Games: A New Scenario for Software And Knowledge Reuse

Carlos Linares López Ubaldo David Estévez Garrido Asunción Gómez-Pérez
Luis Manuel Sánchez Acera

Laboratorio de Inteligencia Artificial
Facultad de Informática
Universidad Politécnica de Madrid
28660 - Boadilla del Monte
Madrid - Spain
E-mail: {clinares,udavid,asun,lsanchez}@delicias.dia.fi.upm.es

Abstract

Games are a well-known test bed for testing search algorithms and learning methods, and many authors have presented numerous reasons for the research in this area. Nevertheless, they have not received the attention they deserve as software projects.

In this paper, we analyze the applicability of software and knowledge reuse in the games domain. In spite of the need to find a good evaluation function, search algorithms and interface design can be said to be the primary concerns. In addition, we will discuss the current state of the main statistical learning methods and how they can be addressed from a software engineering point of view. So, this paper proposes a reliable environment and adequate tools, necessary in order to achieve high levels of reuse in the games domain.

1. Introduction

Since the Minimax procedure [27] was first introduced, many researchers have tried to improve it. Only a few games have been solved so far [1, 13, 29, 39] and the vast majority of them remain unsolved. There are still a lot of people researching in this area and this feverish activity will not stop until the most important games have been solved. However, this is not expected to happen in a near future.

Underlying this efforts, there is a great fascination for the human mind, and the exciting prospect of its simulation. This is what led to great expectations in both the science comunity and the market [9]. Indeed, games constitute a separate domain since:

- They have their own room in the market and many companies have made large investments in this area.
- Scientists consider them a good scenario for testing new search algorithms and learning methods.
- Finally, there are periodic championships among computers and, occasionally, between human beings and computers. Actually, the current world champion in checkers is *Chinook*, a program [35].

Games are software projects. They therefore fall within the scope of software engineering and its applicability. This should never be obviated. Indeed, software reuse should assist the development of new games if the proper conditions are established.

In the following sections, we will describe how we reuse software and knowledge in the domain of games. First, section 2 reviews the main achievements in game theory. In section 3, we will apply domain analysis to this area and we will present a model for implementing new applications. We conclude, in section 4, by presenting the summary and conclusions.

2. Overview of games theory

In recent years, researchers have made a great effort to devise smarter and smarter search algorithms. The first success was the *alpha-beta* algorithm [15], which was based on the *cutoff* concept. This algorithm demonstrated that, if the necessary conditions are met, a whole subtree could be pruned without further considerations, assuring that its backed-up value is not relevant for the final outcome. Then, the immediate concern was how to increase the percentage of cutoffs without sacrificing decision quality. Later new algorithms [11], like Falpha-beta, Lalpha-beta, Palphabeta and Scout, and techniques [33], like fixed and dynamic node ordering, iterative deepening, transposition tables, refutation tables, minimal windows, aspiration search,

killer heuristic or history heuristic, appeared. At the same time, G. Stockman devised SSS* [38], whose main advantage over the alpha-beta algorithm is that it expands —at least theoretically— less nodes. However, the difference is not enough to offset the time spent, so this algorithm is not attractive for practicioners. Later, B* appeared [28], which was programmed for playing chess. Its most important contribution was to demonstrate that it was feasible to apply other strategies (called *provebest* and *disproverest*) than those applied by the minimax principle. Indeed, these two strategies have been widely considered in other algorithms as well, like proof numbers [2], which was based on conspiracy numbers [21, 34]. Besides, new paradigms appeared, such as, for instance, for handling uncertainty in chess [14], how to use knowledge to control tree searching [40], and Bruce W. Ballard [5] presented **-minimax* for searching trees with probability nodes that were called **nodes*.

Later, Richard E. Korf observed that while the computer considers millions of positions, its human opponent only needs to study, selectively, about ten to win. Although the importance of *selective search* was first recognized by Shannon [36] in 1950, only a few contributions have been made in this respect. Thomas Anantharaman et al. presented the singular extensions [3] and Smith and Nau offered an analysis of forward pruning [37]. Recently, Richard E. Korf presented a different manner for traversing and exploring games trees, best-first minimax search [16, 17], an algorithm which improves the quality of decisions by expanding terminal nodes selectively.

