CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Implementación de un algoritmo de búsqueda aleatoria en programación lógica

Autor: Inés Blázquez Ballesteros

Directores: Manuel Hermenegildo Salinas

José Francisco Morales Caballero

MADRID, JUNIO 2017

Agradecimientos:

A José M., sin cuya ayuda no habría sido posible este proyecto.

Y a Mike y a Enrique, por las horas de apoyo moral pasadas al otro lado del teléfono.

Tabla de contenido

Resumen: vi

Abstract: vi

INTRODUCCIÓN Y OBJETIVOS 1

1.1. INTRODUCCIÓN 1

1.2. MOTIVACIONES 2

TRABAJOS PREVIOS 4

2.1. LA PROGRAMACIÓN LÓGICA PURA 4

2.2. LA ESTADÍSTICA EN LA PROGRAMACIÓN LÓGICA: *PROBABILISTIC PROGRAMMING & STATISTICAL MODELING* 8

DESARROLLO 11

3.1. DISEÑO DE UN ALGORITMO DE BÚSQUEDA ALEATORIA 11

3.2. ESTABLECIMIENTO DE MEDIDAS PARA EVITAR LA REPETICIÓN DE SOLUCIONES 13

3.3. CÓDIGO DEL ALGORITMO 15

PRUEBAS DE RENDIMIENTO 20

4.1. LA BÚSQUEDA ALEATORIA FRENTE A LA BÚSQUEDA EN ANCHURA 20

4.2. LA BÚSQUEDA ALEATORIA FRENTE A LA BÚSQUEDA EN PROFUNDIDAD 24

4.3. RESULTADOS DE LOS CASOS DE PRUEBAS 27

4.4. CASOS PRÁCTICOS DE USO 37

4.5. COMPARATIVA DEL ALGORITMO CREADO FRENTE A BÚSQUEDAS CLÁSICAS 38

CONCLUSIONES 42

Bibliografía 45

Tabla de ilustraciones

Ilustración 1- Ejemplo de traza ... 14
Ilustración 2- Traza distinta con mismos valores de unificación ... 14
Ilustración 3- Árbol de búsqueda en anchura sin explorar ... 20
Ilustración 4- Primeros pasos de la búsqueda en anchura ... 21
Ilustración 5- Primera solución en anchura ... 21
Ilustración 6- Primera solución en anchura rechazada ... 22
Ilustración 7- Segunda solución en anchura ... 22
Ilustración 8- Segunda solución en anchura rechazada ... 23
Ilustración 9- Fin del árbol de búsqueda por anchura ... 23
Ilustración 10- Árbol de búsqueda por profundidad sin explorar ... 24
Ilustración 11- Primera solución por profundidad ... 25
Ilustración 12- Primera solución por profundidad rechazada ... 25
Ilustración 13- Segunda solución por profundidad ... 25
Ilustración 14- Segunda solución por profundidad rechazada ... 26
Ilustración 15- Fin del árbol de búsqueda por profundidad ... 26
Ilustración 16- Torres de Hanoi ... 38
Ilustración 17- Problema del lobo, la oveja y la rana ... 39

Resumen:

En el presente trabajo se explican los detalles de la creación de un algoritmo de búsqueda aleatoria. Este algoritmo permite recorrer los grafos obtenidos en el campo de la programación lógica de forma alternativa a la ofrecida por las búsquedas en anchura y en profundidad. Además se justifican los motivos por los que se considera necesario este algoritmo y los objetivos previos a su desarrollo.

Asimismo, se exponen los resultados obtenidos tras la aplicación de este algoritmo a distintos programas de muestra, un estudio sobre estos y diversas conclusiones extraídas a raíz de ellos.

Abstract:

On this paper we will discuss about the creation details of a random search algorithm. Such algorithm allows an alternative way of exploring the graphs obtained in logic programming to the breadth first and depth first searches. In addition, reasons why this new way of obtaining solutions is considered necessary and motivations for the project are also justified.

Besides, the results obtained after applying this algorithm to some sample programs are shown. This document also contains a study on these results and some conclusions drawn from it.

Capítulo 1

INTRODUCCIÓN Y OBJETIVOS

1.1. INTRODUCCIÓN

La programación lógica[1] se encuentra estrechamente unida al pensamiento científico. Con el desarrollo de los primeros ordenadores, surgieron diversos lenguajes que forman parte del paradigma de la programación imperativa y permiten describir de forma explícita los pasos de ejecución necesarios para lograr un objetivo.

Con el avance de la informática, a finales de los años sesenta y principios de los setenta, surge un paradigma de programación basado en la unión de la programación declarativa y la lógica pura. Las bases para el nacimiento de este tipo de programación fueron asentadas por científicos del Instituto Tecnológico de Massachusetts y de la Universidad de Stanford, que necesitaban para sus estudios en Inteligencia Artificial una aproximación más natural al lenguaje matemático basado en la aplicación de reglas, hipótesis y teoremas a un conjunto de hechos con un objetivo final. Es decir, en lugar de indicar la secuencia de instrucciones a ejecutar para lograr un objetivo, se describe el problema que se quiere solucionar y se dota al programa de mecanismos de inferencia de la información proporcionada.

Aunque son varios los lenguajes que forman parte del campo de la programación lógica, en este proyecto nos centramos en el estudio del lenguaje Prolog, que si bien comparte las bases con el resto de lenguajes –lenguaje basado en representación de hechos, reglas y el logro de un objetivo- tiene también particularidades que hacen necesaria su evaluación individualizada. La elección de este lenguaje y no otro se debe a que actualmente es el lenguaje más extendido y asentado en la comunidad científica dentro del campo de la programación lógica.

El uso de este lenguaje crea grafos conocidos como árboles de búsqueda[2]. Se trata de un tipo particular de grafos que constan de una raíz a partir de la cual surge la búsqueda, distintas ramas (que se interpretan como las distintas unificaciones llevadas a cabo en un momento dado del grafo) y hojas o nodos sin hijos, que constituyen las soluciones del problema de la raíz. Como para el resto de grafos, existen multitud de formas de recorrer sus nodos y por tanto distintos órdenes para generar las soluciones. Tradicionalmente, Prolog hace uso de la búsqueda en profundidad, pudiendo utilizar también la búsqueda en anchura si así se especifica. Estos dos tipos de búsqueda, cuyas diferencias se explicarán en capítulos posteriores, han probado su eficacia a la hora de encontrar todas las soluciones existentes.

Aunque estas búsquedas se encuentran ampliamente aceptadas en el campo de la programación lógica y han probado ser capaces de dar solución a todos los problemas

planteados hasta la fecha, el orden en el que muestran las soluciones es siempre el mismo. Esta característica hace que sean poco representativas para algunos escenarios donde lo que se pretende es realizar un estudio de los distintos tipos de soluciones y, por tanto, se necesita que estas se encuentren uniformemente distribuidas entre el espacio total de soluciones.

Siendo así, se ha considerado necesario desarrollar un nuevo tipo de búsqueda que permita recorrer el grafo de manera que el orden en el que se encuentran las soluciones no sea predecible. De esta forma, se ha procedido a realizar un diseño que consiga que el orden de generación de ramas del árbol sea aleatorio y unos mecanismos que permitan que, aun siendo aleatorio el orden de las soluciones, estas no se repitan.

Por último, la creación de este nuevo mecanismo de búsqueda permite ampliar las aplicaciones de la programación lógica y vincularla de manera más estrecha a la programación probabilística.

1.2. MOTIVACIONES

La finalidad de este proyecto es, en resumidas cuentas, encontrar un modo de complementar las búsquedas clásicas que ya se encuentran desarrolladas para programación lógica. De esta forma se plantea un doble objetivo: desarrollar un algoritmo que consiga cubrir el vacío que nos dejan la búsqueda en anchura y en profundidad, y probar dicho algoritmo con el fin de establecer unas pautas para decidir cuándo los resultados de este algoritmo pueden superar a los de otros métodos de búsqueda.

Como se explica en el apartado anterior, la programación lógica es un tipo de programación basada en la lógica formal cuyo método resolutivo consiste en la construcción de árboles de búsqueda. Estos árboles conforman el espacio de soluciones posibles para satisfacer una meta fijada. Tradicionalmente existen dos maneras de recorrer dicho árbol que son las más extendidas: búsqueda en anchura y búsqueda en profundidad. A lo largo del presente documento se explicarán detalladamente las diferencias entre ambas y se abordará sobre el funcionamiento del *backtracking* y los problemas que este ha supuesto para el desarrollo de la búsqueda aleatoria. Además, se demostrará cómo la utilización de cualquiera de estas búsquedas en programación supone un alto coste potencial de recursos de memoria y de tiempo, que se ve incrementado de manera exponencial para vastos espacios de búsqueda debido a la necesidad de mantener en memoria el árbol desarrollado y al tiempo que es necesario para recorrer la totalidad de sus nodos.

Por tanto, la motivación principal de la creación de este algoritmo ha sido la de mejorar el uso de recursos en determinadas ejecuciones. Si bien otros métodos de exploración de

soluciones han probado ya su eficacia a la hora de encontrar todas las soluciones existentes, la búsqueda aleatoria propone mejorar su eficiencia, reduciendo costes de tiempo y memoria.

Otra de las razones que motivaron el proyecto fue la de adentrarse en una investigación con potencial para ser ampliada en nuevas direcciones en el futuro, por ejemplo hacia el campo de la estadística o en su integración con otras aplicaciones.

En esta memoria, el lector podrá encontrar un resumen del estado del arte, una descripción del código del algoritmo desarrollado, su diseño y todos los mecanismos de control establecidos para su funcionamiento, un resumen de las pruebas realizadas y una comparación de los resultados frente a otros tipos de búsquedas, y un resumen de las ventajas y beneficios que ofrece el proyecto al campo de la programación lógica.

Capítulo 2

TRABAJOS PREVIOS

2.1. LA PROGRAMACIÓN LÓGICA PURA

Como ya se mencionó, la programación lógica es un tipo de programación declarativa cuyo origen data de mediados de los años sesenta y principios de los setenta[1]. Su funcionamiento se caracteriza por la capacidad de decidir qué acciones realizar en función de una descripción dada del problema y unas restricciones sobre el escenario.

Existen diversos lenguajes basados en la programación lógica, siendo el más representativo de entre todos ellos Prolog, que es el que se encuentra más extendido entre la comunidad científica. Este proyecto ha utilizado por este motivo Prolog tanto para programar el algoritmo principal como para la implementación de sus módulos complementarios. Además, el método de búsqueda aleatoria desarrollado interpreta programas que están a su vez escritos en Prolog.

Prolog[3] (formado por las palabras en francés *PROgrammation en LOGique*) fue implementado por primera vez en la Universidad de Marsella (Francia) en los años setenta, por un equipo dirigido por A. Colmeraeur, que se basó a su vez en resultados teóricos aportados por R. Kowalski, de la Universidad de Edimburgo. Se trata de un lenguaje de programación lógica de propósito general con sintaxis estandarizada en el documento ISO/IEC 13211-1 (conocido como ISO-Prolog).

Para una mejor comprensión global del presente documento, en este apartado se detalla parte del funcionamiento básico de Prolog, así como descripciones de sus componentes principales.

Para comenzar, es necesario entender que Prolog está formado por distintos elementos[4]. De todos ellos, las **variables** son las más pequeñas. Se representan empezando o bien por una letra mayúscula o por guión bajo seguidos de cualquier secuencia de caracteres alfanuméricos (A, Var, T1, _, _3, G_p,…). Existen dos tipos de variables. Se habla de **variables libres** cuando aún no han sido unificadas, es decir, cuando aún no se ha hallado un valor que hace verdad el predicado. Por otra parte, cuando una variable ya ha sido unificada se denomina **variable ligada** y mantiene su valor hasta el final de la ejecución.

