$ and $\text{collapse-SCCs}(\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}), [\mathcal{V}_\mathcal{O}^x]_\simeq) = \mathcal{G}'$. □

```

function compute-condensed-hypergraph( $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ , algorithm)
1: if (algorithm = naïve-dlg) then
2:    $\mathcal{G}' := \text{collapse-SCCs}(\mathcal{G}, \text{tarjan-dg}((\mathcal{V}, \mathcal{E})))$ 
3:   if (contains-complex-dependencies( $\mathcal{G}'$ ) = false) then
4:     return  $\mathcal{G}'$ 
5:   end-if
6:    $\mathcal{G}' := \text{collapse-SCCs}(\mathcal{G}', \text{naïve-dlg}(\mathcal{G}'))$ 
7: else
8:    $\mathcal{G}' := \text{collapse-SCCs}(\mathcal{G}, \text{tarjan-dlg}(\mathcal{G}))$ 
9: end-if
10: return  $\mathcal{G}'$ 

```

Fig. 6.14 Function for condensing an ADH encoded by a labelled graph.

We use the function $\text{compute-condensed-hypergraph}(\cdot, \cdot)$ (cf. Figure 6.14) to condense an ADH represented by a directed labelled graph. This function orchestrates the different algorithms that can be used to identify the strongly-connected components of the graph fragment of an ADH or the strongly- Σ -connected components of the labelled graph representation of the ADH. Recall that we discuss in the thesis two algorithms for computing the strongly-connected components of a directed graph and two more algorithms for computing the strongly- Σ -connected components of the labelled graph encoding of an ADH. For the design of the function $\text{compute-condensed-hypergraph}(\cdot, \cdot)$, we have only considered three of the four algorithms: $\text{tarjan-dg}(\cdot)$, $\text{naïve-dlg}(\cdot)$ and $\text{tarjan-dlg}(\cdot)$. The function $\text{tarjan-dg}(\cdot)$ corresponds to the function $\text{main}(\cdot)$ of the original algorithm by Tarjan [39] for directed graphs depicted in Figure 2.8; $\text{naïve-dlg}(\cdot)$ corresponds to the function $\text{main}(\cdot)$ of the naïve algorithm for directed labelled graphs depicted in Figure 6.12; and $\text{tarjan-dlg}(\cdot)$ corresponds to the function $\text{main}(\cdot)$ of the algorithm for directed labelled graphs depicted in Figure 6.7.

Depending on the selected algorithm for computing the strongly- Σ -connected components of the labelled graph representation of an ADH, the function $\text{compute-condensed-hypergraph}(\cdot, \cdot)$ completes one or two main steps. If $\text{naïve-dlg}(\cdot)$ is the function requested, then $\text{compute-condensed-hypergraph}(\cdot, \cdot)$ completes two main steps. In the first step, the function computes the partially condensed ADH (Line 2). Only the simple hyperedges of the input labelled graph $\mathcal{G}$ which are stored in $\mathcal{E}$, are considered. The strongly connected components are determined in linear time using $\text{tarjan-dg}(\cdot)$. The reduction of nodes after the partially condensed ADH is computed can decrease the time needed to identify the strongly- Σ -connected components using the function $\text{naïve-dlg}(\cdot)$ (Line 6). After the strongly- Σ -connected components are identified, the function collapse-SCCs produces the

labelled graph representation of the condensed ADH (Line 6). This completes the second step. If $\text{tarjan-dlg}(\cdot)$ is the function selected, then the strongly- Σ -connected components are identified and later collapsed in Line 8. Note that the function $\text{tarjan-dlg}(\cdot)$ would not improve its performances significantly if the partially condensed ADH is computed before. This occurs because $\text{tarjan-dlg}(\cdot)$ explores the successors of the node that is currently visited, connected by simple hyperedges. If all the nodes of the labelled graph representation of an ADH are strong- Σ -connected by simple hyperedges then $\text{tarjan-dlg}(\cdot)$ becomes similar to $\text{tarjan-dg}(\cdot)$.

Proposition 6.48. *Let $\mathcal{H}_{\mathcal{O}}^x = (\mathcal{V}_{\mathcal{O}}^x, \mathcal{E}_{\mathcal{O}}^x)$ be the x -ADH of an ontology $\mathcal{O}$, where $x \in \{\perp, \top\}$. Let $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ be the graph fragment of $\mathcal{H}_{\mathcal{O}}^x$, a directed graph where $\mathcal{V}' = \mathcal{V}_{\mathcal{O}}^x$ and $\mathcal{E}' = \{e \in \mathcal{E}_{\mathcal{O}}^x \mid |T(e)| = 1\}$. Let $\mathcal{G}$ be the labelled graph representation of $\mathcal{H}_{\mathcal{O}}^x$. Let $\simeq$ be the strong-B-connectivity relation in $\mathcal{H}_{\mathcal{O}}^x$, the strong-connectivity relation in $\mathcal{G}'$ or the strong- Σ -connectivity relation in $\mathcal{G}$. Let $\text{compute-condensed-hypergraph}(\mathcal{G}, \text{alg})$ be the function depicted in Figure 6.14, where $\text{alg} \in \{\text{naïve-dlg}, \text{tarjan-dlg}\}$.*

Then: $\mathcal{G}/\simeq = \text{compute-condensed-hypergraph}(\mathcal{G}, \text{alg})$. $\dashv$

Proof. The proposition can readily be seen. Note that the algorithm computing $\text{collapse-SCCs}(\cdot, \cdot)$ yields the condensed graph representation of the ADH by Prop. 6.47, which in turn is equivalent to the graph representation of the condensed ADH by Prop. 6.9. $\square$

