University of Alberta

Library Release Form

Name of Author: Jack van Rijswijck

Title of Thesis: Computer Hex: Are Bees Better Than Fruitflies?

Degree: Master of Science

Year this Degree Granted: 2000

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

. . . .

Jack van Rijswijck
03a 8913 112th Street
Edmonton, Alberta
Canada T6G 2C5

Date: . . . .

University of Alberta

COMPUTER HEX: ARE BEES BETTER THAN FRUITFLIES?

by

Jack van Rijswijck

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

in

Computing Science

Department of Computing Science

Edmonton, Alberta
Fall 2000

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Computer Hex: Are Bees Better Than Fruitflies?** submitted by Jack van Rijswijck in partial fulfillment of the requirements for the degree of **Master of Science** in *Computing Science.*

. . . .
Jonathan Schaeffer

. . . .
Ryan Hayward

. . . .
Andrew Liu

. . . .
Martin Müller

Date: . . . .

Contents

Preface

Acknowledgments

1 Introduction **1**
- 1.1 Game playing 1
- 1.2 Problem description 3
- 1.3 Objective 4
- 1.4 Motivation 4
- 1.5 Related work 5
- 1.6 Summary 5

2 Search algorithms **7**
- 2.1 Game tree 7
- 2.2 Evaluation function 8
- 2.3 Alpha-beta search 9
- 2.4 Enhancements 12
- 2.5 Selective search 15
- 2.6 Machine learning 16

3 Hex **19**
- 3.1 History 19
- 3.2 Rules 20
- 3.3 No draws 21
- 3.4 First player wins 22
- 3.5 Shannon switching game 23
- 3.6 Computational complexity 24

4 Hex strategy **27**
- 4.1 Double threats 28
- 4.2 Edge patterns 28
- 4.3 Ladders 30
- 4.4 Outposts 31
- 4.5 Forcing moves 32

4.6 Opening moves 34

5 Queenbee's evaluation function **37**
5.1 Two-distance 37
5.2 Potentials 39
5.3 Strategic relevance 40
5.4 Move badness 41
5.5 Move tension 42

6 Queenbee's search algorithm **45**
6.1 Beam search and tapered search 45
6.2 SEX search 46
6.3 Move categories 47
6.4 Pattern database 48
6.5 Unsuccessful enhancements 50

7 Results **53**
7.1 Selective search 53
7.2 Branching factor 55
7.3 Sex search 56
7.4 Evaluation function 59
7.5 Tension and badness 60

8 Conclusions and future work **63**
8.1 Discussion 63
8.2 Comparison with Hexy 65
8.3 Future work: Learning search control 66
8.4 Future work: Opening book learning 67
8.5 Future work: Pattern search 67
8.6 Future work: Combinatorial game theory 70

A Sample games **73**

Bibliography **79**

Index **81**

Preface

Traditionally, chess has been called the "fruitfly of Artificial Intelligence". In recent years, however, the focus of game playing research has gradually shifted away from chess towards games that offer new challenges. One of these challenges is a large branching factor, as is the case in games such as Go and Hex. The game of Hex offers some interesting properties that make it an attractive research subject.

This thesis presents the key ideas behind Queenbee, the first Hex playing program to play at the level of strong human players and the first Hex playing entity of any kind to achieve perfect play on board sizes up to 6×6. At the heart of the program lies the evaluation concept of "two-distance", a new way to measure connectivity in a graph which appears to be naturally suited to two-player adversary games. The program's strength also derives from the application of state-of-the-art search and machine learning methods.

Acknowledgments

Special thanks go to...

Jonathan Schaeffer, for convincing me to invest a little time to turn an existing hobby project into a thesis, and subsequently being an excellent advisor to it;

Ryan Hayward, for constantly encouraging me to get the thesis done with, since Queenbee will forever be a work in progress anyway;

Yngvi Björnsson, for his machine learning help and expertise;

Alice Nodleman, for starting the java applet, and **Aaron Davidson,** for finishing it;

the GAMES Group, particularly **Darse Billings** and **Andreas Junghanns,** for their support and boundless enthusiasm and fascination with games;

and the online Hex playing community at Playsite and Maitreg's Hex Club, for challenging Queenbee in games and with puzzles.

Finally, a word of thanks to **Vadim Anshelevich,** for making Hexy available and for making it quite a challenge to beat.

Chapter 1

Introduction

Queenbee, born in 1995, is a program that plays the board game Hex. Based on a novel idea for an evaluation function, it is the first Hex program to surpass "novice" level in human terms. Indeed, it now plays at the level of very strong human players, if not quite yet at the level of the top players. Queenbee has also carried out the first complete analysis of all opening lines on a 6 × 6 board. The program has its own web page, `http://www.cs.ualberta.ca/~queenbee`, which includes the 6 × 6 opening analysis.

1.1 Game playing

> Chess is the touchstone of intellect. – *Johann Wolfgang von Goethe*

Game playing has often been described as an ideal test bed for Artificial Intelligence research. Game playing is a nontrivial task, which when performed by humans is associated with a certain degree of intelligence. It would thus by definition require Artificial Intelligence to enable a machine to play a game well.

> Chess is the Drosophila of Artificial Intelligence. – *Alexander Kronrod, 1965*

In comparison with other intelligent tasks, games offer the advantage of being confined to a limited abstract domain with clearly defined rules for behaviour and clearly defined criteria for success or failure. For these reasons, games — chess in particular — were recognized as an excellent research subject for Artificial Intelligence as early as the 1950s by scientists such as Alan Turing and Claude Shannon.

From a Computer Science point of view, the game playing domain is of considerable complexity. Informally, "game" type problems tend to be harder than "puzzle" type problems; in a puzzle, one merely needs to find a single road to the goal, while in a game, there is an opponent who actively tries to thwart this objective. A puzzle solution can easily be verified, while a game solution needs to be shown to work against every possible counter-strategy.

> Computer chess has developed much as genetics might have if the geneticists had concentrated their efforts starting in 1910 on breeding racing Drosophila. We would have some science, but mainly we would have very fast fruit flies. – *John McCarthy, 1997* [McC97]

After about half a century of game playing research, the contributions of games research to Artificial Intelligence are not often fully acknowledged. Yet game playing has been a driving force behind many Artificial Intelligence developments. Several important ideas, techniques, and algorithms that are now common tools originated in game playing research.

- **Iterative deepening** is a search technique that performs successive searches that are constrained by certain parameters. The parameters, usually involving the search depth[1], are adjusted at the start of each iteration to increase the scope of the search.

The common reference to iterative deepening in Artificial Intelligence is to Korf's 1985 paper [Kor85]. Yet iterative deepening was already described by Scott in 1969 [Sco69], referring to his chess playing program.

- **Memory-assisted search**, where results are cached for re-use later during the search, is a generalization of the transposition tables[2] used in virtually all game playing programs.

Transposition tables were introduced by Greenblatt in 1967 [GEC67], also in a paper involving a chess program. The technique works especially well in tandem with iterative deepening, as described in Slate and Atkin's famous 1977 paper about their program Chess 4.5 [SA77].

- **Reinforcement learning** is a learning environment where there is no teacher who knows what the correct action is; rather, training is based on a future reward depending on the success of the student's actions.

Temporal Difference learning, a well-known and effective reinforcement learning algorithm, is mainly known for its success in the game of Backgammon [Tes95]. The algorithm was first introduced and analyzed by Sutton [Sut88] in a paper that was not specifically games related, although he did use game playing as an example. Yet reinforcement learning was already pioneered more than twenty years prior, in Samuel's seminal work on checkers [Sam59, Sam67].

---

[1] See Section 2.2.

[2] See Section 2.4.

- **Brute force search** is the foundation for almost all state-of-the-art game playing programs today, but is also widely used in other areas of Artificial Intelligence.

While brute force methods have always been used in other areas as well, it is perhaps game playing that has been responsible for the acceptance of brute force as an Artificial Intelligence technique. Indeed, it is misleading to think there is no intelligence in brute force. Large search trees implicitly contain a high degree of knowledge. A common observation used to be that "dumb chess programs" had to do a lot of search to compensate for their lack of knowledge, but it is more accurate to put it the other way around: heuristic knowledge is a compensation for lack of search. Explicit heuristic knowledge, by its very nature, is an imprecise approximation of the truth. By contrast, implicit knowledge from massive searches is exact knowledge, and is thus of higher value.

1.2 Problem description

Traditionally, game playing research has focussed mainly on chess. Several decades of research have produced some powerful techniques, mostly geared at the efficient traversal of large game trees. It has also produced some notable triumphs; humans have been surpassed by programs in games such as checkers [Sch97], Scrabble [Sch00], and Othello [Bur97], while other games such as Connect-4 [UHA89], Nine Men's Morris [Gas90], and Go-Moku [All94] have even been solved.

With the advent of the checkers world champion program *Chinook* and the chess program *Deep Blue*, researchers started to realize that the techniques that drive these programs had been all but stretched to their limits. Yet there are other classes of games for which these methods would be of little use in constructing a program that can play on par with the strongest humans. These classes include the *imperfect information games* such as bridge and poker, where not all of the information is available to each player, as well as the *stochastic games* such as backgammon, where the player's options are partially determined by chance.

Another class is the one containing games whose *branching factor*, defined as the typical number of available options for a player when it is time to make a move, is too large to make brute force tree search algorithms feasible. A direct relationship between playing strength and search depth exists for many games, such as chess [Tho82]. Due to the exponential nature of the search tree, a large branching factor significantly reduces the possible search depth, which in turn diminishes a program's performance. The most well-known of these games is the Oriental board game Go, for which no strong programs exist despite considerable effort and expertise that has been devoted to it.

Another member of the class of high branching factor games is Hex. It is similar to Go, but offers additional advantages as a subject for research due to the simplicity of the goal and the rules. The game has several interesting properties. It can be played on a board of any size, thus becoming arbitrarily complex in terms of the branching factor. The rules are simple, yet they give rise to elaborate strategic ideas. Thus, Hex offers an interesting game that bridges the complexity of chess and Go.

1.3 Objective

The objective of this thesis is to demonstrate the application of search and evaluation techniques to a game which offers particular challenges to both of these factors. The difficulty with search in Hex lies in the large branching factor, while the difficulty with evaluation function is of a more game-specific nature. It is not immediately evident how to identify computable concepts that are strategically relevant to Hex, and many concepts that are often used in game playing programs, such as material count and mobility, are of no use. The key idea behind the evaluation function used by Queenbee is the concept of the "best second-best alternative", which is naturally suited to two-player adversary games.

When dealing with a high branching factor, game playing programs are faced with the dilemma of choosing between exhaustive search or selective search. Both variants carry the risk of producing unreliable results; exhaustive search may do so because the search will be shallow, while selective search may do so because key moves are overlooked. The search techniques used in Queenbee form a generalization of more conventional methods, in which selectivity is effectively emulated by adjustable parameters.

1.4 Motivation

Hex is a fun game. Its rules are so simple that they can be described in one short sentence, yet behind these simple rules hide an unexpectedly deep and rich strategy and dynamic local tactics. The game is a prime example of "a minute to learn, a lifetime to master". Thus, the game is particularly interesting to game players.

Hex is a beautiful game. Its simple structure gives rise to intricate and intriguing mathematical properties. Several strategic properties can be proved by "reductio ad absurdum", without giving any information on the actual strategies that achieve these properties. At least one property ties into advanced topological theorems.[3] Thus, the game is particularly interesting to mathematicians.

Hex is a difficult game. Due to the issues mentioned in the previous section, it is surprisingly difficult to write a program that can do well against human players. Many algorithms that have been very successful in other board games are either inadequate or need to be generalized or modified to work well for Hex. The algorithmic complexity of Hex is likely to be very high.[4] Thus, the game is particularly interesting to computer scientists.

---

[3] The "Brouwer Fixed Point Theorem"; see Section 3.3.

1.5 Related work

The first Hex playing machine was devised by Shannon in 1953 [Sha53]. It was an analog machine that used an electrical circuit to represent the board. Moves were selected by measuring the potential across the playing field, and locating certain specified saddle points. Apparently the machine played reasonably well strategically, but its main weakness was endgame tactical play. Shannon reports the machine won about seventy percent of its games against human opponents when taking the first move, but it is unclear how strong the opponents were and what board size the machine played on.

Until recently, Hex had received only marginal attention from game programmers. Few Hex playing programs existed before Queenbee. To date, the strongest Hex playing program, and the only one other than Queenbee to rise above the level of human novices, is Anshelevich's program *Hexy* [Ans00]. Hexy was introduced in 1999 and tested extensively on the online games server Playsite,[5] where it achieved a rating close to that of the top human players. More detailed information about its playing strength is available in Section 7.1.

Rather than doing a large scale game tree search, Hexy employs a deep and non-uniform search to identify certain aspects of the position known as *virtual connections*. These virtual connections are used to guide a selective and relatively shallow game tree search, as well as in the evaluation function. Since Hexy is currently the strongest Hex playing program, it serves as an excellent benchmark to test Queenbee's strength. A comparison between the two approaches is presented in Section 8.2.

1.6 Summary

The contributions of this thesis can be summarized as follows:

- **First high-level Hex playing program.** Queenbee, originally written in 1995, became the first program to successfully play against experienced human players.
- **Application of non-uniform search methods in a high branching factor game.** Full-width search, the basis of many world class game playing programs, is not feasible in games

---

[4] Hex is an instance of a PSPACE-complete game; see Section 3.6.

[5] See `http://www.playsite.com/games/board/hex`.

with a high branching factor, such as Go. Indeed, humans typically are still stronger than computers at these games. The game of Hex is starting to become an exception to this rule.

- **New evaluation function idea.** The key behind Queenbee's selective search as well as its evaluation function is the concept of "two-distance", an new way to measure connectivity in a graph which appears to be naturally suited to two-player adversary games.

In addition, Queenbee has significantly enhanced theoretical knowledge of Hex play on small boards, by achieving perfect play on any board size up to 6 × 6.

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of the state of the art in search algorithms. The history, rules, and mathematical background of game of Hex are introduced in Chapter 3. To gain an understanding of the difficulty of the game, Chapter 4 explains the basics of Hex strategy. Chapter 5 details the two-distance evaluation that forms the heart of Queenbee, and Chapter 6 describes the search methods used by the program. An analysis of Queenbee's playing strength follows in Chapter 7. Conclusions and future work are presented in Chapter 8.

Chapter 2

Search algorithms

The subject of this thesis is the Hex playing program Queenbee. This chapter introduces the necessary background theory and information on search and learning algorithms. All these techniques have been useful to build high performance programs in other games. This chapter does not contain an exhaustive overview of all state-of-the-art game playing algorithms, but it presents all the algorithms that are incorporated into Queenbee.

2.1 Game tree

A two player game is formally defined as a five-tuple $\{C, c_0, M, L, S\}$, in which

C = a set of *board states*, also referred to as *positions*;
$c_0 \in C$ = an *initial position*;
$M : C \rightarrow \mathcal{P}(C)$ = a *successor* function;
$L \subset C$ = a set of *leaf positions*: $L = \{c \in C | M(c) = \emptyset\}$;
$S : L \rightarrow \Re$ = a *score function*.

A *game* is a sequence of board states $\{c_0, c_1, c_2, \ldots, c_m\}$ where $c_i \in M(c_{i-1})$ for $1 \leq i \leq m$ and $c_m \in L$. The outcome of the game is $S(c_m)$.

The two players, *White* and *Black*, take turns in moving from board state to board state. The set C contains all possible positions than can occur. The game starts in position c_0; without loss of generality, it can be assumed that White is to move in this position. The function M indicates which positions a player can move to from a given position. The game ends when a leaf position, or *terminal position*, is reached. At this point, the score function represents the payoff to one of the players. The payoff can be positive, zero, or negative. Again without loss of generality it is assumed that White receives the payoff. Thus it is White's task to maximize this score, while Black tries to minimize it.

The *game theoretic value* of a position is defined as the outcome of the game if both players follow *optimal play* starting from that position.[1] With optimal play, White always moves such that a score at least equal to the game theoretic value is reached, regardless of Black's choice of moves. Similarly, Black always moves such that a score of at most equal to the game theoretic value is reached. The game theoretic value $v(c)$ of a position c can therefore be defined recursively. For a *white position*, which is a position in which White is to move, we have:

$$v(c) = \begin{cases} S(c) & \text{if } c \in L; \\ \max_{c' \in M(c)} v(c') & \text{otherwise.} \end{cases} \tag{2.1}$$

This reflects the fact that White always chooses the continuation that guarantees the maximum possible outcome. Similarly, for black positions:

$$v(c) = \begin{cases} S(c) & \text{if } c \in L; \\ \min_{c' \in M(c)} v(c') & \text{otherwise.} \end{cases} \tag{2.2}$$

The game theoretic value can thus be computed by means of a recursive algorithm called *minimax*. Game playing programs actually use an equivalent variant of this algorithm, which computes the function $v^*(c)$ defined as $v^*(c) = v(c)$ for white positions and $v^*(c) = -v(c)$ for black positions. In other words, where $v(c)$ is the payoff for White at the end of the game, $v^*(c)$ is the payoff for the player to move. If $S^*(c)$ is analogously defined relative to whose turn it is, the calculation of $v^*(c)$ is identical for white and black positions:

$$v^*(c) = \begin{cases} S^*(c) & \text{if } c \in L; \\ \max_{c' \in M(c)} -v^*(c') & \text{otherwise.} \end{cases} \tag{2.3}$$

The algorithm that computes $v^*(c)$ this way is called the *negamax* algorithm.The set of positions it generates to compute the value $v^*(c)$ is called the *game tree*, with c being the *root node* of the tree. Any move in position c that leads to a position c' for which $v^*(c') = -v^*(c)$ is an optimal move; accordingly, the position c' is called an *optimal successor*. The *branching factor* of a game is defined as the average number of available moves in a position. In other words, it is the average cardinality $|M(c)|$ over all positions c in the game. If the branching factor is b, then the number of nodes that a search tree of depth d contains is of the order of b^d.

2.2 Evaluation function

For most games, the search tree is too large to enable the game theoretic value of the root node to be determined in a reasonable amount of time. To produce a reliable estimate for $v^*(c)$, a game

---

[1] The definition of the game theoretic value is more involved when there is the possibility of cyclic play, in which a sequence of moves can lead back to the same position, or infinite play, in which there exists an infinite sequence of moves that never leads to a leaf position. However, Hex is clearly non-cyclic and non-infinite.