Para conseguir que una variable pase de ser libre a estar ligada hace falta que se dé una **unificación** con constantes. La unificación, como en lógica pura, consiste en hallar un valor para la variable de tal forma que al ligar este valor a dicha variable en todas sus apariciones de la misma regla, el resultado se verifique. Es importante tener en cuenta que este proceso representa la igualdad lógica y no es el mismo que la asignación en

programación imperativa, ya que en este caso se trata de encontrar valores que cumplen la regla y no imponerlos.

Por otra parte, los programas en Prolog están formados por **predicados**. Estos predicados están formados por un **functor** que puede estar seguido de un número de argumentos entre paréntesis separados por comas. Un functor es un término de Prolog que identifica el predicado. Al ser una constante, comienza por letra minúscula y puede contener a continuación cualquier número de caracteres alfanuméricos (func_A, max, aux1,…). Al número de argumentos que acompañan al functor se le denomina **aridad**. Al conjunto de predicados con el mismo functor y la misma aridad se les representa por el functor seguido de la aridad separado por una barra oblicua (max/2, func/3, aux/0,…). Si un predicado tiene argumentos, estos pueden ser variables (libres o ligadas) o constantes.

Las **reglas** son conjuntos de cláusulas de Horn[5] que definen relaciones de causalidad entre sus predicados. De esta forma son capaces de expresar condiciones del tipo "si se cumplen A, B y C entonces se cumple D". Al conjunto de predicados que deben cumplirse para que se verifique el primer predicado (antecedentes) se le denomina **cuerpo** de la regla. Se trata de un conjunto de predicados separados entre sí por conjunciones o disyunciones. El consecuente del cuerpo es conocido como **cabeza** de la regla y se compone de un único predicado. Para escribir reglas en Prolog se deberá escribir la cabeza seguida del cuerpo y separados por un operador que varía dependiendo del tipo de búsqueda que se desee emplear para encontrar la solución (en Ciao Prolog[6] ':-' para búsqueda en profundidad y '<-' para búsqueda en anchura).

Existe un tipo particular de reglas cuyo cuerpo se cumple siempre: los **hechos**. Dado que se trata de una condición de consecuencia lógica si el cuerpo se cumple la cabeza también se cumple, por lo que en este caso la cabeza se cumple siempre. Este tipo especial de predicados suelen representarse omitiendo el cuerpo, de forma que permanece únicamente la cabeza de la regla.

Una vez comprendidos los componentes del lenguaje es preciso hacer lo mismo con su funcionamiento. Como ya se ha dicho, para encontrar una solución es necesario encontrar una unificación para todas las variables que haga que el cuerpo de las reglas se verifique y de esta forma su cabeza también se cumpla.

Para indicar al programa qué búsqueda se quiere hacer, el usuario formula una ***query***. De esta forma, es posible interpretar las *queries* como peticiones de unificación que el usuario realiza al programa.

Cada *query* crea un **árbol de búsqueda**. Los árboles de búsqueda son tipos particulares de grafos donde los nodos se colocan siguiendo relaciones padre-hijo, de forma que existe un único nodo sin padre al que se denomina **raíz** del árbol y diversos nodos sin hijos que se denominan **hojas**. En Prolog, la raíz del árbol representa la *query* realizada, mientras que las hojas representan las posibles soluciones para dicha búsqueda. Existe además un único modo o camino para llegar de la raíz a cada una de las soluciones, que

representa cada una de las unificaciones que tienen lugar durante la ejecución del programa para poder encontrar dicha solución.

En el presente documento se hace especial hincapié sobre las diferencias entre realizar una *query* y repetirla a continuación y realizar una *query* y continuar su ejecución con nuevas búsquedas de soluciones. Si un usuario decide realizar una petición al programa y como consecuencia obtiene unos valores de unificación para su *query*, podría decidir volver a realizar la misma petición en un futuro, lo que en el caso de una búsqueda ordenada produciría las mismas soluciones. Por otro lado, el mismo usuario podría realizar una petición al programa y una vez obtenida la primera solución, decidir seguir ejecutando la misma *query* con el fin de obtener nuevas soluciones, lo que en su lugar resultaría en la continuación de la exploración del mismo árbol de búsqueda. Esta diferencia resulta crucial a la hora de comprender el correcto funcionamiento del algoritmo desarrollado y sus objetivos.

Como para recorrer un grafo, existen múltiples maneras de recorrer los nodos de un árbol que permiten encontrar las soluciones. Las dos maneras más extendidas en la programación lógica son la **búsqueda en anchura** y la **búsqueda en profundidad**, de las que se hablará en posteriores capítulos. La presente investigación constituye en sí misma el desarrollo de una nueva forma de exploración del árbol y el estudio de sus posibles ventajas.

Otros importantes mecanismos que presenta Prolog y que resultan relevantes para este proyecto son el *backtracking* y el corte.

El ***backtracking***[7] es un mecanismo interno que permite que habiendo llegado a un nodo del árbol que no tiene más nodos adyacentes por explorar, la búsqueda continúe volviendo hacia atrás en el árbol de búsqueda y llegando al último nodo explorado que sí presente un nodo adyacente aún no visitado, y siga el recorrido desde ahí.

Por otro lado, el **corte**[8, 9] es un mecanismo que permite controlar el orden de ejecución de las ramas del árbol, pudiendo llegar incluso a modificar el significado del programa. Se representa con el operador **!** y su ejecución se cumple siempre, es decir, nunca da fallo. Resulta una herramienta útil para los programadores ya que permite acortar los espacios de búsqueda y por lo tanto consigue ahorros en tiempo de ejecución.

Veamos un ejemplo de los conceptos previos. El programa siguiente escrito en Prolog permite distinguir los vínculos familiares entre distintas personas.

```
:- module(_,_).

% predicado padre/2
padre(juan,ana).
padre(benito,carlos).
padre(carlos,jorge).

% predicado madre/2
```

```
madre(julia,carlos).
madre(maria,ana).
madre(ana,jorge).

%predicado abuelo/2
abuelo(A,B):- padre(A,C), padre(C,B).
abuelo(A,B):- padre(A,C), madre(C,B).

%predicado abuela/2
abuela(A,B):- madre(A,C), padre(C,B).
abuela(A,B):- madre(A,C), madre(C,B).
```

En el ejemplo anterior vemos cuatro predicados distintos (padre/2, madre/2, abuelo/2 y abuela/2) que permiten definir las relaciones entre los distintos miembros. Los predicados padre/2 y madre/2 constituyen la base de hechos del programa relacionando a los siete miembros de la familia, que aparecen como constantes del programa. Los predicados de abuelo/2 y abuela/2 están definidos como cabeza de reglas cuyo cuerpo se forma a su vez por la conjunción entre dos predicados y cuyos argumentos son variables.

Recordemos que la aparición de variables permite la resolución de consultas al programa mediante el método de unificación.

Supongamos que queremos comprobar que Benito sea abuelo de Jorge. La *query* o consulta relacionada sería del siguiente tipo:

```
?- abuelo(benito,jorge).
```

Una vez introducida la consulta, el mecanismo de búsqueda hallaría la primera ocurrencia del predicado abuelo/2 y unificaría las variables libres A y B del programa con las constantes introducidas en la consulta. Así, se pasaría de la regla '`abuelo(A,B):- padre(A,C), padre(C,B).`' a la regla '`abuelo(benito,jorge):- padre(benito,C), padre(C,jorge).`'.

De esta forma, las variables A y B pasarían a ser variables ligadas. Como se ve, al unificarse o ligarse en su primera aparición como argumentos del functor 'abuelo' también se han unificado en su resto de apariciones en la regla.

Como la variable C aún no ha sido unificada, habría que buscar una constante que cumpliera '`padre(benito,C).`' y '`padre(C,jorge).`' en la base de hechos. Como se puede ver, la constante 'carlos' podría unificarse con la variable C. Al haber conseguido una unificación válida para todas las variables de la regla, el cuerpo se cumple y da lugar a que la cabeza también sea verdadera. Por tanto podemos concluir que Benito sí es abuelo de Jorge.

Por otra parte, el usuario puede realizar también *queries* con variables en lugar de constantes. Esto nos permite saber exactamente qué valores cumplen una determinada regla.

Por ejemplo, la consulta:

```
?- abuela(X,Y).
```

da lugar a una búsqueda en la base de hechos de dos constantes que cumplan el cuerpo de la regla y se unifiquen con la cabeza de esta.

Como ya se habló, Prolog permite además repetir consultas de un mismo tipo hasta agotar todas las soluciones, de forma que, si al encontrar una unificación que da validez al predicado se le indica que se desea ver otra solución distinta, hace uso de la técnica de *backtracking* para volver hacia atrás en el árbol de búsqueda hasta el último nodo explorado y continuar desde allí. De esta forma, en el ejemplo anterior el usuario podría obtener todos los pares de variables que cumplen la consulta realizada y de esta forma recorrer el árbol de búsqueda en su totalidad.

2.2. LA ESTADÍSTICA EN LA PROGRAMACIÓN LÓGICA: *PROBABILISTIC PROGRAMMING & STATISTICAL MODELING*

Existen diversos campos de estudio matemático con los que el proyecto guarda una estrecha relación, bien sea por los posibles usos que se pueden dar al código resultante o por las fuentes que se han utilizado como inspiración para la creación del mismo.

Por una parte, la estadística es la rama de las matemáticas que estudia análisis provenientes de una muestra representativa de datos. Se encarga de la elaboración de medidas descriptivas de un conjunto que permiten la toma de decisiones y establecer generalizaciones sobre la muestra. En nuestro algoritmo, el uso de la estadística sobre las soluciones aleatorias obtenidas permitiría el encuentro de patrones o relaciones entre las soluciones, de forma que se podrían extraer conclusiones sobre la aleatoriedad real de los programas.

Por otro lado, la probabilidad es la parte de las matemáticas que se encarga de medir la certidumbre que hay de que un evento o suceso futuro tenga lugar. Suele expresarse con un número entre el 0 y el 1 (ambos incluidos) de forma que el 0 significa total seguridad de que el evento no ocurrirá y el 1 certeza del caso contrario. Estas medidas de la certidumbre o probabilidades se pueden expresar también como porcentajes. La probabilidad se puede medir también como la frecuencia con la que pasa un evento en un determinado periodo de tiempo frente a la frecuencia total de sucesos, medida mediante experimentos aleatorios de los que se conocen todos los resultados posibles. Es por tanto la rama de las matemáticas que se encarga del estudio y la medición de procesos y experimentos aleatorios. Al haber desarrollado un algoritmo que se encarga precisamente de generar soluciones aleatorias, una de las mayores aplicaciones que

presenta el código es la relacionada con el campo de la probabilidad, que permitiría establecer con qué frecuencia se repiten determinadas soluciones dadas unas condiciones totalmente aleatorias.

En el campo de la informática estas dos ramas matemáticas se han desarrollado junto con la programación hasta dar lugar a lenguajes que permiten la combinación de las características de ambos.

Dentro de la programación lógica, existen varios lenguajes de programación que se engloban dentro de la programación lógica probabilística[10, 11, 12] (*probabilistic logic programming*). Estos lenguajes permiten dotar de pesos o probabilidades de ejecución a las cláusulas que forman el programa. Como ya se ha visto, el método resolutivo de la programación lógica consiste en la unificación de las variables por medio de las cláusulas especificadas en el programa. La elección de la cláusula con la que se pretende unificar viene dada, en programación lógica, únicamente por el orden en el que se encuentran escritas las cláusulas dentro del programa. Mediante la introducción de pesos para cada cláusula, la interpretación de estas depende de la probabilidad indicada favoreciendo la interpretación de unas cláusulas frente a otras.

Se hace uso de modelos estadísticos[13] (*statistical modeling*) para representar las muestras de datos generadas con carácter aleatorio. En el mundo de la informática, los modelos estadísticos son una clase de modelos matemáticos que buscan describir el conjunto de soluciones generadas por los programas de programación lógica probabilística y ayudar a realizar predicciones a partir de ellos.

El presente proyecto se puede entender como un caso particular de programación lógica probabilística orientado al lenguaje Prolog, que difiere de otros casos en que la probabilidad se encuentra uniformemente distribuida entre todas las cláusulas que comparten functor y aridad. Es decir, para 10 cláusulas con el mismo functor y la misma aridad la probabilidad de que se interprete cada una de ellas es de 0,1, mientras que si fueran dos la probabilidad sería de 0,5.