Corollary 6.49. *Let $\mathcal{O}$ be an ontology with $\alpha, \beta \in \mathcal{O}$. Let $\mathfrak{a}, \mathfrak{b} \in \text{Atoms}_{\mathcal{O}}^{\perp}$ such that $\alpha \in \mathfrak{a}$ and $\beta \in \mathfrak{b}$. Let $\mathcal{G}$ be the graph representation of the ADH of $\mathcal{O}$. Let $v, v' \in \mathcal{V}$ such that $\text{get_axioms}(v) = \mathfrak{a}$ and $\text{get_axioms}(v') = \mathfrak{b}$. Let $\text{compute-condensed-hypergraph}(\mathcal{G}, \text{alg})$ be the function depicted in Figure 6.14, where $\text{alg} \in \{\text{naïve-dlg}, \text{tarjan-dlg}\}$. Let $\mathcal{G}/\simeq = \text{compute-condensed-hypergraph}(\mathcal{G}, \text{alg})$, where $\mathcal{G}/\simeq = (\mathcal{V}_{\simeq}, \mathcal{E}_{\simeq}, \mathcal{L}_{\simeq})$.*

Then: $\mathfrak{a} \succ_{\mathcal{O}}^{\perp} \mathfrak{b}$ iff $(v, v') \in \mathcal{E}_{\simeq}^$. $\dashv$*

Note that $\mathcal{E}_{\simeq}^*$ is the transitive closure of the $\mathcal{E}_{\simeq}$.

Proof. The corollary follows from Prop. 6.48 and Prop. 6.49. $\square$

6.6 Implementation

HyS is a Java application designed to accelerate the computation of syntactic locality-based modules and the atomic decomposition of OWL ontologies. The current version of *HyS*, version 1.1, supports OWL-EL¹ ontologies and the notion of $\perp$ -locality. *HyS* explicitly

¹OWL-EL profile extended with inverse and functional role axioms

represents $\perp$ -ADHs as directed labelled graphs. The application also includes the algorithms also includes the algorithm by Tarjan [39] for directed graphs (cf. Figure 2.8) and two algorithms for computing strongly- Σ -connected components of the labelled graph-encoding of an ADH: the naïve algorithm (cf. Figure 6.12) and the algorithm inspired by Tarjan’s algorithm (cf. Figure 6.7).

We plan to extend the implementation to support both, OWL 2 ontologies and $\top$ -locality, in the future.

6.6.1 Architecture

The architecture of *HyS* consists of three main components: *Ontology Manager*, *Graph Builder* and *Connectivity Library*. *HyS* also includes two interfaces, an API and a command-line interface, to get access to its functionality. The interfaces are discussed in Section 6.6.2.

Extensibility was one of the main requirements that we have in mind when we designed and implemented *HyS*. In particular, we are interested in incorporating new algorithms that improve the performance of *HyS* for computing the strongly- Σ -connected components of the labelled graph representation of an ADH. This is why the algorithms are organized in a library that can easily be extended with new algorithms. Next, we provide a description of each of the main components.

Ontology Manager

The *Ontology Manager* has been implemented on top of the OWLAPI and it provides the required functionality to load OWL-EL ontologies and store the atomic decomposition of an ontology and any syntactic local module computed for a given signature. The atomic decomposition of an ontology is stored as a set of OWL ontologies (one for each atom) that also includes one text file with the existent dependencies between pair of atoms. Similarly, modules are stored as OWL ontologies.

The Ontology Manager includes a method to compute the minimal non-locality signatures and the signature of each axiom of the ontology. We originally planned to extend the functionality of the Ontology Manager with new methods for normalisation and denormalisation of OWL 2 axioms based on the normalisation rules defined in [34].

Recall that example 4.8 in Section 4.1.2 shows that it is possible to create an axiom with exponentially many minimal non-locality signatures using merely conjunction and disjunction as logical operators. According to Proposition 4.9, an axiom in normal form does not have more than two minimal non-locality signatures. In Section 6.2, we study if normalisation might affect B -connectivity in an ADH and we conclude that it is the case. This

implies that the syntactic local modules and the atomic decomposition of an ontology before or after normalisation might be different. Therefore, we might need new normalisation rules or a compact representation of exponentially many minimal-non-locality signatures. We need also to study how often real ontologies have axioms with exponentially many hyperedges. The result of this analysis will determine how the Ontology Manager should be extended to deal with OWL 2 axioms.

We plan to extend the implementation of the Ontology Manager to support OWL 2 ontologies in the future.

Graph Builder

The Graph Builder is the core component in the *HyS* architecture as it interacts with all other components. The essential task is to build the directed labelled graph and all related data-structures that have been defined to improve the performances of the algorithms for computing connected and strongly-connected components. The Graph Builder also provides methods for collapsing a set of nodes into a single node which is required to compute the (partially) condensed version of an axiom dependency hypergraph. These methods are similar to the functions `collapse-SCCs(.)` and `compute-condensed-hypergraph(.,.)` depicted in Figures 6.13 & 6.14, respectively.

The nodes and symbols of a directed labelled graph are identified internally using integers. The simple directed hyperedges are represented in an adjacency list data structure. We have implemented specific data-structures that can facilitate the execution of subset and superset queries over sets of symbols which are frequent operations during the computation of connected and strongly-connected components. Binary arrays have been proved to be well-suited to building these data-structures.

The manipulation of the data structures defined by the Graph Builder can only be done using specific methods. The idea is to provide an internal interface between the Graph Builder and the algorithms defined in the Connectivity Library. Using this approach, the algorithms for computing (strongly-) connected components will not be affected by changes in the implementation of the Graph Builder and it will facilitate the extension of the Connectivity Library with new algorithms.

The Graph Builder includes support for heuristics that might improve performances for some tasks and ontologies. For instance, there are methods for sorting symbols or set of symbols based on their frequency.

Connectivity Library

This component provides a collection of algorithms to compute the (strongly-) connected components of directed graphs or labelled graph-encoding of hypergraphs. In *HyS* v1.1, the Connectivity Library has a BFS algorithm for traversing directed graphs and another one for traversing directed labelled graphs. Similarly, there is also a DFS algorithm for directed graphs and directed labelled graphs.

The BFS and DFS algorithms are not only used to compute the connected component of a particular node or set of nodes but also to compute the strongly-connected components of a directed labelled graph. DFS alone is able to compute the strongly-connected components of a directed (labelled) graph. In the case of BFS, it is used as subordinated algorithm of the naïve algorithm for identifying the strongly-connected components of a directed (labelled) graph.

Note that these algorithms do not directly manipulate the internal data-structures that the Graph Builder has created. Instead, they use the methods provided by the Graph Builder to access to the data structures.