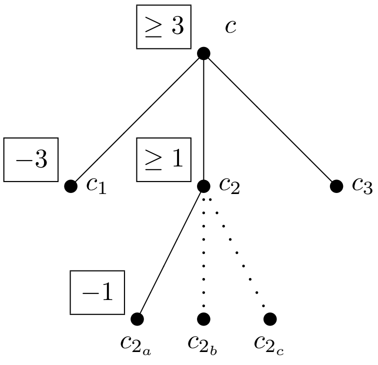


Figure 2.1: An α-β cutoff in a search tree

playing program uses an *evaluation function*. This function computes a heuristic estimate $h(c)$ of $v^*(c)$. The game playing program will expand a subtree of the game tree. This tree is called the *search tree*. For each leaf node c_{leaf} of the search tree, the heuristic value $h(c_{\text{leaf}})$ is computed by the evaluation function. *Interior nodes* are non-leaf nodes; for each interior node c_{int}, the heuristic value $h(c_{\text{int}})$ is backed up by the heuristic algorithm: $h(c_{\text{int}}) = \max_{c' \in M(c_{\text{int}})} -h(c')$. Or, equivalently:

$$h(c_{\text{int}}) = -\min_{c' \in M(c_{\text{int}})} h(c'). \tag{2.4}$$

The search tree is typically expanded to a fixed distance from the root. This distance is called the *search depth*; it is commonly measured in *ply*, where one ply represents one single move by either player. As the search depth increases, the accuracy of the root node value increases. The limit case is reached if the search depth equals the maximum length of the game, in which case the value that is backed up to the root node equals the true game theoretic value provided $h(c) = S^*(c)$ for terminal nodes $c \in L$.

2.3 Alpha-beta search

To find the root node value, it is not necessary to expand the full search tree. Consider Figure 2.1, which depicts a subtree of the search tree. The root position c has three successor positions c_1, c_2, and c_3. Once it is established that $v^*(c_1) = -3$, it is known that $v^*(c) \geq 3$. These values are shown in boxes. As the minimax algorithm is backtracking through this tree, the next step is to compute $v^*(c_2)$. Suppose the value of the first successor position is determined to be -1. At that point, it is known that $v^*(c_2) \geq 1$, and therefore c_2 cannot be an optimal successor position as the value of node c is determined by the minimum value of its successor positions. Thus it is no longer necessary to calculate the values for c_{2_b} and c_{2_c}.

```
int alphabeta(position pos, int $\alpha$, int $\beta$, int depth) {
  int i, value, best;
  move_type moves[];

  if (depth $\equiv$ 0)          /* horizon reached */
    return(evaluate(pos));

  moves = generate_moves(pos);
  if is_empty(moves)     /* terminal position */
    return(evaluate(pos));

  best = -$\infty$;
  for i = 1 to size(moves) do {
    pos = make_move(pos, moves[i]);
    value = -alphabeta(pos, -$\beta$, -max($\alpha$,best), depth-1);
    pos = undo_move(pos, moves[i]);
    if (value > best)   /* new best move found */
      best = value;
    if (best $\geq \beta$)        /* $\alpha$-$\beta$ cutoff */
      break;
  }

  return(best);
}
```

game-specific auxiliary functions:

`evaluate(p)`: returns the heuristic evaluation of position p;
`generate_moves(p)`: returns the legal moves available in position p;
`make_move(pos, m)`: executes move m in position *pos* and returns resulting position;
`undo_move(pos, m)`: retracts move m in position *pos* and returns resulting position;

Table 2.1: The α-β search algorithm

The canonical algorithm that takes advantage of this observation is called the *alpha-beta algorithm*. It discards nodes from the search tree once it is known that they cannot influence the value of any of their parent nodes anymore. These deletions are called α-β *cutoffs*. The name "alpha-beta" refers to the parameters α and β used by the algorithm. The pseudo-code for the algorithm is given in Figure 2.1; the root node value is found by calling the algorithm with $[\alpha, \beta]$ equal to $[-\infty, +\infty]$. The parameters α and β contain the lower and upper bound of the range into which a value must fall if it is to influence the value of its parent. A *fail high* occurs when a node's value is known to be greater than or equal to its β. In the standard α-β algorithm, a fail high implies that the node can be cut off from the tree.

If the first node that the α-β algorithm expands is always an optimal successor, the number n of nodes that the resulting tree contains is of the order of

$$n = b^{\lfloor \frac{d}{2} \rfloor} + b^{\lceil \frac{d}{2} \rceil} - 1, \tag{2.5}$$

as analyzed by Knuth and Moore [KM75]. The size of the tree exactly equals this number if every node in the game tree has exactly b successors.

In the best case, the α-β algorithm effectively reduces the branching factor to its square root by eliminating provably irrelevant nodes. This makes it possible to generate a search tree that is twice as deep yet no larger than the tree that is generated by the minimax algorithm. However, the algorithm cannot be guaranteed to always expand an optimal successor node first in practice.[2] In the worst possible case the algorithm will always expand successor nodes in the order of worst-to-best, building the same search tree as the minimax algorithm. The order in which the successor nodes are expanded is therefore of critical importance. A game playing program needs to contain a *move ordering algorithm* that attempts to select the successor positions in the order of best-to-worst.

Game tree search algorithms are likely to encounter several problems, notably the horizon effect and the odd/even effect. The *horizon effect* occurs because the program will only expand the search tree to some depth d. If the program spots the threat of a strong move sequence for the opponent, it can often erroneously deal with this threat by introducing irrelevant moves that delay the threat. If these moves delay the threat long enough that it takes more than d moves to happen, the threat will have disappeared from the search tree and thus it will appear to have been circumvented. The term refers to the fact that the program cannot see beyond a certain horizon, the depth d, and fails to realize that a problem that has been pushed beyond this horizon has not actually disappeared. The delaying moves that the program plays to achieve this may often weaken the program's position.

The *odd/even effect* occurs because in most games it rarely happens that none of the available moves improve the player's position. This type of situation is called *zugzwang*, from the German word for "forced to move", as the player would prefer to skip the move but the rules of the game do not allow it. It can be proved, however, that zugzwang never occurs in the game of Hex: *any* move is better than no move at all, and nearly all moves are strictly better.[3] Since this means that each player always improves their position by making a move, heuristic evaluations of white positions are essentially incomparable to those of black positions. If the tree is not of uniform depth, meaning that the leaf nodes where the heuristic function is applied are not all at the same depth, then the search algorithm will back up incomparable values.

---

[2] If it could, it would no longer be necessary to generate a search tree at all.

[3] See Section 3.4.

To cope with this, the restriction can be imposed that leaf nodes must always have the same side to move. Suppose that the program generates a search tree of depth d with a white position at the root. If d is even, then all the leaf positions in the tree will be white positions, and so white will have made the last move in every branch of the tree. If on the other hand d is odd, black will have made the last move. The result is that the leaf node evaluations will generally be more in favour of white when d is even, and more in favour of black when d is odd. This can not only cause the program to assess the root node differently, but it can even cause it to choose a different continuation depending on the parity of d. Generally, the program will tend to play more aggressively if d is odd.

2.4 Enhancements

Modern game playing programs use several powerful techniques to improve search efficiency. One of these techniques, *transposition tables* [GEC67, SA77], exploits the phenomenon that the game tree is actually not a tree but a graph in which a node can have more than one parent. A *transposition* occurs when the same position can be reached in two different ways. The transposition table stores the results of the searches for as many nodes encountered in the search as possible. Whenever a previously searched node is reached via a different path, the result of the earlier search can be retrieved from the transposition table. The savings introduced by a transposition table can be very large, depending on the size of the table and the frequency with which transpositions occur. Transposition tables not only store information about the result of the search in a particular position, but also information about the best move found in that position.

Another common technique is called *iterative deepening* [SA77]. The program will start with a 1 ply search, then repeatedly start new searches to successively larger depths until it runs out of time. This method has several important advantages. It is generally not easy to predict ahead of time how long a search to a particular depth is going to take. By using iterative deepening, the program will automatically reach the maximum possible search depth d_{max} in the allotted time. Moreover, the program will always have a move available to play, even when the search is interrupted for some reason. The overhead of first performing all the shallower searches before reaching the search of depth d_{max} is small; due to the exponential nature of the tree, the overall search effort is only increased by a constant factor. With the improved move ordering based on the information gathered during the shallower searches, which is available through the transposition table, an iterative deepening search often even expands fewer nodes than one isolated fixed-depth search to the same depth.

Aspiration search attempts to reduce the tree size by guessing a range $[v_{min}, v_{max}]$ into which the root node value is likely to fall. This range is called the *search window*. At the start of the search, the parameters $[\alpha, \beta]$ for the root node are initialized to $[v_{min}, v_{max}]$ rather than $[-\infty, +\infty]$. If the search eventually returns a value that does fall within the window, then the value is known to be correct. If a fail high occurs, it is known that the value must be larger than v_{max}. In an aspiration search, a fail high at the root node does not lead to a cutoff. Rather, the position must be searched again with a different window. Similarly, the search might fail low by returning a value smaller than v_{min}. Choosing a small window size can greatly reduce the size of the tree, but it also increases the risk of having to re-search the position.

The *Principal Variation / Minimal Window Search* algorithm, PVS/MWS [MC82], is a sophisticated refinement of the basic α-β algorithm. It recognizes that whenever a move with the highest value is searched, the values for the other successor positions need only be proved to be inferior. To prove that the value of a node c is less than some value k, the PVS/MWS algorithm performs a search with a window of minimal size $[k - \epsilon, k]$, where ϵ is the granularity of the evaluation function. If the value of the first searched node was indeed the highest, the PVS/MWS algorithm will generally be able to prove the other successor nodes to be inferior very quickly due to the small window size. The pseudo-code for the algorithm is given in Figure 2.2, where the minimal window size is assumed to be 1. The name of the algorithm is derived from the *principal variation*, which is the sequence of moves that leads from the root node to the node whose value was eventually backed up to the root node. In other words, it is the sequence of moves that were judged to be the best move in their respective positions. The principal variation is the expected line of play.

Rather than expanding a search tree of a fixed depth, it is often advantageous to explore some variations deeper than others. If a particular line has little chance of becoming the principal, the search can be aborted at a shallower level in order to save the search effort. This is known as a *search reduction*. Similarly, a *search extension* can be created in a line that looks promising or critical. The search tree that is generated in this way is no longer fixed-depth. An algorithm that uses search extensions or reductions must contain heuristics to judge when an extension or reduction is appropriate. Typically, reductions are generated in lines where one player's position has deteriorated significantly, indicating that the line is likely to contain inferior moves and is therefore irrelevant. Extensions tend to be used in lines that are unsettled but whose exact resolution is critical.

When a game playing program is to play games under tournament conditions, time management becomes an important issue. Many modern game playing programs contain code that enable the program to think while the opponent is to move. Thinking on the opponent's time is known as *pondering*. Many programs do this by guessing the most probably move for the opponent, and starting to calculate a reply while the opponent is still thinking. If the opponent plays the expected move, the reply can be played relatively quickly because part of the work has already been done. Alternatively, the program can still spend the same amount of time on its move to complete a deeper search. Another method of pondering is to do a search on behalf of the opponent. This fills the transposition table with valuable information that subsequently speeds up the search for the program's reply move.

```
int PVMWS(position pos, int $\alpha$, int $\beta$, int depth) {
  int i, value, best;
  move_type moves[];

  if (depth $\equiv$ 0)                          /* horizon reached */
    return(evaluate(pos));

  moves = generate_moves(pos);
  if is_empty(moves)                        /* terminal position */
    return(evaluate(pos));

                                            /* search first position */
  pos = make_move(pos, moves[1]);
  best = -PVMWS(pos, -$\beta$, -$\alpha$, depth-1);
  pos = undo_move(pos, moves[1]);

  $\alpha$ = best;
                                            /* search remaining positions with null window */
  for i = 2 to size(moves) do {
    pos = make_move(pos, moves[i]);
    value = -PVMWS(pos, -$\alpha$-1, -$\alpha$, depth-1);
    if (value $>$ best)                       /* fail high, re-search?  */
      if (value $> \alpha$) and (value $< \beta$)
                best = -PVMWS(pos, -$\beta$, -value, depth-1;)
    pos = undo_move(pos, moves[i]);
    $\alpha$ = max($\alpha$, best)
    if (best $\geq \beta$)                        /* $\alpha$-$\beta$ cutoff */
      break;
  }

  return(best);
}
```

game-specific auxiliary functions:

`evaluate(p)`:	returns the heuristic evaluation of position p;
`generate_moves(p)`:	returns the legal moves available in position p;
`make_move(pos, m)`:	executes move m in position *pos* and returns resulting position;
`undo_move(pos, m)`:	retracts move m in position *pos* and returns resulting position;

Table 2.2: The MWS/PVS algorithm

game	program	author	b	$\log b$	d	$d \log b$
checkers	Chinook	Schaeffer	3	0.5	23	11
awari	Bambam	Univ. of Alberta	4	0.6	20	12
Othello	Logistello	Buro	12	1.1	12	13
chess	Crafty	Hyatt	35	1.5	9	14
Chinese chess	Abyss	Marsland	50	1.7	8	14
10 × 10 Hex			80	1.9	7	
14 × 14 Hex			160	2.2	6	
19 × 19 Go			300	2.5	< 5	

b = typical branching factor;
d = typical full width search depth for computers.

Table 2.3: Search depth compared to branching factor on a current single processor PC

2.5 Selective search

The exponential nature of game trees limits the depth to which full width search algorithms can explore them under feasible time constraints. This depth limit depends mostly on the branching factor of the game. Table 2.3 lists estimates for the typical branching factors of some common games, and the depths to which state-of-the-art search algorithms can explore them. Due to the use of search extensions and reductions, the search depths are not quite fixed; the table refers to the typical depth to which relevant lines are explored. A time constraint of three minutes is assumed, which corresponds to normal tournament conditions.

The table also contains the estimated full width search depths for Hex and Go, based on the fact that $b \log d$ is roughly the same in every case. The latter fact is a consequence of equation 2.5. The search depths give a good indication of how well computer programs perform in comparison to top human players if ones assumes that top human players search about eight to ten ply deep. The games of Awari and Othello are different; due to the highly nonlocal nature of the moves[4] humans have difficulty searching more than a few ahead. The reachable search depth also depends on the effort involved in computing the evaluation function; Go programs tend to have a complex and slow evaluation function, whereas for example the Awari program Bambam uses an evaluation function that is easy and quick to compute.

Comparing the estimated search depths for humans and computers reveals a good correspondence with the actual balance of power for various games. Computers are significantly better than humans at checkers, almost as good at chess, and not quite as good at Chinese chess. Computers are far better at Awari and Othello, and humans are far better at Go. The numbers also indicate that Hex can be anywhere in the spectrum between chess, Chinese chess, and Go, depending on the size of the board.

---

[4] A move is nonlocal if it changes the state of most of the board, which happens frequently in Awari and Othello.

The reason why humans can look as far ahead as they do is that they employ a *selective search*. Based on experience, insight, and intuition, they quickly dismiss many branches as "uninteresting", and focus only on the interesting ones. In Hex, the ability of humans to search deeply is enhanced by their knowledge of concepts like edge patterns, ladders, and outposts, which will be explained in Chapter 4. For example, the outpost example in Figure 4.9 would require a 12 ply search, but experienced human players can spot the outcome immediately without any lookahead.

The game of Go is possibly the most difficult game for computers. Where programs of near or beyond world championship strength exist or will exist in the near future for most other games, the development of a world calibre Go program is likely to be decades away. Hex can be played on any board size, and the search depth for human players decreases very slowly, if at all, with increasing board size. Thus, Hex becomes relatively more difficult for computers compared to humans on progressively larger board sizes. Because of this, Hex may be viewed as an interesting intermediate goal between the landmark events of creating a world calibre chess program and creating a world calibre Go program. Hex provides an excellent test bed for studying selective search, which will be indispensable for championship Go programs.

2.6 Machine learning

Machine learning applications in game playing can currently be divided into three areas: evaluation function learning, opening book construction, and learning search control. The first of these three areas has always received the most attention, while the other two are comparatively new.

An evaluation function typically takes as input a number of *board features* that have been computed from the game position, and feeds these into a function that returns a single number. This function is often a linear combination of the inputs, but it does not need to be. The function usually contains a number of parameters that express the relative importance of the input features. Automatic tuning of these parameters was explored as early as the 1960s by Samuel in his checkers program [Sam59, Sam67]. The formulation of the Temporal Difference algorithm by Sutton [Sut88] led to impressive results for the game of Backgammon by Tesauro [Tes95].

Any learning entity faces the "exploration versus exploitation" dilemma; the learner must explore in order to learn new things, but it must also use what it has learned so far in order to increase its performance. The roll of the dice in Backgammon naturally forces the learner to explore. Following Tesauro's work, it was therefore often conjectured that the stochastic nature of Backgammon was particularly suited for Temporal Difference learning. Yet more recent results indicate that reinforcement learning can also be applied successfully to deterministic games. Baxter, Tridgell, and Weaver developed a variant on Sutton's algorithm, called TDleaf(λ), that achieved remarkable results in chess [BTW98].

A more difficult problem in evaluation function learning is the discovery of the input features themselves. Buro applied massive regression methods to the game of Othello, eventually identifying a set of very useful board patterns [Bur98]. Utgoff experimented with a function approximating algorithm that constructs features while learning, applied to checkers [UP98]. The author of this thesis applied data mining techniques to large end game databases to discover evaluation features for the game of Awari [Rij00].

Automatic construction of opening books has become an important issue in computer chess, where many programs have a commercial interest in being the strongest. In the early 1990s programmers were engaged in a constant battle to produce “killer books” that were able to exploit weaknesses in other programs’ opening books in order to win games right out of the opening. This eventually forced the programmers to develop opening book learning methods, so the programs would not lose a game twice in the same way. Automatic opening book construction methods are discussed in papers by Hyatt [Hya99] and Buro [Bur99].

The newest area in learning relates to the parameters that control the search itself. Many programs use search extensions and reductions of a fractional number of plies.[5] Björnsson developed an algorithm to learn these numbers automatically [Bjö00].

The significance of these three applications of machine learning is that they all focus on aspects of game playing programs that used to have to be tuned by hand, which constituted the most labour-intensive part of building a high performance program. Automatic learning provides a way to perform this tuning much more quickly and, ideally, more effectively. This is an important development in computer game programming, freeing up the programmer’s time and energy to dedicate to other parts of the program.

---

[5] See Section 6.2.

Chapter 3

Hex

Hex is a board game with simple rules, but a complex strategy. Indeed, winning strategies are only known for board sizes up to 7×7,[1] where the game is commonly played on sizes of 10×10 or larger. The game is a special case of a more general graph colouring game known as the *Shannon switching game*, which was proved to be PSPACE-complete. This chapter presents the history and the rules of Hex, as well as some special properties of the game.