En futuras ampliaciones del proyecto se podría hacer uso de la programación probabilística para colocar pesos de manera iterativa a las cláusulas. De esta forma, la ejecución del programa comenzaría con el algoritmo diseñado (que dota a todas las cláusulas de niveles iguales de prioridad). A medida que la ejecución avanzase se podrían establecer distintos mecanismos que recalcularan el valor de estos pesos en función del número de veces que la cláusula ha sido interpretada. Así, si se observara que una cláusula tiene más tendencia a repetirse que otras, se podría disminuir su peso para equilibrarla frente al resto y obtener nuevos resultados con aleatoriedad condicionada.

Por último, se podrían introducir mejoras al algoritmo utilizando el campo de la programación probabilística junto con el del aprendizaje automático[14, 15] (*machine learning*) y la estadística. Para ello se establecería un mecanismo que recogiera datos sobre las soluciones que fueron bien recibidas por los usuarios frente a las que no y sobre ellos se realizaría un estudio estadístico. En función de estos resultados se

establecerían los pesos para las cláusulas, favoreciendo a aquellas que generan resultados con un mejor recibimiento.

Capítulo 3

DESARROLLO

3.1. DISEÑO DE UN ALGORITMO DE BÚSQUEDA ALEATORIA

Este algoritmo nace con el propósito de poder evaluar programas de un modo distinto a los árboles desarrollados por la búsqueda de soluciones en anchura o profundidad. El desarrollo en memoria de estos árboles de búsqueda se realiza siguiendo el orden en el que se encuentran especificadas las cláusulas en el programa. Dado que el objetivo principal del algoritmo consiste en alterar el orden en el que se forman estas ramas del árbol, el propósito que debe cumplir es el de evaluar las cláusulas en un orden distinto (y aleatorio) al que están escritas cada vez que se ejecute.

Cabe destacar, que el usuario será el encargado de evaluar qué programas se benefician del uso de este algoritmo y para cuáles la generación de soluciones mediante búsqueda aleatoria no resulta suficientemente optimizada. En el capítulo de pruebas, se indican diversos criterios que podrían ser usados para elegir el tipo de búsqueda adecuada para cada programa.

Debido a que el modo que se ha planteado para poder resolver el factor aleatorio en la generación de soluciones consiste en alterar el orden de interpretación para las cláusulas escritas en el programa, resulta crucial que este nuevo orden sea del todo impredecible y que no guarde relación alguna con el orden de interpretación que se estableció en anteriores soluciones de la *query*. Probabilísticamente, es posible que el orden de interpretación de las cláusulas y por tanto el orden de generación de las soluciones coincida con un orden generado previamente. Ya que el objetivo del algoritmo es el de encontrar soluciones totalmente aleatorias cada vez, no se han establecido mecanismos que controlen la no repetición de estos órdenes aleatorios. En posteriores ampliaciones del algoritmo se podrían añadir mecanismos que controlaran cuándo se han hallado todas las posibles soluciones para un camino particular del árbol, de forma que se evitase seguir explorando dicho camino. El establecimiento de este tipo de mecanismos podría evitar, además, la necesidad de usar un control de colisiones, ya que el programa sería capaz de comprobar por sí mismo cuándo habría terminado de explorar el árbol.

Para conseguir que las cláusulas se interpreten en orden aleatorio se ha creado un módulo de traducción que convierte el programa que se desea evaluar en predicados cl/3. Este predicado tiene como primer argumento la cabeza de la regla que se lee en el programa original y como tercer argumento el cuerpo de esta (el segundo argumento hace referencia a la traza y se ve en el siguiente apartado). El cuerpo de la regla se escribe en forma de lista de predicados [P1,P2,…Pn] donde n es el número de predicados que componen el cuerpo. En caso de que se trate de un hecho, es decir una

regla que solo tiene cabeza ya que su cuerpo se cumple siempre, el tercer argumento se representará como una lista vacía []. La transformación del programa que el usuario quiere utilizar con búsqueda aleatoria a un conjunto de predicados cl/3 permite que el algoritmo desarrollado pueda utilizar sus datos durante la ejecución.

Mediante esta forma de representación, cuando el usuario indica al algoritmo que quiere obtener una solución para una cláusula 'A', este obtiene todos los predicados cl/3 que se encuentran en su base de datos cuyo primer parámetro se unifique con A y los almacena dentro de una lista en el orden en el que se encuentran escritos.

A continuación, se toma esta lista ordenada y se hace uso de un predicado auxiliar que la redistribuye de forma aleatoria. Si bien este predicado no resulta ser de los más extensos del algoritmo, representa la clave para obtener el orden aleatorio. Su modus operandi consiste en elegir un miembro al azar de dentro de la lista y colocarlo al inicio de esta. A continuación, repite el proceso con el resto de elementos de la lista sin tener en cuenta el elemento que ya se ha recolocado hasta que todos los elementos toman su nueva posición. El factor que determina qué elemento de la lista se escoge cada vez viene dado por un número aleatorio de módulo igual a la longitud de la lista (lo cual permite que coincida con el índice de uno de los elementos y no se salga de rango). El predicado utilizado para obtener este número es random/3 que se encuentra desarrollado dentro de la librería random de Ciao Prolog[16].

La nueva lista resultante, ordenada de forma aleatoria, indica en qué orden se interpretarán las cláusulas durante este paso de la ejecución. Como se puede ver con la explicación previa, el uso del predicado random/3 asegura que el orden de colocación de los elementos de la lista sea siempre aleatorio.

Por último, cabe destacar que este orden no se mantiene para todas las llamadas al mismo predicado dentro de una misma ejecución: si en una misma ejecución se realizasen distintas unificaciones con el mismo predicado, el orden de interpretación de las cláusulas será aleatorio cada vez.

En resumidas cuentas, se realizará una búsqueda en profundidad pero alterando el orden de elección de las ramas del árbol. Esta alteración del orden de interpretación de los caminos del grafo puede dar lugar a la exploración de ramas infinitas. En el apartado siguiente se tratará qué métodos se han utilizado para controlar la aparición de estas ramas, de forma que no se produzca un fallo del algoritmo.

3.2. ESTABLECIMIENTO DE MEDIDAS PARA EVITAR LA REPETICIÓN DE SOLUCIONES

Para que el algoritmo funcione de forma correcta ya se ha mencionado que se hará uso de una forma particular de representar las cláusulas del programa a evaluar: cl/3. El segundo argumento de este predicado tiene una función clave para evitar que las soluciones ofrecidas al usuario se repitan. Se trata de un identificador único que permite reconocer de forma automática si una solución ya ha sido encontrada previamente mediante la generación de una traza. Es importante aclarar la diferencia entre repetición de valores en una solución y repetición de soluciones. Para esclarecer estos conceptos imaginemos un escenario en el que un cartero quiere llegar de un punto A a otro punto cualquiera. Las soluciones de este caso nos permitirían obtener los distintos lugares a los que puede llegar el cartero, pero si el cartero pudiera llegar de dos formas distintas al mismo punto y solo se tuviera en cuenta el valor de sus soluciones, el algoritmo concluiría indicando que las soluciones son menos de las que realmente existirían. De la misma forma, al recorrer un árbol de búsqueda debe tenerse en cuenta que si bien los valores de unificación de las variables pueden ser los mismos en distintas ramas, cada una de estas ramas representa una solución distinta.

Para representar la traza se indica el identificador de la cláusula elegida para la resolución de cada literal del árbol. Por ejemplo, se tiene el siguiente programa para representar caminos entre distintos puntos (representados con constantes numéricas):

```
:- module(_, _, [rndsearch]).

arc(1,2).
arc(2,3).
arc(2,4).
arc(3,4).
arc(3,1).

path(A,B):-
      arc(A,B).
path(A,B):-
      arc(A,X),
      path(X,B).
```

Cada una de las trazas obtenidas representa una solución única a partir de una *query*. Supongamos que la *query* realizada es la siguiente:

```
?- path(A,B).
```

La traza ‘c2(c2,c2(c5,c2(c1,c2(c2,c1(c5)))))’ representa las distintas elecciones que han tenido lugar desde la raíz del árbol.

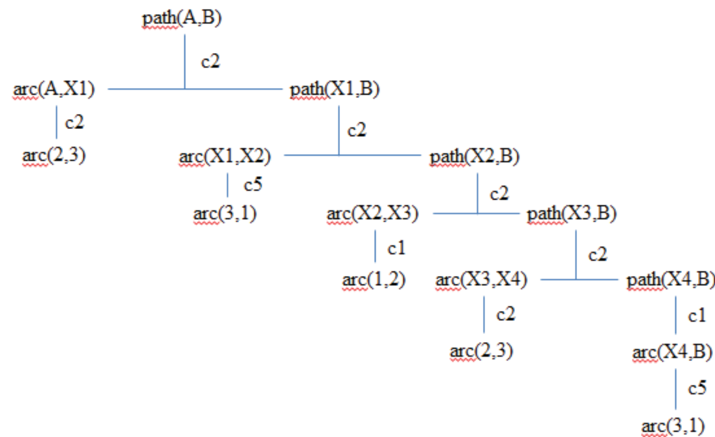


Ilustración 1- Ejemplo de traza

El primer identificador de la traza (c2) indica que se unificó el predicado `path(A,B)` con la cabeza de la segunda regla. Como esta regla tiene dos predicados dentro de su cuerpo, el paréntesis que se encuentra a continuación del identificador 'c2' consta de dos partes separadas por una coma: la primera parte indica la secuencia de unificaciones que se realizaron para `arc(A,X)`, mientras que la segunda indica lo propio para `path(X,B)`. Si se sigue la traza sustituyendo cada uno de los literales que aparecen por las cláusulas del programa se obtienen los valores de unificación A=2 y B=1.

Sin embargo, podemos ver que si se sigue la traza 'c2(c2,c1(c5))' también se llega a los mismos valores de unificación.

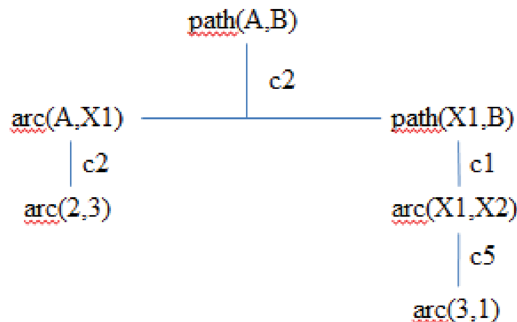


Ilustración 2- Traza distinta con mismos valores de unificación

El objetivo del presente proyecto se basaba en evitar la repetición de soluciones, no de sus valores de unificación, por lo que se ha hecho uso de un mecanismo de comparación de las distintas trazas generadas por cada una de las soluciones de forma que al usuario solo se le ofrezcan soluciones cuyas trazas no habían sido utilizadas previamente. En un origen se consideró que resultaría más sencillo almacenar las soluciones generadas, sin embargo esta idea se descartó ya que el uso de las trazas permite saber con mayor certeza a qué soluciones se refiere, mientras que el almacenamiento de los valores de unificación podría dar lugar a falsos

positivos. Es por ello que el algoritmo asegura la no repetición de soluciones mediante la identificación de sus caminos o ramas y no de sus valores de unificación.

Además de este mecanismo de control, se han establecido otros que permiten incrementar la utilidad del algoritmo para el usuario.

Una de las medidas de control establecidas permite elegir al usuario el número de colisiones consecutivas que está dispuesto a tolerar durante la ejecución del algoritmo. Se consideran como colisiones las ocasiones en las que el algoritmo finaliza su ejecución encontrando una solución (entendiendo por ello todo el camino del grafo y no solo sus valores de unificación finales) que ya había hallado previamente. Así, si se encuentran de forma sucesiva más soluciones repetidas de las que el usuario permite, el programa finalizará indicando antes el motivo de su finalización al usuario y recomendándole aumentar el número permitido de colisiones o bien cambiar de *query*.