On the other hand, as long as search algorithms assume insufficient computation to search all the way to terminal positions, they need an *evaluation function*, which will score any node. If search algorithms are to improve the quality of the game, it is absolutely necessary that they rely on accurate evaluation functions. Otherwise, the errors made by the evaluation functions could lead into undesirable side-effects such as pathologies [26, 30] or biasing [25]. Nevertheless, it is very difficult to get a good evaluation function and often, they are manually tuned. Learning methods have been widely employed in order to strengthen the computer's game. Among them, statistical learning methods are expected to be domain independent insofar as they are based on features which must be related. For this to be true, it is necessary to extract a representative set of features, and this is, a domain-dependent matter. Often, they are easily obtained from expert knowledge. However, how to combine them into a single evaluation function is a cumbersome and far from trivial problem. If it can be proven that the features follow a normal distribution, Bayesian inference is a suitable tool [20]. Otherwise, it is always possible to do not assume any distribution and apply regression analysis in the hope of fitting the samples as well as possible [10].

Moreover, once a position or a concatenation of moves have been proven to be successful, it can be put into a dabatabase for later reuse. In this sense, databases can be used in different stages of the game: *opening*, *middle* and *end game*.

- The *opening game* can be based upon a book containing combinations of moves proven to be successful. These databases reduce the search space insofar as they induce to consideration of only the states stored [10, 23]. Furthermore, the information stored could consist in proverbs instead of moves [8].

- Often, the highest branching factor characterizes the *middle game* (and, in some cases, the end game). Under these circumstances, it may be attractive to make use of a knowledge-based analysis with the aim of setting an estimated value for each move or, even, the correct move [40].

- The final result for some configurations of the game (under a perfect play assumption) can be saved, once they have been explored. These databases will drastically reduce the time spent on the search at the *end game*, as long as they bring up the final outcome for those positions [18].

However, in spite of these efforts, computers do not yet generally defeat their human counterparts. The main problems come from processing uncertainty. Recently, Martin Müller [23] has offered a new perspective based on knowledge engineering. He established four principles which must be met for knowledge acquisition and selection: *consistency*, *completeness*, *relevancy* and *soundness*. These fundamentals must be accomplished in order to avoid the three main sources of uncertainty:

- *Incompleteness*: caused by the fact that only a small fraction of the real world under study can be observed or compiled. The most representative example, is the evaluation function, which is made up of features [20]. Finding a suitable subset of these features is in itself an incomplete task.

- *Inconsistency*: when the system could lead to conflicting conclusions. In this respect: first, pathologies have been widely investigated [26, 30]; second, some search algorithms, like the singular extensions [3], quiescence search [6] and others have been developed in order to avoid the Berliner effect [7].

- *Inaccuracy*: if the final outcome of the current system status cannot be clearly established. In fact, many programs have a database where the final result for

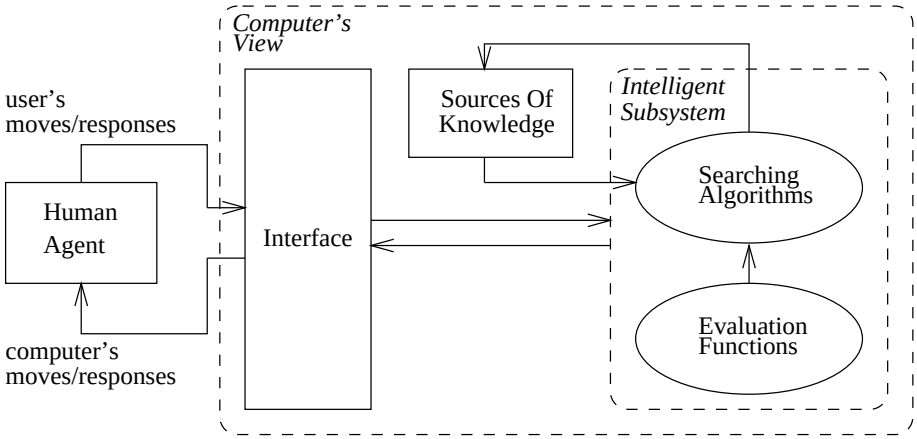

Figure 1. General components of the design of a game

some configurations of the game (under a perfect play assumption) is saved [23]. However, nothing can be said —without assuming an error probability— of any other position, where the search mechanisms and the evaluation function do their best.