One of the properties of Hex is that the game can never end in a draw. A formal proof of this property is given in Section 3.3. Another interesting property is that Hex can be proved to be a theoretical win for the first player. It was John Nash who first realized this. The simple proof of this fact, given in Section 3.4, is however only a proof of existence. No actual winning strategy is known. This makes Hex what is known as "ultra-weakly solved". A general winning strategy is likely to be very hard to find, as will be shown in Section 3.6.

3.1 History

Hex was invented by Danish engineer, poet, and mathematician Piet Hein (1905–1996) at the Niels Bohr Institute for Theoretical Physics at the University of Copenhagen in 1942. It is said he invented the game while contemplating the then unproved four-colour theorem of topology. Piet Hein called the game "Polygon" and published a series of articles on the game in a leading Danish newspaper.

The game was independently rediscovered by mathematics graduate student John Nash (1928–) in 1948 at Princeton, where it became known as "Nash". It should be noted that the often retold story

---

[1] See Section 4.6.

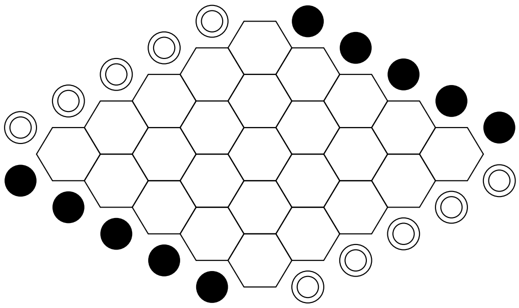


Figure 3.1: A 5 × 5 Hex board

of the game being referred to as “John” since it was often played on the bathroom tiles at Princeton is merely an amusing myth [Nas99]. John Nash invented Hex as an example of a game where no explicit winning strategy was known but that was still a provable win for the first player. The name “Hex” was introduced by Parker Brothers, who marketed the game under that name in 1952.

Hex was again introduced by Martin Gardner in 1959 [Gar59]. Piet Hein marketed the game under the name “Con-Tac-Tix” in 1968. Since then, it has been produced by several different game companies. In the 1990s a Hex playing community emerged on the Internet, first through a play-by-email server, and later a games server that allows live play using a Java interface.[2] Hex continued to gain popularity, witnessing the publication of the first book devoted entirely to the game [Bro00] and being included in the *Olympic list* of games to feature at the 2000 Computer Olympiad.[3]

3.2 Rules

Hex is played on a rhombic hexagonal pattern, as in Figure 3.1. This particular Hex board has 5 × 5 cells, but the game can be played on boards of any size. The board has two *white borders* and two *black borders*, indicated in the figure by rows of discs placed next to the borders. These *edge pieces* are not part of the game; they merely serve as a reminder for the players. Note that the four corner cells each belong to two borders.

Play proceeds as follows. The two players, henceforth to be called *White* and *Black*, take turns placing a piece of their colour on an empty cell. In Hex there is no standard convention on which colour gets the first move. White wins the game by connecting the two white borders with a chain of white pieces, while Black wins by connecting the two black borders with a chain of black pieces.. In Figure 3.2, Black has completed a winning chain.

---

[2] See `http://www.gamerz.net/pbmserv/hex.html` and `http://www.playsite.com/games/board/hex`.

[3] See `http://www.msoworld.com/Olympiad/computer.html`.

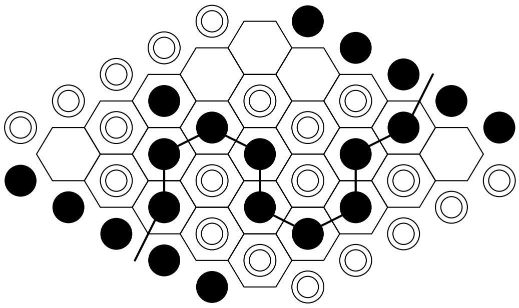


Figure 3.2: A winning chain for Black

Despite the simplicity of the rules, Hex strategy is surprisingly deep and subtle. Chapter 4 contains an overview of the most important strategic concepts. Note that the condition for winning the game is equivalent to having connected the leftmost edge piece and the rightmost edge piece.

3.3 No draws

The fact that Hex cannot end in a draw is usually accompanied by a "handwaving argument", but there is a fairly simple formal proof. This proof was first given by Gale [Gal86], who also showed that this intuitively obvious fact is mathematically "deep" in that it is equivalent to, and leads to a very short proof of, the Brouwer Fixed Point theorem.[4]

Consider a Hex board in which all the cells are occupied, and add four appendages n, s, e, and w to it, as in Figure 3.3-I. Start at the vertex labeled w, and trace a path P by turning either left or right at every intersection in such a way that the path keeps running between cells of different colours. The resulting path is shown in Figure 3.3-II.

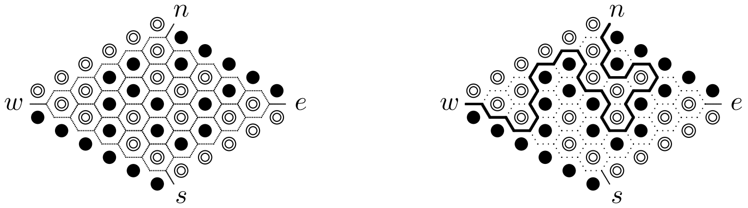
Lemma: This procedure determines a unique path.

Proof: Every vertex has exactly three cells bordering on it, and each of those cells will be coloured. Whenever P arrives at a vertex v, it is running between cells c_1 and c_2 of different colours. Suppose, without loss of generality, that c_1 is black and c_2 is white. If c_3, the third cell that meets at v, is black, then the path must necessarily continue between c_2 and c_3, as c_2 and c_3 are of different colours and c_1 and c_3 are not. Similarly, if c_3 is white, P must continue between c_1 and c_3. □

Thus P continues at every vertex where three edges meet. It never visits the same vertex twice, as

---

[4] The two-dimensional version of this theorem says that if f is a continuous mapping from the unit square I^2 to itself, then there exists $x \in I^2$ such that $f(x) = x$. There exist natural generalizations into higher dimensions of both the Hex no-draw theorem and the Brouwer theorem, which are equivalent as well.



I: A completely filled board II: The path

Figure 3.3: The path that identifies a winning chain

can be seen from the fact that after P visits vertex v, two of the three edges that meet at v have been used up and the third one will never be used as it has equally coloured cells on both sides. As the number of vertices is finite, P must terminate, and the only vertices at which it can do so are n, s, and e.

Lemma: P identifies a winning chain for White or Black.

Proof: If the path ends at n, as it does in Figure 3.3, then the path identifies a winning chain for Black. The reason is that all the cells along one side of the path are of the same colour, the origin of the path at w borders on a black cell on one black border, and the destination of the path at n borders on a black cell on the other black border. The black cells along one side of the path therefore form a chain connecting the two black edges. Similarly, if the path ends up at s then it identifies a winning chain for White. If the path were to end up at e, then there would be winning chains for both players. □

If a game of Hex is played out until the entire board is filled, the lemmas show that there must be a winning chain for one of the players. Thus the game cannot end in a draw. Note that P must terminate in n or s. The path cannot actually end up in e, as it always has black cells on the left and white cells on the right when oriented in the direction in which it is traced, which is not he case for the edge running into vertex e. This fact is not important for the proof, however.

3.4 First player wins

John Nash quickly realized that Hex is a win for the player who goes first. The proof is fairly simple, but it is only a proof of existence. No actual general winning strategy is known. The proof is based on the "strategy stealing argument", which informally says that if there were a winning strategy for the second player, then the first player could "steal" it and use it to win.

Theorem: Hex is a first player win.

Proof: Suppose there were a winning strategy S for the second player. Without loss of generality, say that White goes first, so Black can win by applying S. But then White can also win by playing an arbitrary opening move, and then applying strategy S. After making the opening move, White in effect *becomes* the second player in a new game, in which White goes second and has an extra "bonus" piece already on the board. Whenever S requires White to play a move in an already occupied cell, White makes another arbitrary move. Following this strategy, White wins. As it is impossible that both players win, a contradiction is reached. Hence there can be no winning strategy for the second player. $\square$

Note that this proof relies on the crucial fact that the extra move can *never* be a disadvantage for Black. This is not the case in other games, such as chess and Go.

In practice, there turns out to be a moderately large advantage for the first player in games between human players. One therefore often uses *one move equalization*, also known as the *swap rule*: One player plays an opening move with a white piece, the other player then gets to choose the black or the white pieces. In other words, the other player gets the opportunity to "swap", or switch sides. It is analogous to the *"I cut, you choose"* principle for dividing a cake fairly. The variant of Hex that employs the swap option shall be referred to as *competitive Hex*, as opposed to *unrestricted Hex* where the second player is not allowed to swap. Unless otherwise stated, competitive Hex shall be the default.

After the first move and a possible swap, play continues normally without any more swaps. This way, the first player cannot play a strong opening move, because then the second player will swap and gain the advantage, but the first player cannot play a weak opening move either, because then the second player will *not* swap and retain the advantage. The best the first player can do is to play an opening move that draws, but as draws are impossible in Hex, the swap rule actually turns Hex into a theoretical win for the second player. In practice, the swap rule evens the game out sufficiently for human play.

3.5 Shannon switching game

Hex can be viewed as a graph colouring game. The 5×5 Hex board is drawn as in Figure 3.4. Players now take turns colouring vertices black and white. Four of the vertices are already coloured at the beginning of the game, as in the diagram. It is Black's task to connect the vertices b_1 and b_2 with a connected path of black vertices, while White tries to achieve a connected path of white vertices linking w_1 and w_2.

In this form, Hex is actually a special case of a more general graph colouring game called *SSG*: the *Shannon switching game.* In this game, a graph Γ is chosen with two distinguished vertices s and t. Players take turns colouring a previously uncoloured vertex. White wins by connecting s and t by a path of white vertices, while Black wins by establishing a black *cut set* that disconnects s from t. This game can be played on any graph. The graph in Figure 3.4 can be turned into a SSG graph

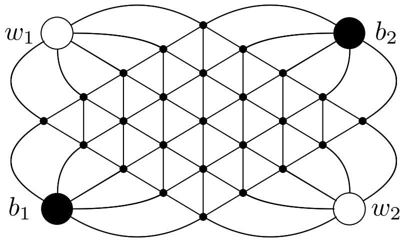


Figure 3.4: 5 × 5 Hex as a graph colouring game

with equivalent game properties by deleting the nodes b_1 and b_2, and let w_1 and w_2 play the role of s and t. When given in the guise of a SSG, the players White and Black are usually referred to as *Short* and *Cut*, respectively.

SSG can also be played by colouring the *edges* of the graph, rather than on the vertices. There actually exist efficient polynomial-time algorithms for finding a winning strategy for SSGE, the Shannon switching game on edges [BCG82]. A winning strategy for the Shannon switching game on vertices, SSGV, is much more difficult to find. The situation is analogous to the difference in complexity of finding an Euler cycle and a Hamilton cycle in a graph, where the latter problem is NP-complete while the former problem is trivial. If Γ happens to be the line graph of some graph Δ, then both the Hamiltonian cycle problem and the generalized Hex problem can be solved by finding the corresponding solutions on the edges of Δ.

3.6 Computational complexity

Theoretical evidence to support the fact that SSGV is a difficult game was supplied by Even and Tarjan [ET76], who proved that the game is PSPACE-complete. The proof uses a reduction from *QBF*, the *Quantified Boolean formula* problem:

$$QBF = \{Q_1 x_1 Q_2 x_2 ... Q_m x_m F\},$$

where the Q_i are quantifiers, the x_i are Boolean variables, and F is a formula in conjunctive normal form with variables $x_1, ..., x_m$. This problem is known to be PSPACE-complete. Any QBF can be represented by a graph Γ such that $(Q_i x_i)F$ is true if and only if Short wins the SSGV on Γ. The graph Γ can be constructed in log-space.

A problem is in PSPACE if there exists an algorithm for solving the problem that requires an amount of memory space that is polynomial in the problem size. In the case of SSGV, the problem is determining whether Short or Cut wins the game. This can be determined by a straightforward depth-first search, corresponding to the minimax algorithm. The amount of memory needed for an exhaustive search is $O(n^2 \log n)$, where n is the number of vertices of the game graph. Hence SSGV is in PSPACE.

The reduction to QBF shows that SSGV is PSPACE-complete. The boolean satisfiability problem SAT, the canonical NP-complete problem, can be solved in polynomial space and is in fact a special instance of QBF. Thus SSVG is "at least as difficult" as NP-complete problems. Even and Tarjan observe that there is no obvious way to determine the winner in polynomial time, even when allowed a nondeterministic algorithm. They therefore suspect that SSGV is actually strictly harder than NP-complete problems. Indeed it is difficult to imagine a way to verify a solution to SSGV in polynomial time.

As Hex is a special case of SSGV, it might therefore have some structure that makes it easier than generalized SSGV. An example of a type of graph where SSGV is "easy" is the class of line graphs; if a graph Λ is known to be the line graph of a graph Γ, then the Shannon switching game on the vertices of Λ can easily be won by playing the switching game on the *edges* of Γ, where as mentioned before an optimal strategy can be found in polynomial time. In the case of Hex, no efficient algorithms have been proposed yet.

As an aside, Even and Tarjan note that the Shannon switching game on the edges of a *directed* graph is also PSPACE-complete, by giving a construction to transform a SSGV-graph Γ into a directed graph on which the SSGE is equivalent to SSGV on Γ.

Chapter 4

Hex strategy

The game of Hex presents specific challenges to game playing programs. To understand these challenges, and to introduce the Hex jargon that will become relevant, this chapter presents an overview of Hex strategy. An understanding of the strategy will also accentuate the difficulty and subtlety of play for both humans and computers. An excellent treatise on this subject is contained in Cameron Browne's book *Hex Strategy: Making the Right Connections* [Bro00].

The Hex game notation used throughout this thesis is analogous to the familiar algebraic chess notation. The board cells are divided into rows and columns. The rows are denoted by numbers, the columns by letters. Figure 4.1 shows the cell coordinates for a 5 × 5 board. Hex board are often printed in different orientations, all of which are topologically equivalent. The coordinate system is equivalent to the one in Figure 4.1 if and only if the a1 cell is in an acute corner.

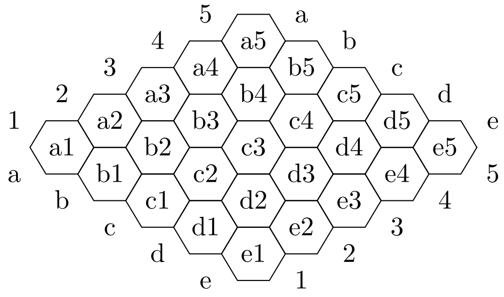
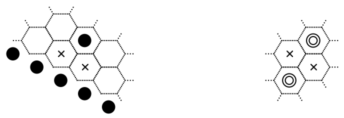


Figure 4.1: Cell coordinates on a 5 × 5 board



I II

Figure 4.2: Double threats

4.1 Double threats

The geometry of the Hex board allows the important concept of a *double threat*, as illustrated in Figure 4.2. The black stone on the left can be connected to the edge through either of the two cells marked '×'. If both those cells are still empty, White cannot prevent Black from connecting the stone to the edge, for if White plays in one of the two ×-cells, Black occupies the other one and connects. Thus, the black stone is already quite securely connected to the edge, even though the connection is not actually established yet.

This concept is called a *virtual connection.* A virtual connection is nearly equivalent to an actual connection.[1] The two white stones in Figure 4.2-II form a particular virtual connection known as a *two-bridge.* Two-bridges form the basis of Hex strategy.

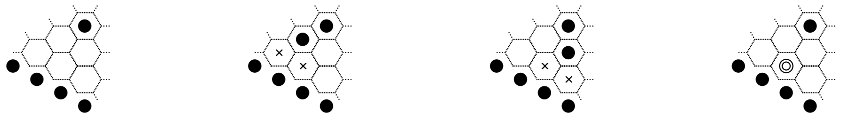
4.2 Edge patterns

The virtual connection in Figure 4.2-I is the most elementary example of an *edge pattern.* Where the white stone in Figure 4.2-I was only one row away from the edge, Figure 4.3-I depicts a white stone on the third row, which is two rows away from the edge. White's first threat is to play as in 4.3-II, establishing a virtual connection to the edge. To prevent this, Black must play in one of the three cells involved in this play; either the one where White threatens to play, or one of the two cells marked '×'. At the same time, White threatens to play as in 4.3-III, forcing Black to play in one of the three cells involved in that play as well. As there is only one cell that is involved in both these threats, Black's reply to block this connection must necessarily be the one shown in Figure 4.3-IV. This pattern is called the 3-triangle. It illustrates an important theme: if there are several threats, the defending player must play in the intersection of all the threats.

Figure 4.4 shows the common pattern for a virtual connection from the third row. As the white stone is involved in a 3-triangle pattern it is clear that Black must play as in 4.4-II to block the connection. But then White replies as in 4.4-II to connect anyway. Thus, the connection cannot be blocked. Another way of seeing this is shown in Figure 4.5. The threat in 4.5-I is the same as

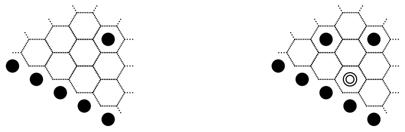
---

[1] Not *entirely* equivalent, as will be shown in Section 4.5.



I II III IV

Figure 4.3: The 3-triangle pattern: forced reply to block a stone on the third row

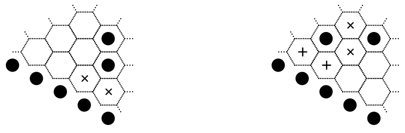


I II

Figure 4.4: A virtual connection from the third row

the one in 4.3-III. The threat in 4.5-II connects the white stone to the edge via two two-bridges indicated with '×' and '+', respectively. Since these threats are disjoint, Black cannot block both of them. The concept of *disjoint threats* is very important in Hex.

There are many more edge patterns. Figure 4.6 shows the simplest pattern for a virtual connection from the fourth row. Human players quickly learn to recognize these patterns, and can quickly assess or discard lines of play based on them. Yet to discover a virtual connection through search can be very taxing; the connection in Figure 4.6, for example, would require an additional 10 ply if the algorithm cannot detect the pattern statically.



I II

Figure 4.5: Two disjoint threats establish the virtual connection

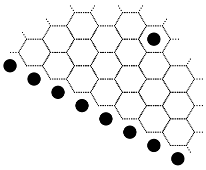


Figure 4.6: A virtual connection from the fourth row

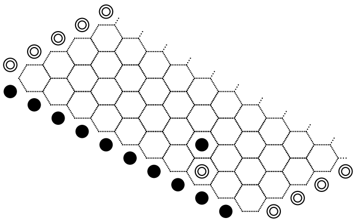


Figure 4.7: The start of a ladder

4.3 Ladders

After learning about two-bridges and edge patterns, the next important concept that human players discover is that of a *ladder*. Consider the position in Figure 4.7, with White to move. White wants to stop the black stone at g2 from being connected to the lower black edge. The only way of doing so is by playing at g1. If Black then threatens again to connect by playing f2, White again has only one move, this time at f1, to block the connection. This exchange of threats and blocks along an edge is called a ladder.