Por otro lado, se ha desarrollado una funcionalidad extra que permite controlar al usuario la profundidad de su árbol de búsqueda. Es decir, si el usuario quiere hallar una solución para una *query* y quiere que la distancia de esta solución a la raíz del árbol sea menor a una profundidad determinada, el algoritmo lo tendrá en cuenta y en el momento en el que la búsqueda sobrepase esta profundidad la descartará e iniciará una nueva búsqueda desde la raíz. De este modo, se permite controlar que la búsqueda no entre por ramas infinitas. Siguiendo el mismo concepto, el usuario también puede establecer un control sobre el número de pasos que se dan antes de encontrar una solución válida. Como se puede ver en las ilustraciones previas de las trazas, ambos límites guardan estrecha relación con el tamaño del árbol generado para hallar la solución, y su control permite limitar las búsquedas a tamaños de árboles determinados de modo que no se generen árboles con mayor tamaño si no se desea.

Por último, se permite también al usuario elegir el número máximo de soluciones que quiere obtener, en caso de que la muestra de soluciones deba ser de una extensión precisa. Si el usuario no tiene preferencias sobre la cantidad de soluciones que desea obtener, el algoritmo también puede usarse de forma que genere nuevas soluciones aleatorias de forma continua hasta el infinito o hasta que estas se agoten.

3.3. CÓDIGO DEL ALGORITMO

Para el correcto uso de este paquete es necesario comprender antes los distintos tipos de búsqueda que el usuario puede realizar. Dentro del módulo del algoritmo se han definido cuatro estrategias distintas de búsqueda que permiten estrechar el cerco de las soluciones en función de los límites que el usuario escoja. Por un lado si el usuario quiere realizar una consulta y que esta se encuentre limitada únicamente por el número de colisiones que tolera elegirá 'rnd_unbounded'. Si además del número de colisiones

quiere limitar la profundidad del árbol y el número de pasos dados antes de encontrar la solución utilizará la estrategia 'rnd_size_limit'. Si en lugar de la profundidad lo que quiere controlar es el número de soluciones que se generan en total y el número límite de colisiones deberá indicar que desea usar la estrategia 'rnd_sol_limit'. Por último, la estrategia 'rnd_sol_size_limit' combina todas las anteriores, permitiendo controlar número total de soluciones, profundidad y número de pasos en el árbol y límite de colisiones.

Para indicar la estrategia y los límites que desea permitir, el usuario debe hacer uso del predicado set_random_params/0 desde el terminal. Si no indica el tipo de estrategia antes de realizar la *query*, el algoritmo utilizará por defecto la estrategia y los límites usados en la consulta anterior. En caso de no encontrarlos, el programa presentará al usuario un mensaje con el siguiente contenido:

```
?- parent(A,B).
Please, select the search parameters (use the
set_random_params/0 predicate)
no
```

Una vez ejecutado el predicado set_random_params/0, se presentará al usuario por pantalla una secuencia de mensajes que le pedirán que introduzca el tipo de estrategia y, en función de la estrategia elegida, los límites que desea.

```
?- set_random_params.
Please, select the strategy from the following list:
  rnd_unbounded
  rnd_sol_limit
  rnd_size_limit
  rnd_sol_size_limit

Strategy:
|: rnd_sol_size_limit.
Please, introduce the collision number (maximum number of
consecutive repeated solutions before the search is
aborted).
Collision limit:
|: 10.
Please, introduce the maximum number of different solutions
you wish to find.
Number of solutions:
|: 5.
Please, introduce the maximum depth of your search tree.
Depth:
|: 5.
Please, introduce the maximum derivation steps of your
search tree.
Max steps:
|: 7.
```

```
yes
```

En caso de que el usuario introduzca una estrategia que no coincida con una de las cuatro mencionadas, el programa mostrará por pantalla un mensaje de error y finalizará su ejecución.

```
?- set_random_params.
Please, select the strategy from the following list:
  rnd_unbounded
  rnd_sol_limit
  rnd_size_limit
  rnd_sol_size_limit

Strategy:
|: not_a_strategy.
Please, select a valid strategy.
no
```

Una vez introducido el tipo de estrategia deseada, el usuario podrá proceder a realizar una *query* tal y como la realizaría para otro tipo de búsqueda.

```
?- nat(X).
```

Como se explicó con anterioridad, para el correcto funcionamiento del código se necesita interpretar las cláusulas con el formato cl/3. Para ello, cuando se compila el paquete se hace partir el módulo de traducción que se encarga de pasar a este formato todas las cláusulas y además proporcionarles un identificador único para su traza. Estos identificadores deberán ser distintos para todos los miembros del conjunto de cláusulas cuya cabeza comparte functor y aridad. Sin embargo, si la cabeza de dos reglas no tiene el mismo functor podrán repetirse los identificadores.

Una vez realizada una consulta, el programa utiliza el predicado rnd_solve/3 para tratar de encontrar una solución. El primer argumento sirve para indicar el tipo de estrategia establecida, en función de la cual se realizarán distintos tipos de controles. El segundo argumento indica el predicado que se quiere resolver según la *query* introducida (en el caso del ejemplo el segundo argumento sería 'nat(X)'). Por último, el tercer argumento representa la traza de la solución encontrada, que se forma mediante los identificadores de los predicados utilizados. Este predicado representa el motor del algoritmo de búsqueda.

El cuerpo de rnd_solve/3 está formado por un gestor de fallos que permite indicar el motivo por el que finalizó la ejecución, un predicado que actualiza el estado y el predicado root_solve/3. Este último predicado recibe hereda los mismos argumentos que tenía rnd_solve/3 y es el encargado de buscar una solución aleatoria nueva desde la raíz del árbol. En caso de encontrarla se actualiza el estado del programa, mientras que

en caso de no encontrarla el gestor de fallos indica el motivo por el que no se encontró mediante el predicado show_status/0 y finaliza la ejecución.

Existen varios motivos por los que podría finalizarse la ejecución. Por una parte, si se realiza una *query* que no tiene solución el programa lo indica mediante el siguiente mensaje:

```
No solutions were found for this query.

no
```

En el caso de que el programa ya haya mostrado todas las soluciones existentes el mensaje mostrado es:

```
There are no more solutions for this query.

no
```

Si se han mostrado tantas soluciones como el usuario quería recibir se indica lo siguiente por pantalla:

```
The maximum number of solutions has been reached.

no
```

Si se ha superado el límite de colisiones permitidas por el usuario el mensaje mostrado es:

```
Number of collisions has been exceeded.

no
```

Si por el contrario el árbol de búsqueda supera la profundidad o el número de pasos permitidos, el algoritmo de búsqueda vuelve a empezar una nueva búsqueda desde la raíz pero no finaliza la ejecución, puesto que se considera que estos límites se utilizan como restricciones para la solución y no como controles de fallo.

En el caso de que el predicado root_solve/3 consiguiese encontrar una unificación válida (una solución aleatoria no repetida), la actualización del estado interno del programa se llevaría a cabo incluyendo la traza de esta solución al conjunto de trazas visitadas, con el fin de guardar un registro para no repetirlas.

El predicado root_solve/3, que como ya hemos visto recibe como argumentos la estrategia, el predicado a solucionar y la traza, es el encargado de encontrar una única solución aleatoria no repetida. Mientras que rnd_solve/3 tiene un mecanismo que consigue que cada vez que comienza una búsqueda ésta se haga desde la raíz del árbol, root_solve/3 utiliza el *backtracking* para ahorrar recursos.

El cuerpo de root_solve/3 se compone del predicado solve/5 y el predicado not_explored/1. El predicado solve/5 es el que se encarga de conseguir soluciones aleatorias tal y como se explicó al comienzo de este capítulo. Para ello, toma todas las cláusulas cuya cabeza coincide con el predicado que pretende resolver y las almacena en una lista. A continuación reorganiza esta lista utilizando el predicado random/3 de la librería random de Ciao Prolog. Este nuevo orden de organización es el que indica en qué orden deben ser interpretadas las cláusulas. Después de seleccionar la cláusula que se va a interpretar se realiza una búsqueda en profundidad hasta encontrar una solución a la *query*, reordenando durante la búsqueda el orden de interpretación de todas las cláusulas que se encuentren.

Por otra parte, not_explored/1 recibe como argumento la traza de la solución encontrada por solve/5 y comprueba que no se encuentre entre el conjunto de las soluciones ofrecidas previamente. Si se trata de una traza repetida incrementa el contador de colisiones y falla, provocando que solve/5 busque una nueva solución por *backtracking*. El uso de backtracking en este paso del algoritmo es posible ya que al haber reordenado las ramas del árbol en la llamada a solve/5, la solución generada será aleatoria. Además, su uso consigue que al buscar soluciones en ramas vecinas al nodo de la solución repetida se ahorre tiempo de búsqueda.

Cuando se encuentra una solución aleatoria que aún no ha sido visitada el programa la ofrece al usuario y es éste quien decide si continuar la búsqueda o no.

Una consideración a tener en cuenta es que el número de colisiones se contabiliza siguiendo el número de veces consecutivas que se encuentra una solución válida cuya traza ya se había encontrado antes, cambiándose a cero cuando se encuentra una nueva solución. Se podrían introducir mejoras en el algoritmo que contabilizaran el número de colisiones particular para cada traza, de forma que el algoritmo finalizase si se repitiera un determinado número de veces una misma traza.

Capítulo 4

PRUEBAS DE RENDIMIENTO

4.1. LA BÚSQUEDA ALEATORIA FRENTE A LA BÚSQUEDA EN ANCHURA

Como ya se ha hablado previamente existen distintos tipos de búsquedas en la programación lógica. Cuando se realiza una *query*, o una petición al programa de hallar la validez de un predicado se genera un árbol de búsqueda. La raíz de dicho árbol coincide con la *query* que se quiere solucionar y los nodos representan cada uno de los pasos que se dan para llegar a las soluciones, que se encuentran en las hojas del árbol. En cada una de estas hojas, todas las variables libres han sido unificadas con constantes para formar una solución válida, de forma que no quedan variables sin unificar.

Los distintos modos de recorrer un árbol de búsqueda en programación lógica coinciden con las formas de recorrer un grafo. De esta forma, dependiendo del modo de generar y recorrer este árbol y del orden en el que se generen las respuestas, se distinguen los tipos de búsqueda. Las dos más conocidas son la búsqueda en anchura y la búsqueda en profundidad.

A continuación, se describe detalladamente cómo gestiona Prolog la búsqueda en anchura[17]. Es necesario recordar siempre que el orden natural de interpretación de las cláusulas en Prolog es de arriba a abajo según estas se encuentren escritas en el programa, y el orden de unificación de variables es de izquierda a derecha. Además, cabe destacar que Prolog hace uso del ya mencionado *backtracking* para generar el grafo.

Ilustración 3- Árbol de búsqueda en anchura sin explorar

Imagínese un programa escrito en Prolog cuya ejecución genera el árbol de búsqueda de la imagen anterior. Los nodos sin color representan nodos del árbol que aún no han sido explorados, mientras que el color azul representa aquellos que ya se han visitado. Se ha hecho uso del color verde para representar soluciones válidas y el color rojo para representar soluciones que han sido descartadas por el usuario.

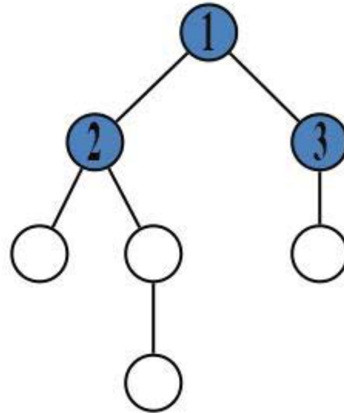


Ilustración 4- Primeros pasos de la búsqueda en anchura

Como podemos ver en la imagen, el orden de visitación de los nodos ha ido por niveles y de izquierda a derecha. Como en el primer nivel no consigue encontrar ninguna solución continúa la búsqueda en el siguiente nivel.