6.6.2 Interfaces

The current version of *HyS* provides two main interfaces: an API and a command-line interface. The API simplifies the access to the main features of *HyS* from other applications. The command line interface allows users to indicate which operation they would like to execute, where is located the input data and where should be store the results of the execution of *HyS*.

Application Programming Interface

The Java class ‘*HyS*’ provides the main methods to compute a (partially) condensed axiom dependency hypergraph and extract atoms, its dependencies and syntactic local modules. Given an input ontology, the class *HyS* provides access to the following features:

- compute the (partially) condensed axiom dependency hypergraph,
- retrieve the axioms that correspond to a given node,
- retrieve the child (parent) nodes of a given node,
- retrieve the set of nodes that do not have incoming hyperedges (root nodes),
- retrieve the set of nodes that do not have outgoing hyperedges (leave nodes),

```

void Example(OWLOntology o, Set<OWLEntity> s) {
1: HyS h = new HyS(o, ModuleType.BOT);
2: h.condense(SCCAAlgorithm.TARJAN-DG);
3: h.condense(SCCAAlgorithm.NAIVE-DLG);
4: Set<Node> allNodes = h.getNodes();
5: Iterator nItr = allNodes.iterator();
6: while (nItr.hasNext()) {
7:   Set<OWLAxiom> atom =
       h.getAxioms(nItr.next());
8: }
9: Set<Node> connectedComponent =
       h.getConnectedComponent(s,CCAAlgorithm.BFS-DLG);
10:Set<OWLAxiom> module =
       h.getAxioms(connectedComponent);
}

```

Fig. 6.15 Example of the *HyS* API

- retrieve the labels of a given node, and
- retrieve the syntactic locality-based module for a given signature node.

The child (parent) nodes of a node v are the nodes that are in the head (tail) of simple directed hyperedges such that v is the tail (head) of these hyperedges. Moreover, the labels of a node v are the minimal non-locality signatures and the signatures of the axiom (or axioms) that are represented by v .

The API of *HyS* is built on top of the OWLAPI. Therefore, it is possible to use the classes and methods provided by the OWLAPI to store set of axioms as OWL ontologies.

Figure 6.15 shows an example on how to create a (partially) condensed axiom dependency hypergraph; how we can retrieve the axioms that are represented by each node in the condensed axiom dependency hypergraph; and how we can compute a connected component for a given signature that corresponds to a syntactic $\perp$ -local module for that signature.

In Line 1, we create a *HyS* object that computes an axiom dependency hypergraph taking an ontology and a syntactic locality notion as an input. We use the algorithm by Tarjan to determine the strongly-connected components of the graph fragment of the axiom dependency hypergraph. The method `condense(.)` will collapse the strongly connected components into single nodes to produce the partially condensed axiom dependency hypergraph (cf. Line 2). Then we use the naïve algorithm to compute the strongly- Σ -connected components of the partially condensed axiom dependency hypergraph. In Line 3, we apply the method `condense(.)` again to collapse the strongly- Σ -connected components to produce the con-

```

java -jar hys.jar -A -o <path-to-ontology>
                        -d <path-to-folder>
java -jar hys.jar -M <-da|-dp|-dc>
                        -o <path-to-ontology>
                        -d <path-to-folder>

options:
-A stores the atomic decomposition
-M stores syntactic locality-based modules
-da computes only the ADH
-dp computes only the pcADH
-dc computes the cADH
-o input ontology
-d folder for reading signatures and saving
  AD/modules

```

Fig. 6.16 Commands available in *HyS*

densed axiom dependency hypergraph. Lines 4 to 8 illustrate how to compute the atoms of the ontology by first collecting the nodes of the condensed axiom dependency hypergraph and subsequently retrieving the axioms represented by each node. In Line 9, we show how to compute a connected component given a signature. As for atoms, it is possible to retrieve the axioms represented by the nodes of the connected component, which corresponds to the syntactic locality based module of the ontology for the signature (cf. Line 10).

Command-Line Interface

It is also possible to execute *HyS* as a standalone application. Using the command-line interface, we can ask *HyS* to compute and store the atomic decomposition of an ontology or the syntactic locality-based modules for a collection of input signatures. Figure 6.16 shows the possible parameters for *HyS*.

For computing the atomic decomposition of an ontology, we have to indicate the location of the input ontology and the folder for storing the atoms together with their dependencies.

For computing syntactic locality-based modules, we have to provide the location of the input ontology, the folder where the input signatures are stored. The input signature of each module that we want to compute using *HyS* must be stored in a text file with extension ‘.sig’ that contains one signature symbol per line. The computed modules will be stored as OWL-ontologies in the same folder that contains the signatures. It is also possible to indicate to *HyS* to compute the modules using the axiom dependency hypergraph or its (partially) condensed version.

6.7 Evaluation

For the evaluation of *HyS* v1.1, we have selected ten well-known biomedical ontologies. Eight of them are available in the NCBO Bioportal.² The version of Full-Galen that we used is available in the Oxford ontology repository.³

We divide the ontologies into two groups: a group consisting of CHEBI, FMA-lite, Gazetteer, GO, NCBI and RH-Mesh, and another group consisting of Biomodels, CPO, Full-Galen and SNOMED CT. The axiom dependency hypergraph of each ontology of the former group can be represented using a directed graph. This means that the axiom dependency hypergraph does not have any complex directed hyperedge, i.e., for all $(T, H) \in \mathcal{E}$: $|T| = 1$, where $\mathcal{E}$ is the set of hyperedges of an axiom dependency hypergraph. In the case of the ontologies of the latter group, the axiom dependency hypergraph also contains complex directed hyperedges.