If both players keep “laddering”, eventually the position of Figure 4.8 will be reached. Black has run out of threats to connect the ladder to the lower black edge. Attempting to keep the ladder going by playing at b2 is a mistake, because White can then connect to the left white edge by playing at b1. In fact, Black is now forced to play at a2, as the white stone at c1 is involved in a 3-triangle. This, in turn, forces a reply by White at b2, which subsequently starts a new ladder that runs along the upper white edge; the ladder has “turned around the corner”. Ladders can turn around both the acute and the obtuse corners of the board.

An important realization is that Black has the initiative while the ladder is running along the lower black edge. Black can opt not to continue the ladder at any point, whereas White is forced to continue the ladder as long as Black does. When the ladder turns around the corner, the initiative switches from Black to White. In practice, ladders are never continued very far. The player with the initiative generally attempts to choose a good moment to interrupt the move sequence.

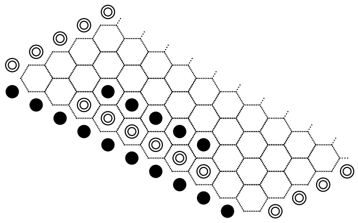
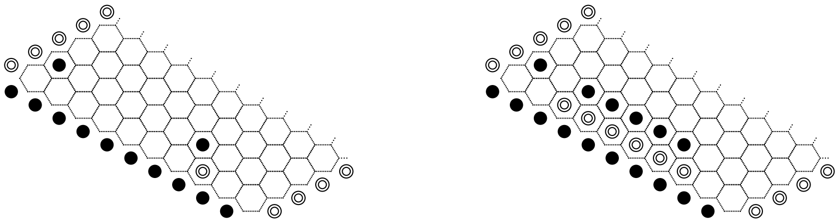


Figure 4.8: The ladder runs into the corner



I: White to move

II: Black to move

Figure 4.9: The outpost decides the outcome of the ladder

Ladders can occur any number of rows[2] away from the edge, but in practice they mostly happen on the second and third row from the edge, where the defending side is forced to keep laddering as long as the attacking side does. If the ladder is farther away from the edge, it can also be interrupted by the defending side. Ladders pose the same problem for Hex playing programs as they do for Go programs. They have to be resolved either by knowledge-based methods, or by very deep additional searches. Knowledge-based methods are hindered by the influence of seemingly unconnected stones on the outcome of the ladder, as will be seen in the following section. For the same reasons, knowledge-based approaches to resolve ladders have proven to be too dangerous in Go; state-of-the-art programs use narrow searches instead for this purpose.

4.4 Outposts

The position in Figure 4.9-I is similar to the one in Figure 4.7, the only difference being an extra black stone on a2. This stone drastically alters the fate of the ladder. If play continues as it did in Figure 4.7, eventually the position in Figure 4.9-II is reached. This time, Black *can* play at b2, because the stone at a2 secures a virtual connection to the lower black edge.

---

[2] In the context of edge patterns and ladders, “rows” strictly speaking refers to “rows or columns, depending on whether Black or White is the attacking side in the pattern.”

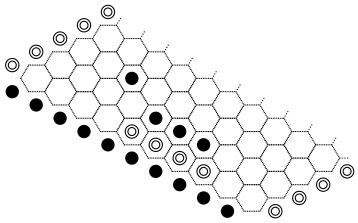


Figure 4.10: Another outpost example

The stone at a2 is called an *outpost*. Another example of an outpost is shown in Figure 4.10, where the outpost is at c3. Black proceeds to play c2 and connects the ladder to the lower black edge. Ladders can be influenced by outposts of either player.

Outposts indicate the depth of Hex strategy. The stone at a2 in Figure 4.9 may have been played there much earlier in the game. Such seemingly innocuous moves can have long term implications.

4.5 Forcing moves

In the position in Figure 4.11, White is to move. Applying the knowledge about two-bridges, White recognizes that the stone at c8 has a virtual connection to the lower white edge. If the stone at c8 could be connected to the upper White edge, White would win.

As for Black's position, White notes that the stones at d4 and d5 are connected to the stone at c7 via a two-bridge. The stone at d4 already has a virtual connection to the lower black edge through an edge pattern of the fourth row, as seen in Figure 4.6. The black stone at a9 already has a virtual connection to the upper black edge, because a white play at a10 would start a ladder that would eventually be decided in Black's favour by the outpost at g9. Thus Black threatens to win the game immediately by connecting the stones at a9 and c7. It therefore appears that White has no option but to play at b8, to stop this connection.

White's move at b8 starts a ladder along the upper white edge. With Black's stones at d4 and d5 being a dangerous influence, the ladder is likely to turn around the corner and continue on the second row along the lower black edge, where Black has an outpost on i2. All in all, White's prospects after playing at b8 are dire at best.

However, White has another option. The surprising move c6 threatens Black's two-bridge connection d5-c7. Black can of course restore the connection by replying at d6, and White's move initially looks futile. But White proceeds to play at b8, starting the same ladder as mentioned above. Soon the position in Figure 4.12 is reached. White can now win immediately by playing at b5.

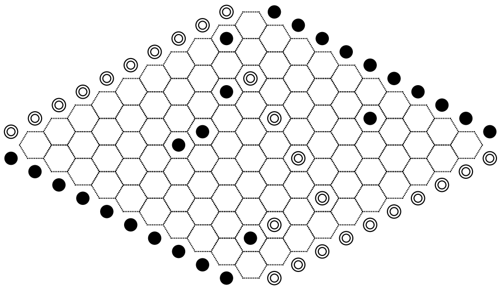


Figure 4.11: Forcing move position

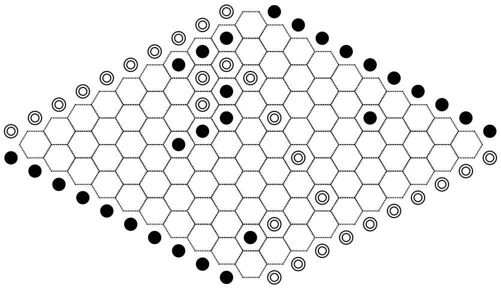


Figure 4.12: White's move at c6 established a winning outpost

In hindsight, White's seemingly useless move at c6 established an outpost which later decided the game in White's favour. A move that threatens a virtual connection is called a *forcing move*, because the opponent is forced to reply to it in order to restore the virtual connection. The power of a forcing move is that it can establish an outpost "for free"; in chess terms, the move does not lose a tempo, because the opponent is forced to reply to it.

Forcing moves can be crucial, and understanding them is important. They show that a virtual connection is not as strong as an actual connection. In reality, a virtual connection is only ever equivalent to an actual connection if its region is disconnected from the rest of the board.

The strategy goes deeper still. After White's move at c6, Black could argue that the reply at d6 is not really forced at all. Black might play some entirely different reply.[3] This would of course allow White to cut Black's connection at d5-c7, but meanwhile Black will have had the chance to play *two* moves elsewhere. These two moves might be more threatening than the d5-c7 connection. Of course, in turn, White might elect not to cut Black's connection at d5 after all, but play somewhere else. What complicates matters further is that Black, in this example, also has many "forcing moves" to choose from. Hex strategy can get highly complicated this way. It is reminiscent of Go, where forcing move fights occur also.

4.6 Opening moves

The opening move presents a particular challenge if the swap rule is in effect. As noted in Section 3.4, the first player cannot play either a very strong opening move or a very weak opening move. The choice would seem to be to play an opening move which leads to an equal position, but in Hex there are no equal positions since draws are impossible. In practice, the choice will be to play a move that makes it as difficult as possible to judge which side is stronger.

The winning opening moves for unrestricted Hex on board sizes up to 6 × 6 were first tabulated by Herbert Enderton [End]. Queenbee confirmed these results, and also computed the length of the perfect game following each particular opening move. A perfect game is a game in which the winning side achieves a winning position as soon as possible while the losing side delays it as long as possible; a winning position is defined as any position in which one player has a connection that may only be interrupted by simple two-bridges.

Figure 4.13 contains the results for the game where Black has the first move; circled numbers refer to winning moves in unrestricted Hex. For example, if Black plays the opening move c3 on a 6 × 6 board, it leads to a win in 16 moves. In contrast, when Black opens in cell a2, it leads to a loss in 19 moves. The latter result is surprising, since strong human Hex players believed this move to be a win. The length of the perfect game following an opening move can be taken as an indication of the difficulty of the opening. If the swap rule applies, then Black may want to choose an opening

---

[3] According to Queenbee this is indeed what happens. Black wins by playing the counter-threat d7, which is also a forcing move. Position 4.11 appears to be a loss for White after all.

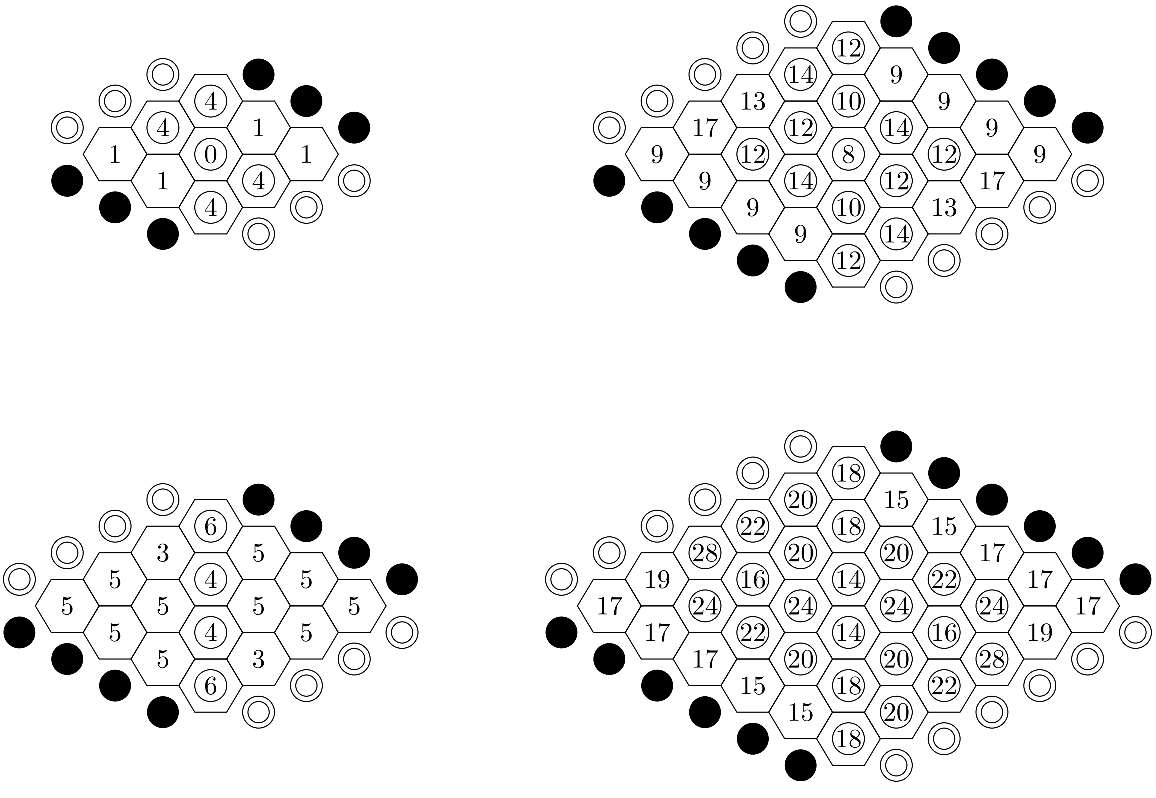

④
④ 1
1 ⓪ 1
1 ④
④
⑫
⑭ 9
13 ⑩ 9
17 ⑫ ⑭ 9
9 ⑫ ⑧ ⑫ 9
9 ⑭ ⑫ 17
9 ⑩ 13
9 ⑭
⑫
⑥
3 5
5 ④ 5
5 5 5 5
5 ④ 5
5 3
⑥
⑱
⑳ 15
㉒ ⑱ 15
㉘ ⑳ ⑳ 17
19 ⑯ ⑭ ㉒ 17
17 ㉔ ㉔ ㉔ ㉔ 17
17 ㉒ ⑭ ⑯ 19
17 ⑳ ⑳ ㉘
15 ⑱ ㉒
15 ⑳
⑱

Figure 4.13: Difficulty of opening moves for Black in unrestricted Hex

with a high number. The trend seems to be that moves near the acute corner, but not on the black border, are good candidates. Strong human players often play the opening move a2.

The largest board size for which winning strategies for unrestricted Hex from the opening position are known is 7 × 7. This strategy was described by Jing Yang [End99]. Anatole Beck [BBC69] proved that playing the first move in an acute corner is a theoretical loss on any board size larger than 1 × 1, so it is known that not all opening moves win.

Chapter 5

Queenbee's evaluation function

A game playing program needs a good evaluation function to help guide the search. It is not immediately obvious how to construct a meaningful evaluation function for Hex. For example, unlike in many other board games, in Hex the concepts of material balance and mobility are entirely unhelpful. This chapter contains new ideas for a Hex evaluation function. These ideas were implemented in the Hex playing program Queenbee. The function calculates the distance to each edge of all the unoccupied cells on the board, according to an unconventional distance metric called "two-distance". The resulting distances are referred to as "potentials".

5.1 Two-distance

Given a graph Γ with an adjacency function $n(p)$ that maps a vertex p onto the set of the vertices that are adjacent to it, there is a distance metric d_z that generalizes the conventional distance metric:

$$d_z(p,q) = \begin{cases} 0 & \text{if } q = p, \\ 1 & \text{if } q \in n(p), \\ \min_k c_k(p) \geq z & \text{otherwise,} \end{cases} \tag{5.1}$$

where

$$c_k(p) = |\{r \in n(p) | d_z(r,q) < k\}|.$$

The conventional distance metric corresponds to $z = 1$, in which case the distance of a cell to an edge on the Hex board represents the number of "free moves" that it would take for a player to connect the cell to the given edge. Unfortunately this distance function is not very useful for building an evaluation function for Hex, as will be shown later. Rather, the concept of *two-distance* is used,

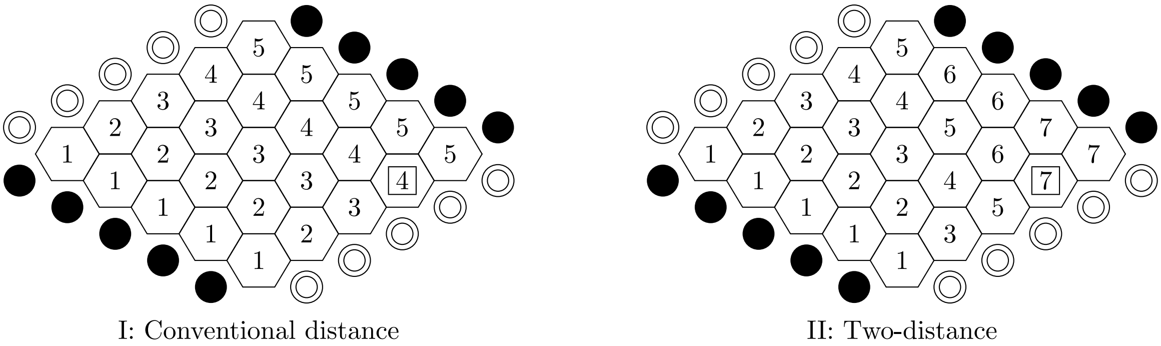

I: Conventional distance II: Two-distance

Figure 5.1: Comparison between conventional distance and two-distance to the lower black edge

where $z = 2$. The two-distance is one more than the *second lowest* distance of p's neighbours to q, with the proviso that the two-distance equals 1 if p and q are directly adjacent.

Figure 5.1 shows the distance of each cell to the lower black edge on a 5×5 board according to these two metrics. The intuition behind the two-distance idea is that, when playing a game, one can always choose to force the opponent to take the second best alternative by blocking the best one. Consider e4, the cell containing the boxed number in Figure 5.1. The conventional distance indicates that direct route to the lower black edge only takes four steps. However, there is only one path originating from e4 that achieves this distance. By contrast, the two-distance equals 7, as the best two adjacent connections are at distance 5 and 6. The two-distance thus captures the essence of "the best second-best alternative".

Notice that the rightmost cell on the board, e5, is at two-distance 7 even though its immediately adjacent neighbours are both also at two-distance 7. The number is nevertheless correct, since the calculation of two-distances needs to take into account the entire *neighbourhood* of a cell. There is an important distinction between adjacency and neighbourhood. Adjacency implies neighbourhood, but not vice versa. Two cells are adjacent if they share a common edge on the board. The notion of neighbourhood takes into account any black and white pieces that are already on the board. Two unoccupied cells[1] are neighbours from White's point of view if either they are adjacent or there is a string of white stones connecting them.

Note that a cell's neighbourhood can therefore be different from White's point of view than it is from Black's point of view. These two neighbourhoods will be referred to as the *W-neighbourhood* and the *B-neighbourhood*. Correspondingly, there will be a distinction between W-distance and B-distance. This explains why the cell e5 is at two-distance 7 from the lower black edge; due to the edge pieces, its B-neighbourhood contains the cells a5 and b5 which are at distance 5 and 6, respectively.

In Figure 5.2, two black stones are added to the board. Figure 5.2-I shows the B-distances of the

---

[1] Neighbourhood is only ever used for empty cells.