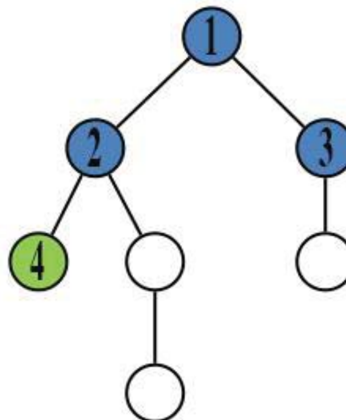


Ilustración 5- Primera solución en anchura

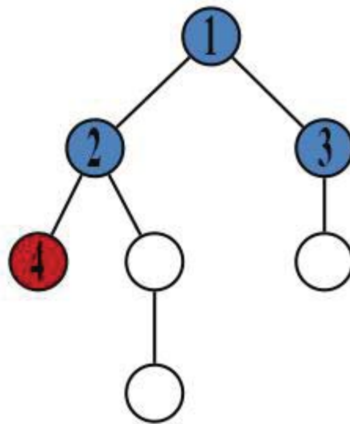


Ilustración 6- Primera solución en anchura rechazada

Al encontrar la primera solución se detiene y se la ofrece al usuario. Podemos ver cómo este ha rechazado la solución y solicita que se continúe recorriendo el árbol. Para continuar con la ejecución de la misma *query* es importante comprender que no se vuelve a generar el árbol desde la raíz, sino que se continúa con su exploración a partir del nodo en el que se encuentra. Dado que este nodo se trata de una hoja y por lo tanto no tiene nodos hijos, Prolog hace uso de la técnica de *backtracking* que le permite volver al último nodo que aun tenga hijos sin explorar.

Cabe destacar las diferencias entre seguir ejecutando una misma *query* o realizar una ejecución distinta de una *query* que ya se ha realizado previamente. Como ya se explicó con anterioridad, la primera hace referencia a continuar explorando un árbol que ya ha comenzado a generarse, mientras que la segunda por el contrario comienza la generación del árbol con cada nueva petición. En este caso podemos ver cómo se continúa con la exploración del mismo árbol, ya que el usuario quiere obtener nuevas soluciones para la misma *query*.

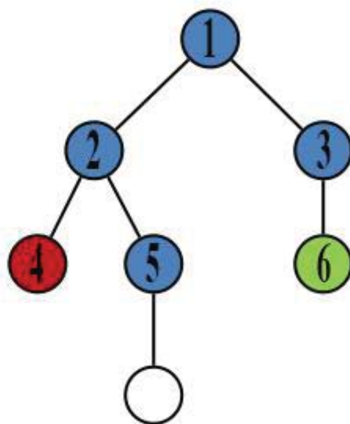


Ilustración 7- Segunda solución en anchura

Como se puede apreciar, dado que el quinto nodo visitado no era una solución, Prolog ha seguido explorando el resto de nodos del nivel hasta encontrar la segunda solución.

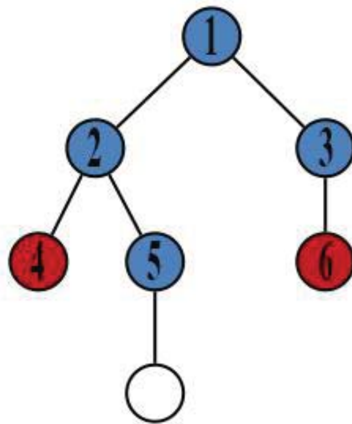


Ilustración 8- Segunda solución en anchura rechazada

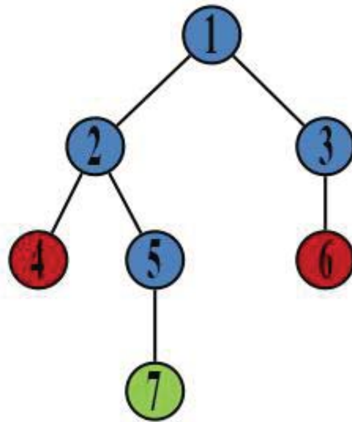


Ilustración 9- Fin del árbol de búsqueda por anchura

La exploración en anchura del árbol concluiría con la generación de la última solución, que se encuentra en el nivel más alejado de la raíz.

Este tipo de búsqueda se conoce como búsqueda en anchura (en inglés *breadth first search*) porque el árbol se desarrolla por niveles de profundidad, explotando toda la anchura de un nivel antes de pasar al siguiente. Es decir, comenzando por el nodo raíz, se exploran todos los nodos vecinos de este nodo. A continuación, para cada uno de los vecinos, se exploran sus respectivos nodos adyacentes hasta que se recorre todo el árbol.

En este tipo de búsqueda, el árbol se guarda en memoria según lo genera el programa, llegando a resultar exponencialmente muy costoso de mantener según aumenta el número de peticiones de distintas soluciones por parte del usuario.

4.2. LA BÚSQUEDA ALEATORIA FRENTE A LA BÚSQUEDA EN PROFUNDIDAD

La búsqueda en profundidad[18] (en inglés *depth first search*) es un modo de explorar los nodos de los árboles de búsqueda que, en cambio, establece un recorrido ordenado pero no uniforme. Su modo de actuación consiste en explorar siempre los nodos que va localizando siguiendo un camino concreto y cuando ya no quedan más nodos por expandir en dicho camino volver atrás (en Prolog esto se consigue mediante la técnica de *backtracking*) para seguir explorando los hermanos del nodo ya procesado. En Prolog, el camino que sigue la búsqueda en profundidad consiste en explorar siempre el nodo que se encuentra más a la izquierda en el árbol de búsqueda.

Los grafos creados mediante este tipo de búsqueda también cuentan con la *query* en su raíz, los distintos pasos de unificación en sus nodos intermedios y las soluciones sin variables libres en las hojas.

A continuación, se muestra un ejemplo ilustrativo de la búsqueda en profundidad usada por Prolog. Como para la búsqueda en anchura, el orden de interpretación de las cláusulas es de arriba a abajo según estas se encuentren escritas en el programa, el orden de unificación de variables es de izquierda a derecha y se hace uso de *backtracking*.

Partiendo del mismo grafo que en el ejemplo anterior, se encuentra la primera solución al explorar hasta el final la rama izquierda del árbol de búsqueda.

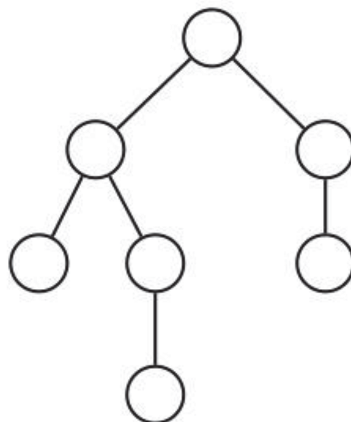


Ilustración 10- Árbol de búsqueda por profundidad sin explorar

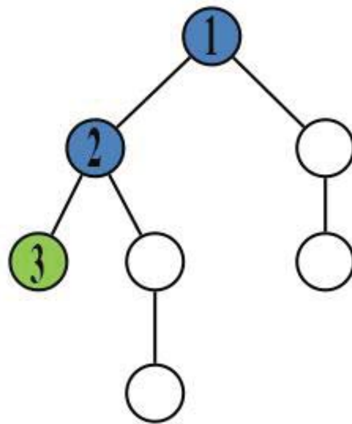


Ilustración 11- Primera solución por profundidad

En el caso de que el usuario quisiera obtener nuevas soluciones, Prolog vuelve usando *backtracking* al último nodo con hijos por explorar y continúa recorriendo el árbol a partir de ahí.

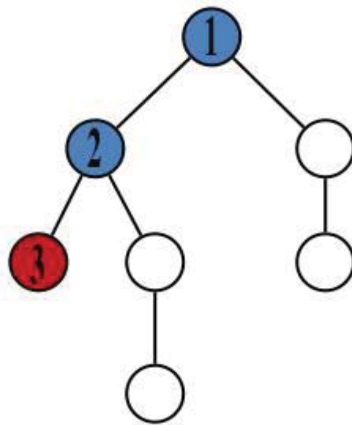


Ilustración 12- Primera solución por profundidad rechazada

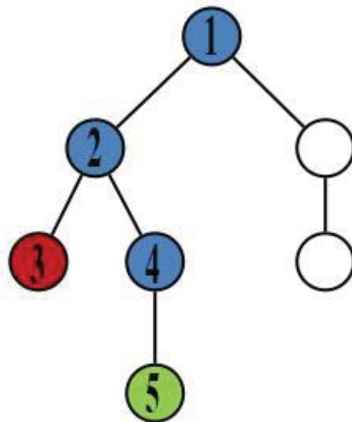


Ilustración 13- Segunda solución por profundidad

Tras rechazar la segunda solución, el programa finaliza la exploración del árbol llegando a la última solución.

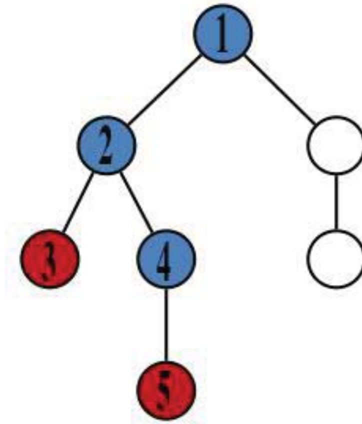


Ilustración 14- Segunda solución por profundidad rechazada

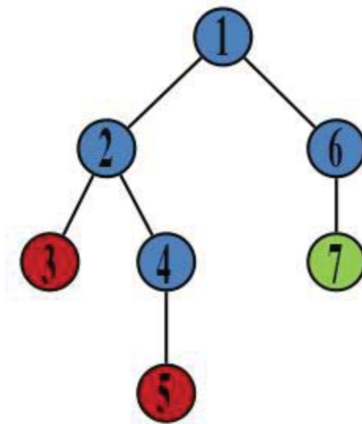


Ilustración 15- Fin del árbol de búsqueda por profundidad

Mientras que en la búsqueda en anchura la primera solución ofrecida es la que se encuentra en el nivel más superficial (si hay varias en este nivel será la que se ubique más a la izquierda) y la última es siempre la del nivel más profundo (si hay varias la de la derecha), en la búsqueda en profundidad[1] la solución primera será la de la rama que se encuentre más a la izquierda y la última la de la derecha.

Este tipo de búsqueda, como la búsqueda en anchura, resulta costosa por la necesidad de mantener el árbol en memoria a medida que este se va generando.

[1] Nótese que en Ciao Prolog, la forma de distinguir entre búsqueda en anchura y profundidad es mediante el functor que une la cabeza y el cuerpo en las reglas. En el caso de cualquier tipo de búsqueda distinta a la búsqueda por defecto se representa con ‘<-’. En el caso de la búsqueda en profundidad, que es la búsqueda por defecto en Ciao, este functor se sustituye por ‘:-’

4.3. RESULTADOS DE LOS CASOS DE PRUEBAS

Para ilustrar el funcionamiento del algoritmo explicado en el anterior capítulo, se muestran a continuación los distintos programas que se han utilizado para probarlo, así como ejemplos de las *queries* realizadas y sus soluciones. Para simplificar la lectura, se indica al comienzo de cada *query* qué estrategia se ha elegido y los valores de los límites introducidos por el usuario mediante el predicado set_random_params/0.