3. Domain Analysis

Three general approaches to reuse [12], have been commonly recognized and will be referred to throughout this section:

- *Adaptive approach*. This is the basic form of reuse. It proposes use of already developed projects as a starting point and to change and refine them.

- *Parameterized approach*. Whenever different families of products have been identified, it is highly desirable to establish common funcionalities and data flows among them. Thus, it should be possible to construct an architecture including such programs. Obviously, the new product is reusable and its behaviour is characterized through a set of values for different parameters.

- *Engineered approach*. Its main goal is to find suitable domain-independent processes that could be applied indifferently, in order to establish commonality and enable software reuse. In recent years, domain analysis [4, 32] has shown itself to be a powerful technique in this field.

Domain analysis should serve to help developers to understand the application domain, significantly increasing their understanding of the subject, and to classify existing reusable components [22]. This can be achieved by various means, such as identifying actions, objects, agents and mediums [19], or by performing three activities [12]: *scoping*, *domain modeling* and *architectural modeling*. While the first alternative is useful for measuring the reusability of existing applications, the second is more appropiate for the design stage. So, we will use the three above-mentioned activities to analyze how software reuse is applied to the games domain. Under this approach, each step addresses a different view of the same domain, which will be studied in the subsections below.

3.1. Scoping

This activity delimits the context of the domain, representing the primary inputs and ouputs, as well as the main interfaces [12]. Basically, the literature distinguishes three modules in any game: the *interface*, the *intelligent subsystem* and the *sources of knowledge*, which are represented in the block diagram shown in Figure 1, identifying the main entities and the operations on them.

The *intelligent subsystem* consists of the search algorithms in conjunction with the set of evaluation functions that guide them. When it is invoked, it returns the best move, which is sent to the interface. It is a very common practice to return its score too.

The *sources of knowledge* provide ad-hoc rules which can be used solely for the benefit of search algorithms during the game, as was pointed out in section 2 when we discussed the use of databases.

The goal of the *interface* is to assure the existence of an adequate procedure for managing the external units of information. There is a distinction between the tasks that must be accomplished by the interface: *administrative tasks*, which concern the execution of service requests as access to databases in which it is possible to load or save games

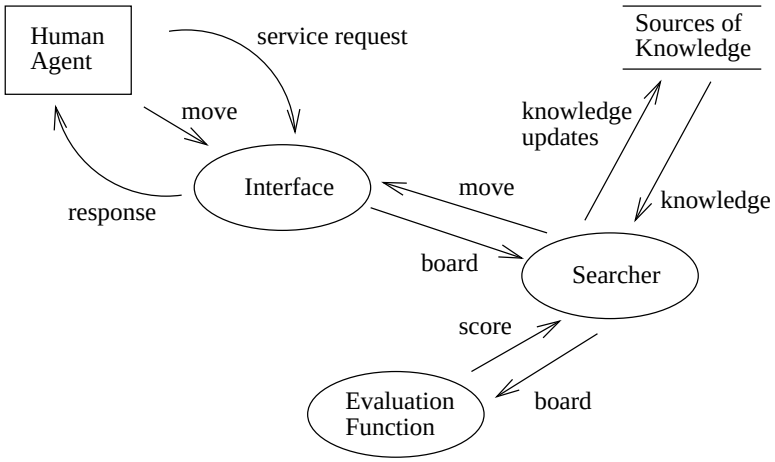

Figure 2. Primary data flows

and access a wide range of commands arranged in menus as: go back/forward, hint, increase/decrease level of difficulty[1], annotations, setup the board, resign, new game, etc.; and *gambling tasks*, which are related to reading of the next move, its validation, printing user and computer moves and other local information, such as the progress of the search algorithm.

