Computer Analysis of Connect-4 PopOut

University of Oulu
Department of Information Processing Science
Master's Thesis
Jukka Pekkala
May 18th 2014

Abstract

In 1988, Connect-4 became the second non-trivial board game to be solved. PopOut is a variant of Connect-4 that was released in 2009. PopOut allows both players to "pop" disks from the bottom row in addition to dropping new disks from above. Connect-4 is a divergent game where the maximum depth of the game tree is 42. Because of the pop moves, PopOut is not divergent but unchangeable and this causes the maximum depth to be much larger. Furthermore, PopOut allows positions to be repeated making the Graph History Interaction (GHI) problem possible.

It is not fully understood why some games have been solved while others remain unsolved. Connect-4 and PopOut are similar games but they have slight changes in some game characteristics. The biggest of these is the change in convergence type. There has not been a previous reported attempt to solve PopOut. By analyzing the game, more light is shed on the relationship between game characteristics and solvability.

We studied whether the search techniques that have been used in solving Connect-4 are also applicable to PopOut and how they should be modified or augmented to be effective. Proof-number search was able to weakly solve PopOut with no modifications. Alpha-beta with its standard enhancements needed heavier modifications to counteract the depth of the game tree and to circumvent the GHI problem.

The game-tree depth was successfully limited by "handicapping" the first player. The first player was prohibited from making non-winning pop moves and any game exceeding a chosen number of moves was declared a loss for the first player. Alpha-beta was able to show that the first player still wins under these handicapping rules. The handicapping technique was essential in solving the game with Alpha-beta. The technique has not been used previously but it might also be applicable to other board games and search algorithms.

PopOut is a first-player win on the standard board size 7x6. Smaller board sizes are either first-player wins or draws by repetition. Some of the smaller boards were strongly solved by retrograde analysis which performed a total enumeration of the state space. No contradictions were found between the three different solving approaches suggesting that the results are reliable.

Keywords
Connect-4, PopOut, Alpha-beta, Proof-number search, Graph History Interaction, Design Science Research

Foreword

I would like to thank three people who contributed to this thesis directly or indirectly. Dr. Ari Vesanen, my supervisor, meticulously read the early drafts of my thesis and suggested improvements. Dr. John Tromp, who strongly solved Connect-4, has done a great service for game programmers by releasing the source code to his program. Without learning from his source code first, I would not have been confident enough to choose this topic. Finally, Johan Nordlund provided expert help in PopOut and tested the program for any errors.

Contents

1. Introduction

The purpose of this study was to analyze the game of PopOut from the point of view of artificial intelligence. The main focus of the analysis was to test the applicability of search techniques on PopOut, which had not been done previously. Through the application of several search techniques, the game was weakly solved for the standard board size 7x6 and smaller board sizes.

Computer games research is a subfield of artificial intelligence research and it has two goals (Heule & Rothkrantz, 2007; van den Herik, Uiterwijk, & Rijswijck, 2002). The first one is to make computers intelligent enough to outperform human players. The second goal is to analyze the outcome of a game when all players play optimally. This outcome is captured in what is called the *game-theoretic value*. This study focused on finding the game-theoretic value of a game for which the value was previously unknown. It is not fully understood why it has been possible to determine the game-theoretic value for some games but not for others (Heule & Rothkrantz, 2007).

Even though games might at first seem to be rather non-serious for a scientific topic, many notable people have participated in computer games research, including at least Alan Turing and Claude Shannon (Schaeffer, 2001). The advantage of studying computer games becomes apparent when it is contrasted with real-world problems. Whereas real-world problems often have changing rules and an intractable number of possibilities, most games provide fixed rules with a more reasonable number of alternative outcomes. Games can therefore be used to benchmark and advance our understanding of artificial intelligence. New techniques can be tested on games before using them on real-world problems, and if they do not work on games, there is little hope they will solve the more challenging real-world problems either. (Schaeffer, 2001.)

The game of Connect-4 with standard rules and board size (7x6) was the second non-trivial game to be solved with the help of artificial intelligence (van den Herik et al., 2002). This solution was independently discovered by two different authors in 1988 (Allen, 1989; Allis, 1988). A strong solution was provided by John Tromp a few years later (Allis, 1994, p. 163). Since then, Connect-4 has been solved for various board sizes, including an infinite board size (Yamaguchi, Yamaguchi, Tanaka, & Kaneko, 2012). In 2009, two variants of Connect-4 were released by the company that invented and manufactured the original game. These variants change the rules slightly. This study investigated one of these variants called *PopOut*.

This study had two research questions. The first research question asked how the search techniques that were used in solving Connect-4 should be modified or augmented in order to be applicable to PopOut. The second research question asked what the game-theoretic value of PopOut is. The second research question could be answered only if the first question was successfully answered first.

The study was carried out using the design science research paradigm. We built and evaluated an artifact that described what steps should be taken in order to find the game-theoretic value of PopOut. The steps included what search techniques to use and how to

resolve encountered obstacles. Such an artifact is called a *method* in the context of design science research. We evaluated the artifact by creating an instantiation of it. The execution speed and memory usage of the instantiation were then measured, and the measurements acted as a proof that the method worked.

The main contributions of this study are the game-theoretic values of PopOut for different board sizes and the creation of a new search technique. The new search technique works by *handicapping* the stronger player. If the technique is successfully applied, search effort is reduced because certain move sequences do not have to be investigated. The game-theoretic value of PopOut indicates that the game is a first-player win on the standard board size 7x6.

The structure of this thesis is as follows. Chapters 2 and 3 are based on a literature survey of the knowledge base. Chapter 2 introduces board game characteristics and their relationship to solvability. Chapter 3 contains search techniques that have been used or can be used in solving board games that are similar to PopOut. Chapter 4 explains the research problem and the used research method in detail. Chapter 5 describes the build process of the method. An evaluation of the method and the game-theoretic values are given in chapter 6. Discussion and conclusions come in chapter 7.

2. Board Games and Their Solvability

In this chapter, we examine board games that are *two-person zero-sum games with perfect information* and especially the relationship between their characteristics and solvability. As the name suggests, this group of games is defined by three separate properties. They are played by two players. These two players cannot cooperate. If one player wins, the other player has to lose (a draw is also one possibility in some games), hence the name zero-sum. The third property is perfect information. This excludes games with random elements and hidden information. Most card games fail to be perfect information because players cannot see what cards other players have or what the order of the shuffled deck is.

Games that possess these three properties can be assigned something called the *game-theoretic value*. This value indicates the result of the game if both players make only optimal decisions. It can have three possible values: win for first player, loss for first player, or a draw. The game-theoretic value does not exist for games in which there is random or hidden information or cooperation (Heule & Rothkrantz, 2007, p. 107). In order to obtain the game-theoretic value, different solving techniques must be applied. The applicability of these solving techniques in turn depends on the characteristics of the game.

We first describe the rules of PopOut. A more detailed definition of what constitutes a solution comes second. Then a list of solved board games fulfilling the above three properties is reviewed. The fourth part summarizes what kind of characteristics different researchers have identified in board games. The fifth part analyzes the relationship between these characteristics and solvability.

2.1 Connect-4 and PopOut

PopOut is played on a vertical two-dimensional board. The standard board size has seven columns and each column consists of six cells. Two players take turns and at each turn, the player to move can make one of two types of moves: drop a new disk, or pop an existing bottom disk of his own color. When dropping a new disk, the disk falls down as much as it can due to gravity. When popping a disk, all disks above the removed one fall down. At any point, gravity ensures that there can be no gaps in any of the columns. The aim of the game is to create a four-in-row, i.e. at least four disks of the same color arranged horizontally, vertically or diagonally. See Figure 1 for allowed moves in a chosen position.

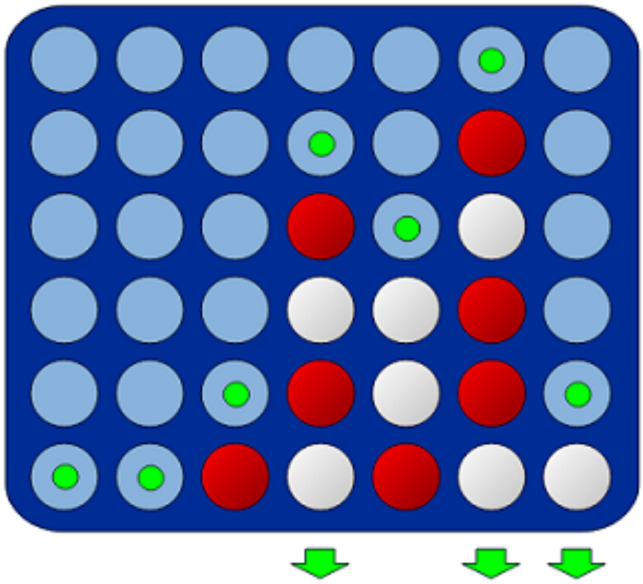

Figure 1. White to move and can choose among 7 drop moves (green dots) and 3 pop moves (green arrows).

PopOut is a variant of Connect-4 and differs from Connect-4 by the introduction of pop moves and three additional rules to handle draws and wins. These three rules are needed to remove ambiguity about what the result of the game is. The first rule deals with simultaneous four-in-rows. If a pop move creates four-in-rows for both players, the player who made the pop move is declared a winner and the other player's four-in-row is ignored. The second rule allows the game to be drawn if the board is full. In such a case, the player to move decides whether he wants to make a pop move or end the game as a draw. The third rule handles repetitions. If the same position is repeated three times, either player can declare the game drawn. (Allen, 2010.)

The three additional rules are not applicable to Connect-4 because of lack of pop moves. Drop moves cannot create four-in-rows for both players at the same time and repeated positions are not possible without pop moves. It can therefore be said that PopOut is a superset of Connect-4: what is legal in Connect-4 is legal in PopOut. The converse is not true.

We have modified the third rule slightly for practical reasons. Instead of requiring the state to be repeated three times, we only require it to be repeated two times. This should not affect the game-theoretic value of any position although we do not have any formal proof for it. Intuitively, if best play leads to a repeated state, playing optimally from that position again will eventually lead to the same state.

We briefly clarify some game-specific terminology. The game under study is called PopOut while Connect-4 refers to the game without pop moves. If the board size is omitted, it should be assumed to be 7x6 (seven columns and six rows). The first player is called White and the second player is called Red.

2.2 Definition of solving

Determining the game-theoretic value for games has been one of the most important goals in AI research (Heule & Rothkrantz, 2007; van den Herik et al., 2002). If a two-

player zero-sum game with perfect information has only a finite number of legal positions, the three properties guarantee that the game-theoretic value can be assigned to each position of the game (Knuth & Moore, 1975). Solving the game then means finding the game-theoretic value for the initial position. Some games allow infinite sequences of moves by repeating certain positions. A game-theoretic value can still be assigned to these games by identifying repeated positions and giving the repeated positions some appropriate value (Knuth & Moore, 1975).

Allis (1994) defined three different levels of solving. An ultra-weak solution identifies the game-theoretic value without providing an algorithm that could calculate the correct move for each step. A weak solution provides an algorithm that determines the next optimal move only when all earlier moves have also been optimal since the initial position. A strong solution extends the algorithm to all legal positions, including those that require non-optimal moves to be played. Allis also emphasizes that the algorithm should run within "reasonable resources". This limitation excludes algorithms that could theoretically solve the game weakly or strongly but no existing computer could complete the algorithm within reasonable time and using a reasonable amount of memory.

2.3 Solved games

There is no official list of games that belong to the class of two-player zero-sum board games with perfect information. However, there is an annual event called the Computer Olympiad where AI programs play different games and compete against each other. Allis (1994) created a list of games called the *Olympic List* from the games played at the early Computer Olympiads. This list contains twelve games fulfilling the three properties (and three games with imperfect information). Van den Herik et al. (2002) compiled a list that contains all the games from the Olympic List and six additional games. This list of 18 games should be adequately representative of the class of two-person zero-sum games with perfect information.

Heule and Rothkrantz (2007) determined that eight games from the list of 18 games have been solved. Table 1 shows the solved games augmented with Checkers, which was solved shortly after the publication of their paper (Schaeffer et al., 2007). These nine solved games all belong to the list used by van den Herik et al. (2002) meaning that half of the games are currently solved.

Table 1. Solved two-person zero-sum games with perfect information (Heule & Rothkrantz, 2007, p. 108).

Game	**Year of solution**
Qubic	1980
Connect-4	1988*
Gomoku	1994
Nine Men's Morris	1996
Pentominoes	1996
Domineering	2000

Renju	2001
Awari	2003
Checkers	2007

*Heule & Rothkrantz (2007) mistakenly give year 1989

It should be noted that even if a game is listed as solved, it does not mean that further analysis is necessarily pointless. Most games have a standardized board size and solving the game usually means solving the game only for that board size. The game may remain unsolved for larger board sizes. Another way to modify the game is to change the rules in some way. Different board sizes are usually not counted as separate games but it is unclear whether variants resulting from a rule change should be regarded as separate games or not. In the above list, Renju and Gomoku are listed separately even though Renju is a variant of Gomoku.

Connect-4 with the standard board size 7x6 (seven columns and six rows) was weakly solved by Allen in 1988. Tromp (2010) has later strongly solved the standard board size and weakly solved larger board sizes (see Table 2). Yamaguchi et al. (2012) solved Connect-4 for infinite and semi-infinite boards (either width or height is infinite, or both) by mathematically proving that both players can prevent each other from winning. PopOut and Pop10 are variants of Connect-4 based on a rule change but neither of these has been solved.

Table 2. Game-theoretical values of Connect-4 for different board sizes (Tromp, 2010; Yamaguchi et al, 2012).

First player wins	6x7, 6x9, 7x6, 7x8, 8x5, 8x7, 9x5, 10x5
Second player wins	6x4, 6x6, 6x8, 8x4, 8x6, 9x4, 9x6, 10x4, 11x4
Draw	4x4–4x11, 5x4–5x10, 6x5, 7x4, 7x5, 7x7, semi-infinite, infinite

2.4 Characteristics of board games

In addition to the three defining properties, board games of this type exhibit additional characteristics that affect their solvability. Different authors have identified different characteristics (see Table 3). The complexity of a game is one major way to classify games (van den Herik et al., 2002) and the complexity can be measured in several ways that complement each other. The two most important of these complexity measures are state-space complexity and game-tree complexity. Convergence is a concept related to state-space complexity and it roughly divides games into two groups. Other characteristics such as sudden death and zugzwang appear in some games and can have a significant impact on their solvability.

Table 3. Identification of characteristics of board games by different authors.

Allis (1994)	Convergence, sudden death, complexity
Uiterwijk & van den Herik (2000)	Initiative, zugzwang

van den Herik et al. (2002)	State-space complexity, game-tree complexity, initiative, convergence
Heule & Rothkrantz (2007)	Convergent endgames, equivalent positions, sudden-death threats, local endgames

In total, there are eight distinct characteristics by the four authors and we explain all of these in more detail. We start by defining state-space and game-tree complexities. Then convergence, sudden-death threats and equivalent positions are treated each in its own subsection. Finally the advantage of the initiative, zugzwang and local endgames are combined and treated in a single subsection due to their limited applicability. The order of these sections reflects our judgment of the relative importance of the eight characteristics.

2.4.1 State-space complexity

Any board game has a well-defined state or position at all times. This state refers to the arrangement of different pieces together with any additional information required to completely describe the state of the game. We use the terms *state* and *position* interchangeably. The term state is used when describing a problem space (Korf, 1985, p. 98) whereas the term position, in our opinion, is more natural to use when the game-specific nature of the problem is evident.

When a move is made, the current state changes to another. The starting position of a game is called the initial state. The initial state corresponds to an empty board for some games and for other games it means the state where none of the pieces has been moved. The state-space complexity refers to the number of different states that can be reached from the initial state by making valid moves (van den Herik et al., 2002). Even though some positions may look legal, they are not counted in the state-space complexity unless there is a move sequence starting from the initial state that would lead to that specific position (see Figure 2 for a seemingly legal but unreachable PopOut position).

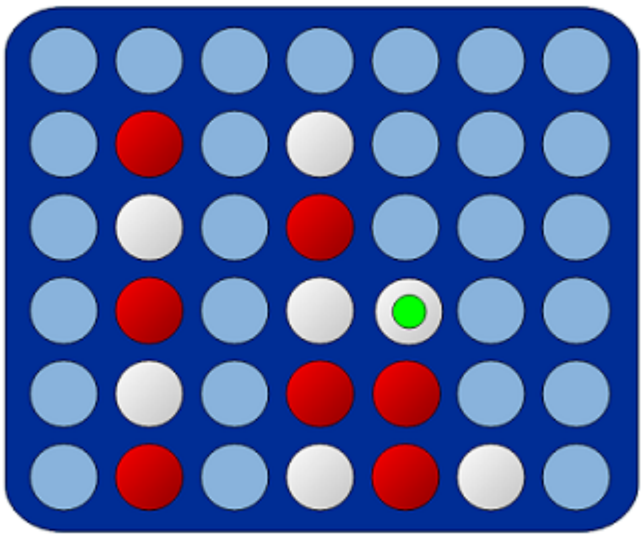

Figure 2. An example of a PopOut position that looks legal but is not reachable.

Allis (1994, p. 159) provides one method of approximating the state-space complexity that is suitable especially in cases where finding the exact complexity is infeasible. The method starts by finding a superset of all legal positions. The superset contains all the legal positions but also some illegal positions. One way is to multiply the number of tiles in the board by the number of different values for one tile (e.g. empty, first player's piece, second player's piece). The set can be further refined as long as none of the legal positions is discarded. Next, a random element of the generated set is taken and it is checked if the position is legal or illegal. This is repeated many times and the ratio of legal and illegal positions in the set is calculated. The calculated ratio should approach the true ratio in a process called Monte-Carlo simulation.