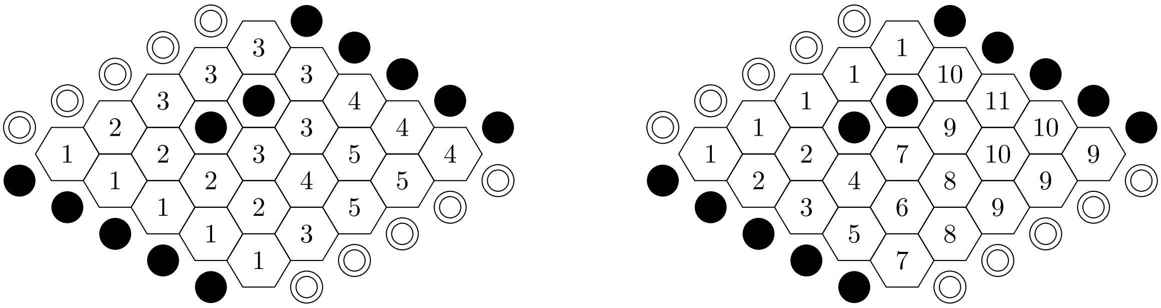

I: B-distance to lower black edge II: W-distance to upper white edge

Figure 5.2: Two-distances on a non-empty board

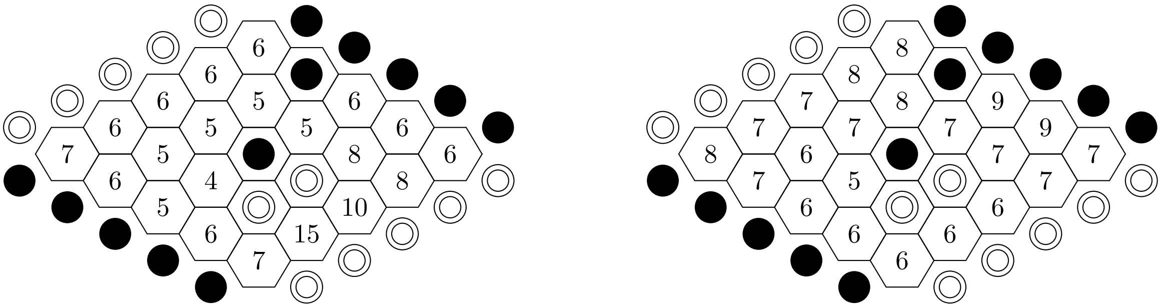
unoccupied cells to the lower black edge, and Figure 5.2-II shows the W-distances to the upper white edge. The addition of the two black pieces results in, for example, cells c2 and b5 being B-neighbours but not W-neighbours.

5.2 Potentials

The goal in Hex is to connect two sides of the board. To help achieve this, one might look for an unoccupied cell that is as close as possible to being connected to both sides, as this would be a promising candidate for being part of a winning chain. The evaluation function calculates *potentials* that capture this concept. Each unoccupied cell is assigned two potentials, based on the two-distance metric. A cell's W-potential is defined as the sum of its W-distance to both white edges; its B-potential is the sum of its B-distance to both black edges. Figure 5.3 shows the potentials for a position with two white and two black stones on the board.

Cells with low W-potentials are the ones that are closest to being connected to both white borders by White. If White can connect a cell to both white borders, this would establish a winning chain. White will therefore focus on those cells that have the lowest W-potentials. The white *board potential* is defined as the lowest W-potential that occurs on the board. In the example of Figure 5.3 the white board potential is 5, and the black board potential is 4. As lower potentials are better, it appears that Black is ahead.

In the same figure, it can be seen that both Black and White have only one cell that actually realizes their board potential. For both players it is the cell c2. It would be better to have more than one cell that realizes the board potential, so as to have more attack options and be less easy to block. The *attack mobility* is defined for each player as the number of cells that realize that player's board potential.



I: B-potentials

II: W-potentials

Figure 5.3: Cell potentials

Queenbee's evaluation function uses only the two concepts of board potential and attack mobility. The evaluation function returns the following number:

$$e = M(p_B - p_W) - (a_B - a_W), \tag{5.2}$$

where

p_W = white board potential;
p_B = black board potential;
a_W = white attack mobility;
a_B = black attack mobility;
M = a large number.

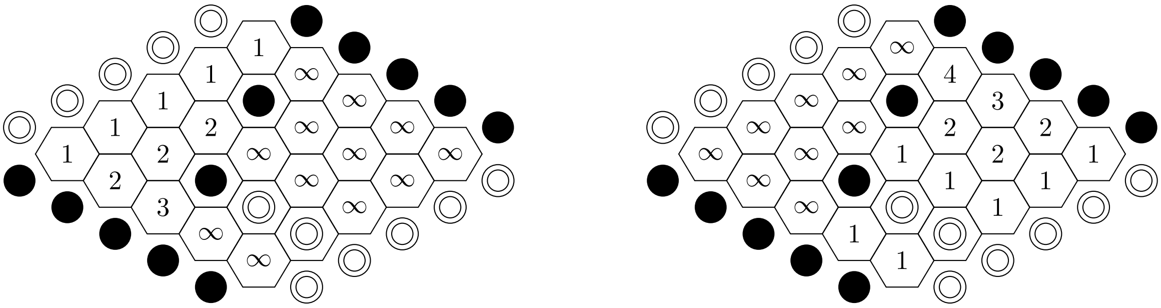
If M is set to be sufficiently large, the evaluation function will prefer one position over another if its board potential difference is better, and only use the attack mobility difference as a tie-breaker for positions with equal board potential difference. The potential of a cell, if it is finite, cannot be larger than $2n^2$ on an $n \times n$ board. Therefore, using a value for M of the order of magnitude of n^2 will achieve this. Queenbee uses $M = 100$.

5.3 Strategic relevance

The idea behind the two-distance metric is directly related to the importance of double threats.[2] Indeed the two-distance implicitly takes into account the two-bridges that occur in a Hex position. Consider the position in Figure 5.4. The white distance to each edge cannot percolate through the black two-bridges. As Black already has a winning connection made up of two-bridges, the result is

---

[2] See Section 4.1.



I: W-distance to upper white edge

II: W-distance to lower white edge

Figure 5.4: The two-distance cannot percolate through two-bridges

that the white board potential is infinite. Thus, the two-distance metric also implicitly recognizes a winning chain that consist of virtual connections through two-bridges, even if the chain is not actually solidified yet.

By contrast, using the regular distance metric in the position of Figure 5.4 would yield board potentials of 4 for both White and Black, suggesting that both players are equally close to establishing a winning connection. This indicates that the two-distance metric is far more suited to Hex than the conventional distance metric is.

5.4 Move badness

As mentioned before, White will want to play in cells that have a low W-potential, as those are the cells that are closest to being connected to both white edges. Simultaneously, White will also want to focus on cells that have low B-potentials. Those are the cells where Black is closest to establishing a winning connection, and therefore White will want to play in those cells to block Black's connection. Combining this, White will prefer to play in cells that have a low *total potential*, where the total potential of a cell is the sum of its W-potential and its B-potential. By symmetry, Black will prefer to play in the same cells. This is analogous to the heuristic for the game of Go that says: "Your opponent's most important play is your most important play."

Consider again the position of Figure 5.3. The total potentials of the unoccupied cells are shown in Figure 5.5. Analogous to the black and white board potentials, the total board potential is defined as the lowest total potential on the board. In this case, the total board potential is 9. It is realized only by cell c2. In this case, the cell with the lowest total potential also has the lowest black and white potentials. In general, however, the two sets of cells with lowest black and white potentials need not be equal. In some cases they can even be disjoint.

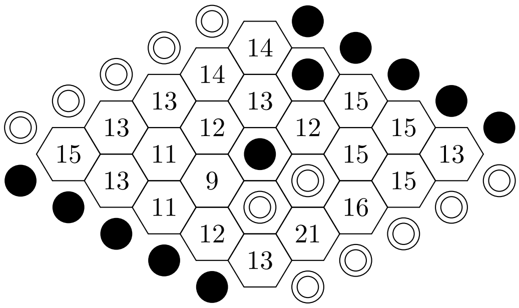
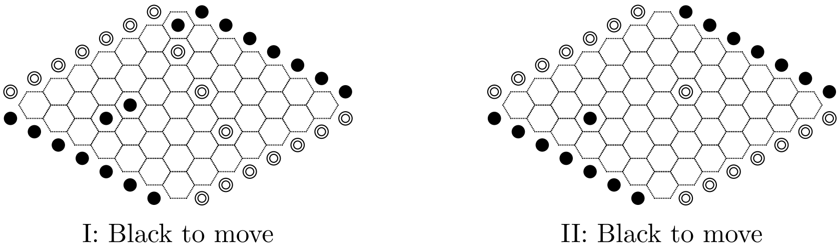


Figure 5.5: Total potentials


I: Black to move

II: Black to move

Figure 5.6: A high tension position and a low tension position

For each cell, the *badness* is defined as the difference between its total potential and the total board potential. For example, in Figure 5.5 it can be seen that the move e2 is the worst-looking move on the board, with a badness of $21 - 9 = 12$. This concept of badness plays a crucial role in Queenbee's search algorithm, as will be described in Chapter 6.

5.5 Move tension

An alternative way to judge the apparent strength of a move is *move tension.* The tension of a move is defined as the swing in board potential difference caused by the move. Selecting the move with the highest tension is therefore equivalent to performing a one ply search. The importance of move tension is that it allows a comparison of the apparent strength of moves in *different* board positions. This information may be very useful in deciding whether or not to extend the search on a given move. Conceptually, extra search effort is likely to be needed after high tension moves, since the evaluation is apparently unsettled.

Figure 5.6, with Black to move, illustrates the difference between quiet and tense positions. Position 5.6-I is very dynamic; there is a ladder fight going on at the upper white edge. The move with the highest tension is a6, whose tension is 4. By contrast, position 5.6-II is very quiet. The highest tension move here is c5, whose tension of 2 is only about half that of the ladder blocking move a6 in 5.6-I.

Move tension is a general, game-independent way to assess the apparent local strength of a move. It is not efficient to use this measure to generate the move ordering, since this would amount to a full-width one ply search with one call to the evaluation function for each available move. It would be too expensive to do this merely to sort the moves. If a game playing program calculates the static evaluation at every node, as Queenbee does, the move tension information becomes available "for free" *after* the move is made. However, at that point the tension information can be very valuable, since it can still be used to generate a local search extension or reduction.

Chapter 6

Queenbee's search algorithm

Queenbee uses an iterative deepening α-β search enhanced with MWS/PVS and transposition tables.[1] The move ordering is based on the "move badness" metric described in Section 5.4. Queenbee's search incorporates the fractional ply searching ideas of the "SEX search algorithm" [LBT89]. The large branching factor of Hex makes regular full-width searching methods inadequate, even when enhanced with conventional search extensions and reductions. On the other hand, a highly selective search is too unreliable due to its inability to cope with moves such as forcing moves and outpost establishing moves. The Sex search algorithm is essentially a generalization of search extensions and reductions, whose behaviour can range smoothly over the spectrum between full-width and fully selective. Moreover, it is amenable to automatic learning.

6.1 Beam search and tapered search

Beam search is a standard search technique applied to search domains where the branching factor of the search tree is too large to reach a search depth required for acceptable performance within realistic time limits. The search algorithm chooses the best k children of each node, as determined by a move ordering function, and expands only those. The parameter k is referred to as the *beam width.* With beam search, the quality of the move ordering is of critical importance. Whenever a particular move is not among the best k according to the move ordering, a beam searcher will not find this move regardless of how much search time is allotted.

A more robust search method is *tapered search.* This algorithm behaves like a beam search, but the beam width varies with the search depth. The natural choice would be to use a wide beam near the root node, and narrower beams at larger depths. At the start of each new iteration, the beams are widened. The *tapering function*, which controls the beam width, takes as input just two parameters: the total search depth D at the current iteration, and the distance d to the root of the current node. If the function is strictly increasing with D for every value of d, the search will eventually find any move when allowed enough time to search.

---

[1] See Chapter 2.

Tapering functions are typically chosen to decrease with d. A simple case would be the function $D - d$. Note that a full-width search in Hex would in fact be a tapered search; since there is one fewer empty cell on the board after every move, the number of children to be expanded decreases as the distance to the root node increases.

Rather than tapering the beam width, it is also possible to taper some function associated with the moves. Suppose each move is assigned a local score, reflecting the apparent strength of the move. At each node, the search algorithm only expands the children whose local score are better than a particular threshold. This threshold is determined by the tapering function. Queenbee can use the *move categories*, described below in Section 6.3, for its tapering.

6.2 Sex search

As described in Section 2.5, selective search is inescapable for developing a high performance Hex program. State-of-the-art search algorithms do behave selectively to a limited degree, using search extensions and reductions. The Sex search algorithm, as described by Levy, Broughton, and Taylor [LBT89], generalizes this idea. The name "Sex Search" stands for "search extensions".

Sex search proposes to assign a weight, or *cost*, to every move in the search tree. Rather than exploring lines until a certain fixed depth is reached, the Sex algorithm explores lines until their moves add up to a fixed cost. This cost limit may be called the *budget*. The idea is that "interesting" moves have low cost, while "uninteresting" moves have high cost. This way, branches with many uninteresting moves are not explored very deeply, which corresponds to search reductions. At the same time, branches that contain many interesting moves will be explored more deeply, corresponding to search extensions.

If all moves are assigned a cost of 1, then a Sex search with a budget of n is equivalent to a full-width fixed-depth search to n ply. If the moves have varying cost, but the average cost is 1, then the Sex search is comparable to an n ply search with extensions and reductions. A move cost of k effectively extends the search by $1 - k$ ply if $k < 1$, and reduces the search by $k - 1$ ply if $k > 1$.

If the range of costs of the available moves is large, then the Sex search algorithm behaves much like a selective search. Consider, for example, a move m with cost 4. This cost ensures that the subtree below m will be explored to an average of 3 ply less than subtrees below m's siblings, assuming they both contain moves with an average cost of 1. Due to the exponential nature of the search tree, the search effort required to explore m becomes insignificant in comparison with the effort required to explore m's siblings. Thus the behaviour is much like that of a selective search that would discard move m altogether. A selective search suffers from the unavoidable risk of discarding moves that turn out to be critical. SEX search does not run this risk, as it does not actually discard any moves.

SEX search is used in some high performance game playing programs, most notably in the chess program Deep Blue [CHH99]. However, in most cases the fractional move costs are only assigned to certain special cases of moves, while the majority of the moves receives weight 1. Queenbee's search is fully fractional, in that each move category has a fractional weight.

Beam search and tapered search are *forward pruning* methods, which means that they discard moves before they are searched. Sex search is a more flexible way to achieve a selective search. It is not a forward pruning method, since the algorithm does expand either all moves or no moves at every node. The search tree can be very non-uniform, where some lines go much deeper than other lines. But this non-uniformity is only relative to the metric of the search tree. When one measures distance by arc length in the search tree, where the arc lengths correspond to the move weights, the tree is of uniform depth up to rounding-off effects.

6.3 Move categories

The crux of the SEX search algorithm is finding a good cost function for moves. Moves are partitioned into equivalence classes, or *move categories*. Each move category has a weight associated with it. The cost of a particular move is obtained by retrieving the weight of its move category. In Queenbee, the partitioning can be based on move badness, as described in Section 5.4, or on move tension, as described in Section 5.5.

When move badness is used, the category $c(m)$ of a move m with badness $b(m)$ is

$$c(m) = \begin{cases} -b^* & \text{if } b(m) = 0 \\ b(m) & \text{otherwise} \end{cases} \tag{6.1}$$

where b^* is the move badness of the second best move overall. The parameter b^* is included to assign different costs to a move of badness zero depending on whether it is the unique best looking move, and on how much better it is than the next best looking move. If $c(m) = 0$, then m is the best move but not the unique best move. If $c(m) < 0$, then the static evaluation of m is $-c(m)$ better than that of the next best looking move.

When move tension is used, Queenbee's standard evaluation function according to Formula 5.2 is used, but the attack mobility is disregarded in order to keep the number of equivalence classes limited. In practice, the move tension has been observed to be as high as 12 in some positions.

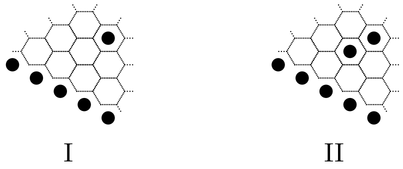


Figure 6.1: Adding a ghost piece to detect a virtual connection to the edge

6.4 Pattern database

Sections 4.2 and 4.3 described the concepts of edge patterns and ladders. Human players use these patterns extensively. It seems likely that a Hex-playing program needs to be able to recognize these patterns as well. Indeed, even the simple pattern of Figure 6.1 would require an additional six plies of search to discover the virtual connection. Bigger edge patterns and ladders can require many more plies. Lack of knowledge about these patterns creates a large risk of incurring a horizon effect. The search algorithm can use the pattern moves to push other tactical events beyond the horizon.

Suppose knowledge about the third row edge pattern of Figure 6.1-I is to be added to the program. Two different ways of doing this have been tried in Queenbee. The first approach was to add "ghost pieces"; imaginary extra pieces that secure the edge connection. The second method was to consider the pieces with a virtual connection to actually be part of the edge of the board. The latter method is called "edge extension". Both methods are only used at the leaf node level; they can be thought of as a *pre-evaluator*, which is a function that transforms a position before it is to be evaluated.

Ghost pieces

In Figure 6.1-I, the black piece has a virtual edge connection thanks to the edge pattern. To take this virtual connection into account in the evaluation, an extra *ghost piece* can be added. This has been done in Figure 6.1-II. As Queenbee's evaluation function recognizes two-bridges correctly, it will conclude that the black piece on the third row is connected to the edge, and that White connections are blocked by this pattern.

This approach suffers from several major drawbacks. It is usually not clear where to add ghost pieces. In the pattern in Figure 6.1-I, a ghost piece can be added in any of four locations to establish the two-bridge connection. Moreover, the addition of ghost pieces severely overestimates the strength of Black's position, as Black will have more pieces on the board. Any compensation that White would receive for making Black actually establish the connection by playing forcing moves is under-represented.

The most important drawback is that the ghost pieces may interfere with other local battles. A ghost piece can inadvertently take on a double role, not just establishing the edge connection but

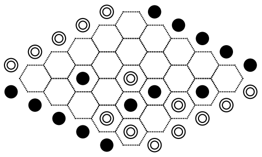


Figure 6.2: The dangers of ghost pieces

also blocking a nearby opponent's connection. In tests with ghost pieces, Queenbee consistently misplayed positions by explicitly aiming for positions in which the ghost pieces played such double roles.

Edge extension

An alternative approach to taking edge patterns into account is dubbed "edge extension". This approach leaves the position undisturbed, but explicitly makes the cell containing the Black piece part of the black border. This effectively puts the cell at distance zero from the border, reducing its neighbours' B-distances to the border by as much as two, in the case of the third row edge pattern. This avoids the disadvantages of ghost pieces.

Edge extension has a tendency to under-represent Black's advantage, as White connections through this pattern are no longer blocked. This is offset by a tendency to overestimate Black's advantage when more than one pattern is found. In such a case it could happen that the various edge patterns interfere with each other.

Pathological cases

There does not appear to be a flawless way to incorporate edge pattern knowledge into an evaluation function. Pathological cases where a position is incorrectly assessed can be constructed for both ghost pieces and edge extension. In practice, the danger turns out to be much less with edge extension.

Figure 6.2 illustrates the dangers of ghost pieces. The position is part of the analysis of the Hex puzzle originally devised by Piet Hein, and given in Martin Gardner's article [Gar59]. The position is claimed to be a losing line for White, for after White's move at c2 Black wins by playing at a4 and eventually laddering over to e4. However, the position is in reality a win for White. White's unique winning move is c4.