EJEMPLO 1:

El primer programa que se ha evaluado como ejemplo consiste en la representación de los caminos entre los nodos de un grafo. Se han representado los nodos con constantes numéricas y los arcos entre estos con el predicado arc/2. Se ha definido también el predicado path/2 que permite evaluar si existen caminos entre dos nodos. La primera línea del programa indica qué módulos se utilizan en su ejecución. En el tercer argumento se importan los paquetes de traducción para el programa. Como se puede ver, en este caso se utiliza el paquete rndseach en el que se encuentran todos los ficheros de código con el algoritmo desarrollado. De esta forma, se indica al programa que debe ejecutarse con búsqueda aleatoria, y no búsqueda en profundidad.

```
:- module(_, _, [rndsearch]).

arc(1,2).
arc(2,3).
arc(2,4).
arc(3,4).
arc(3,1).

path(A,B)<-
      arc(A,B).
path(A,B)<-
      arc(A,X),
      path(X,B).
```

A continuación se puede ver un ejemplo de las *queries* que se pueden realizar con este programa y las soluciones que ha obtenido.

```
STRATEGY: rnd_sol_size_limit.
COLLISION LIMIT: 5.
NUMBER OF SOLUTIONS: 10.
DEPTH: 10.
STEPS: 10.

?- path(A,B).
trace: c2(c2,c2(c5,c2(c1,c2(c2,c1(c5)))))
A = 2,
B = 1 ? n
trace: c2(c2,c1(c4))
```

```
A = 2,
B = 4 ? n
trace: c1(c5)
A = 3,
B = 1 ? n
trace: c1(c1)
 A = 1,
B = 2 ? n
trace: c1(c4)
A = 3,
B = 4 ? n
trace: c1(c3)
 A = 2,
B = 4 ? n
trace: c2(c5,c1(c1))
 A = 3,
B = 2 ? n
trace: c1(c2)
 A = 2,
B = 3 ? n
trace: c2(c1,c1(c2))
 A = 1,
B = 3 ? n
trace: c2(c5,c2(c1,c2(c2,c1(c4))))
 A = 3,
B = 4 ? n
The maximum number of solutions has been reached.
Program finished

no
```

En las soluciones pueden distinguirse las aportaciones del usuario en letra cursiva (se han editado para que puedan diferenciarse de las salidas del programa). Cuando el programa encuentra una solución indica al usuario los valores de unificación con los que la ha conseguido así como la traza que identifica dicha solución. El operador '?' después de la solución da la opción al usuario de indicar si quiere continuar con la búsqueda o está satisfecho con la solución obtenida. En los ejemplos de esta memoria se ha utilizado la letra 'n' para continuar con la búsqueda y la letra 'y' para detenerla.

En el ejemplo, el mensaje final indica que la búsqueda ha sido interrumpida ya que el número de soluciones que el usuario quería obtener se ha alcanzado (diez soluciones distintas). Si esta misma *query* se realiza con búsqueda en anchura o búsqueda en profundidad las soluciones se obtendrían de forma ordenada (de forma que sus trazas seguirían el orden c1, c2, c3,…), sin embargo se observa que el orden de las trazas del ejemplo no sigue ningún patrón.

Además, si se observan detenidamente las soluciones podrá verse que algunos valores de unificación se repiten (por ejemplo A=2 y B=4). Como ya se indicó previamente, el

hecho de que los valores se repitan no indica que sea una solución repetida, ya que lo que identifica cada solución es su traza. Siendo así, si nos fijamos en las trazas del ejemplo podremos ver que no hay ninguna repetida.

EJEMPLO 2:

El siguiente programa contiene tres predicados distintos para conseguir números naturales a partir del 0 (para este ejemplo se ha considerado el 0 como parte del conjunto de números naturales). Cualquiera de los tres predicados ejecutados con las búsquedas clásicas obtendría sus resultados ordenados del 0 al infinito.

```
:- module(_, _, [rndsearch, clpq]).

% Peano encoding
nat(0) <- .
nat(s(X)) <- nat(X).

% Integer encoding (using is/2)
inat(0) <- .
inat(N) <- inat(N0), N is N0 + 1.

% CLP(Q) encoding (note that inc/2 can appear before
qnat/1)
qnat(0) <- .
qnat(N) <- inc(N0, N), qnat(N0).

inc(N0, N) :- N .=. N0 + 1.
```

Un ejemplo del uso del primer predicado es el siguiente:

```
STRATEGY: rnd_size_limit.
COLLISION LIMIT: 9.
DEPTH: 6.
STEPS: 10.

?- nat(X).
trace: c2(c1)

X = s(0) ? n
trace: c2(c2(c1))

X = s(s(0)) ? n
trace: c2(c2(c2(c2(c1))))

X = s(s(s(s(0)))) ? n
trace: c2(c2(c2(c2(c2(c1)))))

X = s(s(s(s(s(0))))) ? n
```

```
trace: c2(c2(c2(c1)))

X = s(s(s(0))) ? n
trace: c1

X = 0 ? n
There are no more solutions for this query.

no
```

La notación que emplea para representar estos números está basada en los axiomas de Peano[19], considerando la primera solución el 0 y representando los siguientes números como sucesores de este. Por ejemplo, la representación del 1 es s(0), mientras que el 4 se representa como s(s(s(s(0)))).

Como se ha indicado que la profundidad máxima del árbol debe ser 6, solo se consideran soluciones válidas de este árbol los números del 0 al 5. El número 5 es el mayor número natural que se puede obtener con el predicado nat/1 ya que la traza de la solución X=s(s(s(s(s(0))))) es trace:c2(c2(c2(c2(c2(c1))))) y tiene profundidad 6.

Puede verse que el programa no ha finalizado por haber sobrepasado la profundidad máxima, sino que ha restringido la búsqueda en función de esta. Una vez terminadas las soluciones del árbol lo indica al usuario y finaliza.

Si se realizara una *query* con los mismos límites que no tuviera ninguna solución posible el programa en lugar de indicar que no hay más soluciones para la query indicaría que estas no existen:

```
?- nat(no_number).
No solutions were found for this query.

no
```

Por otra parte el segundo y tercer predicado unifican con números árabes, no con números de Peano. Para ello se han utilizado algunos predicados *built-in* de Ciao Prolog como el predicado is/2. Estos predicados no necesitan ser traducidos para que el algoritmo los interprete y por tanto no tienen un identificador que sea incluido en la traza.

```
?- inat(X).
trace: c2(c2(c2(c2(c2(c1)))))

X = 5 ? n
trace: c2(c2(c2(c2(c1))))

X = 4 ? n
trace: c2(c2(c1))
```

```
X = 2 ? n
trace: c2(c1)

X = 1 ? n
trace: c1

X = 0 ? n
trace: c2(c2(c2(c1)))

X = 3 ? n
There are no more solutions for this query.

no

?- qnat(X).
trace: c2(c2(c1))

X = 2 ? n
trace: c2(c2(c2(c2(c1))))

X = 4 ? n
trace: c2(c2(c2(c2(c2(c1)))))

X = 5 ? n
trace: c1

X = 0 ? n
trace: c2(c2(c2(c1)))

X = 3 ? n
trace: c2(c1)

X = 1 ? n
There are no more solutions for this query.

no
```

EJEMPLO 3:

Este programa representa algunos de los tipos de relaciones presentes en un árbol de familia. Las relaciones entre progenitor e hijo se encuentran instanciadas en forma de hecho, mientras que el resto de relaciones se definen mediante reglas a partir de estas.

```
:- module(_, _, [rndsearch]).

%parent/2
parent(mary,edward).
parent(jhon,edward).
parent(hanna,beth).
parent(kevin,beth).
```

```
parent(edward,james).
parent(edward,arthur).
parent(edward,nicole).
parent(beth,james).
parent(beth,arthur).
parent(beth,nicole).

%grandparent/2
grandparent(A,B)<-
      parent(A,X),
      parent(X,B).

%son/2
son(A,B)<-
      parent(B,A).

%grandchild/2
grandchild(A,B)<-
      grandparent(B,A).

%sibling/2
sibling(A,B)<-
      parent(X,A),
      parent(X,B).
```

Si el usuario limita a 0 el número de colisiones permitidas indica al programa que una vez que se encuentre una solución repetida deberá finalizar su ejecución. Para comprobar el mensaje obtenido cuando se produce este caso se ha realizado la siguiente consulta:

```
STRATEGY: rnd_unbounded.
COLLISION LIMIT: 0.

?- grandparent(A,B).
trace: c1(c4,c9)

A = kevin,
B = arthur ? n
trace: c1(c1,c5)

A = mary,
B = james ? n
trace: c1(c2,c6)

A = jhon,
B = arthur ? n
trace: c1(c3,c10)

A = hanna,
B = nicole ? n
```

```
trace: c1(c2,c5)

A = jhon,
B = james ? n
trace: c1(c1,c7)

A = mary,
B = nicole ? n
Number of collisions has been exceeded.

no
```

Para demostrar que el resultado obtenido por las *queries* es realmente aleatorio, se muestra otra solución obtenida para la misma *query* anterior. Se puede ver que aun repitiendo la consulta y los límites establecidos el orden de soluciones cambia y además en esta ocasión no se produce fallo.

```
?- grandparent(A,B).
trace: c1(c1,c5)

A = mary,
B = james ? n
trace: c1(c2,c5)

A = jhon,
B = james ? n
trace: c1(c3,c9)

A = hanna,
B = arthur ? n
trace: c1(c3,c8)

A = hanna,
B = james ? n
trace: c1(c3,c10)

A = hanna,
B = nicole ? n
trace: c1(c4,c8)

A = kevin,
B = james ? n
trace: c1(c2,c6)

A = jhon,
B = arthur ? n
trace: c1(c2,c7)

A = jhon,
B = nicole ? n
```

```
trace: c1(c1,c7)

A = mary,
B = nicole ? n
trace: c1(c4,c9)

A = kevin,
B = arthur ? n
trace: c1(c1,c6)

A = mary,
B = arthur ? n
trace: c1(c4,c10)

A = kevin,
B = nicole ? n
There are no more solutions for this query.

no
```

EJEMPLO 4:

El siguiente programa consiste en un generador de árboles binarios.

```
:- module(_, _, [rndsearch]).

tree(nil).
tree(node(Left,Right))<-
     tree(Left),
     tree(Right).
```

Se ha realizado la siguiente *query* sin limitar profundidad para que la aleatoriedad del algoritmo sea apreciable.

```
STRATEGY: rnd_unbounded.
COLLISION LIMIT: 100.

?- tree(X).
trace: c1

X = nil ? n
trace:
c2(c2(c1,c1),c2(c2(c2(c2(c1,c2(c1,c1)),c1),c1),c2(c1,c2(c1,
c1))))

X =
node(node(nil,nil),node(node(node(node(nil,node(nil,nil)),n
il),nil),node(nil,node(nil,nil)))) ? n
trace: c2(c1,c1)

X = node(nil,nil) ? n
```

```
trace:
c2(c2(c1,c2(c2(c2(c2(c2(c2(c2(c1,c2(c1,c2(c2(c2(c1,c1),c2(c
1,c1)),c2(c2(c1,c2(c2(c2(c1,c2(c1,c2(c1,c1))),c2(c1,c1)),c1
)),c2(c1,c2(c2(c2(c2(c2(c2(c1,c2(c1,c1)),c1),c1),c2(c2(c2(c
1,c1),c1),c2(c1,c1))),c1),c1)))))),c1),c2(c2(c2(c2(c2(c1,c1
),c2(c1,c1)),c2(c1,c1)),c1),c2(c2(c2(c1,c1),c2(c1,c1)),c1))
),c2(c2(c1,c2(c2(c2(c2(c2(c2(c1,c1),c1),c1),c2(c2(c1,c2(c1,
c2(c1,c2(c1,c2(c1,c1))))),c2(c2(c1,c1),c1))),c2(c1,c2(c1,c1
))),c1)),c1)),c2(c1,c1)),c1),c1)),c1)

X =
node(node(nil,node(node(node(node(node(node(node(nil,node(n
il,node(node(node(nil,nil),node(nil,nil)),node(node(nil,nod
e(node(node(nil,node(nil,node(nil,nil))),node(nil,nil)),nil
)),node(nil,node(node(node(node(node(node(nil,node(nil,nil)
),nil),nil),node(node(node(nil,nil),nil),node(nil,nil))),ni
l),nil)))))),nil),node(node(node(node(node(nil,nil),node(ni
l,nil)),node(nil,nil)),nil),node(node(node(nil,nil),node(ni
l,nil)),nil))),node(node(nil,node(node(node(node(node(node(
nil,nil),nil),nil),node(node(nil,node(nil,node(nil,node(nil
,node(nil,nil))))),node(node(nil,nil),nil))),node(nil,node(
nil,nil))),nil)),nil)),node(nil,nil)),nil),nil)),nil) ?

yes
```