The state-space complexity of a game has a major impact on the solvability of the game. If the state-space complexity is too high, the game cannot be solved by simply enumerating all the legal positions starting from the initial state. Table 4 shows estimated state-space complexities for various games. The values for most games are only approximations and due to the vast differences between games, a logarithmic scale is used. For example, Chess has an estimated complexity of 10^46 and Chinese Chess has a complexity of 10^48. This means that Chinese Chess is estimated to have 100 times more legal positions than Chess. (van den Herik et al., 2002.)

Table 4. Estimates of state-space complexities for various games (van den Herik et al., 2002, p. 300).

Game	**$\log_{10}$ of State-space Complexity**
Awari	12
Checkers	21
Chess	46
Chinese Chess	48
Connect-4	14
Domineering	15
Go	172
Gomoku	105
Hex	57
Nine Men's Morris	10
Othello	28
Qubic	30
Renju	70
Shogi	226

Edelkamp and Kissmann (2008) calculated the exact state-space complexity of Connect-4 to be 4,531,985,219,092. The logarithmic state-space complexity of Connect-4 in the table should therefore be somewhat under 12.7 instead of 14. Even though the difference may not look like much, the estimate is off by a factor of 22 due to the logarithmic scale.

2.4.2 Game-tree complexity

Game-tree complexity complements state-space complexity. As the name suggests, the game-tree complexity is a measure of the *game tree*. The game tree consists of nodes and each node corresponds to some game position. The initial state has a well-defined set of moves leading to new states. These states in turn have their own successor states and so on. This process creates a tree-like structure where each branch represents a move. In most games, the game tree is finite and the leaf nodes of the tree are terminal positions where either player has won or the game has ended in a draw. (Bruin, Pijls, & Plaat, 1994.)

A full-width search goes through all nodes that are some predefined number of moves away from the root state. In order to determine the game-theoretic value, it is not always necessary to perform a full-width search that is as deep as the whole game tree. For example, if a position has a move that leads to an immediate win and the other moves do not, it is enough to perform a full-width search of one level to determine the value. Game-tree complexity refers to the number of leaf nodes in that portion of the game tree that is necessary to be inspected by the shallowest possible full-width search that can solve the game. (van den Herik et al., 2002.)

The main point of the previous paragraph is that game-tree complexity does not refer to the size of the whole game tree. The reason why it refers to only part of the game tree comes from a fundamental search algorithm that is used to traverse the game tree. More precisely, the game-tree complexity estimates the size of the shallowest Minimax search (see section 3.1.1) that is sufficient to solve the game (Allis, 1994, p. 160).

Game-tree complexity is an indication of how many decisions have to be made in order to obtain a solution. Game-tree complexity replaces another measure called *decision complexity* that was used to directly refer to the number of decisions needed for a solution. Decision complexity was found to be too vague and was therefore replaced by the more exact definition of game-tree complexity. (Heule & Rothkrantz, 2007; van den Herik et al., 2002.)

Heule and Rothkrantz (2007) proposed an alternative measure to game-tree complexity that builds on the obsolete decision complexity. For each position that can come up during optimal play, the best move has to be stored. The best move is also the optimal decision and hence the connection to decision complexity. Certain solving procedures can combine several positions so that a solution has to be stored for only one of the positions. Different solving techniques effectively reduce the number of positions for which the solution has to be stored and this reduced number is called the *solution size*. According to Heule and Rothkrantz, the solvability of a game directly depends on the solution size and not on the game-tree complexity. (Heule & Rothkrantz, 2007.)

2.4.3 Convergence

Convergence can be defined in terms of the state space. Roughly speaking, if the number of possible states becomes lower as the game progresses, the game is *convergent*. If the number grows, the game is *divergent*. If the game is neither convergent nor divergent, it is said to be *unchangeable* (Allis, 1994, p. 157). For convergent games, it can be possible to enumerate all end positions if they are few

enough. These end positions can then be stored in memory for fast retrieval. (Heule & Rothkrantz, 2007; van den Herik et al., 2002.)

The formal definition of convergence uses a concept called *conversion*. A conversion is a move that permanently changes the board so that for any configuration of pieces that could have existed before the conversion, the configuration can no longer occur after the conversion. A typical example of a conversion is the removal or addition of a piece. (Allis, 1994.)

Of the nine solved games mentioned before, Awari, Nine Men's Morris and Checkers are convergent. The rest are divergent, including Connect-4. It is worth noting that none of the solved games is unchangeable. Shogi is the only unchangeable game in Table 4. Shogi allows any removed piece to be put back on the board and therefore there are no conversions. PopOut is also unchangeable and this has a major impact on its solvability.

2.4.4 Sudden death

Sudden death threats exist in situations where one player is about to win the game with a single move ending the game prematurely (see Figure 3). The other player must quickly refute such a threat and thereby his move options are limited. If one player is able to create many sudden threats in succession so that they lead to a forced win for him, the analysis of the game position can be greatly simplified. Such sequences are called *winning threat sequences* and they were essential in solving at least Gomoku (see section 3.1.5). (Heule & Rothkrantz, 2007.)

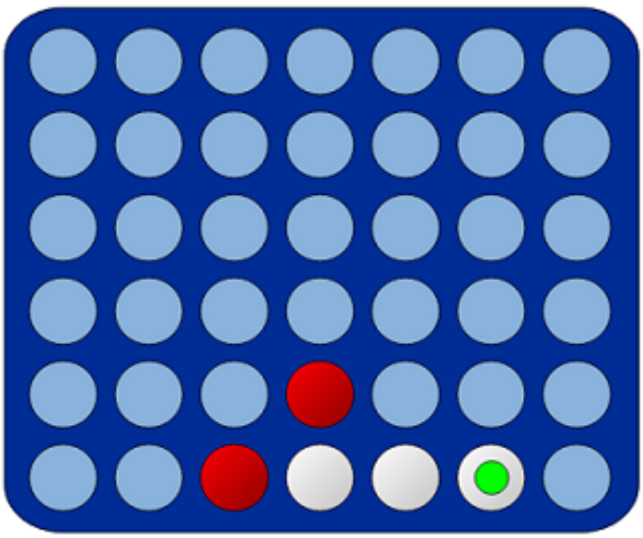

Figure 3. An example of a sudden threat in Connect-4.

Sudden death threats also exist in Connect-4. According to Allis (1994, p. 163), most games in Connect-4 end between moves 37 and 42 and the sudden threat property therefore does not seem to be very important in Connect-4. Heule and Rothkrantz (2007) have however listed sudden threat sequences as moderately useful in solving Connect-4.

2.4.5 Equivalent positions

In most games, it is possible that two different sequences of moves lead to positions that are either the same or effectively the same. There are three ways that this can happen. Either the move sequences are different but the positions are exactly the same in which case they are said to be transpositions. The positions could also be symmetrical. Figure 4 shows how the the left and right columns in Connect-4 must lead to the same result due to symmetry. The third way is to use some sort of generalization that proves that the positions have the same value even though they are not identical or symmetrical. In all three cases, such positions are called equivalent positions. (Heule & Rothkrantz, 2007.)

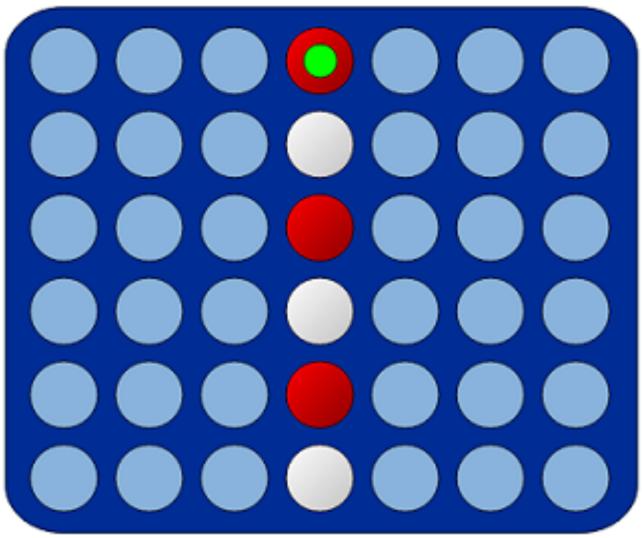

Figure 4. The left and right columns must have the same values due to symmetry.

2.4.6 Initiative, zugzwang and local endgames

The advantage of the initiative refers to the phenomenon that most games are either first-player wins or draws. This is explained by the fact that the first player usually has several alternatives for his first move. It is enough that only one of them leads to a win whereas if the game were a second-player win, all of them would need to lead to a loss for the first player. (Uiterwijk & van den Herik, 2000.)

Zugzwang originates in Chess and it refers to any situation where the player is forced to move even though he would be better off if he did not make a move at all. The concept of zugzwang is not applicable to most games but notably it was crucial when Allis solved Connect-4. He created nine strategic rules and many of them depended on zugzwang. Figure 5 shows an example of zugzwang in Connect-4. (Allis, 1988; Uiterwijk & van den Herik, 2000.)

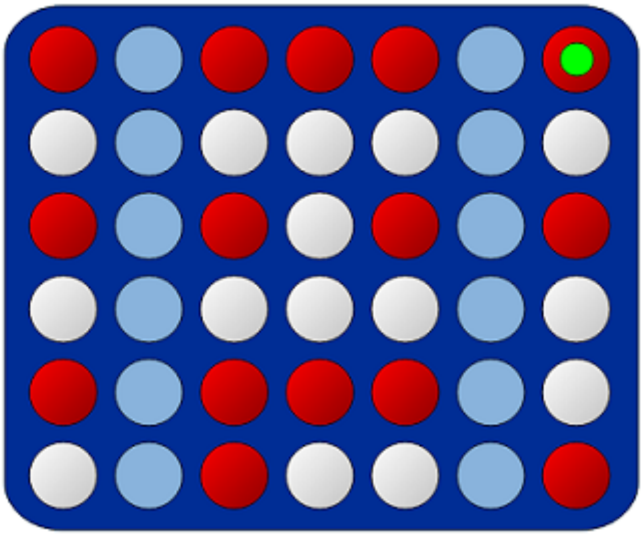

Figure 5. Zugzwang causes White to lose in this Connect-4 position.

Sometimes it is possible to ignore most of the board and analyze only some specific part of the board to determine the solution. Such situations are called local endgames. Allis (1988) relied on local endgames in the formulation of some of his strategic rules for Connect-4 (Heule & Rothkrantz, 2007). Figure 5 demonstrating zugzwang is also an example of a local endgame because everything above the lowest threats can be ignored.

2.5 Solvability

Van den Herik et al. (2002) analyzed the solvability of games in terms of their state-space and game-tree complexities. They did this by applying a *double dichotomy* (see Figure 6). They divided games into two groups based on whether they had a low or high state-space complexity. Then they did a similar division for game-tree complexity. This gives four categories in total. According to van den Herik et al. (2002), the solvability of a game depends on which category the game belongs to.

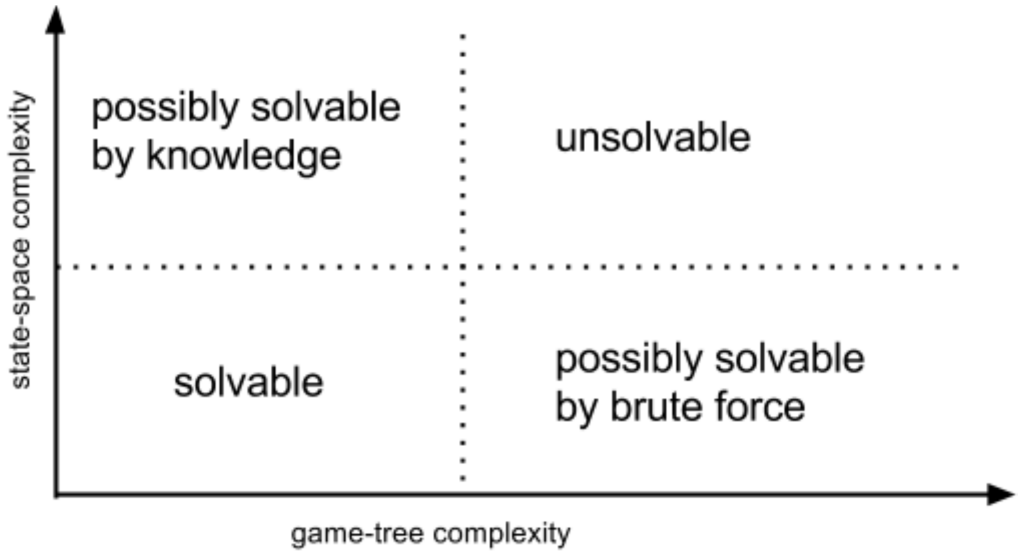

Figure 6. A double dichotomy by van den Herik et al. (2002, p. 279).

If the state-space complexity of a game is too high, it cannot be solved by brute-force methods but must be solved by knowledge-based methods. Conversely if the game-tree complexity is too high, it must be solved by brute-force methods instead of knowledge-based methods. If both are too high, the game cannot be solved. The difference between brute-force and knowledge-based methods is explained in chapter 3.

Van den Herik et al. (2002, p. 299) claimed that a low state-space complexity is more important than a low game-tree complexity when determining the solvability of a game. They estimated in 2002 that a game whose state-space complexity is below 3*10^12 could be solved by complete enumeration using computers of that time. Five years later, Heule and Rothkrantz (2007, p. 106) gave a more careful estimate of 10^10 for the same boundary.

Heule and Rothkrantz (2007) criticized the use of double dichotomy as being able to determine the solvability of games. They pointed out that the historical order in which games have been solved does not have any recognizable pattern in terms of state-space or game-tree complexities. They illustrate this with a diagram that shows the order in which games have been solved together with their complexities (see Figure 7).

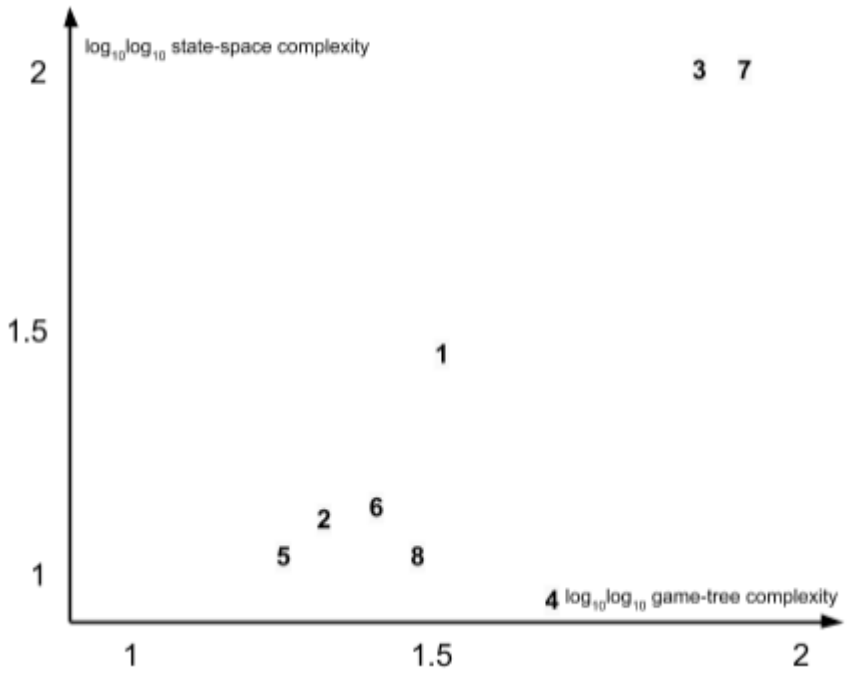

Figure 7. The historical order of solved games (Heule & Rothkrantz, 2007, p. 108).

As an alternative to the double dichotomy, Heule and Rothkrantz (2007, p. 117) posited that the solvability of games "could effectively be explained by the applicable solving procedures". The applicable solving procedures can in turn be determined by analyzing the game characteristics. The solving procedures reduce the solution size and if this reduction is effective enough, computers are able to cope with the whole solution size. (Heule & Rothkrantz, 2007.)

3. Search Techniques

Schaeffer and Plaat (2000, p. 4) defined three concepts that are used in designing game-playing programs. An *algorithm* traverses the game tree or the game graph and it forms the backbone of the program. *Enhancements* are added to the algorithm to improve the algorithm's performance. Finally the term *search technique* is used to refer to both algorithms and enhancements.

The search techniques can be divided into two classes: brute-force methods and knowledge-based methods. Brute-force methods use only information about the initial state, how to derive children states and how to recognize a terminal state (Korf, 1985). Knowledge-based methods take advantage of the structure of the game and their largest contribution often is a good move ordering and node selection (van den Herik et al., 2002).

This section has three parts. The first part introduces different search algorithms. The second part focuses on techniques that enhance the performance of the Alpha-beta search algorithm. The third part briefly explains how game-dependent knowledge rules could potentially be applied to PopOut.

3.1. Search algorithms

Board games are divided into categories which dictate the search algorithms that can be used. These algorithms have different characteristics and an analysis of these characteristics aids in selecting the best one among them. The most typical category for board games is the category of game trees. Another noteworthy category is based on *AND/OR* trees which can also be regarded as two-valued game trees. (Allis, 1994, p. 15.)

Search algorithms can traverse the nodes of a game tree in three different orders: depth-first, best-first, and breadth-first. Depth-first search always expands a child of the current node until a depth limit is reached or a solution is found. Depth-first search only has to store the current path in memory. A drawback is the need for a depth limit which can sometimes be completely arbitrary. Best-first search chooses the next node it expands according to some heuristic. The definition of *best* depends on the used heuristic. Best-first search has to keep all nodes in memory which is its biggest drawback. Breadth-first search expands all nodes at a certain depth before going deeper. (Allis, 1994; Korf, 1985.)