The last entity involved in this domain is the *human agent* who plays with the computer or sends requests, asynchronously, to the *interface*.

3.2. Domain modeling

The goal of this activity is to provide a description of the requirements met by software within this domain [12]. In addition, we will detail every process involved in the development of a new game, using data flow diagrams to represent the relationships between the entities identified in *scoping*.

Games can be classified according to different criteria: number of players or agents; complete or incomplete information; zero or non-zero sum, and cooperatives or non-cooperatives. Within the context of this paper, we will address two-person, zero-sum, non-cooperative, complete or incomplete information games, such as othello, checkers, go, stratego, etc. However, there are many other kinds of games such as solitaire, diplomacy, risk, dominoes, ... which are irrelevant to our analysis, from this perspective at the present time.

Figure 2 shows the primary data flows and the information exchanged by entities. As this study is related to two-person games, there are only two external entities: the *human agent* and the *sources of knowledge*. Both delimit the behaviour of the computer: the human player, for obvious reasons; the *sources of knowledge*, because they contain essential information that could affect the chosen move.

The *interface* receives either moves or any service request from the human player (like "go back"). If it can satisfy the request (for example, move back a step and generate a new board), it will meet it by executing the necessary procedures without exchanging information with the *searcher*. However, if it receives a move or a special request (such as "give me your advice on the next move"), it passes the new board to the searcher and waits patiently for the response in order to draw up the new configuration or to satisfy the request. Besides, the search algorithm could make use of or modify the knowledge available in the *sources of knowlege*. This is why they can be considered reusable components, because it is possible to write new information for later reuse. More specifically, opening databases are actually a knowledge reuse of all the knowledge available in form of theory on literature.

In addition, the *searcher* will give boards to the *evaluation function* and will obtain numerical scores which will be handled inside the searcher.

From the data flows representation in Figure 2, it is feasible to extract properties of objects that are likely to be effectively reused. To do so, let us focus on the internal entities (external entities will be discussed later), and it will become clear that there are templates for them as shown in the Figure 3, where: `move_t` refers to the data flow labelled "move"; `score_t` is "score" and `board` stands for "board" in Figure 2. The existence of the argument `player` gives flexibility to both procedures and allows both the computer or the human opponent to be evaluated (which is useful for programming options like "hint"). The type `side_t` could be any structure capable of distinguishing between both players. The suspension points in the

[1]Commercial programs usually allow the customization of the personality of the program, even in imitation of some famous human players.

procedure `search` suggest that there could be other parameters like the bounds α or β used in many search algorithms or, at least, one such bound which must be typed as `score_t`. Obviously, any of these types, or even the possibility of receiving a variable number of arguments, can be easily implemented in many languages without the need of object-oriented techniques. The other internal entity, the *interface*, cannot, however, be clearly characterized as long as it is the main procedure of the program.

3.3. Architectural modeling

The goal of architectural modeling is to provide a model for implementation, suitable for constructing applications within the specified domain [12].

The architecture, shown in Figure 4, respects the blocks shown in Figure 1 and the data flows described in Figure 2. The architecture emphasizes the allocation of resources for the main entities identified in the previous activities. As is shown, both learning and search procedures are classified and stored in their respective repositories. So, these items will be reused pursuant to certain heuristic or statistical reasons; it is assumed that the developer knows which component is more appropiate for the case at hand.

To construct search functions as reusable packages, it is necessary to adopt a parameterized approach. The most difficult process is to identify common data flows. However, within the context of this paper, this is rather easy because many pieces of code are strongly related. So, an adaptive approach to the minimax procedure is feasible. Once the search algorithms are programmed, they can be successfully reused in different games because the data flow labelled as "board" in Figure 2 is not handled inside the searcher. Instead, it is passed to other procedures which must be programmed for every game. The same applies to the data flow labelled as "move", which only contains a move. For instance, Figure 5 shows the minimax and scout pseudocodes in a C/Pascal-like language [11, 24]. Obviously, the procedures for generating the sons, scoring the boards or deciding whether a node is terminal are domain-dependent and must be programmed for every game. However, once they have been successfully implemented, they can be reused across all the search algorithms contained in the repository. Similarly, it is apparent that an adaptive approach to other search algorithms is feasible.