In addition, we compared the systems on four artificial ontologies Branch, Cycle, Tree and Tree-cycle that are defined as follows: Branch = $\mathcal{T}_{\text{tree}}(1, 5000)$, Cycle = $\mathcal{T}_{\text{tree-cycle}}(1, 5000)$, Tree = $\mathcal{T}_{\text{tree}}(10, 4)$ and Tree-cycle = $\mathcal{T}_{\text{tree-cycle}}(10, 4)$, where

$$\begin{aligned}\mathcal{T}_{\text{tree}}(b, d) &:= \{A_\pi \sqcap B_\pi \sqsubseteq A_{\pi \cdot j} \sqcap B_{\pi \cdot j} \mid \pi \in [b]^i, j \in [b], 0 \leq i < d\}, \\ \mathcal{T}_{\text{tree-cycle}}(b, d) &:= \mathcal{T}_{\text{tree}}(b, d-1) \cup \{A_\pi \sqcap B_\pi \sqsubseteq A_0 \sqcap B_0 \mid \pi \in [b]^{d-1}\}.\end{aligned}$$

The axiom dependency hypergraph for $\mathcal{T}_{\text{tree}}(b, d)$ is an b -ary tree of depth d . The only difference to the axiom dependency hypergraph for $\mathcal{T}_{\text{tree-cycle}}(b, d)$ is that the leaf nodes are set to be the root node.

The structure of these artificial ontologies represents a tree. In the case of the branch ontologies, they represent a tree with only one branch. The tree ontologies used in the example have multiple branches (each node has 10 child nodes). In the no-loop versions, the module of each axiom of a tree ontology should be smaller on average than in a branch ontology. For the loop version of tree and branch ontologies, the module of each axiom have the same size (all axioms are in the same atom).

Table 6.1 contains, for each ontology, the size of the signature, the number of axioms, the number of axioms of the form $A \sqsubseteq C$, where A is a concept name, the number of axioms of the form $C \equiv D$, the number of role axioms contained in the ontology.

For testing atomic decomposition, we compare *HyS* against two systems: FaCT++ v1.6.3 and OWLAPI-AD v3.5.0. They represent the state of the art applications for computing the atomic decomposition of OWL 2 ontologies and they implement the same algorithms. This

²<http://bioportal.bioontology.org/>

³<http://www.cs.ox.ac.uk/isg/ontologies/>

Table 6.1 Selected ontologies for the evaluation of *HyS*.

Ontology $\mathcal{O}$	Properties of $\mathcal{O}$				
	Signature size	#axioms	#axioms $A \sqsubseteq C$	#axioms $C \equiv D$	#role axioms
CHEBI	37 891	85 347	85 342	0	5
FMA-lite	75 168	119 561	119 558	0	3
Gazetteer	517 039	652 361	652 355	0	6
GO	36 945	72 669	72 667	0	2
NCBI	847 796	847 755	847 755	0	0
RH-Mesh	286 382	403 210	403 210	0	0
Biomodels	187 598	660 188	–	–	–
CPO	136 090	379 668	306 111	73 461	96
Full-Galen	24 088	37 696	25 563	9 968	2 165
SNOMED CT	291 207	291 156	227 698	63 446	12
Branch	–	5 000	0	0	0
Cycle	–	5 000	0	0	0
Tree	–	11 111	0	0	0
Tree-cycle	–	11 111	0	0	0

is interesting to measure the impact of using two different programming languages (C++ and Java).

For testing module extraction using syntactic locality, we compare *HyS* against FaCT++ v1.6.3 and OWLAPI v3.5.0 (module extraction using the OWLAPI v4.0.1, failed for some ontologies).⁴ OWLAPI is a widely used tool for computing syntactic locality-based modules. We choose OWLAPI instead of OWLAPI-AD because the latter does not include in its API the methods to compute locality-based modules. Moreover, we would like to verify how much FaCT++ has improved the execution of the OWLAPI when it computes the same syntactic locality-based modules. We decide to do not include CEL in the evaluation. The last version of CEL was released in 2009 and it seems that the development and maintenance of this tool stopped some years ago.

In the case of *HyS*, we compare the implementation of two different algorithms for directed labelled graphs: the Naïve and Tarjan algorithms depicted in Figures 6.12 & 6.7, respectively. We also show the impact of condensing the graph fragment of an ADH before the entire ADH is condensed. We refer to Naïve-C (Naïve-NC) when the execution of the Naïve algorithm (does not) includes the condensation of the graph fragment.

All experiments were conducted on an Intel Core i5-4590 CPU @ 3.30GHz with 32G RAM running Debian GNU/Linux 8 and Java 1.8.0_40.

⁴The problems with OWLAPI v4.0.1 seems to be solved in version v4.0.2 or later.

Table 6.2 Evaluation of *HyS* for Atomic Decomposition.

Ontology $\mathcal{O}$	Time for Atomic Decomposition of $\mathcal{O}$				
	FaCT++ v1.6.3	OWLAPI-AD v3.5.0	Naïve-NC	<i>HyS</i> v1.1 Naïve-C	Tarjan
CHEBI	91.3 s	1 425.7 s	15.9 s	2.2 s	2.1 s
FMA-lite	11 777.4 s	OOM	OOM	6.0 s	5.9 s
Gazetteer	20 525.9 s	TO	912.7 s	16.3 s	15.9 s
GO	32.3 s	761.9 s	12.4 s	2.0 s	1.9 s
NCBI	36 422.6 s	60 237.1 s	1 759.8 s	40.4 s	39.4 s
RH-Mesh	4 301.7 s	5 935.4 s	385.1 s	10.5 s	10.3 s
Biomodels	OOM	OOM	4 056.9 s	1 029.6 s	44.5 s
CPO	6 210.9 s	19 327.7 s	1 051.7 s	371.5 s	34.3 s
Full-Galen	255.8 s	678.1 s	206.3 s	14.1 s	11.7 s
SNOMED CT	11 790.0 s	33 439.7 s	2 061.9 s	866.1 s	61.7 s
Branch	1 410.2 s	31.0 s	81.3 s	81.2 s	2.0 s
Cycle	29.3 s	49.0 s	124.7 s	124.9 s	1.8 s
Tree	20.7 s	11.5 s	4.9 s	4.7 s	4.2 s
Tree-cycle	181.6 s	334.2 s	7.9 s	8.0 s	2.0 s

Table 6.2 lists the time needed for each system to compute the atomic decomposition of the ontologies. The time values are the average of 10 executions. We applied a timeout (TO) of 24h, which aborted the execution of OWLAPI-AD on the ontology Gazetteer. OWLAPI-AD also could not deal with FMA-lite due to an out-of-memory (OOM) error after about 190 min. Similarly, *HyS* executing Naïve-NC also fails with FMA-lite due to the same problem. FaCT++ and OWLAPI-AD each ran out of memory with Biomodels after about 880 min and 1 440 min, respectively.