There are many search algorithms, but only three of them have been used to successfully solve games (Heule & Rothkrantz, 2007, p. 111). Alpha-beta is the most common one of these. The other two algorithms that have been able to solve games are conspiracy-number search and proof-number search. These two are closely related and conspiracy-number search can be said to be a "direct ancestor" of proof-number search (Allis, 1994, p. 60). More precisely, proof-number search uses the same underlying idea from conspiracy-number search but in a more refined form.

In this subsection we take a look at five different search algorithms or methods. Minimax is needed to understand Alpha-beta. Proof-number search comes after those two. Retrograde analysis, while not technically a search algorithm, comes fourth. Finally threat-space search and Monte-Carlo search are briefly explained because of their usage in the knowledge base. Both Alpha-beta and proof-number search are explained in greater detail because our implementation used them. Conspiracy-number search is skipped because proof-number search has practically replaced it.

3.1.1 Minimax

As mentioned in section 2.4.2, a game tree represents all possible lines of play from the initial position. All terminal nodes of a game tree can immediately be assigned some integer value that indicates the outcome of the game. Minimax is a depth-first algorithm that determines the game-theoretic value of the root node by investigating the game tree. It does this by calling itself recursively for each move. (Campbell & Marsland, 1983.)

When discussing the Minimax algorithm, it is customary to call the first player *Max* and the second player *Min*. The values of terminal nodes indicate how good the end positions are for Max. Due to the zero-sum property of the game, a position that is good for Max must be bad for Min. For each node, Max will always choose the value of that child node that has the greatest value. Min will conversely always try to choose the minimum value. This is enough to guarantee that the correct values will cascade all the way down to the root node.

Figure 8 shows how values propagate in Minimax when applied to PopOut 4x4. The value of each node is represented by the board color. A white board represents the value 1, a red board represents -1, and a yellow board represents 0. Max, which in this case is the white player, always prefers to choose the maximum value of these. The terminal nodes (i.e. the ones with no children) have been assigned a value by an evaluation function. The evaluation function checks if either player has won or if the game is drawn. White can choose to reject a draw in PopOut which happens once in Figure 8 but the exact details of how White knows to do so in the figure should be ignored.

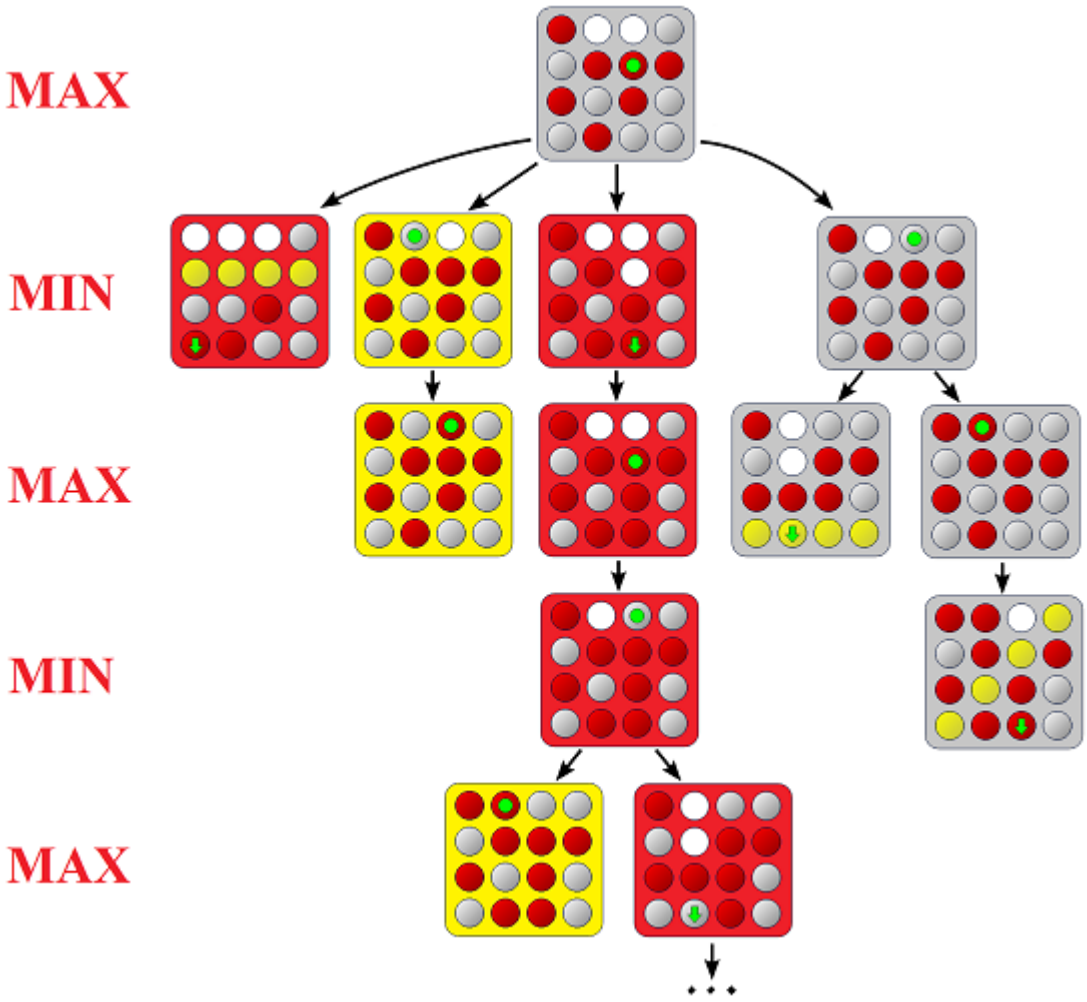

Figure 8. A game-tree of a PopOut 4x4 position and how Minimax values propagate (some moves neglected for simplicity).

There is a slightly different way of implementing Minimax called *Negamax*. In Negamax, the integer values assigned to nodes indicate how good the position is for the moving player, and not how good it is for Max. When Negamax is called recursively, the negative of the returned value is taken. This simplifies the implementation because Max and Min nodes can use the same logic. (Campbell & Marsland, 1983.)

3.1.2 Alpha-beta

The Alpha-beta algorithm is a brute-force method that cuts off subtrees that provably cannot affect the value of the root node in Minimax. Alpha-beta pruning has been used in game-playing programs since the 1950s and it is still the best-known algorithm for game trees. Several people have independently discovered it at least partially but a full analysis was first done by Knuth and Moore in 1975. Alpha-beta pruning reduces the number of nodes that have to be investigated in Minimax without any loss of information. Therefore it is able to give the same result as Minimax but takes less time to execute. (Allis 1994; Knuth & Moore, 1975.)

Alpha-beta passes two values to each recursive function call: alpha and beta. Alpha is the score that Max is guaranteed to get, and beta is the score that Min is guaranteed to get. When alpha and beta become equal, a cutoff can be made because the line of play cannot be optimal. A distinctive feature of Alpha-beta is its ability to make so-called deep cutoffs in addition to shallow cutoffs (Knuth & Moore, 1975).

The black board in Figure 9 is a node that has been pruned off by a deep cutoff. The nodes are evaluated from left to right and after investigating the second child of the root node, Max knows that it can get at least a draw. Later in the tree, Min also is guaranteed at least a draw and this causes a deep cutoff. The ordering of nodes is not optimal here; if Max had evaluated the fourth child first, the three other children and their subtrees would have been pruned off immediately. Notice how the third child has now been marked as a draw instead of a loss due to the cutoff.

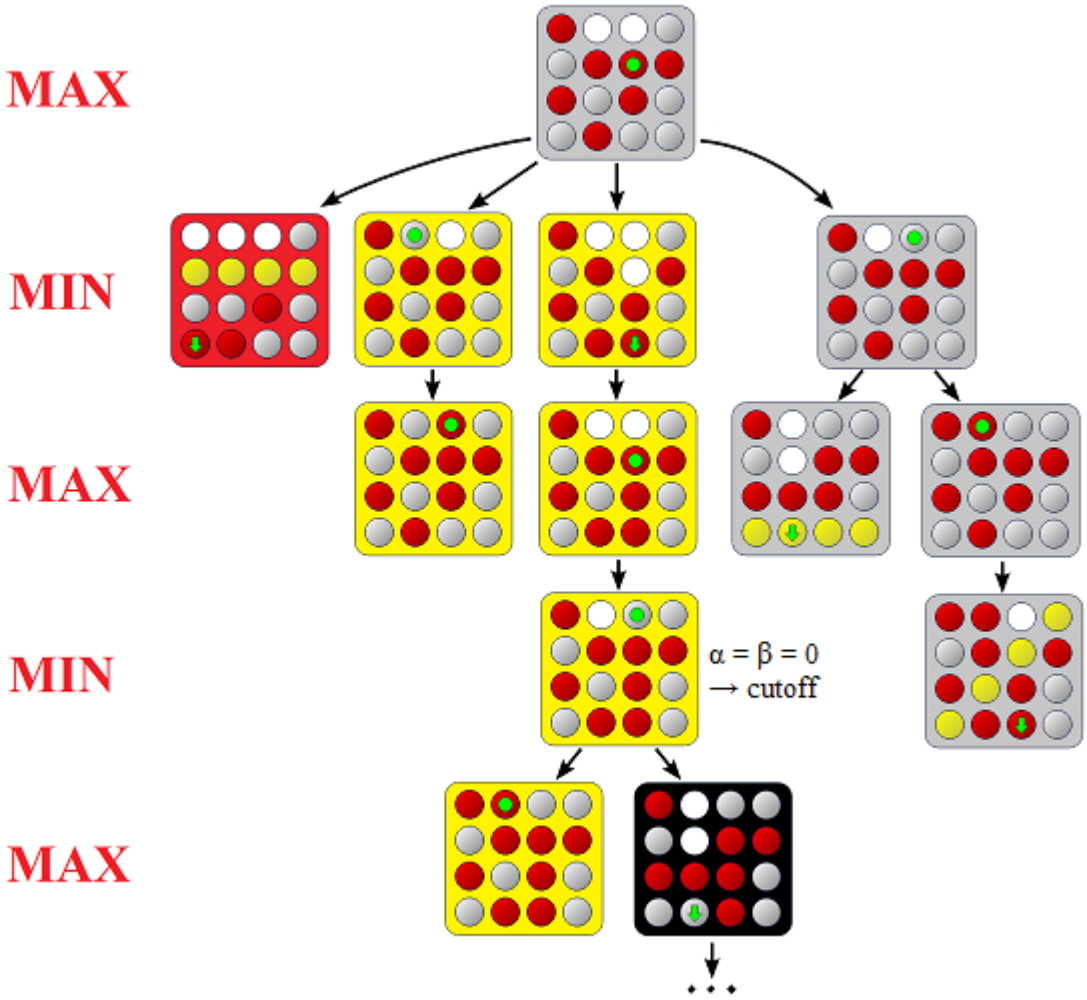

Figure 9. The same position with Alpha-beta pruning.

When Alpha-beta pruning works completely optimally, it produces a *minimal* game tree. The nodes investigated by Minimax constitute a *maximal* game tree. The number of nodes in the maximal tree is the square of nodes in the minimal tree (Knuth & Moore, 1975). For example, if the maximal tree has 10^12 nodes and it takes 24 hours for Minimax to determine the root value, the minimal tree then has 10^6 nodes and Alpha-beta can determine the root value in less than 10 milliseconds. This assumes that

both Minimax and Alpha-beta work at the same speed of a little over 10 million nodes per second. The Alpha-beta implementation in our study had a speed close to that value.

According to Schaeffer (2001, p. 30), the reason for the popularity of Alpha-beta can be attributed to Chess. Chess has historically been the main focus of computer game researchers and Alpha-beta has been especially suitable for Chess. Because of this research focus, numerous search enhancements have been developed for Alpha-beta. These search enhancements allow many programs to use Alpha-beta pruning close to its best-case performance.

3.1.3 Proof-number search

Proof-number search (PNS) is a knowledge-based method (van den Herik et al., 2002, p. 304) that has been especially designed for solving games. While Alpha-beta has been the traditional choice as the search algorithm, it has weaknesses especially in the endgame phase. Sometimes these weaknesses can be remedied by enhancements such as endgame databases but that is not always possible. Therefore the state-of-the-art implementations of many games use PNS nowadays. PNS also outperforms Alpha-beta in solving positions that require narrow but deep lines of play to be investigated. It is difficult to modify Alpha-beta to play well in such lines. (Kishimoto, Winands, Müller, & Saito, 2012; Saito, Winands, & van den Herik, 2010.)

Unlike Alpha-beta, PNS is not based on Minimax but instead uses an AND/OR tree. An AND/OR tree has two types of nodes: AND nodes and OR nodes. These nodes can be either evaluated or unevaluated. An evaluated value can be either true or false. When a node is true, it is said that the node has been proven. When it is false, the node has been disproven. The purpose of PNS is to find out whether the root node can be proven, i.e. whether its value is true or false. (Allis, 1994, p. 17.)

PNS attaches two numbers to each node in the tree: proof number and disproof number. The proof number indicates how many descendants have to be proven in order to prove the node itself. The disproof number conversely indicates the number of descendants that have to be disproven to disprove the node. PNS uses these two numbers to find the *most-proving node* and investigates it. PNS is a best-first search that uses the shape of the tree to find the next node to expand. Because of this, no domain-specific knowledge is needed to implement a move ordering. (Allis, 1994; van den Herik & Winands, 2008.)

The biggest disadvantage of PNS is its high memory consumption. Therefore a myriad of variants have been developed to reduce the memory requirements. Usually the memory reduction has come at the expense of speed. The earliest and fastest variant is PN^2. Four depth-first variants have also been developed, usually to correct shortcomings of the earlier variants. All variants are slower than PNS but they are able to traverse a much larger number of nodes before running out of memory. We briefly review PN^2 of these as an example of how PNS can be modified. (Kishimoto et al., 2012; van den Herik & Winands, 2008.)

PN^2 was proposed by Allis (1994) and it uses two levels of PNS. The first level uses another PNS to evaluate nodes. The nodes created by the second-level search are immediately deleted after evaluation. If the second-level search evaluates to true or

false, it does not need to be called again. If it evaluates to unknown, the proof and disproof numbers are updated and it may be necessary to rerun the second-level search. If PNS is able to investigate N nodes before running out of memory, PN^2 can investigate N^2 nodes. A naive implementation of PN^2 will take three times as much time to solve the problem but it can be made faster by gradually changing the size of the second-level search. (Allis, 1994; Breuker, 1998.)

PNS and its variants have been used in solving several games. Checkers is the largest game that has been solved and it used PNS as its forward search component (Schaeffer et al., 2007). Connect-4 has been solved by combining PNS and domain-specific knowledge (Allis, 1994). Additionally, PNS has been applied to at least Awari, Chess, Shogi and Go (van den Herik & Winands, 2008, p. 2).

3.1.4 Retrograde analysis

Retrograde analysis is a form of exhaustive search. Alpha-beta and PNS work by searching forward from the root state. Retrograde analysis differs from them by taking another direction. Instead of starting from the root state, retrograde analysis starts from the terminal states. Retrograde analysis can be used only if there is an efficient way to enumerate all terminal states. (Heule & Rothkrantz, 2007; Nievergelt, Gasser, Mäser, & Wirth, 1995.)

Retrograde analysis maps each state to a unique index in a database or some other memory storage. Ideally, a perfect hash function is used as the mapping but speed can often be gained by using a looser mapping that consumes some extra memory. Initially, all terminal states in the database are marked as solved and their game-theoretic values are stored. The predecessor states of solved states can now also be solved iteratively. If a state has at least one child that leads to a win for the current player, the state itself is also a win for that player, otherwise it is a loss or a draw. (Gasser, 1996.)

Retrograde analysis has been an essential technique in solving several games, including Nine Men's Morris, Awari and Checkers (Gasser, 1996; Romein & Bal, 2003; Schaeffer et al., 2007). Usually retrograde analysis is not the only search method applied but is used in combination with some forward-search algorithm. The program that solved Checkers used PNS as the forward search part and retrograde analysis as the backwards search part (Schaeffer et al., 2007). These two forms of search were able to meet up with each other at the middle of the game.

3.1.5 Threat-sequence search

Threat-sequence search is used when the opponent's replies are practically forced because of sequential sudden-death threats. This allows a two-player game to be regarded as single-agent search (van den Herik et al., 2002, p. 303) simplifying analysis considerably. Gomoku was solved by combining threat-space search, PNS and retrograde analysis (van den Herik et al., 2002). Figure 10 shows a threat sequence in PopOut consisting of eleven moves. White is able to make an immediate threat after each move, and Red has only one way of blocking the threat.

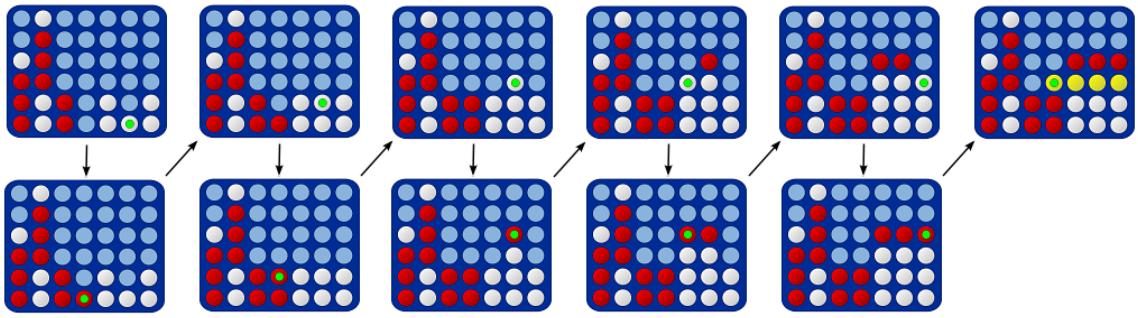

Figure 10. A threat sequence in Connect-4 and PopOut.