The confusion arises because the black piece at b2 has a virtual connection to the lower black edge, and the black piece at d3 has a virtual connection to the upper black edge. Therefore White's move at c2 is seemingly forced. But White in turn has a forcing move at c4 that turns the tables. If ghost pieces were used to evaluate this position, the black piece on d3 will be connected to the upper black edge by adding a ghost piece at c4, c5, or d4. In each of those cases, the resulting position is a loss for White. The only correct solution in this case is to add a black ghost piece at d4 *and* a white

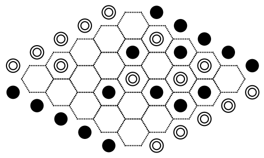


Figure 6.3: The dangers of edge extension

ghost piece at c4, but this would actually require additional search.

A simple case where edge extension evaluates a position incorrectly is shown in Figure 6.3. The two black pieces at b4 and d3 in Figure 6.3 both have a virtual connection to the upper black edge. Since the black piece at c2 has a virtual connection to the lower black edge and can always be connected to either b4 or d3, it appears that Black has a secure virtual connection all the way across. However, White next plays at c4 and wins. The flaw in Black's line of reasoning was that b4 and d3 each have a virtual connection locally, but the connections overlap. White can attack both connections at once; Black can save either one, but White can then cut the other connection.

This leads to the important conclusion that when a collection of pieces each have virtual connections to the edge, it merely implies that the player in question can choose to connect any one of them to the edge – but not necessarily *all* of them. This could lead to a serious error in the assessment of a position, as this example shows. In practice, however, the edge templates used are rather too sparse to cause this kind of destructive interference.

6.5 Unsuccessful enhancements

In addition to the methods described in this Chapter, several other ideas were tried. Some of the methods had no significant beneficial effect on the search efficiency, or proved to be too risky and unstable in tests. Such unsuccessful attempts include null move searches, forward pruning, and balanced evaluation.

Null moves

An apparently very promising search enhancement for a Hex playing program would be *null move search* [GC88]. A null move is effectively a pass, where the turn is handed back to the opponent without making a move at all. The remaining position is searched with a reduced search depth, usually two ply less. If this null move search returns a value that exceeds the β bound, a cutoff is generated. This cutoff is relatively safe, since it is to be expected that there are moves available that are better than a pass move; if even a pass move exceeds the β bound, there are likely to be valid moves that do so as well. Informally, if the opponent cannot improve their position even when making two moves in a row, then the first of these two was a mistake which can be pruned away.

Null moves are "cheap"; since the remaining search is shallower they take relatively little time. Their power is that they can thus establish a cutoff very quickly. The danger that threatens null move algorithms is that they implicitly assume that there are always moves that are better than a pass move. This is however not the case in zugzwang positions.[2] Fortunately, zugzwang does not occur in Hex. There are always moves that are better than a pass move.

Experiments with null move searches in Queenbee were unsuccessful. The extra search effort did not slow down the program much, but hardly any cutoffs were generated. The reason may be a pass move in Hex is so bad that indeed *any* move is guaranteed to be better. Since no move can weaken a player's position, a pass move by the opponent always makes it possible to reach a better position than that before the previous move, even when the previous move was not the best possible.

Forward pruning

Due to the large odd/even effect in Hex and the fact that a move can never weaken a player's position, the evaluation of a position tends to oscillate during a game. The evaluation is usually in favour of the last player to move, except near the end of the game where one player is significantly behind. This suggest a forward pruning mechanism where a position is ignored if the evaluation is already in favour of the player to move even before a move is played, since this would indicate that the player could actually afford to skip the move which is a significant advantage in Hex.

The pruning method can be generalized by introducing a pruning threshold, where a position is ignored if its evaluation prior to making a move already exceeds the threshold. During experiments, searches would often actually terminate by completion, meaning that it had been found that one player was guaranteed to achieve this threshold evaluation. Moreover, this forward pruning mechanism led to significant reductions in node counts. Eventually this method was abandoned, since there turned out to be too many "pathological" positions where critical lines were cut off. This tended mostly to happen in ladder positions.

Singular extensions

Singular extensions were introduced by the Deep Thought / Deep Blue team [ACH90]. The method proposes to extend the search in positions where one move is found to be clearly better than all the other moves. In Queenbee, this would be a move whose potential is lower than that of any other move; in other words, a move with negative move badness. Singular extensions are thus a special case of SEX search. It can be conjectured that this method may be able to deal with ladders by extending its way through them, but in experiments it turned out to be too unstable. The reason for this may be that it is dangerous to apply Queenbee's evaluation function at different ply depths in the search; see Section 8.1 for a discussion on this subject.

Balanced evaluation

Queenbee's evaluation function experiences a large odd/even effect. Its search can be set to "aggressive" or "defensive", corresponding to odd ply searches and even ply searches, respectively. The odd/even effect is particularly large in ladder positions. The danger of aggressive search is that these ladder positions are too often considered to be favourable, since the player to move always gets the last move in each line of the search. Similarly, a defensive search is too pessimistic about ladders. In playing tests it appears that aggressive search is clearly the better choice of the two.

---

[2] See Section 2.3.

The problems caused by the odd/even effect could perhaps be fixed by taking the average value of a node's evaluation and the evaluation of its parent node, and passing these averaged values back up the search tree. This might for example favour a line where the evaluation oscillates between +3 and +1 over a line where, due to a ladder, the evaluation fluctuates between +5 and -5. However, in tests the balanced evaluation search behaved nearly identical to the aggressive search.

Chapter 7

Results

The main goal when developing a game playing program such as Queenbee is simply to maximize the program's playing strength. The most important experiments are therefore those that test Queenbee's strength. This was mainly done by self-play matches, as well as test matches against its main computer rival Hexy. In the near future, feedback will also be obtained from online play. To this end, an applet was developed through which human players can play against Queenbee on the Internet.

Other experiments were performed to assess the effectiveness of the various search and evaluation ideas incorporated in Queenbee. These include tapered search and sex search, the move ordering by move badness, the move weighting based on move tension and move badness. All experiments were performed on a 7 × 7 board; the same set of experiments should be repeated on size 11 × 11 and perhaps other sizes in order to assess how well the findings scale up to larger boards.

7.1 Selective search

The strength of human players in a game is commonly measured by their *rating*. The ELO-ratings for chess players are a well-known example. For Hex, similar ratings are calculated by the online games server Playsite.[1] Typical ratings for human players are listed in Table 7.1.

Queenbee first started rivalling strong humans around 1997. Two years later, the very strong program Hexy [Ans00], written by Vadim Anshelevich, appeared. Hexy is available for download online,[2] and thus proved to be an excellent opponent to benchmark Queenbee's performance. Its

---

[1] See `http://www.playsite.com/games/board/hex`.

[2] See `http://home.earthlink.net/~vanshel`.

range	playing level
1200–1400	novice
1600–1800	advanced
2000–2200	expert
2200–2300	champion

Table 7.1: Ratings for human players on Playsite

Queenbee version	wins	draws	losses	percentage	
full width	3	14	8	40%	27%
beam search, width 10	3	19	3	50%	50%
beam search, width 5	10	12	3	64%	77%
move class tapering	5	19	1	58%	83%

Table 7.2: Queenbee at 100,000 nodes versus Hexy at level 1, on a 7 × 7 board

rating on Playsite is reported to be in the range of 2100 to 2200, but no detailed information is available. To date, there is no other known Hex playing program that can approach Hexy's strength.

Queenbee played games against Hexy on a 7×7 board, using each possible opening move once. Due to the symmetry of the board, there are 25 distinct opening moves. Each program played two games for each opening move, taking White's side in one game and Black's side in the other. The opening was scored as a draw if both programs won one of the games. The swap option was not used. Since many openings may be lopsided, and the programs are forced to play each opening once from the side they consider to be the weakest, drawn encounters may be disregarded.

Results are shown in Table 7.2. The table shows two winning percentages for each match; the first one considers all games played, whereas the second one disregards drawn encounters. This match was played with Queenbee using 100,000 nodes per move and Hexy set to "beginner level", taking on average about 10 seconds and about 2 seconds to move. To assess the relative strength of the programs, experiments would need to be to establish how much search Queenbee would need to break even with Hexy at advanced levels. However, the purpose of the experiments in Table 7.2 is to compare the strength of beam search and tapered search versus regular full-width search, not to compare the strength of Queenbee and Hexy.

The beam search only considered the best five or ten moves, according to Queenbee's potential-based move ordering, in each position. The move class tapering considered all the moves with a badness of at most $f(d,h)$, where f is a function of the depth of the node and the height of the subtree remaining below it. In this case, a hyperbolic function is used: $f(d,h) = d/h$. This function was chosen over the more straightforward $f(d,h) \simeq h/(d+h)$ in order to have less selectivity near the root and more selectivity near the fringe of the tree. The results clearly show that beam search outperforms regular full width search, and that the ad-hoc tapering function in turn outperforms

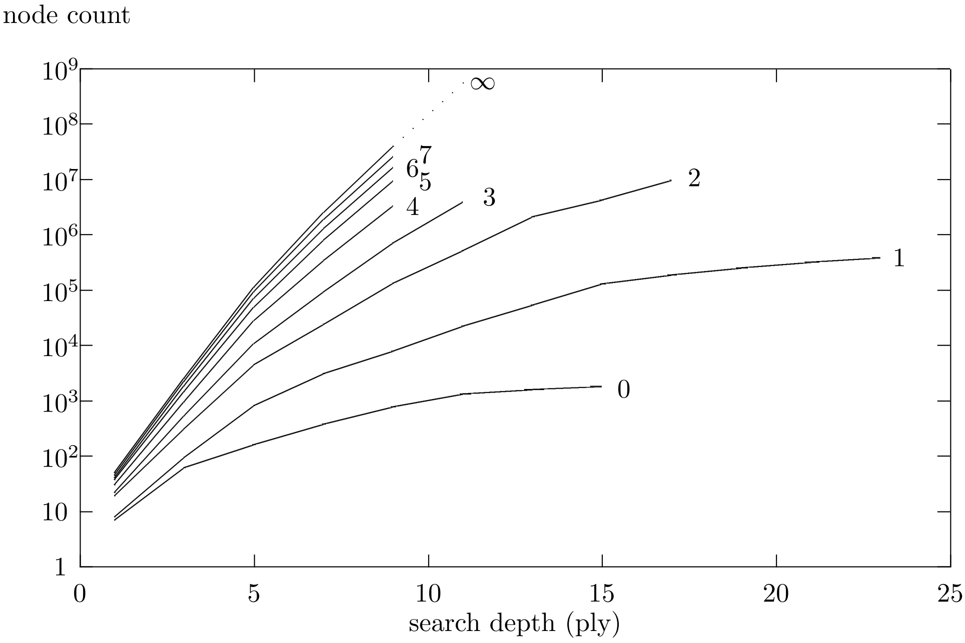


Figure 7.1: Node counts for searches with indicated beam width, 7 × 7 board

beam search.

7.2 Branching factor

Since the move selection in tapered searches or Sex searches uses the move badness metric, it is relevant to know how much the branching factor of the search tree is affected by pruning away moves based on this criterion. Figures 7.1 and 7.2 show the node counts as measured on 7 × 7 and 11 × 11 boards, when only moves of a given maximum badness are considered. The lines in the graphs are labelled with the maximum badness of the moves considered in the search. The labels '∞' correspond to a search that considers move of arbitrary badness; in other words, a full width search.

The results show a clear trend when the logarithms of the node counts are taken relative to those of a full width search. Figures 7.3 and 7.4 show the resulting lines. It can for example be established that a tolerance beam width of zero has a branching factor approximately equal to the fifth root of the branching factor of a full width search on an 11 × 11 board. This would enable a search to go five times as deep. A tolerance beam width of zero corresponds to a search that only considers moves that have optimum potential; note that there may be more than one such move in a given

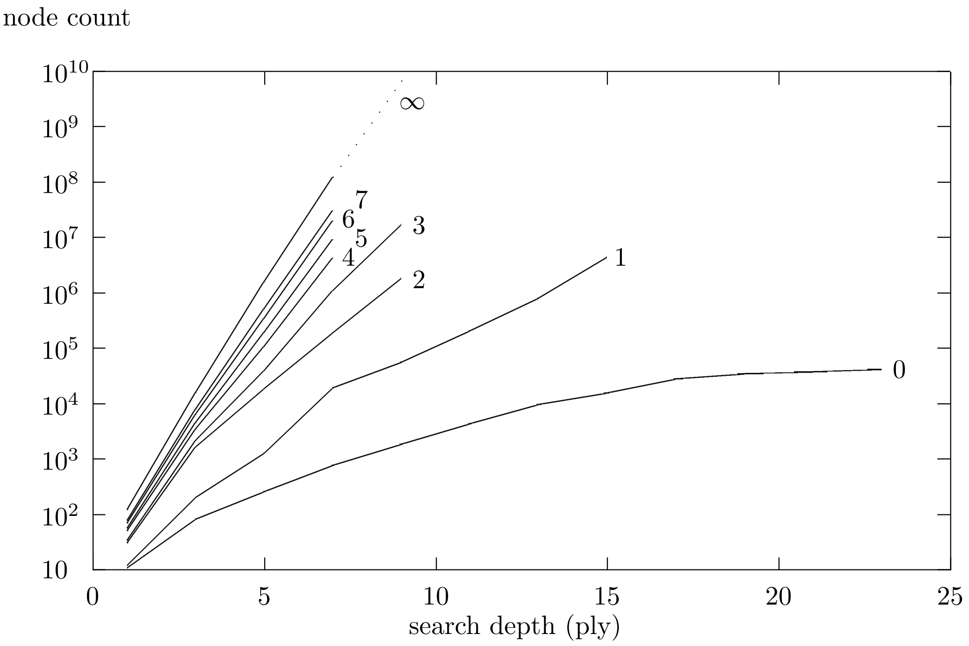


Figure 7.2: Node counts for searches with indicated beam width, 11 × 11 board

move category	≤ -4	-3	-2	-1	0	1	2	3	4	5	6	7	≥ 8
simple extensions	0.5	0.5	0.5	0.5	0.5	1	1	1	1	1	1	1	1
simple reductions	1	1	1	1	1	2	3	4	5	6	7	8	9

Table 7.3: Move category weights used in the simple search extensions and reductions settings

position.

7.3 Sex search

The Sex search algorithm forms a natural generalization of search extensions and reductions. A move that is assigned a weight $w > 1$ is effectively a search reduction by $w - 1$ ply, while a weight $w < 1$ corresponds to an extension by $1 - w$ ply. Experiments were performed where Queenbee used a hand-crafted set of weights that emulated search extensions or search reductions. The actual weights used are listed in Table 7.3.

Special care must be taken in implementing the Sex search. Queenbee's evaluation function suffers

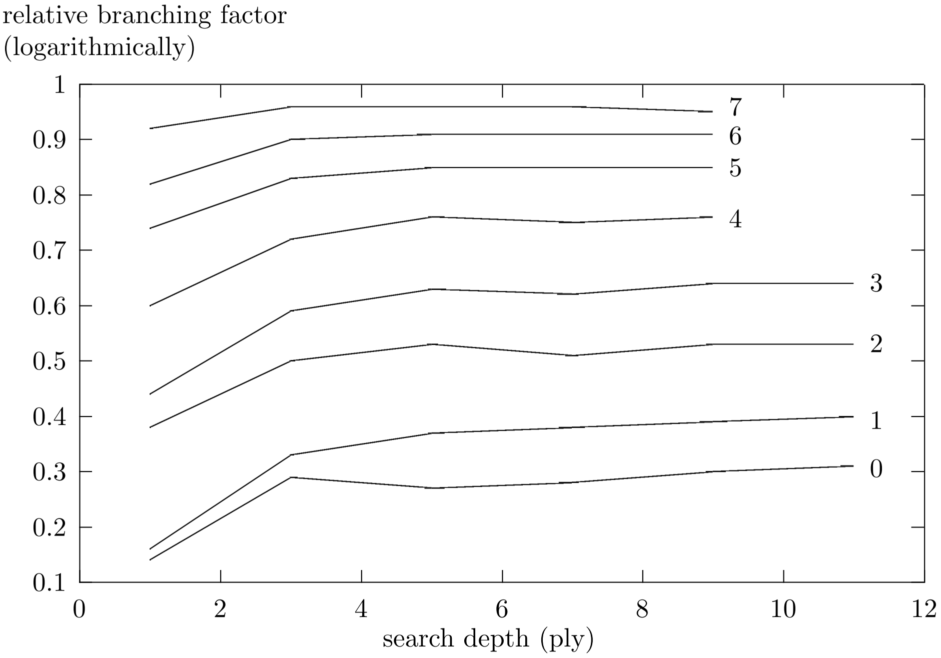


Figure 7.3: Branching factor relative to full width search for searches with indicated beam width, 7 × 7 board

Sex search variant	wins	draws	losses	percentage	
simple extensions, two budgets	4	15	6	46%	40%
simple extensions	3	22	0	56%	100%
simple reductions	7	16	2	60%	78%

Table 7.4: Queenbee at 100,000 nodes, sex search versus full width search, on a 7 × 7 board

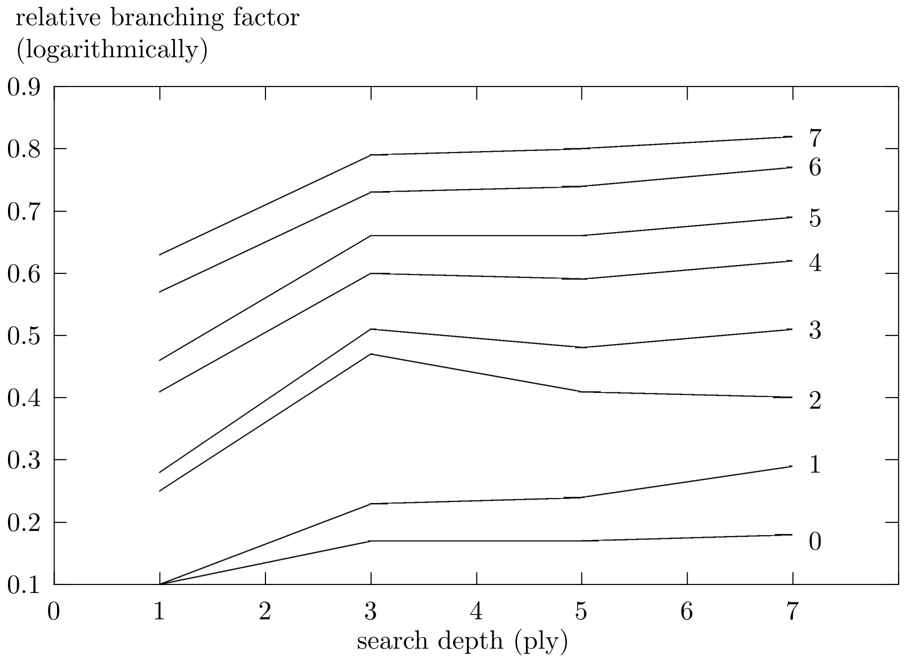


Figure 7.4: Branching factor relative to full width search for searches with indicated beam width, 11 × 11 board

from a considerable odd/even effect.[3] It must therefore be guaranteed that the evaluation function is always called in positions with the same player p to move. To achieve this and at the same time keep all lines at roughly the same cost, Queenbee uses the following solution. In positions where p is to move, only the moves whose cost will not reduce the budget to below zero are explored. In positions where p's opponent is to move, all moves are explored.