A continuación se pueden ver los resultados de repetir la *query* disminuyendo los límites de profundidad y pasos. Se puede ver que los árboles obtenidos tienen un tamaño mucho menor respecto a los obtenidos en el ejemplo anterior.

```
STRATEGY: rnd_size_limit.
COLLISION LIMIT: 100.
DEPTH: 5.
STEPS: 9.

?- tree(X).
trace: c1

X = nil ? n
trace: c2(c2(c1,c1),c2(c2(c1,c1),c1))

X = node(node(nil,nil),node(node(nil,nil),nil)) ? n
trace: c2(c2(c2(c1,c2(c1,c1)),c1),c1)

X = node(node(node(nil,node(nil,nil)),nil),nil) ? n
trace: c2(c1,c1)

X = node(nil,nil) ? n
trace: c2(c1,c2(c2(c2(c1,c1),c1),c1))

X = node(nil,node(node(node(nil,nil),nil),nil)) ? n
```

```
trace: c2(c1,c2(c2(c1,c2(c1,c1)),c1))

X = node(nil,node(node(nil,node(nil,nil)),nil)) ? n
trace: c2(c2(c2(c1,c1),c2(c1,c1)),c1)

X = node(node(node(nil,nil),node(nil,nil)),nil) ? n
trace: c2(c2(c2(c1,c1),c1),c1)

X = node(node(node(nil,nil),nil),nil) ? n
trace: c2(c1,c2(c2(c1,c1),c2(c1,c1)))

X = node(nil,node(node(nil,nil),node(nil,nil))) ?

yes
```

EJEMPLO 5:

Por último, se tiene un programa que define un lenguaje a partir de dos variables (x e y) y distintas operaciones bit a bit (xor, and, < y -). El predicado expr/1 determina las expresiones que son correctas para dicho lenguaje. Por otra parte se ha añadido el predicado good_expr/2 que se encarga de generar expresiones aleatorias en este lenguaje y posteriormente unificarlas con otra que se pasa como argumento. De esta forma, si en el segundo argumento de good_expr/2 se pasa una expresión concreta, el algoritmo dará con ella.

```
:- module(_, _, [rndsearch]).

expr(x) <- . % variable x
expr(y) <- . % variable y
expr(X#Y) <- expr(X), expr(Y). % xor
expr(X/\Y) <- expr(X), expr(Y). % bitwise and
expr(X<Y) <- expr(X), expr(Y). % bitwise and
expr(-X) <- expr(X). % unary minus

% Get solution, then unifies
good_expr(X, Y) <- expr(X), X = Y.
```

Para probar su funcionamiento se ha ejecutado la siguiente consulta. El segundo argumento introducido en la *query* es una expresión que computa el mínimo entre dos números sin utilizar ninguna rama if-then-else:

```
STRATEGY: rnd_sol_size_limit.
COLLISION LIMIT: 10.
NUMBER OF SOLUTIONS: 10.
DEPTH: 6.
STEPS: 13.

?- good_expr(X, (y # ((x # y) /\ (-(x < y))))).
trace: c1(c3(c2,c4(c3(c1,c2),c6(c5(c1,c2)))))
```

```
X = y#(x#y/\ - (x<y)) ?

yes
```

Tal y como se ve en el ejemplo, al introducir una expresión válida en el segundo argumento de good_expr/2, el algoritmo ha podido encontrarla. Realizando esta misma consulta utilizando una búsqueda en anchura (*breadth first search*), la búsqueda incrementa su tiempo de ejecución notablemente. Como en búsqueda aleatoria podemos limitar el número de pasos previos a una solución y en este caso lo sabemos de antemano, esta restricción limita el árbol de búsqueda generado por el algoritmo. Por otro lado el grafo generado por la búsqueda en anchura resulta mucho más exhaustivo provocando que tarde más en encontrar la solución. Este caso, por tanto, es un ejemplo para el que sería aconsejable utilizar el algoritmo desarrollado frente a búsquedas clásicas.

De los ejemplos anteriores se puede comprobar que el algoritmo funciona correctamente de manera aleatoria y que sus soluciones no se repiten. También se puede observar en las distintas consultas realizadas los distintos modos de gestionar los límites establecidos por el usuario.

4.4. CASOS PRÁCTICOS DE USO

Debido a su diseño, que incluye un módulo de traducción que permite al algoritmo interpretar las cláusulas como necesita, cualquier programa escrito en lenguaje Prolog puede ejecutarse haciendo uso de la búsqueda aleatoria.

En el apartado anterior se han mostrado los resultados de utilizar este tipo de búsqueda para encontrar soluciones en programas sencillos ya que resultaban ejemplos muy ilustrativos para la comprensión del algoritmo. Sin embargo, a pesar de tratarse de ejemplos adecuados para la explicación de su funcionamiento, no resultan lo suficientemente representativos en cuanto a sus posibles usos y aplicaciones.

Además de aplicaciones más formales, este algoritmo permite la integración sencilla de búsquedas aleatorias en otros tipos de aplicaciones que se pueden modelar fácilmente como programas lógicos. Una de las posibles aplicaciones del algoritmo sería la creación de una aplicación que generase menús equilibrados y variados para los usuarios. De este modo, cada usuario podría elegir si restringir las opciones de su búsqueda (establecer un límite de calorías para personas con dieta hipocalórica, eliminar determinados componentes de los alimentos para personas celiacas o alérgicas, dietas hiperprotéicas para deportistas, etc.) y obtendría un menú aleatorio siguiendo las recomendaciones de salud de nutricionistas.

Por otra parte, se podría integrar el uso del algoritmo a una plataforma de Internet de forma que se formasen paquetes de viajes aleatorios en función de variables establecidas por el usuario (límite de precios, rango de fechas, etc.).

Además, el algoritmo se puede emplear con fines estadísticos, ya que como se ha visto se pueden hacer estudios sobre la distribución de sus soluciones y la probabilidad de obtener determinadas soluciones variando las entradas del programa.

4.5. COMPARATIVA DEL ALGORITMO CREADO FRENTE A BÚSQUEDAS CLÁSICAS

A pesar de que se han mencionado posibles usos de la búsqueda aleatoria, existen casos en los que el uso de este algoritmo para la obtención de soluciones sería incorrecto o poco útil.

Por ejemplo, la programación lógica juega un papel importante en la creación de programas que tratan de encontrar las soluciones optimizadas para distintos escenarios. En este tipo de escenarios, generalmente, se trata de llegar a un estado final (solución) en el menor número de movimientos posibles partiendo de un estado o condiciones iniciales y teniendo en cuenta una serie de restricciones.

En el caso de las torres de Hanoi[20], se tiene un rompecabezas con un escenario en el que se encuentra un número de discos de radio creciente que se apilan insertándose en una de las tres estacas del tablero. El objetivo del juego es crear la pila en otra de las estacas teniendo en cuenta que nunca se podrá colocar un disco sobre otro de menor tamaño y que todos los discos deben encontrarse en una de las tres estacas.

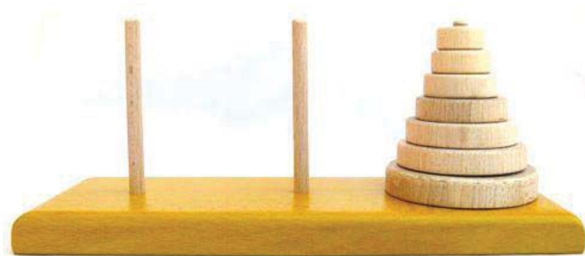


Ilustración 16- Torres de Hanoi

Otro problema ampliamente discutido en los libros de texto es el del barquero que debe cruzar el río con una oveja, un lobo y una rana (en algunas fuentes aparecen variantes como un zorro en lugar del lobo). En este caso, se debe conseguir que los cuatro

elementos crucen de la orilla en la que se encuentran al comienzo del juego a la orilla contraria, en el menor número de movimientos y teniendo en cuenta que en la orilla no pueden quedarse juntos la oveja y la rana o el lobo y la oveja si no es en presencia del barquero. Además, la barca solo puede pasar de una orilla a otra dirigida por su dueño y solo puede llevar una carga además de él (también puede ir sin carga).

Ilustración 17- Problema del lobo, la oveja y la rana

En cualquiera de los dos ejemplos anteriores el objetivo es llegar al estado final en el menor número de movimientos posible. En este tipo de problemas donde se persigue encontrar una solución óptima y existen heurísticas conocidas, no se ha llegado a conclusiones claras de que la alternativa ofrecida por el algoritmo que se ha desarrollado sea beneficiosa. El uso de una búsqueda aleatoria podría interactuar con las optimizaciones y heurísticas del programa y hacer que se explorasen caminos con un mayor número de nodos en el grafo, innecesariamente. De forma ilustrativa, sería como permitir al barquero del segundo ejemplo realizar tantos viajes como quisiera con su barca, de una orilla a otra, aun no llevando carga alguna (podría realizar cientos de viajes antes de encontrar la solución final).

Por otra parte, existen casos en los que el uso de este tipo de búsqueda frente a la búsqueda en profundidad o en anchura no solo generaría soluciones poco prácticas, sino que podría generar soluciones erróneas o no generarlas al encontrar bucles infinitos dentro del código.

Tal y como se habló en el capítulo de introducción, en el entorno de la programación lógica existen ciertos mecanismos que permiten controlar el orden de ejecución de los predicados, entre los cuales se encuentran los cortes. Existen dos tipos de cortes[8, 9]: cortes verdes y cortes rojos. Los cortes verdes permiten simplificar el árbol de búsqueda sin modificar el significado del programa. Por otro lado, se habla de que un corte es rojo cuando a diferencia del corte verde sí que afecta a la semántica declarativa.

Por ejemplo, se tiene un programa para encontrar el menor de dos números naturales:

```
min(X,Y,Z):- X>=Y, Z=Y.
```

```
min(X,Y,Z):-Y>X, Z=X.
```

Para el uso de este programa con números naturales, es sencillo comprobar que la ejecución solo va a poder concluir por una de las dos ramas, puesto que un número no puede ser a la vez mayor y menor que otro.

La declaración anterior es fácil de deducir para una persona con conocimientos matemáticos, pero Prolog no es capaz de predecir qué predicados van a producir fallo antes de probarlos. Por ello, aunque un par de números ya hayan unificado con la primera regla, al continuar su ejecución Prolog comprobará si es posible que unifiquen con la segunda.

Para evitar la ejecución de ramas del árbol de búsqueda que ya se sabe que provocarán fallo, se puede modificar el programa previo de la siguiente forma:

```
min(X,Y,Z):- X>=Y, !, Z=Y.

min(X,Y,Z):-Y>X, Z=X.
```

El corte introducido en la primera regla es un corte verde, puesto que no cambia el significado del programa y solo "poda" algunas ramas del árbol de búsqueda. El programa ofrece las mismas soluciones que si el corte no existiera, aunque su búsqueda resulta más eficiente.

Sin embargo, se podría realizar una última modificación al programa. Dado que ya se ha dicho que X>Y e Y>X son mutuamente excluyentes, podría decidir omitirse la condición de la segunda regla. Como existe un corte detrás de la condición 'X>=Y' en la primera regla, está claro que si llega a la segunda regla es porque dicha condición no se satisfizo y por tanto resultaría innecesario comprobarlo.

```
min(X,Y,Z):- X>=Y, !, Z=Y.

min(X,Y,Z):- Z=X.
```

En este caso, el corte que muestra el programa es un corte rojo puesto que cambia su significado. Puede apreciarse que la presencia de este corte resulta necesaria para el último programa ya que si se eliminase se ofrecerían resultados incorrectos.

Como puede deducirse del ejemplo anterior, si se modificase el orden de exploración de las ramas del árbol, podría suceder que se interpretara la segunda regla antes que la primera, de forma que el corte rojo se omitiría. De esta forma, se generarían soluciones incorrectas. Asimismo, otros programas hacen uso de cortes rojos para controlar iteraciones en bucles, por lo que en algunos casos su omisión implicaría la aparición de bucles infinitos.