Threat-sequence search tries all possible threats. In addition to blindly testing all threat sequences, there are two more sophisticated threat-based searches. *Threat-space search* does not try all possible threats but selects only a subset of them. *Lambda search* adds the concept of *null* moves to threat-sequence search. Null moves are moves where the opponent passes his turn to move. Analyzing such null moves can reveal the possibility of exploiting non-immediate threats as well. (Heule & Rothkrantz, 2007; van den Herik et al., 2002.)

3.1.6 Monte-Carlo tree search

Monte-Carlo Tree search (MCTS) cannot solve games (Browne et al., 2012, p. 7) but we briefly discuss it here because it has been gaining a lot of attention among game researchers lately. MCTS is a best-first search algorithm that builds the game tree by performing randomized simulations. It is useful when writing a heuristic evaluation function is difficult or not done. The game length should also be reasonably limited. (Chaslot, 2010.)

When Chess programs surpassed the best human players, the interest in Chess and Alpha-beta search waned and much of the focus shifted to Go. MCTS has been surprisingly successful in playing Go and programs utilizing MCTS can compete with the best human players on smaller boards. Prior to MCTS, Go programs were on par with amateur human players on the same smaller boards. It is expected that the focus on MCTS only keeps growing during the coming decade. (Browne et al., 2012.)

We speculate that MCTS could possibly be applicable to very large Connect-4 or PopOut boards. If no domain-specific knowledge is used, Alpha-beta and PNS might fail to work. In such a case, these algorithms might be completely helpless against human players. MCTS on the other hand could still pose a respectable challenge against human players. To our knowledge, there has not been any MCTS implementation of Connect-4.

3.2. Alpha-beta enhancements

Even though the search algorithm forms the backbone and the most central part of the program, choosing the algorithm is usually straightforward (Hlynka & Schaeffer, 2006, p. 1; Schaeffer & Plaat, 2000, p. 4). Due to the ease of choosing the algorithm, most of the design focus is consequently on selecting the right enhancements that aid the algorithm and improve its efficiency. Enhancements work by taking advantage of

properties that exist in the search space. These properties can be either game-dependent or game-independent. (Schaeffer & Plaat, 2000.)

Because of the high number of different search enhancements and the fact that they can interfere with each other, choosing the right combination is time-consuming and difficult. Hlynka and Schaeffer (2006) tried to automate the selection of enhancements. Their proof-of-concept program greedily searched among different combinations of enhancements and adjusted their parameters in order to find the optimal set of enhancements. Their program provided promising results but it seems that there has not been further development of automated search selection. The task of choosing enhancements thus remains manual. (Hlynka & Schaeffer, 2006.)

Enhancements that are based on Alpha-beta pruning can have one of the following four purposes. They can change the order in which moves are selected making the best move more likely to be analyzed first. They can dynamically change the search depth. They can also change the alpha or beta bounds in an attempt to reduce the search effort. Finally, they can utilize memory to cache repeated positions. (Hlynka & Schaeffer, 2006.)

3.2.1 Transposition tables

A transposition table caches repeated positions and it is the most important search enhancement for Alpha-beta. According to Kishimoto and Schaeffer (2002), utilizing a transposition table in a Chess program is equal to a tenfold increase in performance. Any efficient search algorithm based on Alpha-beta almost necessarily has a transposition table. The details of how the transposition table should be implemented depend on the domain. (Kishimoto & Schaeffer, 2002.)

A transposition table is a hash table that stores calculated search values for game states. Because the same state can appear many times in the game tree and all such states have the same value, ideally the value needs to be calculated only for one node and the other nodes can get the value from the transposition table. In some cases, a transposition table can even enable a tree that is smaller than the minimal Alpha-beta tree, but usually the reduction is 99% of possible reductions in the maximal tree if a good move ordering is also used (Schaeffer, 1989).

A transposition table usually cannot store all found states due to memory constraints. This means that some found states cannot exist simultaneously in the table. Each state can be associated with an *index* that points to a slot in the transposition table. Several states can have the same index and a *key* is used to distinguish between states with the same index. The index and the key can be used to uniquely identify the state. When a state is to be saved in the transposition table, its index is used to find the correct slot. The key and the score are then saved in that slot along with other information. The other information usually contains at least the work that went into calculating the score. The work can be expressed as either the depth or the number of nodes in the subtree. (Breuker, Uiterwijk, & van den Herik, 1994.)

Due to the fact that some states can map to the same slot in the transposition table, a mechanism is needed to resolve collisions. In other types of hash tables, it may be possible to retain all values with the same index by associating a linked list with every

slot. The linked list can grow indefinitely when new values are appended to it. While it is possible to use linked lists in the transposition table as well, the more logical choice is to have a fixed number of entries per slot (Breuker, 1998, p. 19).

A *replacement scheme* is used to handle collisions in transposition tables. The replacement scheme dictates whether to discard or keep the old value when a collision happens. Breuker at al. (1994) identified and empirically compared seven different replacement schemes (see Table 5). They found that two-level transposition tables outperform one-level tables. In a two-level transposition table, each slot has room for two positions instead of one. The disadvantage is that the number of slots has to be halved. However, experiments have shown that doubling the transposition size reduces the number of investigated nodes by only 5% on average. (Breuker et al., 1994.)

Table 5. Replacement schemes by Breuker et al. (1994).

Name	**Collision resolution**
TwoBig1	Two entries per slot. Keep the new node and the node with the larger subtree.
TwoDeep	Two entries per slot. Keep the new node and the node with the deeper subtree.
Big1	Keep the node with the larger subtree. If the value of a node in the subtree was fetched from the transposition table, it counts only as one node.
BigAll	Keep the node with the larger subtree. If the value of a node in the subtree was fetched from the transposition table, all nodes of the node's subtree are counted.
Deep	Keep the node with the deeper subtree.
New	Always replace the old node.
Old	Always keep the old node.

3.2.2 Move ordering

The minimal tree is achieved if interior nodes are evaluated in the best possible order so that the largest number of Alpha-beta cutoffs can be made. The maximal tree is achieved if the ordering is the worst possible. Because of the enormous size difference, a good move ordering is essential. A perfect move ordering cannot be reliably achieved. If it were possible, Alpha-beta search would be unnecessary because the result of the move ordering function could be used instead.

Most game-playing programs have traditionally used domain-specific knowledge for move ordering. The *history heuristic* is an Alpha-beta enhancement that does not require domain-specific knowledge. It is in fact able to approximate domain-specific move ordering by utilizing the implicit experience it gains from visiting different parts of the tree. Schaeffer (1989) evaluated different search enhancement combinations. He discovered that a transposition table and the history heuristic together were able obtain

99 percent of all possible reductions in the tree size. The other search enhancements he tried had an insignificant impact. (Schaeffer, 1989.)

The history heuristic orders moves by how good they have been in earlier searches. Each move has an associated score and the move with the highest score is tested first. A move is said to be sufficient either if it is the best move or if it causes an Alpha-beta cutoff. Whenever a move is found to be sufficient, the history score of that move is increased. The score increase is usually weighted based on depth. The weight with the best experimental results has been two to the power of the subtree depth. (Schaeffer, 1989.)

Iterative deepening is another way of ordering moves. It also allows the search to be terminated at any point without harming the quality of the search too greatly. Iterative deepening imitates breadth-first search by making many depth-first searches and by increasing the depth limit in each iteration (Reinefeld & Marsland, 1994). The results of earlier iterations can be used to order the moves in later iterations. (Korf, 1985.)

3.2.3 Distributed and parallel search

Most modern computers have multi-core processors. To utilize all cores, the search has to be done in parallel instead of sequentially. A similar situation happens when the search is distributed among several computers. Alpha-beta cannot do parallel search without modifications. Unfortunately, it is notoriously hard to modify Alpha-beta to do parallel search reasonably well (Kishimoto & Schaeffer, 2002).

There are two main causes of overhead when using parallel Alpha-beta search: *search overhead* and *communication overhead*. Search overhead occurs as a result of performing search that sequential Alpha-beta could have avoided. For example, if sequential Alpha-beta is capable of pruning some subtree, the parallel version might not notice the pruning opportunity if the information required for that is scattered among multiple processors. Communication overhead is caused when processors must communicate with each other in order to share their results and organize search activities. (Kishimoto & Schaeffer, 2002; Marsland & Popowich, 1985.)

Work division has to be done before starting distributed or parallel search. Usually the search space is divided into independent positions so that each processor can complete its search in isolation and simply communicate the results after finishing. Traditionally, each processor has a list of work that it has to perform. If one processor finishes before others, it can *steal work* from the other processors. A modern alternative to work stealing is so-called *transposition-driven scheduling* (TDS). TDS allows communication to be done asynchronously and removes the difficulty of sharing the transposition table. This is done by making the transposition table responsible for work scheduling. (Romein, Bal, Schaeffer, & Plaat, 2002.)

When using work stealing instead of TDS, one of three strategies has to be used when Alpha-beta search is used with a transposition table. The simplest strategy is to assign a local transposition table to each processor. This avoids communication overhead but causes search overhead if work could have been saved by having access to another processor's table. The second strategy is to replicate the same table among all processors. When one processor updates the table, it sends a notification to all other processors. Reading on the other hand causes no communication overhead. The third

strategy uses one table where each processor is responsible for some part of the table. If a processor wants to access or update a partition that does not belong to it, it must communicate with the owner. (Kishimoto & Schaeffer, 2002.)

3.3. Game-dependent knowledge rules

The search techniques can further be improved by incorporating game-dependent knowledge rules or expert rules. We have used no expert rules in our implementation but they still merit some discussion because they were essential when Connect-4 was solved for the first time. Therefore they could also potentially have a significant impact in PopOut.

Expert rules fall under knowledge-based methods. Together with PNS and threat-space search, they take advantage of the structure of the game. In order to apply these methods effectively, good knowledge of the game is required. Expert rules are made by hand and they have been used to achieve a better move ordering and to identify won positions long before reaching a terminal node. (van den Herik et al., 2002.)

When Allis solved Connect-4 in 1988, the formulation of several easy-to-understand expert rules was enough to solve the game (van den Herik et al., 2002, p. 305). There are sixty-nine ways for one player to create a four-in-row on the standard 7x6 board. Allis (1988) formulated and informally proved nine strategic rules that could be used to show that a player could be prevented from completing any of his remaining potential four-in-rows. The interaction of the nine rules was carefully analyzed to determine which rules could be used together and which combinations would make other rules invalid. Figure 11 is an example position where disks that cannot be part of a four-in-row are shown in a darker tone.

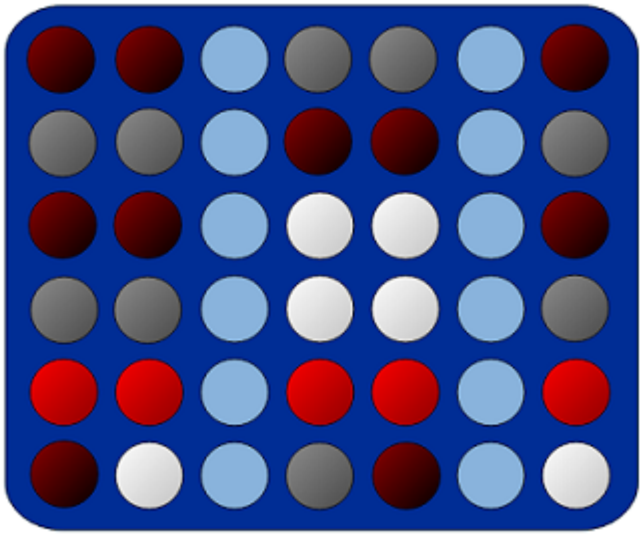

Figure 11. Most of the disks cannot be part of a four-in-row in this Connect-4 position.

Many of Allis's rules depend on zugzwang. Zugzwang also exists in PopOut albeit in a more limited form. Figure 11 is a position that is a loss for White in Connect-4 due to zugzwang but a win for White in PopOut. In general, pop moves allow a player to avoid zugzwang by breaking the opponent's threat. If similar expert rules were used in PopOut, they would most likely need a major reformulation together with the addition of new rules.

4. Research Problem and Method

James D. Allen, the first person to solve Connect-4 in 1988, writes in his recent book (Allen, 2010, p. 229): "As far as I know, there has not yet been a complete computer analysis of PopOut." The book was published four years prior to this research and the lack of computer analysis on PopOut seemed to remain as of 2014. The quoted situation acted as the motivation for this research. We now describe the research questions and methodology used in our study.

4.1 Research questions

Since no one had tried to solve the game, it was not known if there would be any obstacles that would have rendered commonly used techniques unusable. The first research question was therefore whether the same search techniques that were used in solving Connect-4 would also be applicable to PopOut and how they should be modified or augmented if they are not. PopOut has different game characteristics because of the rule changes, and it is not fully understood why some games have been solved and others have not been (Heule & Rothkrantz, 2007). Any difference between the applicable techniques in Connect-4 and PopOut might help to clarify the relationship between solvability and game characteristics.

Trying every possible technique that has been used in solving Connect-4 is of course hardly possible. We therefore selected techniques only from such reported solutions that we personally deemed to be seminal or otherwise important. We believe the selected solutions to be representative of how programs usually solve Connect-4. More details about the selected solutions are given in section 5.2. Another limitation is that we avoided using expert rules and focused only on search techniques and their enhancements.

A second research question was also formulated. It asked what the game-theoretic value of PopOut is. This was not an independent question but the two research questions can be regarded as two sides of the same coin. The second research question clarifies and acts as a reminder for what it means for a search technique to be applicable. If certain search techniques are successfully applied to a game, they will necessarily also yield the game-theoretic value of the game.

RQ1: How should the search techniques that were used in solving Connect-4 be modified or augmented in order to be applicable to PopOut?

RQ2: What is the game-theoretic value of PopOut?

There are two audiences that might be interested in the results of this study: the AI research community and board-game enthusiasts. People who play the game for fun are probably more interested in RQ2 whereas RQ1 is expected to be more useful for AI researchers.

4.2 Research method

This study was carried out as design science research (DSR). DSR is a research paradigm that studies the artificial world rather than the natural world. The researcher investigates something that does not exist naturally but is created by humans. The researcher first builds an artifact and then evaluates how well the artifact fulfills its purpose. The evaluation is used as feedback in another build phase and the cycles are repeated until the research question is satisfactorily addressed. New knowledge is then produced through the creation of the artifact. It is important to notice that the artifact is not the goal itself but a means to the goal. The goal is to produce new knowledge. (Hevner & Chatterjee, 2010.)

The game-theoretical value of PopOut is a mathematical fact and DSR, being an empirical approach, arguably cannot be applied to mathematics as such. However, the game-theoretical value cannot be calculated by hand but requires computer assistance. The research can therefore be regarded as a computer-assisted proof where the focus is not on the mathematical aspect of the proof but on the software and algorithms that enable the proof to be generated.

The performance of an AI program can be empirically investigated by defining independent and dependent variables (Simon, 1995, pp. 99-101). The independent variables in our study were the search techniques that were being applied. The dependent variables were the execution speed and memory consumption of the program. The definition of solving requires the program to run within reasonable resources, i.e. fast enough and without consuming too much memory (Allis, 1994). These two metrics consequently provided an empirical way to measure how applicable different combinations of search techniques are.

4.3 DSR guidelines

Hevner and Chatterjee (2010) proposed seven guidelines that good DSR should follow. They also compiled a checklist that is meant to ensure that none of the key aspects of DSR is neglected. The items of the checklist embody the guidelines although there is no one-to-one correspondence between the checklist and the guidelines. Table 6 provides a summary of the guidelines in the form of Hevner's and Chatterjee's checklist. The rest of this section describes all of the seven guidelines in detail one by one.

Table 6. DSR Checklist by Hevner and Chatterjee (2010, p. 20).

Questions	**Answers**
1. What is the research question (design requirements)?	How should the search techniques that were used in solving Connect-4 be modified or augmented in order to be applicable to PopOut?
2. What is the artifact? How is the artifact represented?	A method that can weakly solve the game. The method is represented as a description of used techniques and solutions to encountered problems.
3. What design processes (search heuristics) will be used to build the artifact?	Techniques used in solving Connect-4 and other related games are applied to PopOut until a satisfactory design is reached.

4. How are the artifact and the design processes grounded by the knowledge base? What, if any, theories support the artifact design and the design process?	The applied techniques come from the existing knowledge base. The design process uses Allis's (1994) description of AI problem solving.
5. What evaluations are performed during the internal design cycles? What design improvements are identified during each design cycle?	Execution speed, memory usage, and number of nodes investigated by the search algorithm are observed. Improvements are chosen to reduce execution speed and memory usage.
6. How is the artifact introduced into the application environment and how is it field tested? What metrics are used to demonstrate artifact utility and improvement over previous artifacts?	The final utility of the artifact is tested by its ability to solve the game. No previous attempts to solve PopOut exist as far as we know.
7. What new knowledge is added to the knowledge base and in what form (e.g., peer-reviewed literature, meta-artifacts, new theory, new method)?	The method embodies a list of applicable search techniques which helps in understanding the solvability of games. The game-theoretic value of PopOut is also added.
8. Has the research question been satisfactorily addressed?	Yes. A list of applicable search techniques has been found and given as a method.