Another danger that threatens the SEX search, especially when pattern databases[4] are used, is what is called the "SEX horizon effect". When exploring a line that contains a winning combination, the algorithm will often create a horizon for itself by including very bad moves for the defending side. This causes the variation to be searched less deeply, pushing the win beyond the budget limit.

Two ways to circumvent the SEX horizon effect are implemented in Queenbee. The first method is to make sure than whenever a move is or becomes a principal variation move, it is considered to be of the lowest available move category. This way, no high move weights can occur on the principal variation. The second method, as described in [LBT89], is to maintain two separate budgets, one for each side.[5] This ensures that a strong move sequence will be found regardless of the opponent making uninteresting moves.

Table 7.4 lists the results of self-play matches. Note that both extensions and reductions win roughly the same percentage of games, but extensions appear to be more robust since they did not lose any matches. It can also be seen that the two-budget approach did not work well. Possible reasons for this are discussed in Chapter 8.

7.4 Evaluation function

Since Queenbee has completely solved all the positions after one move on a 6 × 6 board, as well as many opening positions two or more moves into the game, it is possible to compare the evaluation function's assessment of these positions with perfect knowledge. In a winning position we distinguish between *good moves* and *bad moves*. A bad move loses against perfect play, while a good move preserves the win. There are two types of good moves: *optimal* moves and *suboptimal* moves. A move is optimal if it maintains the shortest possible win. In a losing position there are no good or bad moves, as every move will lead to a loss against a perfect opponent. Yet there still is a distinction between optimal and suboptimal moves. Optimal moves are those that delay the loss as long as possible, while suboptimal moves do not.

Figure 7.5 displays the frequency in percentages of optimal and good moves for various move categories.[6] The data was obtained from 27 winning positions and 34 losing positions that Queenbee has analyzed. The figure indicates that the lower move categories contain a significantly higher

---

[3] See Section 2.3.

[4] See Section 6.4.

[5] This is also used in Deep Blue.

[6] See Section 6.3.

position type	move type	move category							
		-5	-2	-1	0	1	2	3	5
winning	optimal	-	1	1	9	6	2	7	1
	good	-	1	1	12	7	2	3	1
losing	optimal	3	3	5	21	1	-	-	1

Table 7.5: Category containing the lowest potential move

percentage of good moves than average. Note that negative move categories, intuitively corresponding to apparently "forced" moves, appear to be better in losing positions than they are in winning positions. This may be because the winning side typically has several options to choose from, while the losing side is more often forced to reply to a threat.

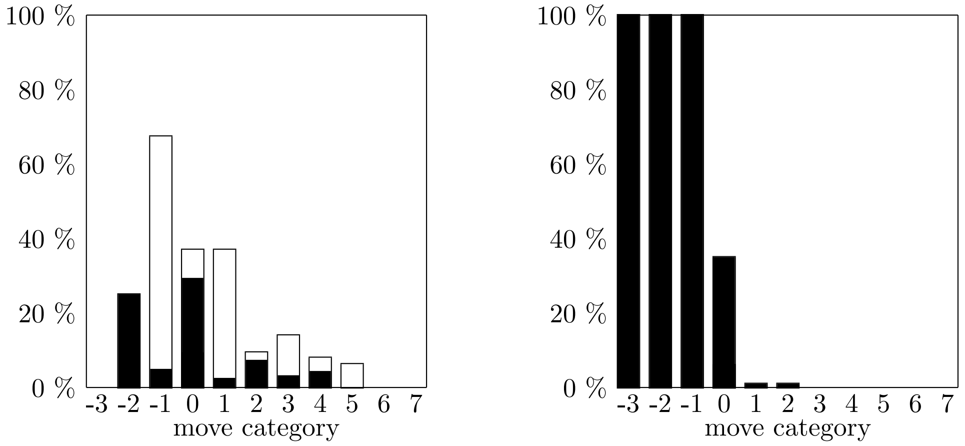
The effectiveness of the Sex Search relies on the move categories' capacity to distinguish good moves from bad moves. Any category that has a significantly different distribution of good and bad moves compared to the overall distribution is therefore valuable. Categories can be assigned low or high weights according to whether good moves are relatively common or uncommon, respectively. The bottom graph in Figure 7.5 indicates that almost all categories do show a large deviation from the average frequency. Moreover, the frequency of optimal moves decreases almost monotonically as the move category increases, indicating that the cell potential is indeed a good estimator of move strength.

In order for the search to produce reliable results, it is not necessary that all good moves in a position be found. What is important is that at least one good move is found. Table 7.5 lists the lowest move categories in which optimal and suboptimal moves were encountered in the same set of positions. It is clear that in most cases there is an optimal or suboptimal move to be found in categories at most 0 or 1. Positions in which good moves only occur in higher move categories are rare.

From these tables it becomes apparent that the evaluation function enables a good partitioning of moves into classes of different move quality. It is also clear that the cell potentials provide a good estimate of the quality of the moves, since good moves are more common in low move categories.

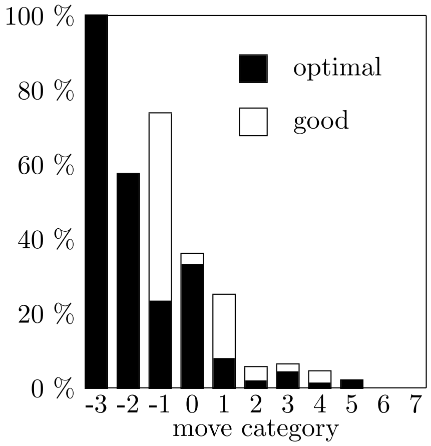
7.5 Tension and badness

The same experiments as in Section 7.3 were performed using move tension, rather than move badness, to partition the moves into categories. The rationale behind this is that the reason to generate a search extension is not because the preceding move is *good*, but because it is *volatile*. The extension hopefully causes the position after the volatile move to be resolved. Move tension is a natural measure of volatility, and hence it appears to be a suitable choice as an extension control mechanism.



winning positions

losing positions



all positions

Figure 7.5: Move types per move category

move tension	0	1	2	3	4	5	6	7	≥ 8
simple extensions	1	1	0.7	0.5	0.4	0.3	0.2	0.1	0.1
simple reductions	3	2.5	2	1.5	1	1	1	1	1

Table 7.6: Move category weights based on move tension used in the simple search extensions and reductions settings

Sex search variant	wins	draws	losses	percentage	
simple extensions	1	20	4	44%	20%
simple reductions	2	20	3	48%	40%

Table 7.7: Queenbee at 100,000 nodes, sex search using move tension versus full width search, on a 7×7 board

Table 7.6 lists the weights that were chosen to emulate reductions and extensions. The match results are displayed in Table 7.7. It is clear that move tension does not perform as well as move badness; in fact, the move tension with the given weights actually decreased Queenbee's playing strength relative to a normal full-width search.

Chapter 8

Conclusions and future work

The general conclusion to be drawn at this stage in Queenbee's development is that the program is strong, but that two key areas of weakness can be identified. The first is the program's play in the opening phase, when the game is still very quiet and strategic in nature. The second weakness is the way the search is focussed. This chapter discusses the results and presents ideas for future work towards strengthening these weak links.

8.1 Discussion

The importance of selectivity and search depth are clearly illustrated by the beam search results in Table 7.2. The best results are obtained when only five moves are explored in each positions in the search tree. Table 7.5 partially explains this; the best moves for the losing side are almost always in the low move categories. Yet the best moves for the winning side are frequently not at the top of the move list. A beam search algorithm would in that case never find a winning move, no matter how much time is allocated for the search.

The results show that increased search depth with narrow beam search outweighs the risk of frequently not finding a good move for the winning side. The apparent importance of search depth indicates that it is imperative that unsettled positions be resolved through additional search. Resolving unsettled positions may be more important in Hex than in many other games since draws are impossible in Hex. In games like chess, a position whose static evaluation is roughly even may well be a drawn position. But since there are no draws in Hex, every position will eventually tip in favour of one of the players. Finding out which way the position is likely to tip may be more important in Hex than exploring other alternatives near the root level.

Tapered search combines the advantage of selectivity and depth with the capacity of in principle being able to find any move, no matter how bad its static move evaluation is. The search is very selective near the fringe of the tree, allowing it to reach greater depth than a full-width search, and less selective near the foot, allowing it to find non-obvious moves. However, this method still misses potentially critical lines in the search tree that contain important non-obvious moves near the fringe rather than near the root. Sex search generalizes the tapered search method, since it allows the non-obvious moves to appear at any distance from the root, while maintaining the same search effort.

When Sex search is used, high move weights generally do not matter. A high move weight leads to a search reduction, but this does not tend to make much difference in the search effort since the Minimal Window Search is so efficient that it proportionally does not spend much time on the bad looking moves anyway. For the same reason, Sex search with two budgets did not work very well. The two-budget approach correctly plays out a line after one player inserts a bad move, but this is what the alpha-beta algorithm would do anyway. The two-budget method works correctly, but does not lead to any significant search savings over full-width search.

The Sex horizon effect is particularly destructive when edge patterns are used in the evaluation function. The problem is likely mainly caused by the non-uniform depth of the search tree. Beam search and tapered search are also selective, yet all the lines that are explored do reach down to the same depth. In Sex search they do not. Perhaps it is very dangerous to apply Queenbee's evaluation function at different ply depth in the tree. The game of Hex contains a natural measure of "time"; late game positions are essentially incomparable to early game positions since they contain a different number of stones.

The following conclusions can now be drawn about game tree search in the game of Hex.

- The search must be non-uniform, concentrating the search effort on moves that appear good based on static evaluation or initial search.
- The search must in principle consider all moves eventually; no moves can be discarded without further ado.
- The search should be flexible enough to consider non-obvious moves at any depth, not just near the root.
- The search is recommended to go down to the same ply depth in every line, meaning that the static evaluation function must be applied at the same or at a comparable depth everywhere in the tree.

A search-based solution to this might be a tapered search in which the tapering function itself is dynamic. Each position P in the search tree could be assigned a branching factor $b(P)$, indicating the number of moves or the number of move categories that are to be explored. A move m leading to child node P_m would then be assigned a new branching factor $b(P_m)$ based on the move class of

m. The worse m appears to be, the more $b(P_m)$ is decreased relative to $b(P)$. This way, a bad move does not lead to a shallow tree, as it does in regular Sex searches, but it leads to a sparse tree.

8.2 Comparison with Hexy

Anshelevich's approach used in his program Hexy seems excellently suited to Hex. The program's search component is mainly geared towards discovering virtual connections, although it also does some additional game tree search. The virtual connections are partitioned into a hierarchy of "generations", where a connection of the n-th generation consists of a conjunction or disjunction of connections of generations at most $n-1$. The evaluation function is based on an electrical network model, where apparently each cell is a node in the network which is connected to all adjacent cells on the board. When Hexy can find a virtual connection between the two edges of the board for one of the players, the position is solved. When no winning virtual connection can be found, the virtual connections are used in the evaluation function. There are two reasons why a winning virtual connection might not be found. The connection may be too deep and thus require too much search, or it might indeed be impossible to find it since the connection rules that Hexy uses have been proved to be incomplete [Ans00].

Since a winning virtual connection of the n-th generation is equivalent to a win in n ply, it could also be found by an n ply game tree search. Part of the power of Hexy's search derives from its non-uniformity; some connections are pursued to a much higher generation than others. Anshelevich uses some metric to decide whether or not to expand a new connection, apparently based on the estimated amount of search effort involved. Connections are no longer expanded when they cross a certain "effort threshold". This is directly equivalent to a SEX search, where the estimated search effort per connection corresponds to the move weights, and the effort threshold corresponds to the search budget. Hexy's capabilities to detect very deep connections, such as ladders, hint that its playing strength would be decreased if it were to expand all virtual connections to a fixed depth. Thus the move weighting is of singular importance, but it is unclear how this is done in Hexy.

An important difference between the discovery of a deep winning virtual connection and a deep winning line in a game tree is that the virtual connection immediately proves the win, while the game tree line only does so once all alternative lines for the losing side have been disproved. As such, the virtual connections behave more like *threat patterns*, to be described below in Section 8.5. These patterns also prove wins and losses without needing to disprove all other lines for the opponent.

When a winning connection cannot be found, a heuristic value is returned based on the resistance of the electrical network corresponding to the position. Opponent's pieces are represented by insulators, which corresponds to raising the resistance of the wires to infinity. Friendly pieces correspond to zero-resistance conductors. Hexy's playing strength in positions where a win is not yet proved is considerable, despite its relatively shallow game tree search. This indicates that the inclusion in the evaluation function of knowledge gathered during the virtual connection search is of vital importance. Here, too, it is unclear how it is done in Hexy. Note that it cannot be a matter of introducing a zero-resistance wire between virtually connected cells; Figure 6.3 on page 50 gives an example where local virtual connections appear to prove a win in a position that is in reality a loss.

The details concerning the search control and the evaluation function are thus part of the key to Hexy's strength. It may or may not be possible that a game tree searcher using a suitable SEX move weighting can emulate Hexy's capabilities to prove a win or a loss, and that a Queenbee-type evaluation function can emulate Hexy's use of virtual connection knowledge at least where it comes to standard edge patterns. It will be possible to speculate on these matters once the information about the two said aspects of Hexy becomes available.

8.3 Future work: Learning search control

Queenbee's evaluation function does not contain any parameters whatsoever. Thus, in contrast to other game playing programs, Queenbee's evaluation function remained unchanged from the beginning, and is not a target for learning. A clear target for machine learning, however, are the move category weights described in Section 6.3. Recent work by Björnsson at the University of Alberta enables the automatic tuning of the fractional ply extensions [Bjö00]. Learning can proceed either online, by playing games, or offline, by compiling a suite of test positions along with their solutions.

Björnsson's methods were implemented in Queenbee. Tests with online learning have thus far been unsuccessful. The main problem appears to be that the method requires Queenbee to keep track of the full principal variation[1] with every search. This is difficult to achieve in combination with a transposition table, since transposition table cutoffs can cause principal variation information to get lost. Attempts to force full principal variations to be returned have not yet been successful.

The alternative method, offline learning, requires building a set of test positions. The method of choice to achieve this is to have the program play a large number of games against external opponents. With the implementation of the applet that allows users to play against Queenbee via the Internet, this has become feasible. Whenever the evaluation value on some move k is significantly lower than the value at move $k-1$, it is clear that the program found a particular continuation during the search at move k that was missed during the search at move $k-1$. The position at move $k-1$ is then added to the test suite, along with the principal variation found at move k. The learner's task is to minimize the search effort required to find the given variation in the given position.

---

[1] See Section 2.4.

8.4 Future work: Opening book learning

As noted in Section 2.6, the area of automatic opening book construction has recently become prominent in computer chess. Not only does an opening book need to be built, but it also needs to be updated constantly during game play, to avoid losing a game twice in the same way. One property that an opening book learning algorithm should ideally have is that, since it never plays an identical losing line twice, it would eventually construct a perfect opening book. In practice, the number of games required to reach this goal would doubtlessly be too large; however, the property of being able to do this *in principle* can still serve as a good guide line.

This section presents a high level description of such an algorithm, based on Buro's algorithm for Othello [Bur99]. An opening book is maintained in which each node contains two values: a *local value* and a *negamax value*. The local value is obtained by performing a regular game tree search in the position in question, *excluding* the available book moves. The local value thus represents the value of the best *alternative move*, which is the best move that does not appear in the book. The negamax value of a node in the book is found by comparing the value of the alternative move to the values of the book moves, and choosing the best one according to the negamax paradigm.

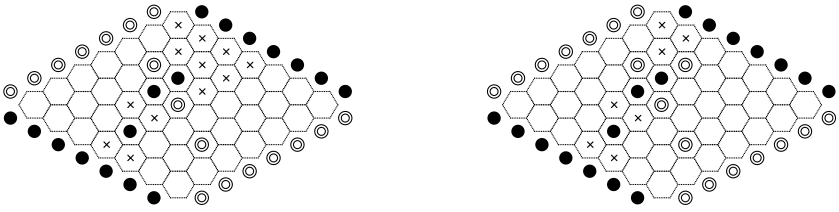
The opening book does not necessarily contain information about *all* child nodes for each of its internal nodes. Indeed, in a game with a high branching factor such as Hex, hardly any nodes will have all their children expanded. Learning can theoretically start with an empty opening book, and is performed by playing games, either against an external opponent or by self-play. Whenever the alternative move in an opening book position is found to be better than the available book moves, the new move is added to the book and the negamax values of all its parent nodes are updated accordingly. This is a *horizontal addition* to the opening book.

Horizontal additions occur when the alternative move becomes more advantageous than the available book moves. Since the book moves were originally chosen because they appeared to be better, the alternative move can only be chosen when the book move's values drop. When a position drops its negamax value, it necessarily implies that one of its children changed its value as well. Such a change in value is ultimately caused by a *vertical addition* to the book.

A vertical addition occurs when a leaf node in the opening book acquires a child node, and the child node backs up a different value than its parent's local value. Vertical additions occur according to some specified criterion; one could imagine adding a new node every time the program leaves the opening book, in which case one new node would be added during every game.

8.5 Future work: Pattern search

Partition search, described by Ginsberg in his work on Bridge [Gin96], refers to a search paradigm where moves are partitioned into classes. During the search, only one move from each of the partition


I: White to move

II: Black to move

Figure 8.1: Only the cells marked '×' are relevant

classes is searched. This reduces the branching factor of the search tree, resulting in a considerable speedup if the classes are large. For this method to work, the search results necessarily have to be identical or nearly identical for all moves in each given class.

One example of such a class occurs frequently in almost every game, when a human player considers that a particular threat must be dealt with. The player identifies a broad collection of moves that do not meet the threat, and that are all refuted identically. This is the inspiration behind *threat space search*, developed by Allis and used to solve Go-Moku [All94]. In a threat search, the defending side is allowed to play *all* forced replies to a given threat. If it can be proved that the attacking side still wins, then it follows that the position was a win under the normal game rules as well.