For the first two groups of ontologies, *HyS* consistently outperforms FaCT++ which in turn (considerably) outperforms the OWLAPI-AD. In the case of the first group of six ontologies, an over 1 000-fold speedup could be achieved compared to the performance of FaCT++ on FMA-lite and Gazetteer. For the smallest ontology in this group, which is GO, *HyS* is 17 times faster than FaCT++. For the second group of four ontologies, *HyS* is also considerably faster than FaCT++. *HyS* is about 200 times faster than FaCT++ on SNOMED CT and CPO.

The comparison between the performances of FaCT++ and *HyS* was done with respect to *HyS* using Tarjan for directed labelled graphs. For the first group of ontologies, Tarjan and Naïve-C show similar performances. This result is not a surprise because these group of ontologies can be represented using an ADH without complex hyperedges. This corresponds

to a directed graph. *HyS* executes Tarjan for directed graphs before the Naïve algorithm for directed labelled graphs. When Tarjan is executed, all strongly-connected components are found and the execution of the Naïve algorithm is not required. Without condensing the strongly-connected components of the graph fragment, Tarjan is considerable faster than the Naïve algorithm for the first group of ontologies. In particular, Tarjan is 57 times faster on Gazetteer and 45 times faster on NCBI. Note also that the implementation of the Naïve algorithm in *HyS* produces an out-of-memory error on FMA-lite.

The performances of Tarjan and Naïve-C on the second group of ontologies are not so close any more. This is because the ADH of any ontology of the second group has complex hyperedges and the Naïve algorithm must be executed by *HyS*. However, this approach is very effective on reducing the execution time of the Naïve algorithm 2–15 times. Like in the first group of ontologies, Tarjan is significantly faster than the Naïve algorithm. For instance, Tarjan is around 90 times faster on Biomodels and around 30 times faster on CPO and SNOMED CT.

The differences in performance are emphasised by the third group of artificial ontologies. These ontologies are designed in a way to deactivate in *HyS* the computation of the strongly-connected components using Tarjan for directed graphs (which corresponds to computing the pcADH). It can readily be seen that *HyS* significantly outperforms the rest of systems when Tarjan for directed labelled graphs is used. We were expecting that all systems are slower on the cyclic versions of the artificial ontologies (because all modules of these ontologies are as large as the entire ontology and, consequently, the ontology consists of just one atom). However, this is not the case for FaCT++ on the ontology Branch as it needs over 23 min as compared to about 30 s for the ontology Cycle. This is surprising as OWLAPI-AD supposedly implements the same algorithm for atomic decomposition that was used in FaCT++ and it can handle both ontologies rather well. We also notice that the execution of the Naïve algorithm in *HyS* (without computing pcADH) becomes more than 2.5 times slower than OWLAPI-AD on the ontologies Branch and Cycle. This is not the case of Tarjan that significantly outperforms all other systems on the artificial ontologies.

Table 6.3 shows a significant reduction of the number of nodes and directed edges after the axiom dependency graph is partially condensed. This reduction of nodes and directed edges contributes to decrease the time needed to compute all the strongly connected components of the axiom dependency hypergraph.

We also compare the performance of *HyS* for extracting $\perp$ -locality modules with the performance of FaCT++ v1.6.3 and the OWLAPI v3.5.0. Table 6.4 presents, for every system, the time needed to extract a module from an ontology for a signature consisting of

Table 6.3 Size of the labelled graph-encoding of an ADH for $\mathcal{O}$.

Ontology $\mathcal{O}$	Statistics of the labelled graph-encoding of an ADH for $\mathcal{O}$					
	ADH		pcADH		cADH	
	#nodes	#edges	#nodes	#edges	#nodes	#edges
CHEBI	85 344	341 079	32 913	74 204	32 913	74 204
FMA-lite	165 786	991 137	100 873	56 948	100 873	56 948
Gazetteer	652 361	3 703 218	517 028	1 089 717	517 028	1 089 717
GO	273 082	202 175	245 259	69 023	245 259	69 023
NCBI	847 755	847 719	847 755	847 719	847 755	847 719
RH-Mesh	403 210	1 032 421	286 263	401 390	286 263	401 390
Biomodels	660 188	–	365 196	–	365 192	–
CPO	379 734	3 725 565	131 595	779 228	129 157	764 956
Full-Galen	37 696	323 836	20 396	58 759	15 211	38 746
SNOMED CT	573 255	2 494 041	304 770	913 347	304 523	911 581

500 symbols selected at random. Every time value in the table is the average taken over 1000 executions.

Table 6.4 Evaluation of *HyS* for module extraction.

Ontology $\mathcal{O}$	Time for Extraction of $\perp$ -local modules from $\mathcal{O}$				
	FaCT++ v1.6.3	OWLAPI v3.5.0	<i>HyS</i> v1.1		
			ADH	pcADH	cADH
CHEBI	39.3 ms	175.3 ms	3.8 ms	2.3 ms	2.1 ms
FMA-lite	329.2 ms	1 041.7 ms	55.4 ms	3.7 ms	3.2 ms
Gazetteer	181.3 ms	1 502.4 ms	27.2 ms	15.9 ms	15.7 ms
GO	519.5 ms	1 395.8 ms	8.2 ms	6.2 ms	5.9 ms
NCBI	238.7 ms	9 189.2 ms	22.5 ms	15.8 ms	15.9 ms
RH-Mesh	93.0 ms	1 810.7 ms	10.4 ms	9.1 ms	8.8 ms
Biomodels	900.8 ms	1 262.4 ms	287.9 ms	23.2 ms	23.1 ms
CPO	574.9 ms	3 024.2 ms	84.3 ms	53.3 ms	51.4 ms
Full-Galen	76.2 ms	214.8 ms	13.1 ms	3.7 ms	2.9 ms
SNOMED CT	534.3 ms	2 839.9 ms	92.9 ms	87.2 ms	83.9 ms

HyS outperforms FaCT++ and the OWLAPI in all cases. For the first group of six ontologies, the best speedup of over 90 times was achieved in the case of FMA-lite. Notice that module extraction times using the pcADH and the cADH (last two columns) are nearly the same as the two graphs are equivalent. The small variation in extraction time is due to noise in the execution environment. The differences in the times values in the third column and the last two columns correspond to the differences in size of the ADH and the pcADH/cADH. For the second group of four ontologies, the best performance improvement was realised in the case of Full-Galen with a speedup of over 20-times. However, we note

that using the cADH instead of the pcADH does not yield a large performance difference despite the fact that the cADH is slightly smaller than the pcADH. In the particular case of Full-Galen, there appears to be a trade-off between condensation and increased time needed to perform signature containment checks. Computing the partially condensed ADH (using a linear time algorithm) is generally much faster than computing the condensed ADH. Given the that module extraction times are similar when using the pcADH and the cADH (cf. the times in the last two columns), it seems more efficient to only compute modules using the partially condensed ADH.