On the other hand, Aske Plaat [31] has shown how, using certain parameters, an algorithm using the function he called *MT driver*[2], can behave like many others. So, a parameterized approach to reuse is definitely feasible.

As for the search enhancements, they are easily adaptable and could be properly parameterized for use with a wide variety of search algorithms and games. Dynamic ordering is based upon a sorter which is often included in the language chosen for the development. Transposition tables or killer heuristics, consist of tables which store information relating to the search tree. So, their contents can be typified in the same way as search algorithms. Therefore, the routines for their management can be successfully reused. Iterative deepening can be adopted without further problems, including the search algorithm into a loop, which iteratively invokes the search algorithm, possibly with different values every time.

As far as the interface is concerned, there are packages for every environment which implement specific protocols. Fortunately, most of the *gambling tasks* identified in section 3.1 can be put together in a common module, leaving it to specialized libraries to perform specific actions related to each environment. However, we carry out the main tasks with a tool specially programmed, which is called *Puzzle* in figure 4.

The administrative tasks could be programmed in a separate module, but strictly speaking, they cannot be considered as reusable, because they could be highly implementation dependent. For instance, "go back" should be programmed decreasing the value of a specific variable and then, redrawing the board. Which variable to decrease and how to interpret the contents of the new board are implementation dependent matters. However, the gambling tasks can be effectively reused by means of reuse of specifications.

The notion is to adapt the specification to natural language and directly produce code which could be linked with the rest of the components shown in Figure 4. This can be accomplished with the help of a code generator that would ask about different aspects, and finally, the code generated should be embedded in the application. It should be noted that this method favours prototyping life cycles, allowing a lot of refinements. As the resulting code must be linked with the rest of the application, two major issues must be covered: information on the parameters exchanged between the interface and other procedures, and information on the design itself. Information on the first issue should be used to integrate the system, whereas information on the design must be kept within the interface package.

Finally, the presence of a couple of learning packages in Figure 4 is noteworthy. They consist of procedures for reading games saved in a database which label each position of the game as winning or losing and other procedures which try to recognize feature patterns using *Bayesian inference* or *regression analysis*. This analysis should result in a evaluation function suitable for the search algorithm.

Although learning procedures could be treated as reusable packages, their reusability is limited to the game under development. Obviously, it is not the same to learn to

[2] MT stands for *Memory-enhanced Test*

Searcher	move_t search (board:board_t, player:side_t, . . .)
Evaluation Function	score_t evaluate (board:board_t, player:side_t)