The first guideline states that the result of DSR should be an artifact. There are four different types of artifacts: constructs, models, methods, and instantiations. Constructs constitute the vocabulary that the research community uses to describe things that it finds interesting. Models form relationships between the constructs. A method performs some task by dividing it into several steps. A method can be an algorithm or a guideline. Finally an instantiation demonstrates that a model or a method is feasible and effective by realizing it. (March & Smith, 1995.)

The artifact of this research was a method that described what steps were used to build a program that was capable of solving the game of PopOut. These steps included the search techniques that were applied and how encountered problems were fixed. Following the steps should result in a similar program that is also capable of solving PopOut. The method was formulated on a high level and it did not specify many of the implementation details, such as which programming language to use.

The second guideline requires that the research is relevant to its respective community. In the context of information systems, the research addresses a problem that occurs as a result of "the interaction of people, organizations, and information technology" (Hevner, March, Park, & Ram, 2004, p. 84). Our research was intended for the AI community which is closer to computer science than information systems and therefore the organizational aspect is not as relevant. Any research in IT must however address some issue that is valuable to practitioners and not merely of theoretical importance (March & Smith, 1995). The artifact of this research should provide guidance for practitioners who try to solve similar games or problems. The research community benefits from knowing how applicable the existing techniques are and if there is need for further research.

Guideline 3 involves evaluation of the artifact. Evaluation is an essential part of DSR because it is through evaluation that the utility of the artifact can be demonstrated. It also provides feedback that can be used to plan and carry out new iterations. The evaluation is done with methods and metrics taken from the knowledge base. If the knowledge base does not contain feasible evaluation methods, the evaluation can also be performed descriptively. (Hevner et al., 2004.)

Method artifacts can be evaluated in terms of whether they are able to perform the task, how efficient they are, and how generalizable and easy-to-use they are (March & Smith, 1995, p. 261). Our artifact was a method that was required to be able to solve the game. The most important metric therefore had to indicate if the method achieved the solution. The definition of solving requires that it be done within "reasonable resources" (Allis, 1994, p. 8). The completeness of the method is therefore intertwined with its efficiency in terms of execution speed and memory usage. It is widely known within computer science that methods and models may work on paper but fail to work when tested in a real environment, and only instantiations can provide the ultimate proof (March & Smith, 1995, p. 260).

We accordingly included a secondary artifact whose purpose was to prove that the method works. This secondary artifact was an instantiation and it was used as a proxy to evaluate the method indirectly. By measuring how much memory the instantiation used and how long it took to run, we could measure the completeness and efficiency of the primary artifact. To prove the completeness of the method, the instantiation was required to finish solving within a couple of days on a modern desktop computer using no more than 8 GB of physical memory. The instantiation was implemented in C++. The relative efficiencies of different iterations were also compared by counting how many nodes the search algorithm had to investigate. Metrics based on the number of nodes are widely used within AI (Schaeffer, 1989, p. 13) and because they are less dependent on the implementation, they can be used to measure two iterations of the method itself and not only the instantiation.

Guideline 4 emphasizes the importance of clear research contributions. The research contribution is what distinguishes DSR from routine design. Routine design only applies state-of-the-art knowledge and tends to avoid taking risks. This risk aversion usually precludes it from producing new knowledge because everything that is unknown is avoided in order to reduce risks. DSR on the other hand is easily identified by its focus on the unknown. (Vaishnavi & Kuechler, 2004.)

The applicability of different search techniques was the primary contribution of this research. The search techniques included one novel technique that had not been used in other board games. Even though there are many different search techniques, the number of solved games is not very high. Any new knowledge about search techniques and their effect on solvability should be valuable. To our knowledge, this is the first reported attempt to solve the game of PopOut. A secondary contribution was the game-theoretic values for several board sizes.

Guideline 5 involves research rigor. Rigor comes from the utilization of the knowledge base, which provides the foundations and the methodologies for constructing and evaluating the artifact (Hevner et al., 2004, p. 88). In this research, the applied search techniques came from the knowledge base. The construction was modeled after Allis's (1994) description of problem solving in AI. The metrics that were based on node

counting are also commonly used in AI, as already mentioned. We have tried to utilize the knowledge base whenever possible but we are disqualified from judging how well this was ultimately done.

Guideline 6 describes design as a search process. According to Hevner et al. (2004), the search process determines how the solution is obtained and it usually requires the use of heuristics to decide what to try next. They further remark that the used heuristics do not always lead to the best solution but only to a solution that is satisfactory. We have looked at how similar games, especially Connect-4, have been solved when choosing the search techniques. This reliance on past solutions of Connect-4 might have led us to a suboptimal way of solving PopOut.

The seventh guideline says that the research must be communicated to both technology-oriented and management-oriented audiences. The management-oriented audience refers to people responsible for leading an organization. DSR as described by Hevner and Chatterjee is tailored for information systems and this explains their focus on organizational aspects which is inapplicable in our case. The technology-oriented audience however exists. They need sufficiently detailed information about how the artifact was built so that they can reproduce the implementation. Chapter 5 is devoted to the description of the build process. (Hevner et al., 2004.)

5. Build

The build phase in design science research is performed in iterations. Each iteration consists of constructing or improving the artifact and performing evaluation. The result of each iteration is a design alternative. The evaluation of the design alternative is used as feedback in the next iteration. Iterations are continued until a satisfactory design is found. (Hevner & Chatterjee, 2010.)

Problem solving in AI includes two distinct phases: knowledge representation and searching. Knowledge representation refers to how the problem is analyzed, conceptualized and formalized. According to Allis (1994, p. 14), knowledge representation is the more important part of problem solving because in the most extreme cases, a good knowledge representation can eliminate the need for search altogether. This can happen, for example, if the representation makes it clear that some property must remain invariant. Usually instead of eliminating the search altogether, the knowledge representation only manages to reduce the state space and helps the search to be done more efficiently in terms of execution time and memory usage. (Allis, 1994.)

The implementation of a game-playing program can be considered to have three components. A move generator determines the allowed moves for the current position. An evaluation function assigns a value to each position. This value indicates how good the position is for one player and thus also for the other player due to the zero-sum property. The search algorithm with all its enhancements form the third part. (Lorenz & Tscheuschner, 2006; Marsland, 1986.)

As was seen in chapter 3, there are several ways to implement the search part. We have experimented with three different ways of searching: Alpha-beta, PNS, and retrograde analysis. Alpha-beta search received most of the development focus. PNS was implemented in its original non-variant form. Retrograde analysis was implemented as a means to strongly solve the smaller boards. The move generation and the terminal evaluation function were the same in all three search implementations.

This section first provides a knowledge representation that fulfills the rules of PopOut. The rest of the section concerns the construction of the search algorithm. The three different search approaches that were experimented with and the problems that were encountered with them are described. The sections are ordered mostly chronologically in an attempt to reflect the different iterations although the three search approaches are virtually independent of each other.

5.1 Bitboard representation

Before the construction of the algorithm could be initiated, we had to develop a way of representing the board. This phase belongs to the above-mentioned knowledge representation and it is therefore an important factor when it comes to execution speed and memory usage. There are especially two properties that a good Connect-4 board

representation should have. It should use only a minimum amount of memory and it should be able to efficiently discover when there are four disks in a row.

A naive solution would be to use a two-dimensional array where the dimensions are the width and the height of the board. A single cell in the array could then assume three values: empty, occupied by White, and occupied by Red. This solution however wastes memory and a loop is required to find any four-in-rows. We have instead used a technique called *bitboards*. The bitboard technique is widely used in Chess and other board games.

A bitboard is an unsigned integer where the bits in its binary format have some kind of semantic meaning. Using a bitboard provides two advantages. The space complexity is reduced because every bit can be assigned some meaning. The second advantage is in the reduction of time complexity. Bitwise operations are fast and they sometimes allow actions to be performed in parallel. The same actions with a non-bitboard encoding might have to be performed sequentially. (San Segundo, Galan, Matia, Rodriguez-Losada, & Jimenez, 2006.)

The Connect-4 bitboard representation we selected for our algorithm has previously been used at least in Tromp's Fhourstones implementation. Tromp (2010) claims that it is the best way to represent Connect-4 bitboards. It is not clear if Tromp was the first person to invent the representation or if someone else has used it prior to him. We will now describe this bitboard representation with required accommodations for PopOut.

There are two unsigned 64-bit integers, one for each player. Each bit in the integer maps to a certain cell of the board. The first six bits map to the six cells of the first column. After each column, an extra bit is reserved. These extra bits are used to make sure that bits do not overflow from one column to another when performing certain bit operations. In total, 49 bits are needed to encode the disks for one player in the standard board size 7x6. If the player's bitboard has 1 at some bit position, it signifies that he has a disk in the corresponding board cell. If neither has 1 at some bit position, the cell is empty. It is an error for both players to have the same bit position set to 1. See Figure 12 for how a Connect-4 board is encoded as two integers. The least significant bit is the one in the bottom left corner.

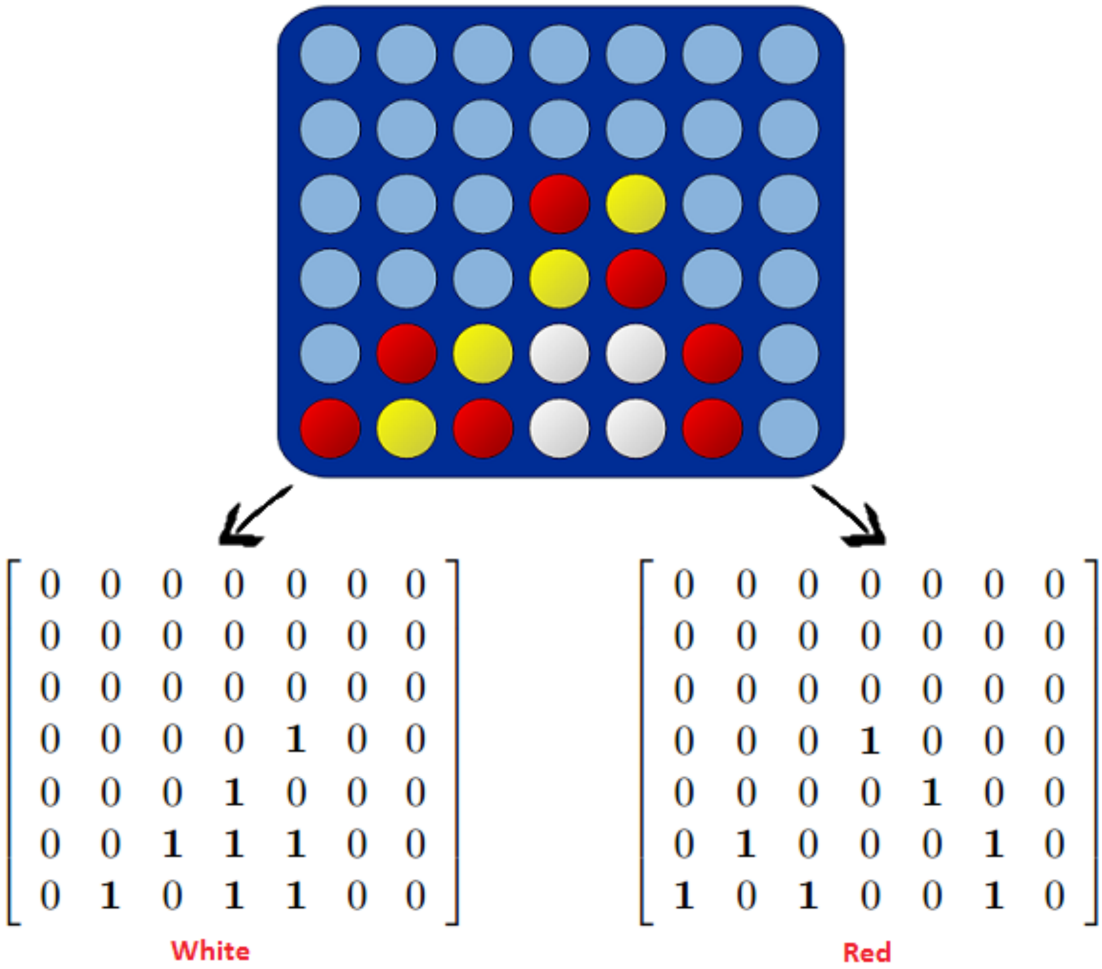

Figure 12. Bitboard encoding for Connect-4 and PopOut.

Using this bitboard encoding, it takes only 4 bit operations to check if there is a four-in-row horizontally anywhere in the player's bitboard. These four bit operations check all six rows in parallel. Analogously, it takes 4 bit operations for vertical connections and 4 for each type of diagonal connections. In total, it takes 16 bit operations to check if a player has a four-in-row anywhere in the board in any direction. The C++ implementation below shows how to check all directions for one player. There is a win only if at least one of the bits remains set after the four operations. Figure 13 further illustrates how the bits behave when the slash (/) diagonal is checked.

```
//bitboard is defined as unsigned 64-bit integer

bool hasWon(bitboard board) {
      bitboard horizontal = board & (board >> HEIGHT + 1);
      horizontal &= horizontal >> 2 * (HEIGHT + 1);

      bitboard vertical = board & (board >> 1);
      vertical &= vertical >> 2;

      bitboard slash = board & (board >> HEIGHT + 2);
      slash &= slash >> 2 * (HEIGHT + 2);

      bitboard backslash = board & (board >> HEIGHT);
      backslash &= backslash >> 2 * HEIGHT;
```

```
        return horizontal | vertical | slash | backslash;
}
```

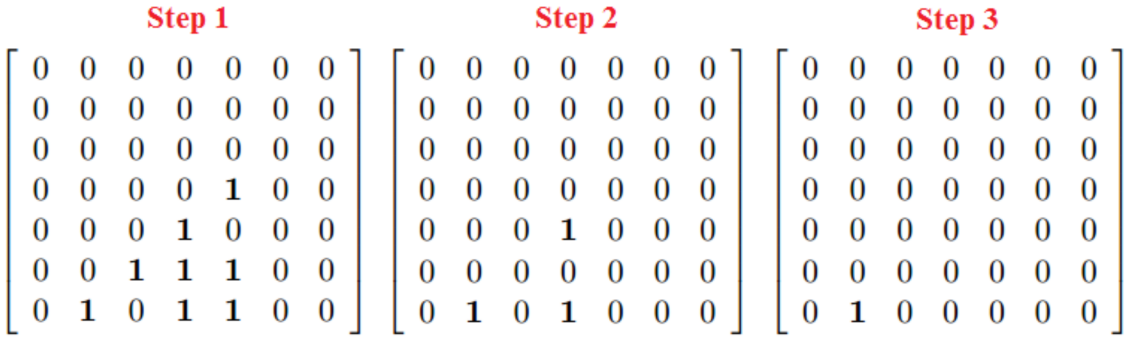

Figure 13. Checking for a win diagonally (/).

When making a move, there are two steps that need to be performed. The legality of the move has to be confirmed and the bitboards have to be updated to reflect the move. In the case of a drop move, the height of the column has to be computed first. The column and its height are then used to create a bitboard where only the bit corresponding to the new disk is set. The player's bitboard is combined with this new bitboard through an OR-operation. If the resulting bitboard has no bits set in the reserved top row, the move is legal.

Making a pop move is slightly more complex than making a drop move because several bits may need to be adjusted. The legality of the move is confirmed by checking that the player has a disk in the bottom cell of the column. To update the bitboards, bit masks are used to shift only the disks belonging to the column. The number of required bit operations is several times larger than in a drop move but due to the fast nature of bit operations, it is unlikely to be a bottleneck

Using two 64-bit integers is optimal when it is necessary to check for four-in-rows. It is however not the most optimal encoding memory-wise; the two integers can be combined into one 64-bit integer to uniquely encode any position. In order to achieve the memory-efficient encoding, we first create a bitboard having all bottom cells set to 1. One player's bitboard is added to that bitboard twice while the other player's bitboard is added only once. The resulting bitboard has zeroes for all empty cells except for the lowest empty cell. This allows us to find the height of each column because the bit for the lowest empty cell effectively marks the height of the column. After the height-marking bit, zeroes and ones are used to distinguish between the two players' disks and there is no need to mark further empty cells because they cannot occur.

5.2 Initial analysis

Before setting out to construct the first iteration of the search algorithm, we decided to estimate the state-space and game-tree complexities of PopOut in order to get an idea of the amount of computing resources required to solve it. Every legal position and legal move in Connect-4 is also legal in PopOut. Therefore at least the state-space complexity has to be greater than that of Connect-4. The number of nodes in the game tree is also

higher, but this does not guarantee that the game-tree complexity is higher (see section 2.4.2).

It has been calculated that the state-space complexity of Connect-4 is exactly 4,531,985,219,092 (Edelkamp & Kissmann, 2008; Tromp, 2010) or approximately 10^12.7. This must be a lower bound for PopOut. In order to get a reasonable upper bound, we first observe that there are seven possibilities for the height of a single column. Each occupied cell can be one of two values. For each height, the number of possibilities is consequently 2^height. Summing these heights for a single column, we get (2^7 - 1) possibilities for one column. There are seven columns in total so one reasonable upper bound is (2^7 - 1)^7 which is around 10^14.5.

Heule and Rothkrantz (2007, p. 106) estimated in 2007 that a game whose state-space complexity is around 10^10 can be solved by "elementary solving procedures that are applicable to all games". To account for Moore's law, we adjust this estimate to around 10^11. The calculated lower and upper bounds are still higher than that. Heule and Rothkrantz do not clearly define what they mean by elementary solving procedures but unenhanced Alpha-beta presumably falls into that category. The solvability of PopOut with help of Alpha-beta search then depends on how much the different enhancements can compensate for the excess size of the state space.