6.8 Summary

In this chapter we introduce a compact representation of an axiom dependency hypergraph using directed labelled graphs. We showed that Σ -connectivity in the labelled graph corresponds to B -connectivity in the axiom dependency hypergraph. Therefore, a (strongly-) Σ -connected component in the labelled graph corresponds to a (strongly-) B -connected component in the axiom dependency hypergraph. This implies that we can use the labelled graph representation an axiom dependency hypergraph to compute syntactic local modules and the atomic decomposition of an ontology.

In Example 4.8, we showed that from the signature and syntax of an axiom, we can obtain exponentially many minimal non-locality signatures. It is possible to normalized OWL 2 axioms using the normalization rules defined in [34] and generate normalized axioms with only one or two minimal non-locality signatures. We found a couple of simple examples that demonstrate that B -connectivity in an axiom dependency hypergraph might be altered if the ontology is normalized. As a result, we cannot use this normalization rules in OWL 2 ontologies and expect that the modules and atoms are still correct.

In this chapter, we have introduced two new algorithms for traversing directed labelled graphs. One explores the graph similar to BFS and the other similar to DFS. The analysis of these two algorithms concludes that their complexity is the same. Recall that in the case of directed hypergraphs the complexity of BFS and DFS when DFS produces a DFS-graph that it is an approximating graph is different. Based on these two algorithms, we showed how to compute syntactic local modules and the atomic decomposition of ontologies. These algorithms are implemented in *HyS*, a Java application for computing syntactic local modules and the atomic decomposition of OWL-EL ontologies. We compare *HyS* against state-of-the-art implementations using large and well-known biomedical ontologies. The evaluation shows that *HyS* significantly outperforms FaCT++ and OWLAPI-AD on the computation of

the atomic decomposition of ontologies. HyS also demonstrates to be faster than FaCT++ and OWLAPI during the extraction of syntactic $\perp$ -local modules.

Chapter 7

Conclusions and Future Work

In this thesis, we investigate ways to compute the strongly-connected components of directed hypergraphs. We present two new algorithms and we show their correctness and computational complexity. One of these algorithms is inspired by Tarjan’s algorithm [39] for directed graphs. The second algorithm follows a simple approach to compute the strongly-connected components. This approach is based on the fact that two nodes of a graph that are strongly-connected can also reach the same nodes. In other words, the connected component of each node is the same.

Both algorithms are empirically evaluated to compare their performances. To this end, we have produced a selection of random directed hypergraphs inspired by existent and well-known random graph models like Erdős-Rényi [15, 16, 19] and Newman-Watts-Strogatz[44, 30].

In this thesis, we also apply these algorithms for the computation of syntactic local modules and the atomic decomposition of OWL ontologies. To this end, we introduce the notion of an axiom dependency hypergraph which is a generalization of the atomic decomposition of an ontology. A module in the ontology corresponds to a connected component in the hypergraph, and the atoms of the ontology to the strongly-connected components. To demonstrate the viability of this approach, we have implemented the algorithms in the application HyS which computes the modules of ontologies and calculate their atomic decomposition. In the thesis, we provide an experimental evaluation of HyS with a selection of large and prominent biomedical ontologies, most of which are available in the NCBO Bioportal. HyS outperforms state-of-the-art implementations in the tasks of extracting modules and computing the atomic decomposition of these ontologies.

7.1 Research Methodology

The research presented in this thesis has been conducted on an exploratory manner. We started with one or more related observations. From these observations, we define one or more hypotheses. Next, we try to show these hypotheses theoretically or by conducting an experiment. In both cases, it might be required to define new notions and prove certain properties of these notions to show that the hypotheses are correct. When it was possible, we submitted a publication to obtain feedback on our approaches and results.

The beginning of the research presented in this thesis was motivated by an observation during the execution of the module extraction algorithm depicted in Figure 4.1 on a very simple ontology. The ontology used in the experiment was similar to the ontology of Example 4.5. This module extraction algorithm iteratively adds new axioms of the ontology into the module until no more axioms can be found. In each iteration, the algorithm extends the input signature with the new symbols of the axioms recently added in the module. The algorithm produces a finite sequence of subsets of the input ontology. We observed that this sequence induces a binary relation on O representing the successive inclusion of the axioms into the module (**O1**). We also noticed that a directed graph or hypergraph might provide a suitable representation of this relation. So, our first hypothesis (**H1**) was defined as follows: **H1**: *the binary relation induced by the module extraction algorithm can be represented by a directed hypergraph*. The definition of such hypergraph representation motivates the introduction of the notion of axiom dependency hypergraph (cf. Definition 5.1) which generalizes the previous notion.

After analysing some simple axiom dependency hypergraphs, we observed that connected components in the hypergraph might correspond to syntactic local modules and strongly-connected components to atoms in an ontology (**O2**). Based on this observation, we define the following hypothesis: **H2**: *syntactic local modules and atoms in an ontology correspond to connected and strongly-connected components, respectively, in the axiom dependency hypergraph*. Proposition 5.15 shows the correspondence between the atoms of an ontology and the strongly-connected components of an axiom dependency hypergraph. However, Proposition 5.6 only shows the correspondence between syntactic local modules of an ontology and connected components in the axiom dependency hypergraph if the signature to compute the module is the union of the signatures of a set of axioms of the ontology. To compute a syntactic local module for any possible signature Σ , we introduced the notion of Σ -extension of an axiom dependency hypergraph (cf. Proposition 5.9).