Figure 3. Templates for the searching and evaluating processes

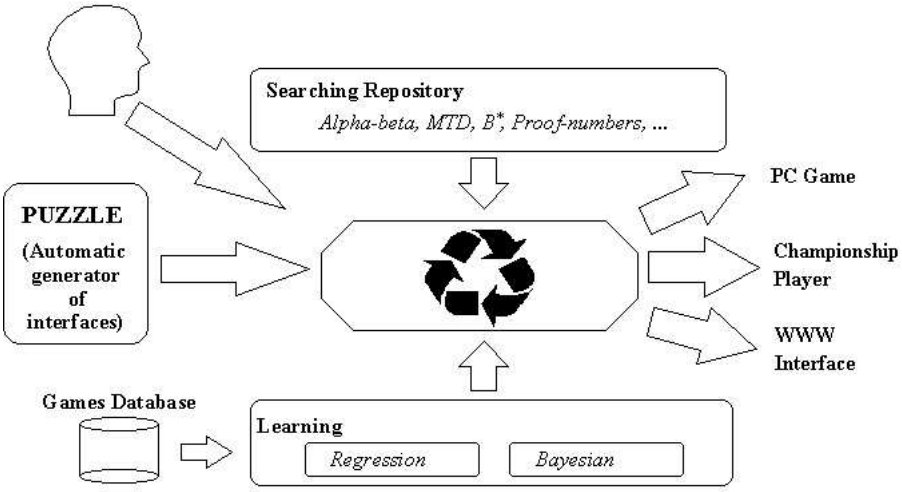

Figure 4. Architecture for the implementation of games

```
score_t minimax (b:board_t, side:side_t)
{
    if (terminal (b)) then
        return (evaluate (b,side));
    w=generate_sons (b);
    if (kind (b)=alpha) then
        m=-∞;
    else
        m=+∞;
    for i=1 to w do
    {
        t=minimax(b_i);
        if (t>m) and (kind (b)=alpha) then
            m=t;
        if (t<m) and (kind (b)=beta) then
            m=t;
    }
    return (m);
}
```

```
score_t scout (b:board_t, side:side_t)
{
    if (terminal (b)) then
        return (evaluate (b,side));
    w=generate_sons (b);
    m=-scout(b_1);
    for i=2 to w do
        if (not (test (b_i,-m,side))) then
            m=-scout(b_i);
    return (m);
}

score_t test(b:board_t, p:score_t, s:side_t)
{
    if (terminal (b)) then
        if (evaluate (b,s)>p) then
            return (TRUE);
        else
            return (FALSE);
    w=generate_sons (b);
    for i=1 to w do
        if (not (test (b_i,-p))) then
            return (TRUE);
    return (FALSE);
}
```

Figure 5. Pseudocodes of minimax and scout procedures

play Go 19×19, or Othello or chess. Nevertheless, the same package can be used —under a parameterized approach— for different features, probability functions or approaches as local/global, weighted/non-weighted learning. Anyway, the underlying idea is always the same and it is possible to program other learning methods for other games under an adaptive approach.

To conclude, it is important to note that there is implicit knowledge reuse at this point, as long as the lost (and drawn) games can be re-examined once the program is operating, with the aim of detecting the mistakes made and adjust the evaluation function.

4. Summary and Conclusions

Although games are a well-known domain for testing search algorithms and learning methods, they have received little attention as software products. We have addressed both software and knowledge reuse with the aim of developing games rapidly, as long as the use of a common framework serves to increase the productivity and the reliability of future projects. As domain analysis has shown itself to be a powerful technique for software reuse, we have applied it successfully with the hope of obtaining a suitable architecture which could be shared by different games. This study lead to the identification of three very important modules: search algorithms, learning methods and interface programming. We have shown how these tasks can be addressed independently.

Our experience has demonstrated that this approach is advantageous and appropiate. However, it is important to note that the computation time spent by the search algorithms can sometimes be slightly improved by means of tricks relative to the rules of the game or any other characteristic. Anyway, it is possible to apply prototyping life cycles and therefore refinements can be regularly made.

5. Acknowledgments

The authors would like to thank Mariano Fern´andez L´opez for his assistance in reviewing the first drafts and giving insight, to M´onica Roll´an Galindo and Rachel Elliott for reviewing the format and language and to the SEARCHING AND LEARNING research group of the Artificial Intelligence Laboratory for their comments and suggestions.

References

[1] L. V. Allis, H. J. van den Herik, and M. P. H. Huntjens. Go-moku solved by new search techniques. In *Proceedings of the AAAI Fall Symposium on Games: Planning and Learning*, 1993.

[2] L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. Proof-number search. *Artificial Intelligence*, 66:91–124, 1994.

[3] Thomas Anantharaman, Murray S. Campbell, and Feng hsiung Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43:99–109, 1990.

[4] G. Arango. Domain analysis - from art to engineered discipline. In *Proceedings of the 5th International Workshop on Software Specifications and Design*, pages 152–159, 1989.

[5] Bruce W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21:327–350, 1983.

[6] Don F. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43:85–98, 1990.

[7] H. Berliner. *Chess as problem solving: the development of a tactics analyzer*. PhD thesis, Carnegie Mellon University, 1974.