Es cierto que en el desarrollo del proyecto se han establecido controles para interrumpir la ejecución de bucles infinitos: en las búsquedas que ofrecen las estrategias rnd_size_limit y rnd_sol_size_limit se permite al usuario decidir la profundidad máxima del árbol de búsqueda y el número máximo de pasos que se permitirán antes de

encontrar una solución. Sin embargo, supone un uso incorrecto el uso de estos predicados para controlar las ejecuciones que deberían manejarse con un corte rojo.

Por otro lado, el uso de la búsqueda aleatoria frente a los dos tipos de búsqueda más generalizados sí que sería recomendable en algunos casos de comprobación de la validez de ciertas soluciones. Por ejemplo, si de un programa queremos realizar una *query* en la que sabemos que la solución que queremos obtener se encuentra en el último nodo del árbol de búsqueda, convendría utilizar el algoritmo de búsqueda aleatoria para incrementar la probabilidad de adelantar su resolución. También se debería usar este tipo de búsqueda en escenarios en los que el orden de las soluciones no tiene ningún valor o incluso resulta conveniente variarlo (por ejemplo en programas que buscan ofrecer soluciones variadas a sus usuarios).

Como para las elecciones entre búsqueda en anchura y profundidad, queda a criterio del usuario decidir en qué casos conviene usar un tipo de búsqueda frente a las otras.

Capítulo 5

CONCLUSIONES

El desarrollo de la búsqueda aleatoria consigue completar distintos vacíos presentes en las formas clásicas de recorrer los grafos. Este tipo de búsqueda permite acercar la programación lógica al estudio estadístico y a entornos en los que el orden de obtención de las soluciones no solo es irrelevante, sino que es recomendable que sea inexistente.

La idea original fue la creación de un algoritmo que permitiese, dado un programa escrito en Prolog, obtener soluciones válidas aleatorias del espacio de soluciones teniendo en cuenta que estas no debían repetirse dentro de una misma ejecución de una *query*. Para ello, además de un programa que alterase el orden de interpretación de las cláusulas especificadas, era necesario introducir mecanismos de control que controlaran la unicidad de las trazas de estas soluciones.

Actualmente, se ha logrado desarrollar un algoritmo completo que cumple con todos los objetivos planteados en el comienzo del proyecto. La aleatoriedad de las soluciones se ha conseguido mediante la alteración del orden de las cláusulas cada vez que estas deben ejecutarse, lo cual asegura que el orden de las soluciones sea totalmente impredecible. Por otro lado, aunque en un comienzo se barajó la posibilidad de guardar en memoria las soluciones previas para poder confirmar que las nuevas soluciones no hubieran sido generadas anteriormente, se descartó la idea y se sustituyó dicho planteamiento por la conservación en memoria de las trazas de las soluciones. El motivo de este cambio se debió al ahorro que suponía para la memoria y a que se pueden dar casos en los que distintas hojas del árbol (distintas soluciones) tengan los mismos valores para las variables libres pero distintos caminos para llegar al nodo final (distintas cláusulas utilizadas), por lo que el mecanismo de la idea original las identificaría como soluciones idénticas aunque no lo fuesen.

Además, se han establecido para mayor utilidad del usuario, diversos controles que permiten enfocar la búsqueda. Así, el algoritmo resultante permite controlar variables como la profundidad máxima que tendrá el árbol de búsqueda, el número máximo de pasos que se podrán dar hasta encontrar la solución, el número de soluciones distintas que se desea que se muestren o el número máximo de colisiones que se está dispuesto a tolerar (número máximo de veces consecutivas que se encuentra una solución que ya se ha mostrado previamente).

Para permitir el uso de todos los controles mencionados se han creado distintos predicados que permiten utilizar combinaciones de estos, de forma que sea posible en ciertos casos realizar búsquedas que no controlen algunas de las variables.

El uso de este tipo de búsqueda, si bien ofrece ventajas en la realización de estudios estadísticos o proyectos donde no se tenga preferencia de unos resultados frente a otros, no sería apropiado para buscar soluciones optimizadas de escenarios con estado inicial y meta (si se variase el orden de interpretación de las cláusulas se obtendrían soluciones válidas pero no optimizadas) y sería totalmente desaconsejado para programas en los que existen cortes rojos, ya que la variación del orden de interpretación de sus cláusulas podría dar lugar a búsquedas con bucles infinitos.

Su diseño en forma de paquete permite que sea utilizado por cualquier usuario de Ciao Prolog. Además, su división por módulos separa la interfaz de usuario, encargado de obtener los límites para las variables de control y generar los mensajes de salida, del algoritmo y el módulo de traducción. De esta manera, se simplifican los costes de mantenimiento y se evita la propagación de fallos. Por otra parte, en el desarrollo de este algoritmo se ha premiado la sencillez de su diseño frente a la optimización del código, lo cual podría tomarse como un punto de partida para posibles mejoras.

En un futuro, se podrían aumentar los usos del algoritmo incluyendo la existencia de pesos probabilísticos en las cláusulas que permitiesen que la recolocación de estas no dependiese de un factor meramente aleatorio sino que tuvieran distintas probabilidades de ser ejecutadas unas frente a otras (integración del algoritmo dentro del campo de *probabilistic logic programming*). Siguiendo esta línea, también se podría desarrollar un método relacionado con el campo de aprendizaje automático (*machine learning*) que calculara pesos para las cláusulas en función del número de veces que estas fueran utilizadas, de forma que al comienzo de la ejecución todas las cláusulas tuvieran los mismos pesos y estos se fuesen recalculando según avanzara la ejecución. Por otra parte, eliminar el mecanismo que evita que se repitan soluciones podría tener aplicaciones estadísticas, ya que podría estudiarse qué soluciones se repiten más a menudo a pesar de que se generen de forma aleatoria y por qué. Para finalizar, se podría desarrollar un algoritmo que combinase la búsqueda aleatoria desarrollada con la búsqueda en anchura o la búsqueda en profundidad iterativa[22] (*iterative deepening*) y permitiese obtener soluciones aleatorias dentro de un límite de profundidad que se iría ampliando durante la ejecución del programa. De esta forma, el orden de las soluciones no sería plenamente aleatorio, sino que estaría ordenado por profundidad.

Por último, este tipo de búsqueda permite un ahorro en memoria respecto a las búsquedas en anchura y profundidad. Mientras que estas últimas mantienen el árbol en memoria durante toda la ejecución, la búsqueda aleatoria solo guarda constancia de las

trazas que ya han ofrecido soluciones previamente. De esta forma, aunque puede tener pérdidas en eficiencia (el tiempo de ejecución se podría ver incrementado al probar ramas del grafo que ya han sido exploradas previamente), se trata de un algoritmo que posee ventajas en cuanto a ahorro en memoria frente a las búsquedas clásicas.

BIBLIOGRAFÍA

[1] Wikipedia (s.f.). "*Logic programming*". Recuperado en mayo de 2017 [Online] Disponible en: https://en.wikipedia.org/wiki/Logic_programming

[2] Wikipedia (s.f.). "*Search tree*". Recuperado en mayo de 2017 [Online] Disponible en: https://en.wikipedia.org/wiki/Search_tree

[3] Wikipedia (s.f.). "*Prolog*". Recuperado en junio de 2017 [Online] Disponible en: https://en.wikipedia.org/wiki/Prolog

[4] Sterling, L. & Shapiro, E., "*Basic constructs*", in The Art of Prolog. Cambridge, USA: The MIT Press. (1986)

[5] Horn clause logic. [Online] Disponible en: https://cs.nyu.edu/courses/spring02/G22.2560-001/horn.html

[6] The Ciao System. [Online] Disponible en: https://ciao-lang.org/

[7] Wikipedia (s.f.). "*Backtracking*". Recuperado en mayo de 2017 [Online] Disponible en: https://en.wikipedia.org/wiki/Backtracking

[8] Apuntes: Curso en programación lógica (2008). "*Control en la programación lógica*". [Online] Disponible en: http://www.sc.ehu.es/jiwhehum2/prolog/Temario/Tema3.pdf

[9] Sterling, L. & Shapiro, E., "*Cuts and negation*", in The Art of Prolog. Cambridge, USA: The MIT Press. (1986)

[10] Riguzzi, F., "*MCINTYRE: A Monte Carlo Algorithm for Probabilistic Logic Programming"*, in Ferrara, Italia: Università di Ferrara. (2001)

[11] De Raedt, L., Kimmig, A. & Toivonen, H., "*ProbLog: A Probabilistic Prolog and its Application in Link Discovery*", in Freiburg, Alemania: Machine Learning Lab, Albert-Ludwigs-University. (2006)

[12]Poole, D., "*The Independent Choice Logic and Beyond*". In: De Raedt L., Frasconi P., Kersting K., Muggleton S. (eds) Probabilistic Inductive Logic Programming. Lecture Notes in Computer Science, vol 4911. Springer, Berlin, Heidelberg. (2008)

[13] Wikipedia (s.f.). "*Statistical model*". Recuperado en junio de 2017 [Online] Disponible en: https://en.wikipedia.org/wiki/Statistical_model

[14] Wikipedia (s.f.). "*Aprendizaje 45utomatic*". Recuperado en mayo de 2017 [Online] Disponible en: https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico

[15] Aprendizaje automático [Online]
Disponible en: http://searchdatacenter.techtarget.com/es/definicion/Aprendizaje-automatico-machine-learning

[16] The Ciao system. "*Random library*". [Online]
Disponible en: http://ciao-lang.org/legacy/files/ciao/ciao_html/ciao_174.html

[17] Wikipedia (s.f.). "*Breadth-first search*". Recuperado en junio de 2017 [Online]

Disponible en: https://en.wikipedia.org/wiki/Breadth-first_search

[18] Wikipedia (s.f.). "*Depth-first search*". Recuperado en junio de 2017 [Online]
Disponible en: https://en.wikipedia.org/wiki/Depth-first_search

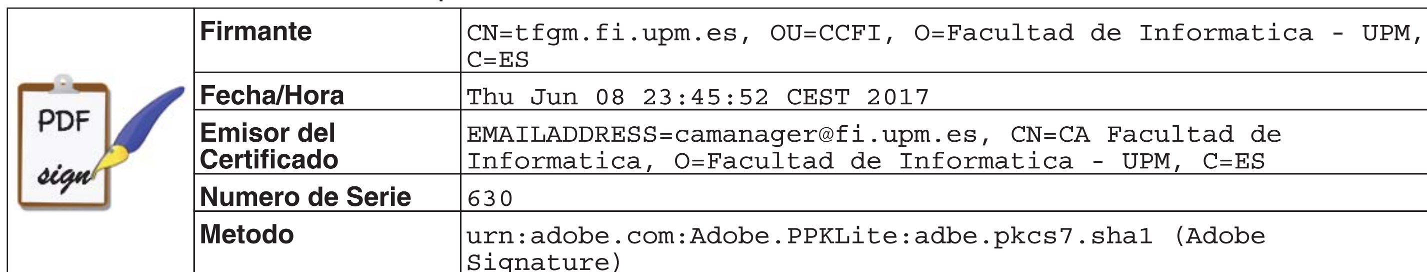
[19] Wikipedia (s.f.). "*Axiomas de Peano*". Recuperado en mayo de 2017 [Online]
Disponible en: https://es.wikipedia.org/wiki/Axiomas_de_Peano

[20] Wikipedia (s.f.). "*Tower of Hanoi*". Recuperado en junio de 2017 [Online]
Disponible en: https://en.wikipedia.org/wiki/Tower_of_Hanoi

[21]Association for Logic Programming (ALP). [Online]
Disponible en: http://web.archive.org/web/20080723172635/http://vl.fmnet.info/logic-prog/

[22] Wikipedia (s.f.). "*Iterative deepening*". Recuperado en mayo de 2017 [Online]
Disponible en: https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Thu Jun 08 23:45:52 CEST 2017
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)