After we established some relevant notions and properties, we were wondering how efficient could be the computation of the atoms of an ontology using an axiom dependency hypergraph. We noticed that there was not any algorithm for computing strongly-connected

components in directed hypergraphs. However, we observed that large fragments of well-known biomedical ontologies can be represented using a directed graph instead of a hypergraph (**O3**). For directed graphs, there are algorithms that can compute all strongly-connected components in linear time with respect to the size of the graph. Based on this information, we define the following hypothesis: **H3**: *Computing atoms using Tarjan’s algorithm in a directed graph might be faster than other known approaches*. Our experiments confirmed hypothesis H3 and even more, our prototype was able to compute the atomic decomposition of large fragments of ontologies in seconds whereas state-of-the-art applications require many hours. These results were published in [25]. Extending the implementation to compute the modules and the atomic decomposition of complete ontologies was problematic because there were no algorithms for computing strongly-connected in directed hypergraphs (cf. Section 2.6). Moreover, storing the entire axiom dependency hypergraph was not feasible for large ontologies. The former problem motivates the design of new algorithms for traversing directed hypergraphs and to compute the strongly-connected components of these hypergraphs (Chapter 3). The need of a compact representation of an axiom dependency hypergraph motivates the introduction of the notion of labelled graph-encoding for axiom dependency hypergraph (Chapter 6).

7.2 Significance of Main Contributions

The main contributions of the thesis are:

- C1:** A new DFS algorithm for directed hypergraphs.
- C2:** Two new algorithms for computing strongly-connected components in directed hypergraphs.
- C3:** The notions of axiom dependency hypergraph and its labelled graph representation.
- C4:** The application HyS for computing syntactic local modules and the atomic decomposition of OWL-EL ontologies.

C1 & C2 are relevant for researchers and practitioners interested in modelling many-to-many relations using a graph representation. For instance, some social networks might include supra-dyadic relations where more than two people are involved [8]. There are also metabolic reactions that require several source compounds and produce more than one target compound [22]. For any of these hyper-networks [17], a directed hypergraph might be a suitable representation that requires algorithms for traversing the hypergraph or computing its strongly-connected components.

C3 & C4 are relevant for researchers and practitioners interested in ontologies. An axiom dependency hypergraph provides detailed information about how modules are computed. Ontology engineers might find this information useful when they are modifying or importing an ontology. For a practical point view, it is important to have an implementation that can significantly reduce the time needed to compute modules or to compute the atomic decomposition of an ontology. Recall that for some on ontologies of the evaluation conducted in Chapter 6, computing their atomic decomposition might require hours or even days with state-of-the-art systems. *HyS* reduces this time to minutes or even seconds.

The notion of an axiom dependency hypergraph also compliments the theory of modularization and atomic decomposition. In particular, it establishes a relation with graph theory. As we did in the thesis, notions and algorithms from graph theory might be applicable for problems related to modularization and atomic decomposition of ontologies.

7.3 Future Work

To date the theory of directed hypergraphs has not been developed as much as it is the case for directed graphs. Notions and algorithms available for directed graphs do not have a counterpart for directed hypergraphs. For instance, there are several algorithms to compute the shortest path between two nodes of a directed graph in polynomial time. This problem is relevant in the analysis of networks [29]. However, such algorithms do not exist for directed hypergraphs and only algorithms to compute an “approximation” of the shortest path of a directed hypergraph have been suggested [18]. To encourage the adoption of directed hypergraphs in real applications, new notions and algorithms should be defined, implemented and evaluated.

The theory on axiom dependency hypergraphs can also be extended. An axiom of an ontology might have exponentially many minimal non-locality signatures. Normalisation rules for OWL 2 ontologies do not appear the solution for this problem and we might need to find an alternative in the near future. Also it is unclear if this is a problem in real ontologies. A study on a representative collection of ontologies should be conducted to determine the maximal number of minimal non-locality signatures generated by a single axiom.

The functionality of *HyS* must be also extended to support OWL 2 ontologies and new syntactic locality notions like $\top$ -locality.

References

- [1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.
- [2] Allamigeon, X. (2014). On the complexity of strongly connected components in directed hypergraphs. *Algorithmica*, 69(2).
- [3] Ausiello, G., D’Atri, A., and Saccà, D. (1983). Graph algorithms for functional dependency manipulation. *Journal of the ACM (JACM)*, 30(4):752–766.
- [4] Ausiello, G. and Italiano, G. F. (1991). On-line algorithms for polynomially solvable satisfiability problems. *The Journal of Logic Programming*, 10(1):69–90.
- [5] Ausiello, G., Italiano, G. F., Laura, L., Nanni, U., and Sarracco, F. (2010). Classification and traversal algorithmic techniques for optimization problems on directed hyperpaths. *Department of Computer and System Sciences Antonio Ruberti (Università de Roma)*, 2(18).
- [6] Baader, F., Brandt, S., and Lutz, C. (2008). Pushing the el envelope further. In Clark, K. and Patel-Schneider, P. F., editors, *In Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions*.
- [7] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2007). *The description logic handbook: theory, implementation, and applications*. Cambridge University Press.
- [8] Bonacich, P., Holdren, A. C., and Johnston, M. (2004). Hyper-edges and multidimensional centrality. *Social networks*, 26(3):189–203.
- [9] Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J. (2000). Graph structure in the web. *Computer networks*, 33(1):309–320.
- [10] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.
- [11] Cuenca-Grau, B., Horrocks, I., Kazakov, Y., and Sattler, U. (2007). Just the right amount: extracting modules from ontologies. In *Proceedings of WWW’07: the 16th international conference on World Wide Web*, pages 717–726, New York, NY, USA. ACM.
- [12] Cuenca-Grau, B., Horrocks, I., Kazakov, Y., and Sattler, U. (2008). Modular reuse of ontologies: theory and practice. *JAIR*, 31:273–318.