In addition to estimating the complexities, knowing which techniques have been used in solving Connect-4 and other related games can be used to guide the implementation and to get a clearer idea of what to expect. Connect-4 has been solved by several authors and programs. Here we will briefly summarize the information we could publicly find about the different ways of solving Connect-4 (see Table 7). After that, the solution of Checkers is also described because Checkers is the most recent well-known game to be solved and it shares some characteristics with PopOut that do not exist in Connect-4.

Table 7. Techniques used in solving Connect-4.

Author	**Year**	**Used techniques**
James Allen	1988	Alpha-beta, transposition table, killer-move heuristic, expert knowledge (van den Herik et al., 2002, p. 288)
Victor Allis	1988	Conspiracy-number search, nine strategic rules (Allis, 1988)
John Tromp	1993	Alpha-beta, transposition table, history heuristic (Tromp, 2010)
Victor Allis	1994	Proof-number search, strategic rules (Allis, 1994)

Allis (1988) weakly solved the game by informally proving nine domain-specific rules for Connect-4. Allen also used some amount of expert knowledge in solving the game and both authors needed about 300–350 hours to run their calculations on a Sun-4 workstation (van den Herik, 2002, p. 288). Tromp was able to solve Connect-4 with no expert knowledge at all, which shows how well the history heuristic is able to make up for the lack of domain-specific knowledge. Allis (1994, p. 163) mentioned having applied PNS to Connect-4. According to him, combining strategic rules and PNS solved the game within 25 hours on comparable hardware that he had used in 1988.

Based on these identified solutions, we chose two search algorithms: Alpha-beta and PNS. Conspiracy-number search was neglected because PNS uses the same idea and PNS is more suitable for solving games than what conspiracy-number search is (Allis, 1994, p. 61). The chosen enhancements were a transposition table and the history heuristic. Schaeffer (1989, p. 10) describes the killer heuristic as "just a special case of the history heuristic". It was therefore not implemented either. We did not try to apply any of the expert rules.

Checkers is the most complex board game that has been solved so far. Solving it took almost two decades of effort. The state-space complexity of Checkers is around 5*10^20, i.e. a million times larger than the upper bound we estimated for PopOut. The program that weakly solved Checkers had three different components: endgame database, proof-tree manager, and proof solver. (Schaeffer et al., 2007.)

The endgame database was constructed using retrograde analysis. The proof-tree manager used PNS that can prioritize the moves to be searched next. The proof solver evaluated the position in three different strengths: proven, partially proven, and a heuristic value. If the evaluated value was either partially solved or a heuristic value, the proof-tree manager could choose to reconsider the position depending on if it was necessary to do so in order to attain the proof. (Schaeffer et al., 2007.)

The correctness of the Checkers solver has been corroborated by doing consistency checks and having some of the computations verified by independent parties. Schaeffer et al. (2007) also note that if an incorrect position is far away from the start, it is unlikely that it would propagate all the way down the tree to change the overall game-theoretic value.

5.3 First Minimax implementation

A simple Minimax implementation was created to solve the smaller boards. The implementation was formulated as Negamax and could return three values: WIN, DRAW, and LOSS. These values described the value from the perspective of the current player. The implementation performed shallow cutoffs but was not Alpha-beta. A shallow cutoff took place if any of the children evaluated to a win for the current player, in which case the function returned immediately and skipped the rest of the children. No enhancements were used.

The first significant problem came apparent already at this point. Whereas standard Connect-4 is a divergent game where every move is a conversion, PopOut is neither diverging nor convergent. In fact, it turned out that you could get from almost any PopOut state to another legal PopOut state without repeating any position twice. To illustrate this with a trivially small board, the game-tree size for Connect-4 2x2 is 19. For PopOut 2x2, the game-tree size is 47,073. The state spaces also differ somewhat but not as wildly. Figure 14 shows the complete state spaces for PopOut and Connect-4 for the board size 2x2.

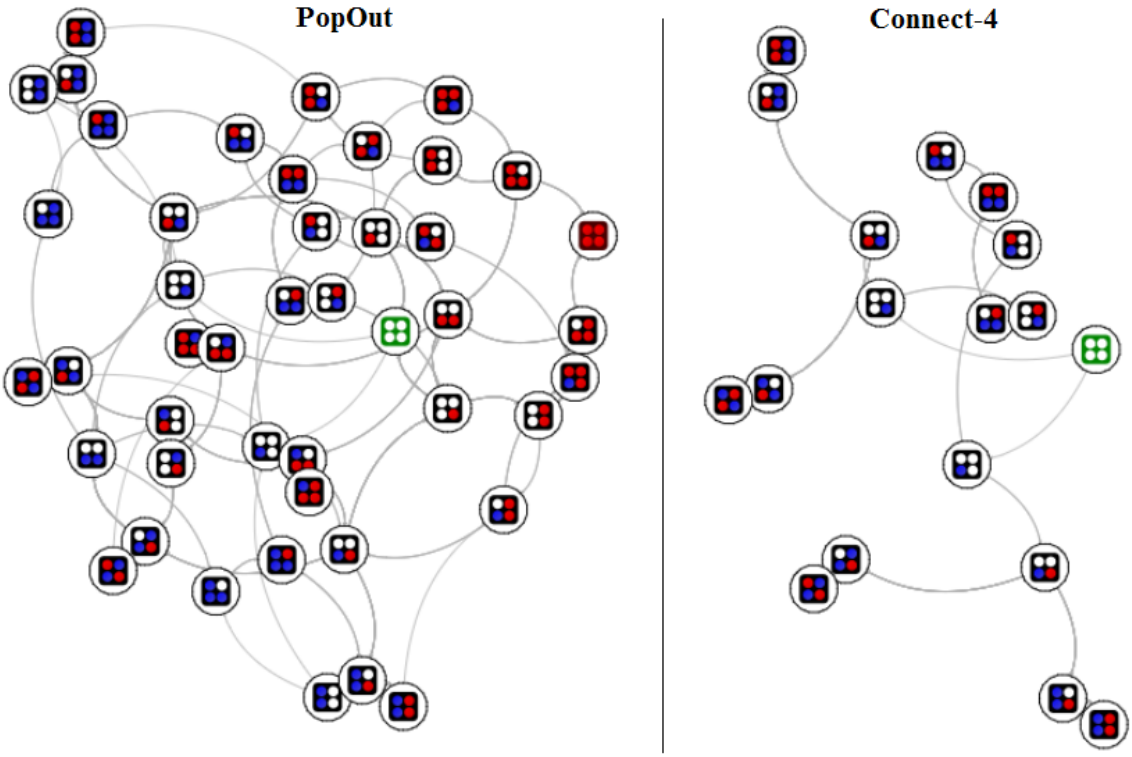

Figure 14. The state space of PopOut 2x2 on the left and the state space of Connect-4 2x2 on the right.

Because the state space remains moderate, attaching a transposition table allowed the search results to be reused massively. Solving PopOut 4x4 with Minimax is already intractable. Using a 5 MB transposition table allowed all legal states to be saved and the game to be solved practically instantly. Instead of trying a Minimax-based approach, we thought it was more useful to try to enumerate all states instead of traversing the game tree.

5.4 Retrograde analysis

Retrograde analysis is meant for convergent games which PopOut is not. We nevertheless tried applying it because no more suitable method for enumerating the state space was found. It turned out to be sufficient to strongly solve PopOut for boards that had no more than 25 cells, i.e. up to 5x5. As an added benefit, the retrograde analysis also computed the number of legal states for PopOut and Connect-4. The calculated numbers of Connect-4 states match those derived by Tromp (2010).

An array was created that had room for all PopOut states. Each array cell stored information about the state such as a counter for the number of children. The size of the array was actually an upper bound for the state space. This means that some space was wasted for the sake of simplicity. Starting at the root state, its successors were generated and the number of successors was stored in the array. This was continued until all legal states had been initialized. When a terminal position was encountered, it was appended to a list of unprocessed states. These terminal positions were then processed one by one. For each terminal state, its parents were computed and the parents' counters decremented by one.

A non-terminal state was added to the list of unprocessed states if its counter reached zero. At this point, its game-theoretic value could be determined. If any of its children was a win from the parent's perspective, the value of the parent was a win too.

Otherwise if it had any children that was marked as a draw, the parent's value was also a draw. The parent was a loss if all children were losses.

Retrograde analysis has huge memory requirements and this explains why we could not easily apply it to larger board sizes. Our implementation kept the array and the list of unprocessed states in main memory. Board sizes 6x5 and 5x6 could have been solved using this method if the state graph was somehow partitioned so that parts of the graph could be saved on disk. There was no apparent way of doing this so it was not pursued. We estimate that solving PopOut 7x6 with retrograde analysis would have taken several petabytes of disk space.

5.5 Alpha-beta and transposition table

Turning our Minimax algorithm into Alpha-beta meant that we had to deal with 5-way logic instead of 3-way logic. In Minimax, the algorithm could return either WIN, DRAW, or LOSS. The Alpha-beta implementation indicated the uncertainty caused by deep cutoffs by including two more values: DRAW_OR_WIN, and DRAW_OR_LOSS. These two new values signified that the returned value was only a bound and not an exact value. It is important to notice that with only three exact values, a cutoff could only be caused when alpha and beta both equaled DRAW.

To increase the performance of the Alpha-beta algorithm, a transposition table was added. The hashing function took the 64-bit position as input and performed a modulo operation on it. The divisor in the modulo operation was the size of the transposition table. For this reason, the transposition table size was chosen to be a prime.

The implemented replacement scheme was TwoBig1 (see section 3.2.1). In this replacement scheme, each slot in the transposition table has room for two positions. The search algorithm would keep track of how many nodes it had to investigate in the subtree to obtain the correct value. If some node in the subtree was resolved with the help of the transposition table, it would count as only one node. The number of investigated nodes determined whether the position was saved in the first or second slot. The first slot was always occupied by the position that had taken most work.

Alpha-beta enhanced with a transposition table was almost able to solve the same board sizes as retrograde analysis but there was a significant problem. The code was thoroughly supplied with assertions to make sure that the code executed as we thought it should. Yet one of the assertions kept failing and for a long time we could not figure out why it did that. It eventually turned out that the assertion failed because of a rather subtle problem called the Graph History Interaction problem.

5.6 Countering the GHI problem

The Graph History Interaction (GHI) problem occurs when transposition tables are used in games where states can be repeated. Repeated positions are usually marked as draws. If some state has been evaluated to a certain value by using the repetition rule, the value may not be unique depending on the path history. If such a value is then stored in a transposition table and reused without checking the path, the result may get corrupted. (Kishimoto, 2005.)

It is difficult to illustrate the GHI problem with a real example. Figure 15 shows an example that is a modification of the one given by Kishimoto and Müller (2005, p. 297). The squares represent states of the player who tries to reach the sole winning state. The circles are the opponent's states who tries to reach the loss state. If one starts from state A, one can follow A→B→C→D→E→Win to correctly evaluate state A as a win. In that line of play, the opponent has only forced moves. If the algorithm however uses a different move ordering and first plays A→F→E→B→C→D, then upon seeing that playing E again would lead to a repeated state, the value draw is propagated back to C. Because C is the only child of B and it has just returned a draw, the algorithm could incorrectly save B as a draw in the transposition table. Now executing A→B would evaluate to a draw even though it should still be a win.

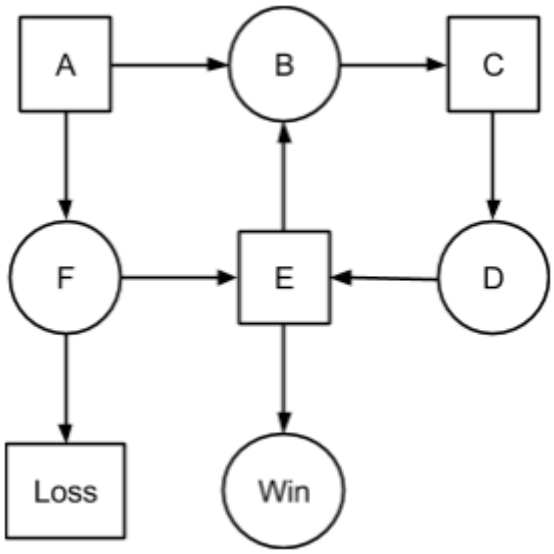

Figure 15. An example of the GHI problem.

There are several ways to deal with the GHI problem. One solution by Kishimoto and Müller (2004) is to store a signature of the history path in the transposition table. When fetching a value for some state from the transposition table, the signatures are checked to ensure that the value can safely be used. Their method does not require the signatures to be exactly the same but it uses simulation to determine the compatibility.

We chose to mark nodes whose values were determined with the help of the repetition rule as "tainted". This was a flag returned together with the value. If the value was marked as tainted, we disqualified it from being saved in the transposition table. Because our Alpha-beta implementation already could return five different values, adding the flag increased the number of different values to ten because any returned value could now either be tainted or untainted.

The logic to handle the taint flag was as follows. If any of the children evaluated to an untainted win, an untainted win could be returned irrespective of whether the other children were tainted or not. Otherwise if any of the children was tainted, the returned value would also be tainted because we did not know if the child could have evaluated otherwise with a different path history. We chose this mechanism simply because it was easy to implement. On the board 5x5, about 10% of interior nodes were marked as tainted.

5.7 Using history heuristic

The history heuristic together with a transposition table has been shown to be able to make 99% of possible reductions in some games (Schaeffer, 1989). Our algorithm used the history heuristic in the following manner. If a move was sufficient, i.e. it was the best one or caused a cutoff, the cell corresponding to that move was incremented by 2^history_depth. The history depth was defined to be 40 minus the distance from the root node. If the history depth was negative, the history was not incremented at all. Drop moves could map to 42 different slots in the history heuristic array whereas pop moves had only 7 possible slots in the array.

The history heuristic was initialized by giving higher values to columns that were closer to the center. For example in 7x6, the middle column received the value 3 while the outermost columns were initialized to 0. The difference was as small as it could be and it was done so that the algorithm would play in the central columns at the start of the game before the history heuristic had gained enough information. For most board sizes, the central columns tend to be more valuable than the outer columns.

Surprisingly, the introduction of the history heuristic deteriorated the performance of the PopOut algorithm. When pop moves were disabled, the history heuristic improved the performance. This behavior was explained by pop moves being sometimes prioritized over drop moves because of the history heuristic. The default move ordering was to try all drop moves first. Altering the default behavior so that all pop moves were tried before drop moves showed even a higher deterioration than what was introduced by the history heuristic.

The above behavior might have been caused by the fact that drop and pop moves had different numbers of slots in the history heuristic. To test this, the number of slots for pop moves was increased to 42. Two pop moves were considered to be the same only if they used the same column and the column had the same height. Previously only the column was used as a criterion. This variation had a similar deterioration in performance.

The final history heuristic variation that was tried was to use the history heuristic only for the drop moves and give the pop moves the lowest possible history score so that they would be forced to be tried last. This was found to improve the performance of the algorithm. We therefore ordered only the drop moves using the history heuristic. The pop moves were tried after the drop moves and they were not ordered in any way. This means that pop moves were tried only as a last resort. This behavior is compatible with Allen's (2010, p. 228) assumption that pop moves usually do not work unless they lead to an immediate win.

5.8 Symmetry recognition

The board in Connect-4 and PopOut is horizontally symmetric. You can take a mirror image of any position and the game-theoretic value does not change. Symmetric positions and transpositions are both types of equivalent positions (Heule & Rothkrantz, 2007). Just as with transpositions, once you know the value of some position, you also know the value for any symmetric position it might have.

We applied symmetry recognition in two ways in our program. The first way was to normalize positions before they were stored into or fetched from the transposition table. This normalization was done by reversing the columns in the bitboard with the help of bit masks. The mirrored bitboard was then compared to the original bitboard by doing an integer comparison, and the smaller of them was chosen to the normalized form.

The second way utilized normalization to find out when the position was symmetric. A symmetric position was one where the mirror bitboard was exactly the same as the original bitboard. This symmetry property could be used to reduce the number of moves that Alpha-beta tested in the symmetric position. For example, once Alpha-beta knew the result of the leftmost move, it also knew what the result of the rightmove move must have been without actually testing it.

When such symmetric positions were recognized in Alpha-beta, it could skip three out of seven drop and pop moves in the best case. The reduction in search effort was however not of the same ratio. For example, if a symmetric position had seven drop moves, the reduction of effort was not 3/7 as long as the transposition table was enabled. This is because the symmetric moves could reuse some of the results that were already calculated by the mirror moves. In fact, it was very likely that the position resulting from the symmetric move was directly fetchable from the transposition table but this was not always guaranteed. If Alpha-beta investigated another move between the two symmetric moves, investigating the subtree might have caused the result to be overwritten in the transposition table.

5.9 Proof-number search

Despite the transposition table and the history heuristic, Alpha-beta was still unable to solve the larger board sizes. We decided to test the effectiveness of basic PNS on PopOut. We chose to implement the original PNS algorithm as described in Allis's thesis (1994). Nodes were immediately evaluated upon creation and unexpanded nodes were initialized by setting both the proof and disproof numbers to one.

As expected, the algorithm used up a lot of memory. Solving 4x5 consumed over one gigabyte of memory and took around half a minute to solve. This was almost 50 times slower than our version of Alpha-beta enhanced with the transposition table and the history heuristic. The memory consumption was almost tenfold of that needed by retrograde analysis for the same board size. Retrograde analysis also finished its calculations in less than one third of the time. However, PNS clearly outperformed the Alpha-beta implementation in 4x5 when the transposition table was disabled.

We could have implemented PN^2 or one of the other variants to reduce memory usage. These variants execute slower than PNS (van den Herik & Winands, 2008) so implementing them would not have improved the performance in 4x5. We made a grave mistake when we neglected our PNS implementation as promising but weaker than the enhanced Alpha-beta implementation. We did not test its performance on larger board sizes until the very end of the study. It turned out that with a little bit of help, the PNS implementation was able to solve PopOut 7x6 in almost 10 seconds without consuming more than one gigabyte of memory.