[8] D. Borrajo, J. R´ıos, M. A. P´erez, and J. Pazos. *Heuristic Programming in Artificial Intelligence, the First Computer Olympiad*, chapter Integration Issues in an Expert Dominoes Player. Ellis Horwood, West Sussex, 1989.

[9] Daniel Borrajo, Natalia Juristo, Vicente Mart´ınez, and Juan Pazos. *Inteligencia Artificial*. Ram´on Areces, Madrid, 1993.

[10] M. Buro. *Techniken für die Bewertung von Spielsituationen anhand von Beispielen*. PhD thesis, University of Paderborn, Paderborn, Germany, October 1994. In German.

[11] Murray S. Campbell and T. A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20:347–367, 1983.

[12] Sholom Cohen. *Process and Products for Software Reuse and Domain Analysis*. Software Engineering Institute, Pittsburgh, PA, 1990.

[13] R. Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 1994.

[14] Helmut Horacek. Reasoning with uncertainty in computer chess. *Artificial Intelligence*, 43:37–56, 1990.

[15] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[16] Richard E. Korf and David Maxwell Chickering. Best-first minimax search: Othello results. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 2, pages 1365–1370, 1994.

[17] Richard E. Korf and David Maxwell Chickering. Best-first minimax search. *Artificial Intelligence*, 84:299–337, 1996.

[18] R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde analysis problems using a network of workstations. *Advances in Computer Chess VII*, 1994.

[19] Ronald J. Leach and Terrence L. Fuller. An illustration of the domain analysis process. *Software Engineering Notes*, 20(5):78–82, December 1995.

[20] Kai-Fu Lee and Sanjoy Mahajan. A pattern classification approach to evaluation function learning. *Artficial Intelligence*, 36:1–25, 1988.

[21] David Allen McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35:287–310, 1988.

[22] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[23] Martin Müller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. PhD thesis, Swiss Federal Institute of Technology Zürich, 1995.

[24] Agata Muszycka and Rajjan Shingal. An empirical comparison of pruning strategies in game trees. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15(3), 1985.

[25] Dana S. Nau. The last player theorem. *Artificial Intelligence*, 18:53–65, 1982.

[26] Dana S. Nau. Pathology on game trees revisited, and an alternative to minimaxing. *Artificial Intelligence*, 21:221–244, 1983.

[27] John Von Neumann. Zur theorie der gesellshaftsspiele. *Mathematical Annals*, 100:295–320, 1928.

[28] Andrew J. Palay. The B* tree search algorithm —new results. *Artificial Intelligence*, 19:145–163, 1982.

[29] O. Patashnik. Qubic 4×4×4 tic-tac-toe. *Mathematics Magazine*, 53:202–216, 1980.

[30] Judea Pearl. On the nature of pathology in game searching. *Artificial Intelligence*, 20:427–453, 1983.

[31] Aske Plaat. *Research Re: Search & Re-Search*. PhD thesis, Rotterdam, March 1996.

[32] Ruben Prieto-D´ıaz and Guillermo Arango. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, California, 1991.

[33] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(1):1203–1212, November 1989.

[34] Jonathan Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43:67–84, 1990.

[35] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook: The world man-machine checkers champion. *AI Magazine*, 17(1):21–29, Spring 1996.

[36] C. E. Shannon. Programming a computer for playing chess. *Philosophical magazine*, 41:256–275, 1950.

[37] Stephen J. J. Smith and Dana S. Nau. An analysis of forward pruning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 2, pages 1386–1391, 1994.

[38] G. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, pages 179–196, 1979.

[39] J. W. H. M. Uiterwijk, H. J. van den Herik, and L. V. Allis. *Heuristic Programming in Artificial Intelligence, the First Computer Olympiad*, chapter A Knowledge-based Approach to Connect-four. The Game is Over: White to Move Wins!, pages 113–133. Ellis Horwood, West Sussex, 1989.

[40] David E. Wilkins. Using knowledge to control tree searching. *Artificial Intelligence*, 18:1–51, 1982.