- [13] Del Vescovo, C., Gessler, D. D. G., Klinov, P., Parsia, B., Sattler, U., Schneider, T., and Winget, A. (2011a). Decomposition and modular structure of bioportal ontologies. In *Proc. of ISWC'11*, pages 130–145. Springer-Verlag.
- [14] Del Vescovo, C., Parsia, B., Sattler, U., and Schneider, T. (2011b). The modular structure of an ontology: Atomic decomposition. In *Proc. of IJCAI'11*, pages 2232–2237.
- [15] Erdős, P. and Rényi, A. (1959). On random graphs. i. *Publicationes Mathematicae*, 6:290–297.
- [16] Erdős, P. and Rényi, A. (1960). On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61.
- [17] Estrada, E. and Rodríguez-Velázquez, J. A. (2006). Subgraph centrality and clustering in complex hyper-networks. *Physica A: Statistical Mechanics and its Applications*, 364:581–594.
- [18] Gallo, G., Longo, G., Pallottino, S., and Nguyen, S. (1993). Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2–3):177–201.
- [19] Gilbert, E. N. (1959). Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144.
- [20] Gunopulos, D., Mannila, H., Khardon, R., and Toivonen, H. (1997). Data mining, hypergraph transversals, and machine learning. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 209–216. ACM.
- [21] Horrocks, I., Kutz, O., and Sattler, U. (2006). The even more irresistible SROIQ. In Doherty, P., Mylopoulos, J., and Welty, C. A., editors, *Proceedings of KR'06: the 10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 57–67. AAAI Press.
- [22] Klamt, S., Haus, U.-U., and Theis, F. (2009). Hypergraphs and cellular networks. *PLoS Computational Biology*, 5(5):e1000385.
- [23] Klein, D. and Manning, C. D. (2005). Parsing and hypergraphs. In *New developments in parsing technology*, pages 351–372. Springer.
- [24] Ma, H.-W. and Zeng, A.-P. (2003). The connectivity structure, giant strong component and centrality of metabolic networks. *Bioinformatics*, 19(11):1423–1430.
- [25] Martín-Recuerda, F. and Walther, D. (2013). Towards fast atomic decomposition using axiom dependency hypergraphs. In *Workshop on Modular Ontologies (WoMO) 2013*, CEUR Workshop Proceedings, pages 61–72. CEUR-WS.org.
- [26] Martín-Recuerda, F. and Walther, D. (2014). Axiom dependency hypergraphs for fast modularisation and atomic decomposition. In Bienvenu, M., Ortiz, M., Rosati, R., and Simkus, M., editors, *Proceedings of the 27th International Workshop on Description Logics (DL'14)*, volume 1193 of *CEUR Workshop Proceedings*, pages 299–310.

- [27] Martín-Recuerda, F. and Walther, D. (2014). Fast modularisation and atomic decomposition of ontologies using axiom dependency hypergraphs. In *Proceedings of ISWC'14: the 13th International Semantic Web Conference*, volume 8797 of *Lecture Notes in Computer Science*, pages 49–64. Springer International Publishing.
- [28] Mittelstädt, P. (2015). Concepts of connection in directed hypergraphs. Master's thesis, TU Dresden, Germany.
- [29] Newman, M. (2010). *Networks: an introduction*. Oxford University Press.
- [30] Newman, M. E. and Watts, D. J. (1999). Renormalization group analysis of the small-world network model. *Physics Letters A*, 263(4):341–346.
- [31] Nguyen, S., Pretolani, D., and Markenson, L. (1998). On some path problems on oriented hypergraphs. *ITA*, 32(1–3):1–20.
- [32] Nielsen, L. R., Munkegade, N., and Pretolani, D. (2001). A remark on the definition of a *B*-hyperpath. Technical report, Department of Operations Research, University of Aarhus.
- [33] Nortje, R., Britz, A., and Meyer, T. (2012). A normal form for hypergraph-based module extraction for SROIQ. In *Proc. of AOW 2012*, volume 969 of *CEUR Workshop Proceedings*, pages 40–51. CEUR-WS.org.
- [34] Nortje, R., Britz, A., and Meyer, T. (2013). Reachability modules for the description logic SRIQ. In *Proc. of LPAR-13*.
- [35] Pretolani, D. (2004). Hypergraph reductions and satisfiability problems. In *Theory and Applications of Satisfiability Testing*, pages 383–397. Springer.
- [36] Sattler, U., Schneider, T., and Zakharyashev, M. (2009). Which kind of module should i extract? In *Proceedings of DL'09*. CEUR-WS.
- [37] Sedgewick, R. and Wayne, K. (2011). *Algorithms, Fourth Edition*. Addison-Wesley, 4th edition.
- [38] Suntisrivaraporn, B. (2009). *Polynomial time reasoning support for design and maintenance of large-scale biomedical ontologies*. PhD thesis, TU Dresden, Germany.
- [39] Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160.
- [40] Thakur, M. and Tripathi, R. (2009). Linear connectivity problems in directed hypergraphs. *Theoretical Computer Science*, 410(27):2592–2618.
- [41] Tsarkov, D. (2012). Improved algorithms for module extraction and atomic decomposition. In *Proc. of DL'12*, volume 846 of *CEUR Workshop Proceedings*. CEUR-WS.org.
- [42] Tsarkov, D., Vescovo, C. D., and Palmisano, I. (2013). Instrumenting atomic decomposition: Software apis for owl. In Rodriguez-Muro, M., Jupp, S., and Srinivas, K., editors, *Proceedings of OWLED-13: the 10th International Workshop on OWL: Experiences and Directions*, volume 1080 of *CEUR Workshop Proceedings*. CEUR-WS.org.

-
- [43] Turlapati, V. K. C. and Puligundla, S. K. (2013). Efficient module extraction for large ontologies. In Klinov, P. and Mouromtsev, D., editors, *Knowledge Engineering and the Semantic Web*, volume 394 of *Communications in Computer and Information Science*, pages 162–176. Springer Berlin Heidelberg.
 - [44] Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of ‘small-world’ networks. *nature*, 393(6684):440–442.
 - [45] Zhao, J., Yu, H., Luo, J.-H., Cao, Z.-W., and Li, Y.-X. (2006). Hierarchical modularity of nested bow-ties in metabolic networks. *BMC bioinformatics*, 7(1):386.