5.10 Handicapping Alpha-beta

We returned to our Alpha-beta implementation and decided to work on cutting the depth that Alpha-beta would search. The very large depth of the game tree caused it to investigate nodes that were hardly relevant. For smaller board sizes, such deep searching was able to fill the transposition table with scores that could be reused at lower depths. This however did not work for larger board sizes.

Figure 16 shows how Alpha-beta behaves when solving the initial state of PopOut 6x6 with a large depth limit. Each letter in the side panel corresponds to a single move. The letter indicates the column (e.g. *a* is the leftmost column) and pop moves are written in uppercase. The first ten moves do not change at all despite running the algorithm for hours.

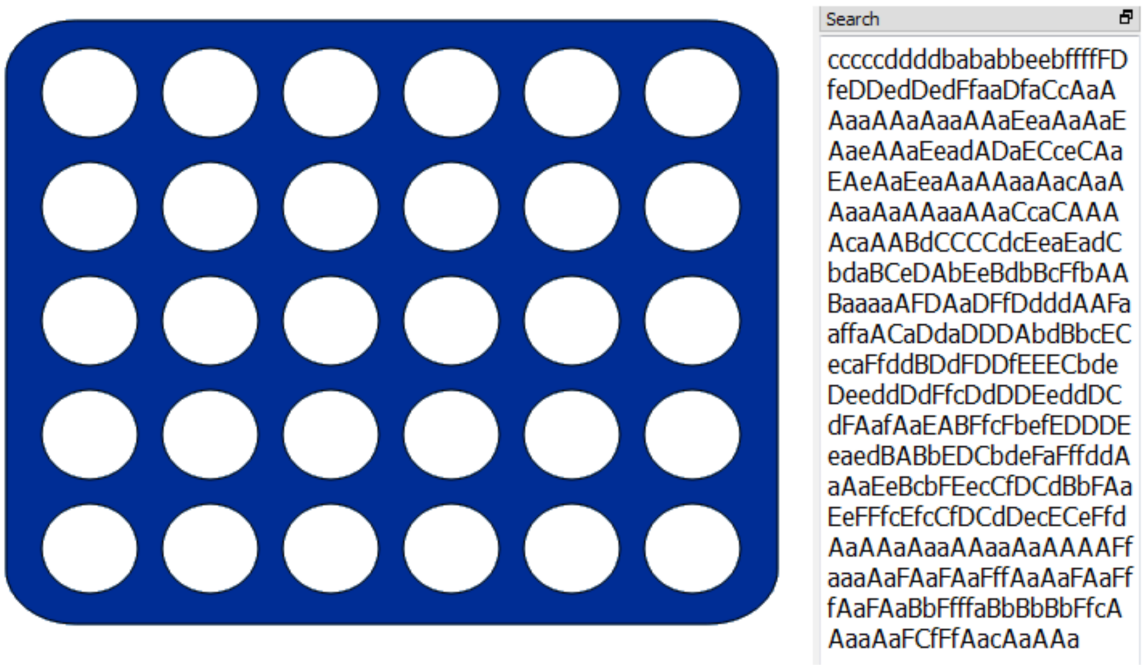

Figure 16. Alpha-beta gets “stuck” in a very deep search.

Earlier we had tried to limit the depth by terminating the search when a certain ply was reached or when a certain number of pop moves had been made. The mistake we made was to return the value UNKNOWN when the search was terminated. This meant that inexact values would propagate through the investigated game tree and the ability of Alpha-beta to prune subtrees was severely hampered. For example, solving 5x5 with a depth limit of 25 took 200 times as long as solving it with a depth limit of 100. With the higher depth limit, more exact values could be determined and stored in the transposition table.

The solution was to return an exact value instead of UNKNOWN. We changed the algorithm to return the value corresponding to “White loses” when the depth limit was reached. This change is equivalent to a rule change stating that the first player must win the game within N number of moves (where N is e.g. 42 for the board 7x6). In other words, we made the game harder for the first player while keeping the same options for the second player.

The advantage of this rule change was immediately evident. Instead of having inexact values propagating throughout the tree, exact values could be used. This in turn meant that Alpha-beta retained its ability to prune subtrees. More importantly, the maximum depth of the game tree now had a fixed limit and Alpha-beta would not get lost in investigating irrelevant variations.

We further modified the Alpha-beta implementation so that any draw would also count as a loss for the first player. This meant that only two values were needed: whether the position was a win for White or not. Because the alpha and beta parameters could now only assume two values and they could not be equal at the start of the function call, the two variables became redundant. To be technically accurate, our algorithm was no longer Alpha-beta because deep cutoffs could no longer occur. It retained its ability to prune any immediate subtrees however. We continue to refer to it as Alpha-beta even though the name is slightly erroneous.

Encouraged by the performance boost, we tried handicapping White even more. Allen (2010, p. 228) writes that pop moves are usually wrong if they do not lead to an immediate win. Based on this idea, we prevented White from making non-winning pop moves. If a node was created as a result of White's pop move, the value "White loses" was immediately returned, unless the pop move created a four-in-row for White. Red was allowed to make pop moves without any limit.

Limiting the pop moves had a greater impact than limiting the depth, although using both of them in combination yielded the best results. Setting the pop limit to 0 allowed the board 7x6 to be solved. The pop limit of 0 was too strict for some smaller boards, implying that optimal play requires White to make non-winning pop moves somewhere in the game. This *handicapping* technique is further discussed in section 7.3.

6. Evaluation and Results

Evaluation is an essential part of DSR and its purpose is to assess the efficiency, utility and performance of the artifact (Hevner & Chatterjee, 2010, p. 109). In our case, the artifact was a method that utilized various search techniques. Instead of evaluating the method directly, we evaluated an instantiation of it. This means that some details, such as the exact elapsed time and memory usage, are not important per se. They are important only insofar as they demonstrate that the method can solve the game within reasonable resources.

Hevner and Chatterjee (2010, p. 115) mention several common mistakes that can be made during the evaluation phase. These mistakes include having no clear goal and using an unsystematic approach. In our case, the goal was to demonstrate the method's ability to solve the game within reasonable resources. Therefore the measurements we used were meant to assess the amount of required resources. We also tried to apply the same measurements for all iterations in an attempt to be as systematic as possible.

We first discuss the correctness of our implementations. Then the game-theoretic values and state-space complexities for different board sizes are given. Finally, the performance evaluation is given separately for retrograde analysis, PNS and Alpha-beta.

6.1 Correctness

Showing that an algorithm has solved the game is not straightforward. Even if the design of the algorithm was rigorously proven to be correct, a mistake in the implementation could go unnoticed and give the wrong result. Allis (1994, p. 9) suggests that the natural way to test the algorithm is to let it play against skilled human players. If the algorithm has shown that one side wins, it should not lose a single game against human players when it gets to play the winning side. A single defeat would disprove the correctness of the algorithm or the implementation.

Even though using human players to evaluate the correctness of the program is the natural way, in our case we had another way of performing the validation. The algorithm was designed in such a way that pop moves could always be easily disabled and the game would transform into normal Connect-4. Because the results of Connect-4 are widely known and established, the results returned by our algorithm could be compared with those. The chosen reference program used for this validation was Tromp's Fhourstones, and it confirmed our results.

The larger boards were solved by both Alpha-beta and PNS. The smaller boards were also solved using retrograde analysis. In total, three different approaches were used to solve the smaller boards. We manually checked that all three approaches produced compatible results in various positions. For these reasons and because no inconsistencies were found, the results should be reliable.

6.2 Game-theoretic values

The board size 5x5 and smaller have been strongly solved by retrograde analysis. Alpha-beta and PNS both confirmed these results and further solved boards 5x6, 6x5, 6x6 and 7x6 weakly. See Table 8 for the results.

Table 8. The game-theoretic value of the root state for various PopOut board sizes.

	Height 4	Height 5	Height 6
Width 4	Draw	First	First
Width 5	Draw	First	First
Width 6	First	First	First
Width 7	First	First	First

The game-theoretic value is a first-player win for all board sizes except for 4x4 and 4x5. For these two board sizes, the optimal play entails drawing by repetition. For comparison, the results for Connect-4 are given in Table 9. These results have been calculated by the Alpha-beta implementation by disabling pop moves. The results are in concordance with those calculated by Fhourstone (Tromp, 2010).

Table 9. The game-theoretic values of the same board sizes in Connect-4 (Tromp, 2010).

	Height 4	Height 5	Height 6
Width 4	Draw	Draw	Draw
Width 5	Draw	Draw	Draw
Width 6	Second	Draw	Second
Width 7	Draw	Draw	First

In Connect-4 7x6, the only winning move is the middle column. In PopOut, all columns except the outermost columns are winning moves (see Figure 17). Based on casual observation, White seems to retain the high number of alternative winning moves at deeper plies as well. Allen (2010, p. 229) writes that White has a much stronger advantage in PopOut than in Connect-4. It should be safe to confirm Allen's assumption.

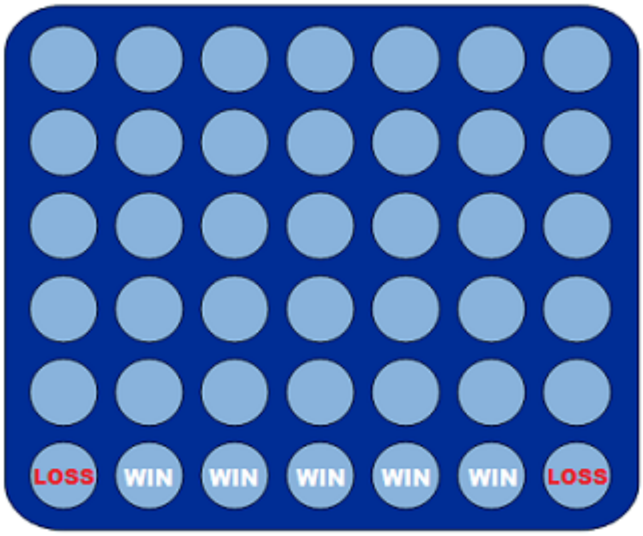

Figure 17. All columns except the two outermost are wins for White in PopOut 7x6.

6.3 State-space complexities

The implementation based on retrograde analysis counted the number of legal states, i.e. the state-space complexities, for the board sizes it could strongly solve. Table 10 shows the state-space complexities for boards 5x5 and smaller. The table is sorted so that the board size with the smallest complexity comes first. The state-space complexity of PopOut seems to be around 5 times as large as that of Connect-4. There seems to be a slight indication that the ratio might grow as the complexity increases.

Table 10. State-space complexities for PopOut and Connect-4 and the ratio between the two.

Board Size	PopOut	Connect-4	Ratio
4x4	692,427	161,029	4.30
4x5	7,942,028	1,706,255	4.65
5x4	18,531,835	3,945,711	4.70
4x6	82,822,024	15,835,683	5.23
5x5	359,220,338	69,763,700	5.15
6x4	486,534,361	94,910,577	5.13

6.4 Performance evaluation

There are several ways to measure the performance of a search algorithm. For the implementor, the best measure is probably elapsed CPU time but that obviously depends on the used computer. Other less machine-dependant metrics count the investigated nodes. There are two major variations for this node counting in Alpha-beta. The first one counts only the terminal nodes. The second one counts also interior nodes. (Schaeffer, 1989, p. 13.)

We chose to provide both node measures for Alpha-beta. For PNS, we give the number of expanded nodes. We ran all calculations on an Intel Core i5-4670K 3.40 GHz. We provide the elapsed CPU time on this processor while using only one core. For PNS and retrograde analysis, we also measured the peak memory usage. The peak memory refers to the peak private bytes and not to the peak working set. Working set is the amount of main memory used whereas private bytes measure how much memory the program has allocated. The operating system can store some of the allocated memory on disk instead of the main memory.

6.4.1 Retrograde analysis

Retrograde analysis was implemented as a means to enumerate the total state space. Table 11 shows the elapsed time and peak memory usage for different board sizes. Larger board sizes could not be tested due to memory limitations. We estimate that solving 5x6 by retrograde analysis would have required around 35 GB and solving 6x5 would have required around 70 GB. For comparison, databases up to 178 GB were created when solving Awari by retrograde analysis (Romein & Bal, 2003).

Table 11. Performance of retrograde analysis for different board sizes.

Board size	Elapsed time	Peak memory usage
4x4	0.6 s	23 MB
4x5	9.2 s	149 MB
4x6	102 s	1,740 MB
5x4	25 s	327 MB
5x5	*559 s	8,158 MB
6x4	*878 s	9,584 MB

*Peak memory exceeded 8 GB of main memory causing significant performance degradation.

6.4.2 Proof-number search

The PNS algorithm was implemented in its original non-variant form. The PNS implementation did not use handicapping rules, symmetry recognition or any other enhancements. It was able to solve all boards. For boards 4x4 and 5x4 which are draws, PNS only proved that White does not win. It would have been trivial to use PNS to prove that Red does not win either and thus arrive at the correct game-theoretic value.

Table 12 gives the elapsed time, number of expanded nodes and peak memory usage while solving the root node for different boards sizes with PNS. The elapsed times for boards 6x5, 6x6 and 7x6 are not comparable with the rest because the peak memory exceeded the amount of main memory. The operating system had to resort to paging which caused a massive degradation of performance, especially on board 7x6. Board 5x4 has a high elapsed time and node count because it is a draw unlike most boards.

Table 12. Performance of PNS for different board sizes.

Board size	Elapsed time	Node count	Peak memory
4x4	7.6 s	7,206,169	675 MB
4x5	15 s	14,280,600	1,774 MB
4x6	21 s	21,726,306	3,662 MB
5x4	121 s	94,417,092	3,663 MB
5x5	32 s	29,205,810	4,854 MB
5x6	65 s	52,635,421	6,891 MB
6x4	51 s	40,153,721	5,728 MB
6x5	*236 s	56,097,931	11,117 MB
6x6	*264 s	69,167,358	13,016 MB
7x6	*1158 s	114,206,314	24,287 MB

*Peak memory exceeded 8 GB of main memory causing significant performance degradation.

6.4.3 Alpha-beta

The fully-enhanced Alpha-beta implementation consisted of a two-valued Alpha-beta search algorithm together with several enhancements. These enhancements were the transposition table, the history heuristic, symmetry recognition and two handicapping rules. Table 13 shows how many interior and terminal nodes the implementation investigated in order to solve the root node for different board sizes. The total number of nodes is the sum of these two node counts. The table columns *pop* and *depth* refer to the values used in the handicapping rules.

Table 13. The number of nodes nodes investigated by Alpha-beta with handicapping rules.

Board size	Time	Terminal	Interior	Pop	Depth
4x5	0.7 s	3,059,695	5,476,281	1	25
4x6	0.3 s	1,133,949	2,168,998	0	25
5x5	1.6 s	6,288,826	12,298,311	1	25
5x6	7.8 s	32,840,424	71,991,030	0	29
6x4	53 s	212,442,600	412,958,895	1	27
6x5	13 s	43,615,532	112,451,684	0	25
6x6	31 s	88,829,470	290,001,528	0	25
7x6	6.2 s	14,710,813	54,777,040	0	21

The transposition table was approximately one gigabyte and the history heuristic was initialized to prefer middle columns. For each interior node, the Alpha-beta looked for all terminal nodes sequentially by starting from the left column and trying all drop moves first. After that it tried all pop moves in the same order. If any of them evaluated to a win, the rest of the terminal nodes were not checked and are not included in the count. The number of interior nodes is the same as the number of calls to the Negamax function.

We customized the handicap limits for each board. We first checked if it was possible to solve the board when the pop limit was set to zero. If it was not, we increased it to one. No board required a higher pop limit. After finding the correct pop limit, we determined the optimal depth limit for each board using a similar process. Increasing the pop limit above its minimum value did reduce the optimal depth limit on some boards but it also increased the number of investigated nodes. For example, if the pop limit was completely removed on 6x6, the optimal depth limit was 23 but over three times as many terminal nodes had to be investigated.

Repetition checking was disabled when the pop limit was zero. It could have been disabled for other boards as well but then it would have affected the number of investigated nodes. We tested the effect of not doing repetition checking on 5x5. The algorithm investigated about 25% more nodes and the total elapsed time was a bit higher despite not having the repetition checking overhead.

The standard board size 7x6 seemed surprisingly easy to solve. Even though the game-tree complexity cannot be directly derived from the given node counts, the node counts still suggest that the game-tree complexity of 7x6 is smaller than that of some of the smaller boards. We would have tested larger boards as well but the bitboard encoding would have required modifications due to using 64-bit integers.

We also compared the importance of different enhancements when solving 5x5 with handicapping rules. The pop limit was 1 and the depth limit was 25. Table 14 shows the number of investigated nodes and the running time for board size 5x5 when only some of the enhancements were enabled. TT refers to a transposition table of around one gigabyte. HH is the history heuristic and SR is symmetry recognition. Symmetry recognition had a noticeable effect only when used in combination with the other two enhancements.

Table 14. The number of nodes investigated by Alpha-beta for 5x5.

Enhancements	**Time**	**Terminal**	**Interior**
None	105 s	489,748,141	1,031,843,809
TT	54 s	231,634,369	537,083,119
HH	5.7 s	25,671,327	45,831,186
TT + HH	2.2 s	8,779,326	17,856,556
TT + HH + SR	1.6 s	6,288,826	12,298,311

The history heuristic was initialized by giving the centermost moves slightly higher values. This meant that the algorithm played the first move in the middle column when the history heuristic was enabled but in the leftmost column when it was disabled. The leftmost column is not a winning column. By instructing the algorithm to play the first move in the middle column when the history heuristic was disabled, the algorithm with no enhancements took 91 s and the algorithm with the transposition table took 53 s. If all moves in the history heuristic were initialized to zero, the fully enhanced algorithm took 7.4 s instead of 5.7 s. The significant performance gain with the history heuristic did therefore not depend on the initialization.

The earlier Alpha-beta iterations did not use handicapping rules and they could return all three game-theoretic values: WIN, DRAW, and LOSS. We tested how important it was to return the value "White loses" upon reaching the limits. We created another Alpha-beta version which we call the limit version. It was based on the three-valued Alpha-beta algorithm and it used the same three enhancements as the handicap version. Then we added limits that were similar to the handicap limits. When the depth or pop limit was reached, the value UNKNOWN was returned instead of "White loses".

When the depth and pop limits were the same as with the handicapped version on 5x5, the elapsed time was 350 times higher. If higher depth limits were used, more exact values could be found and stored in the transposition table. With depth limit set to 35 and pop limit set to 1, the elapsed time was only twice as high as with the handicapped version. However, disabling the transposition table had a larger impact on the limit version. With depth limit 35, the elapsed time was quadrupled. If the depth limit was too low (e.g. 25), the board 5x5 could not be reasonably solved without the transposition table (elapsed time was at least 10 h).

The limit version was slower but still comparable if the depth limit was chosen right on 5x5. If the handicapped version could be solved with a certain depth limit, increasing the depth limit would only degrade the performance. The same was not true of the limit version. Using a smaller depth limit meant that more inexact values could propagate throughout the game tree. A suitable depth limit was therefore a tradeoff between not having unnecessarily deep searches and being able to return exact scores. We could not find a suitable depth limit on 7x6. The limit version seemed unable to solve the board no matter what the depth limit was.

The limit version was not the same as fixed-depth Alpha-beta. Limiting the number of pop moves is more important than limiting the depth. Disabling pop moves for White causes every move of either player to be a conversion (see section 2.4.3) and this in turn also affects the maximum depth. If fixed-depth Alpha-beta was used, i.e. the limit version with no limit on pop moves, the algorithm was barely able to solve the smallest boards despite having all enhancements enabled.

7. Discussion and Conclusions

The game of PopOut has been solved for the standard board size 7x6 and smaller. Both research questions have been successfully answered. To our knowledge, this study is the first published computer analysis of PopOut. We now repeat the research questions and provide the answers.

RQ1: How should the search techniques that were used in solving Connect-4 be modified or augmented in order to be applicable to PopOut?
PNS required no modifications. Alpha-beta enhanced with a transposition table, the history heuristic and symmetry recognition required two modifications. The GHI problem had to be countered and the search depth had to be limited. PopOut suffers from these two problems because unlike Connect-4, it is not a divergent but unchangeable game. This change in convergence type is caused by the introduction of pop moves.

RQ2: What is the game-theoretic value of PopOut?
The game-theoretic value for 7x6 is a first-player win. The game-theoretic values for smaller board sizes are either draws or first-player wins.

The first section describes the design artifact and our adherence to the DSR guidelines. In the second section, open issues in the implementation are discussed. The handicapping technique is analyzed in more detail in the third section. Finally, ideas for future research are suggested in the fourth section.

7.1 Artifact and DSR

We have created and evaluated an artifact that describes what steps should be taken in order to solve the game of PopOut. These steps include what search techniques to apply and how to fix encountered problems. The order in which to perform the steps is not important. Such an artifact is called a method in the context of DSR. Repeating the steps of the method should result in a program that can solve PopOut.

The method had two distinct phases: knowledge representation and searching. The knowledge representation was done using bitboards. The bitboard encoding we used has been described by Tromp (2010) as the optimal encoding for Connect-4. The encoding enabled fast and parallel win detection and a memory-efficient way to store positions.

Two design alternatives were created for the search part: one based on Alpha-beta search and one based on PNS. Both alternatives were found to be able to solve the game within reasonable resources. PNS was implemented as described by Allis (1994) in its original non-variant form. PNS consumed a lot of memory especially on the board 7x6. If it is required to use less memory, the problem can be divided into subproblems: instead of solving the initial state, a selection of 3-ply positions can be solved. Less than one gigabyte of memory is needed when solving 7x6 with 3-ply positions. No other obstacles were encountered.

The Alpha-beta implementation required more iterations than PNS before a satisfactory design was found. The earliest iterations could return three exact scores and two inexact scores. The exact scores were a win, a draw, and a loss. The two inexact scores were the lower bound and upper bound. In the final design, two-way logic was used instead and the score only indicated whether White wins or not. The search was enhanced by a transposition table, the history heuristic and symmetry recognition. The GHI problem was avoided by returning a flag together with the score. If the returned score had the flag set, the position could not be safely stored into the transposition table. To cut the depth of the search, the first player was handicapped (details in section 7.4).

The contributions of this research are the design artifact and the game-theoretic values of PopOut. The design alternative based on PNS was simple to implement and therefore relatively little was learned from its application. The Alpha-beta design alternative provided more contributions by being more challenging. More obstacles were encountered and because of that, the solutions to those obstacles have more potential to be useful for other people.

7.2 Open issues in implementation

The method artifact was not evaluated directly because ideas in computer science often work on paper but fail to work in a real environment (March & Smith, 1995, p. 260). An instantiation of the method was evaluated instead. We now describe several aspects in the implementation that are more closely related to the instantiation rather than the method. Improving these aspects would have shortened the execution time of the implementation but it would not have affected the utility of the method.

The solution to the GHI problem was not perfect and it probably discarded more work than what was necessary. The actual details of how to circumvent the GHI problem are not so important as long as the GHI problem is taken into account in some way. The method should have worked even if another solution to the GHI was used. Implementing the general solution suggested by Kishimoto and Müller (2004) remains the biggest improvement in that regard.

Repeated positions were identified by having all previous positions stored in an array and looping over the array. The only optimization that we used was to skip the positions where it was the other player's turn. Such positions clearly could not have been the same. Profiling revealed that approximately 20% of the running time was taken up by looking for repeats. In other words, a solution that completely eliminated the repetition checking would have improved our implementation by 25%. Repetition checking was not used when the handicapping pop limit was set to zero.

PNS keeps all nodes in memory until they can be proved or disproved. Most of the high memory consumption in PNS comes from this fact. We implemented the nodes as C++ structs consisting of several fields. The sizes of these fields could have been optimized. If the size of the struct had been halved, it is likely that the peak memory usage of PNS would also have been almost halved.

7.3 Handicapping the first player

Alpha-beta is a depth-first search algorithm and this causes problems when the depth of the tree becomes too great. In some convergent games, this issue has been avoided by using endgame databases (van den Herik & Winands, 2008). This is infeasible in PopOut because PopOut is not a convergent game. Instead, we had to come up with another solution for limiting the depth.

We handicapped the first player by imposing stricter rules on him while letting the second player keep all his options. It is our understanding that this handicapping technique has not previously been used in solving other board games. The lack of previous examples is probably explained by the small number of solved games and the fact that there has not been a particular need for it in other games. Due to the unchangeable nature of PopOut, the maximum depth of the game tree was very large and this hampered the effectiveness of Alpha-beta search. By adding handicapping rules for White, the maximum depth was brought to a reasonable and fixed level.

The handicapping rules work by making the game harder for the stronger player. This difficulty is created by prohibiting the stronger player from making moves that he would normally be allowed to do. It is important that the weaker player has the same move options that he would normally have. This rule change has the same effect as if the stronger player out of his own volition decided to use a strategy that is more difficult for him. If it can be proven that the stronger player still wins, he would have won without the stricter rules as well because the weaker player cannot prevent him from using that strategy.

We believe that handicapping rules are easier and safer to apply than expert rules. For example, a strategic rule can be used to prove that one player has won long before a terminal node is reached. If the rule is invalid, the correctness of the calculated game-theoretic value is also questionable. A handicapping rule is less likely to compromise the game-theoretic value. If the handicapping rule is too strict and the algorithm returns a value indicating that the player does not win, we just decide not to infer anything and possibly try again by loosening the handicapping rule.

A natural question is how applicable this technique is to other games. The technique should be logically valid whenever the moves for one player are artificially limited but not for the other player. If this limitation creates a situation where the player seemingly has no moves left, the player is declared to have lost. We have already used two examples of such rules. The first one dictated that the first player is allowed to make only 11 (on 7x6) moves in total. Any more and he has lost. The second rule stated that he cannot make any non-winning pop moves during the course of the game. Both rules limit the number of some type of moves that the player can make. The move type can be completely disabled by setting the limit to zero.

In order for this technique to be applicable, the stronger player should have a clear advantage and there should be an easy way to limit the moves. We could not identify many potential candidates among the solved games. The technique could possibly reduce the search effort in Gomoku because the first player has a clear advantage in the game. For example, the first player could be required to win within some number of moves. It is however uncertain if the reduction in search effort would be significant at all.

We speculate that the technique is mostly applicable to other unchangeable games when a depth-first search is used and there is a need to limit the maximum depth. It could also be applied to such convergent or divergent games where there is an excessive number of moves between conversions. When using a best-first search, such as PNS, the need for limiting the depth is not as essential. Some amount of search effort could still be saved in a best-first search as well.

We must give a warning about the usage of handicapping rules and its possible relation to the GHI problem. A handicapping rule may introduce the GHI problem to a game that normally does not have it. If one player has a limit on the number of moves he can make, then the value of the position may vary depending on the number of remaining moves.

7.4 Future research

PopOut is the first unchangeable game to be solved that we are aware of. Attempting to solve other unchangeable games might shed more light on the relationship between convergence and solvability. Especially, it would be interesting to see how successful a depth-first search such as Alpha-beta would be and whether the depth of the search could be easily limited in those games as well.

Gasser wrote in 1996 that “exhaustive search in large state spaces is still in its infancy” (p. 101). We remark that this still seems to be the case. The only viable method of total enumeration of the state space that we could find in the knowledge base was retrograde analysis. Retrograde analysis is designed mainly for convergent games but we still applied it to PopOut. One future question is whether there is some other form of total enumeration that would be more effective in divergent or unchangeable games than what retrograde analysis is.

We implemented retrograde analysis and it was enough to solve PopOut 5x5. Bigger boards could not be solved because of memory limitations. It was not clear how to divide the state space into subgraphs so that it could be solved piecewise. In many games that have used retrograde analysis, the division was done with the help of conversions. A conversion means that any position that could occur before the conversion can no longer occur after the conversion. Because of this, it is easy to divide positions into classes and determine which classes can safely be ignored when analyzing a certain position. We therefore ask if there is a way to partition the state space in retrograde analysis without using conversions as the basis for partitioning.

Allis (1988) has formulated nine strategic rules for Connect-4. We did not look into how applicable they are to PopOut. It is likely that they would have needed some adaptation but if such expert rules could be applied to PopOut, the solving time could have been shortened considerably. An analysis of expert rules in PopOut thus still remains to be done.

Finally we ask if the handicapping technique can be used to reduce the search effort in other games and search algorithms. We did not apply the handicapping rules in our PNS implementation. We modified our Alpha-beta search to use two-valued logic because draws cannot be determined when the handicapping rules are in effect. PNS already uses two-valued logic so no modifications would be needed. It might in fact have been more natural to implement the handicapping rules in PNS rather than in Alpha-beta.

References

Allen, J. D. (1989). A note on the computer solution of Connect-Four. *Heuristic Programming in Artificial Intelligence, 1*(7), 134-135.

Allen, J. D. (2010). *The complete book of Connect-4: History, strategy, puzzles*. New York, NY: Sterling.

Allis, V. L. (1988). *A knowledge-based approach of Connect-four* (Master's thesis, Vrije Universiteit, Amsterdam, Netherlands). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.2778&rep=rep1&type=pdf

Allis, V. L. (1994). *Searching for solutions in games and artificial intelligence* (Doctoral dissertation, University of Limburg, Maastricht, Netherlands). Retrieved from fragrieu.free.fr/SearchingForSolutions.pdf

Breuker, D. M. (1998). *Memory versus search in games* (Doctoral dissertation, Maastricht University, Netherlands). Retrieved from http://www.top-5000.nl/ps/dennis_breuker_thesis.pdf

Breuker, D. M., Uiterwijk, J., & van den Herik, H. J. (1994). Replacement schemes for transposition tables. *ICCA Journal, 17*(4), 183-193.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., … Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games, 4*(1), 1-43.

Bruin, A., Pijls, W., & Plaat, A. (1994). *Solution trees as a basis for game tree search* (Report No. EUR-FEW-CS; 94-04). Rotterdam, Netherlands: Erasmus University.

Campbell, M. S., & Marsland, T. A. (1983). A comparison of minimax tree search algorithms. *Artificial Intelligence, 20*(4), 347-367.

Chaslot, G. (2010). *Monte-carlo tree search* (Doctoral dissertation, Maastricht University, Netherlands). Retrieved from http://www.unimaas.nl/games/files/phd/Chaslot_thesis.pdf

Edelkamp, S., & Kissmann, P. (2008). Symbolic classification of general two-player games. *Proceedings of the 31st Annual German Conference on Artificial Intelligence*, 185-192.

Gasser, R. (1996). Solving nine men's morris. *Computational Intelligence, 12*(1), 24-41.

Heule, M. J., & Rothkrantz, L. J. (2007). Solving games: Dependence of applicable solving procedures. *Science of Computer Programming, 67*(1), 105-124.

Hevner, A., & Chatterjee, S. (2010). *Design science research in information systems*. New York, NY: Springer.

Hevner, A., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly, 28*(1), 75-105.

Hlynka, M., & Schaeffer, J. (2006). Automatic generation of search engines. *Proceedings of the 11th Advances in Computer Games*, 23-38.

Kishimoto, A. (2005). *Correct and efficient search algorithms in the presence of repetitions* (Doctoral dissertation, University of Alberta, Canada). Retrieved from http://www.is.titech.ac.jp/~kishi/pdf_file/kishi_phd_thesis.pdf

Kishimoto, A., & Müller, M. (2004). A general solution to the graph history interaction problem. *Proceedings of the 19th National Conference on Artificial Intelligence*, 644-649.

Kishimoto, A., & Schaeffer, J. (2002). Distributed game-tree search using transposition table driven work scheduling. *Proceedings of the 31st International Conference on Parallel Processing*, 323-330.

Kishimoto, A., Winands, M. H., Müller, M., & Saito, J. T. (2012). Game-tree search using proof numbers: The first twenty years. *ICGA Journal, 35*(3), 131-156.

Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence, 6*(4), 293-326.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence, 27*(1), 97-109.

Lorenz, U., & Tscheuschner, T. (2006). Player modeling, search algorithms and strategies in multi-player games. *Proceedings of the 11th Advances in Computer Games*, 210-224.

March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. *Decision Support Systems, 15*(4), 251-266.

Marsland, T. A. (1986). A review of game-tree pruning. *ICCA Journal, 9*(1), 3-19.

Marsland, T. A., & Popowich, F. (1985). Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 7*(4), 442-452.

Nievergelt, J., Gasser, R., Mäser, F., & Wirth, C. (1995). All the needles in a haystack: Can exhaustive search overcome combinatorial chaos? In J. van Leeuwen (Ed.), *Computer science today: Recent trends and developments* (pp. 254-274). Berlin, Germany: Springer.

Reinefeld, A., & Marsland, T. A. (1994). Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 16*(7), 701-710.

Romein, J., & Bal, H. (2003). Solving the game of awari using parallel retrograde analysis. *IEEE Computer, 36*(10), 26-33.

Romein, J. W., Bal, H., Schaeffer, J., & Plaat, A. (2002). A performance analysis of transposition-table-driven work scheduling in distributed search. *IEEE Transactions on Parallel and Distributed Systems, 13*(5), 447-459.

Saito, J. T., Winands, M. H., & van den Herik, H. J. (2010). Randomized parallel proof-number search. *Proceedings of the 12th Advances in Computer Games*, 75-87.

San Segundo, P., Galan, R., Matia, F., Rodriguez-Losada, D., & Jimenez, A. (2006). Efficient search using bitboard models. *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence*, 132-138.

Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 11*(11), 1203-1212.

Schaeffer, J. (2001). A gamut of games. *AI Magazine, 22*(3), 29-46.

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., ... Sutphen, S. (2007). Checkers is solved. *Science, 317*(5844), 1518-1522.

Schaeffer, J., & Plaat, A. (2000). Unifying single-agent and two-player search. *Proceedings of the 13th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, 1-12.

Simon, H. A. (1995). Artificial intelligence: An empirical science. *Artificial Intelligence, 77*(1), 95-127.

Tromp, J. (2010). *John's Connect Four Playground*. Retrieved January 26, 2014, from http://homepages.cwi.nl/~tromp/c4/c4.html.

Uiterwijk, J., & van den Herik, H. J. (2000). The advantage of the initiative. *Information Sciences, 122*(1), 43-58.

Vaishnavi, V., & Kuechler, W. (2004). *Design research in information systems*. Retrieved from http://desrist.org/design-research-in-information-systems/

van den Herik, H. J., Uiterwijk, J. W., & Van Rijswijck, J. (2002). Games solved: Now and in the future. *Artificial Intelligence, 134*(1), 277-311.

van den Herik, H. J., & Winands, M. H. (2008). Proof-number search and its variants. In H. R. Tizhoosh & M. Ventresca (Eds.), *Oppositional Concepts in Computational Intelligence* (pp. 91-118). Berlin, Germany: Springer.

Yamaguchi, Y., Yamaguchi, K., Tanaka, T., & Kaneko, T. (2012). Infinite Connect-four is solved: Draw. *Proceedings of the 13th Advances in Computer Games*, 208-219.