In Hex, human players often reduce a position to a small set of relevant board cells. Consider the position in Figure 8.1-I. It is clear that Black has won the game. The important observation is that only the cells marked '×' actually matter. The status of the other cells is irrelevant. Indeed, even if White were to occupy *all* of the other cells, the position would still be a loss for White. The collection of ×-cells is the *threat pattern* that proves the win for Black. Since it is White's move, the pattern is a losing threat pattern.

When White plays c6 in the position of Figure 8.1-I, the position in Figure 8.1-II arises. This position is still a win for Black. Again Black would still win even if White were to occupy all the cells that are not marked '×'. The threat pattern in Figure 8.1-II is a winning threat pattern. This threat pattern is very powerful, since it not only proves that White's c6 is a losing move, but it immediately disproves *all* moves that are not marked '×' at the same time. In this case, the move c6 disproves 35 other moves in one single effort.

This leads to the following definition of a threat pattern.

- **Definition:** A collection Ψ of empty cells in a Hex position P is a *threat pattern* if the game-theoretical value of P is unaltered when the losing side occupies all the empty cells not in Ψ.

Note that a threat pattern is not unique to a position, since adding an empty cell to a threat pattern always creates another valid threat pattern. A *minimal threat pattern* is one from which no cell can be omitted without rendering the pattern invalid. Even minimal threat patterns are not unique to a position; for example, when there are several moves that establish a winning connection, each of those moves represents a winning threat pattern consisting of one cell. There exists a threat pattern in every position, since the pattern that consist of all empty cells is always trivially valid.

Threat patterns can be calculated recursively. If a position is a win, the winning player merely has to find one single winning move m^+ along with a *losing* threat pattern Ψ^- of the resulting position. The winning threat pattern Ψ^+ then consists of the union of m^+ and Ψ^-:

$$\Psi^+ = m^+ \cup \Psi^-. \tag{8.1}$$

If the position is a loss, this can be established by finding a collection of k winning threat patterns $\{\Psi_1^+, \Psi_2^+, \ldots, \Psi_k^+\}$ for the opponent that have an empty intersection. The patterns do not need to be pairwise disjoint. The losing threat pattern Ψ^- then is the union of all these winning threat patterns:

$$\Psi^- = \bigcup_{i=1}^{k} \Psi_i^+. \tag{8.2}$$

The bottom of the recursion is reached in any position where one of the players has a chain that connects both sides; the threat pattern in that case is the empty pattern, since the winning player still wins even if the opponent occupies the entire rest of the board.

The potential power of this method is clear. In a winning position, ideally one move has to be searched in order to verify the win, just like in regular alpha-beta searches. But in a losing position, only a small subset of the moves have to be searched, whereas in alpha-beta *all* moves have to be disproved. In the position of Figure 8.1-I, a pattern search requires 2602 nodes to prove the 10 ply win, corresponding to a branching factor of 2.2. Queenbee requires less than 10 ply to see the win, due to the evaluation function. As it turns out, a full width search by Queenbee requires 6 ply and 63,874 nodes to prove the win, corresponding to a branching factor of 6.3.

In practice, the pattern search algorithm is quite sensitive to move ordering. The position in Figure 8.2 is a trivial win for Black. If the move b5 is searched first, the algorithm will correctly conclude that the pattern in Figure 8.2-I proves the win for Black. If on the other hand the move c4 is searched first, the win is still proved – by searching only two replies for White, rather than all 16 of them – but the resulting pattern contains three cells. It still is a valid and correct pattern, but it is bigger than it needs to be.

When backing up the patterns, they quickly grow too large if special care is not taken to keep them as small as possible. A pattern that covers the entire board is trivially valid, but it is also useless since the search then becomes a standard alpha-beta search. Some overhead would be required to keep the patterns small, which is problematic. The great potential speed of the partition search algorithm is also its own greatest impediment, since the slightest overhead in the algorithm will spoil most of the speed benefits.

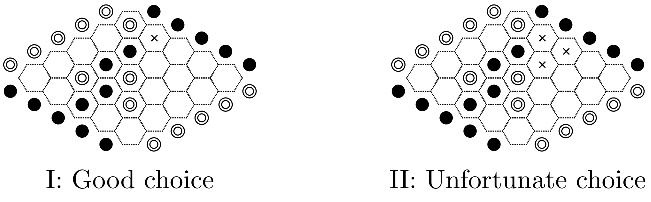

I: Good choice
II: Unfortunate choice

Figure 8.2: The importance of move choice

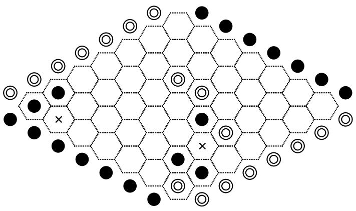


Figure 8.3: Moves marked '×' are reversible for White

It should be noted that the algorithm as formulated can only prove wins and losses. It cannot back up heuristic values. Some preliminary experiments have also been performed where heuristic values were artificially turned into wins and losses by considering a leaf position a win if and only if its evaluation exceeds a certain threshold. Repeated searches can identify the threshold position where the assessment changes from a win to a loss. The findings are still inconclusive. The power of Queenbee's evaluation function could be combined with pattern searches, since the evaluation function can detect a win before the connection is establishes; the remaining two-bridge connections then form the threat pattern. For example, the win in Figure 8.1-I would require just 91 nodes to prove using a pattern search combined with two-bridge detection, as compared to the 63,874 nodes that Queenbee's game tree search takes.

8.6 Future work: Combinatorial game theory

In combinatorial game theory, a move m is said to be *reversible* if there is a reply that leaves the opponent in a strictly better position than before m was played. In Hex, reversible moves are provably irrelevant, and can be deleted from the search.

An example of reversible moves is shown in Figure 8.3. The cells marked '×' are reversible moves for White. The move f3 for White is a forcing move, since it threatens the Black virtual connection between f2 and e4. But human players immediately discard this particular move, since it does not achieve anything. Black could reply by playing e3, after which there are no new W-neighbours.

White's move temporarily created new W-neighbourhood connections between e3 and the four cells g3, g4, f5, and e5, but Black's reply severed these connections. Since White's move did not connect anything that was not already connected, the move is evidently futile; any White move that does introduce new W-neighbourhood connections is necessarily better.

Another reversible move for White is b1. This move actually fails to create any new W-neighbours at all, and is thus reversed by any Black move. Reversible moves can be detected easily. A move is reversible it its neighbourhood, from the point of the player to move, either forms a clique or can be partitioned into a single cell *c* and a clique. In the latter case, an opponent's reply in *c* reverses the move; in the former case, any reply reverses the move. A game tree search can be enhanced by pruning these moves immediately.

Note that the move f1 in Figure 8.3 is not reversible, since even after the Black reply at e2 there is a new W-neighbourhood connection between e1 and the lower White edge. Reversible moves are rare, but the search savings might nevertheless be significant, since reversible moves tend to be forcing moves. The SEX search algorithm in Queenbee spends relatively much effort on forcing moves, since they usually have very low badness and very high tension. Pruning reversible moves would also be a safeguard against a common mistake that Queenbee is prone to make, which is playing a forcing move on the wrong side of the connection.

When combined with pattern search, reversible moves might introduce larger savings, again since reversible moves are usually forcing moves that therefore need to be searched by the pattern search. In the position of Figure 8.1-I, moves d1 and e1 are reversible for White. A pattern search with two-bridge detection and reversible move detection would then require 49 nodes to prove the win in this position.

A powerful method in combinatorial game theory is the decomposition of games into independent sub-games. This is usually the case with winning connections in Hex. The winning pattern in position 8.1-I really consists of three independent sub-patterns. Partial decomposition of a pattern is easy to achieve, since non-contiguous regions in a pattern must be independent. For example, when the algorithm tries move c6 first, position 8.1-II arises. The standard pattern search as described in Section 8.5 then concludes that only the moves within this pattern need to be considered and all other ones can be discarded. If the decomposition of pattern 8.1-II is detected, all three parts could be searched independently; if none of the sub-patterns grow to overlap the others, the win is still proved. Using this method, the 10 ply win in position 8.1-I can be proved by searching 6 nodes.

Appendix A

Sample games

This Appendix contains games that successive incarnations of Queenbee played since 1998, when the program made its debut on the online games server Playsite. [1] Note that ‘1’ and ‘2’ refer not to the outcome of the game but rather to which player went first and which player went second. The first move of the game is for Vertical. After the first move, the first player plays Horizontal if the swap option is used, and continues playing Vertical otherwise.

The opponent in the first game was an expert player who had just beaten Queenbee’s author by eight games to zero; Queenbee subsequently played and almost avenged the losses.

1: Queenbee								19 October 1998
2: Emanuele (2061)								Playsite, 10 × 10
	QB	E		QB	E		QB	E
1.	H8	F5	11.	B4	D1	21.	F8	E9
2.	H4	H5	12.	D2	E1	22.	E7	D8
3.	G5	F7	13.	E2	F1	23.	resign	
4.	C8	E6	14.	F2	G1			
5.	D6	E4	15.	H2	G2			
6.	E5	F4	16.	G3	F3			
7.	C5	D3	17.	G6	G7			
8.	D4	E3	18.	I6	G9			
9.	C3	C4	19.	H6?	H9			
10	A5	B2	20.	G8	F9			

The final position was resigned because 23 E8 D9, 24 D7 B9 and Horizontal ladders over to C4.

---

[1] See `http://www.playsite.com/games/board/hex`. The game notation used here differs from the Playsite convention in that the swap does not cause the first piece to change colour and location; rather, the *players* change colour.

However, Queenbee enjoyed a clearly won position on move 19. Simply playing 19 H9 or 19 I8 would have won the game easily; if Horizontal responds with 19 ... H6 then 20 J4 wins. More recent versions of Queenbee do not miss this win anymore.

The first official win over a strong player occurred in the following game. However, the human player committed a serious elementary error.

1: 1oak1 (1840)			21 October 1998		
2: Queenbee			Playsite, 10 × 10		
	1o1	QB		1o1	QB
1.	E10	E6	11.	I8	I7
2.	D6	C8	12.	G8	G9
3.	B8	B7	13.	A10	A9
4.	C7	B9	14.	E8	D9
5.	F6	F5	15.	J6	J7#
6.	H4	D8			
7.	I2	H6			
8.	G6	E9			
9.	H7	F9			
10.	I5	H8			

The move 10 I5 loses instantaneously by allowing a very simple edge pattern. Vertical could have won the game at that point by playing 10 G8.

Queenbee later recorded online wins against expert players. The following two games both featured players rated in the highest segment at Playsite, where the maximum player ratings are around 2100–2200.

1: JasonKidd (–)			5 June 1999		
2: Queenbee			Playsite, 10 × 10		
	JK	QB		JK	QB
1.	A2	E6	11.	E8	B7
2.	D6	D7	12.	C7	E9
3.	F5	F6	13.	F8	F9
4.	G5	G6	14.	H8	C5
5.	H7	I5	15.	D5	E3
6.	B8	E5	16.	E4	G10
7.	H5	H6	17.	I9	G7
8.	C6	B9	18.	G8	F3
9.	C8	C9	19.	F4	G3
10.	D8	D9	20.	H3	resign

This game looks quite hopeless for Queenbee, but actually it missed a win as late as two moves before the end of the game. The move 18 ... G2 would have won the game.

1: Queenbee			5 June 1999		
2: Twixter (2119)			Playsite, 10 × 10		
	QB	T		QB	T
1.	C2	swap	11.	G5	H5
2.	E6	B7	12.	H4	J3
3.	C7	B8	13.	I3	J2
4.	B10	C8	14.	I4	J4
5.	D9	A10	15.	I5	J5
6.	C5	D6	16.	I7	I6
7.	A9	B9	17.	H7	H6
8.	D7	C6	18.	F8	E8
9.	E4	H3	19.	F7	resign
10.	E5	G6			

This game is Queenbee's first official win against one of the top players; Queenbee actually went on to beat the same player in the next game as well.

Queenbee participated in the Fifth Computer Olympiad, as part of the Mind Sports Olympiad in London. In addition to Hexy, there was only one other entry in the Hex tournament. This entry illustrated the point that Hexy and Queenbee are the only two programs to transcend beginner level; the difference in playing strength between them and the third program is quite considerable.

Queenbee						24 August 2000		
Killer Bee			Computer Olympiad, London, 11 × 11					
	game 1			game 2			game 3	
	KB	QB		QB	KB		KB	QB
1.	H2	F6	1.	B2	F6	1.	A4	B11
2.	G5	G6	2.	E7	G5	2.	F6	E8
3.	E7	E6	3.	D6	E6	3.	G5	G8
4.	F5	H5	4.	D7	E5	4.	E7	C9
5.	D7	D6	5.	D5	F5	5.	G6	F8
6.	resign		6.	D9	resign	6.	D8	D9
						7.	H7	H8
						8.	resign	

Killer Bee's programmer resigned the fourth game without playing. The performance of Killer Bee illustrates how challenging it is to write a strong Hex playing program, given that its author is the very strong Hex player Emanuele Brasa, Queenbee's opponent in the first game in this Appendix.

The final of the Hex competition was played between Queenbee and Hexy, who had both won all their games against Killer Bee.

1: Queenbee											24 August 2000
2: Hexy											Computer Olympiad, London, 11 × 11
	QB	H		QB	H		QB	H		QB	H
1.	B2	swap	11.	E5	D5	21.	B8	A8	31.	D2	D3
2.	F6	G6	12.	D8	C8	22.	B7	A7	32.	H9	J8
3.	F7	G7	13.	D7	C7	23.	B6	A6	33.	I8	J7
4.	I3	H4	14.	C9	A10	24.	F11	H10	34.	I9	J9
5.	G5	I5	15.	A11	B10	25.	G10	H8	35.	I7	J6
6.	H3	G4	16.	B11	C10	26.	G9	F8	36.	resign	
7.	G3	E4	17.	B9	A9	27.	B5	A5			
8.	E3	F3	18.	C11	D10	28.	B3	B4			
9.	F2	G2	19.	D11	E10	29.	C4	C3			
10.	F4	D6	20.	E11	F10	30.	C2	E2			

1: Hexy								25 August 2000
2: Queenbee								Computer Olympiad, London, 11 × 11
	H	QB		H	QB		H	QB
1.	A2	swap	11.	K1	J2	21.	D8	D7
2.	F6	G6	12.	I6	J6	22.	F8	D9
3.	I3	H3	13.	I7	J7	23.	K10	G9
4.	H4	G5	14.	I8	J8	24.	F10	E6
5.	G4	F5	15.	I10	I9	25.	G7	resign
6.	F4	D5	16.	H10	H9			
7.	E5	D6	17.	G10	K9			
8.	E3	K2	18.	J11	G8			
9.	C4	J4	19.	F9	K8			
10.	I5	J5	20.	E8	B9			

Bibliography

[ACH90] T. Anantharaman, M. S. Campbell, and F.-H. Hsu. Singular extensions: adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109, April 1990.

[All94] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* Ph.d. thesis, University of Limburg, The Netherlands, 1994.

[Ans00] V. Anshelevich. The game of Hex: An automatic theorem proving approach to game programming. In *AAAI proceedings*, to appear, 2000.

[BBC69] A. Beck, M. D. Bleicher, and D. W. Crowe. *Excursions into Mathematics.* Worth Publishers Inc., New York, 1969.

[BCG82] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for your Mathematical Plays.* Academic Press, New York, 1982.

[Bjö00] Y. Björnsson. *Learning search control.* Ph.d. thesis, University of Alberta, Canada, 2000.

[Bro00] C. Browne. *Hex Strategy: Making the Right Connections.* A. K. Peters, Natick, Massachusetts, 2000.

[BTW98] J. Baxter, A. Tridgell, and L. Weaver. Experiments in parameter learning using temporal differences. *International Computer Chess Association Journal*, 21(2):84–99, 1998.

[Bur97] M. Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. *International Computer Chess Association Journal*, 20(3):189–193, 1997.

[Bur98] M. Buro. From simple features to sophisticated evaluation functions. In H. J. van den Herik and H. Iida, editors, *Proceedings of the First International Conference on Computers and Games (CG-98)*, volume 1558 of *Lecture Notes in Computer Science*, page 126, Tsukuba, Japan, 1998. Springer-Verlag.

[Bur99] M. Buro. Toward opening book learning. *International Computer Chess Association Journal*, 22(2):98–102, 1999.

[CHH99] M. S. Campbell, A. J. Hoane, and F. Hsu. Search Control Methods in Deep Blue. In *AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*, pages 19–23. AAAI Press, 1999.

[End] H. Enderton. Infrequently asked questions about the game of Hex. Web page `http://www.cs.cmu.edu/People/hde/hex/hexfaq`.

[End99] H. Enderton. Personal communication, 1999.

[ET76] S. Even and R. E. Tarjan. A combinatorial problem which is complete in polynomial space. *Journal of the Association for Computing Machinery*, 23:710–719, 1976.

[Gal86] D. Gale. The game of Hex and the Brouwer fixed point theorem. *American Mathematical Monthly*, pages 818–827, 1986.

[Gar59] M. Gardner. *The Scientific American Book of Mathematical Puzzles and Diversions*, chapter The game of Hex. Simon and Schuster, New York, 1959.

[Gas90] R. Gasser. *Heuristic Search and Retrograde Analysis: Their application to Nine Men's Morris.* Diploma thesis, Swiss Federal Institute of Technology, Zürich, 1990.

[GC88] G. Goetsch and M. S. Campbell. Experiments with the Null-Move Heuristic. In *Proceedings of the 1988 AAAI Symposium on Game Playing Systems*, pages 14–18, 1988. Also in T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 159–168. Springer-Verlag, New York, 1990.

[GEC67] R. D. Greenblatt, D. E. Eastlake, and S. D. Crocker. The Greenblatt chess program. *Fall Joint Computing Conference Proceedings*, 31:801–810, 1967. Also in D. Levy, editor, *Computer Chess Compendium*, pages 56–66. Springer-Verlag, 1988.

[Gin96] M. Ginsberg. Partition search. In *AAAI National Conference*, pages 228–233, 1996.

[Hya99] R. M. Hyatt. Book learning — A methodology to tune an opening book automatically. *International Computer Chess Association Journal*, 22(1):3–12, 1999.

[KM75] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[Kor85] R. Korf. Iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[LBT89] D. Levy, D. Broughton, and M. Taylor. The SEX algorithm in computer chess. *International Computer Chess Association Journal*, 12(1):10–21, 1989.

[MC82] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14(4):533–551, 1982.

[McC97] J. McCarthy. AI as sport. *Science*, 276(June 6):1518–1519, 1997.

[Nas99] J. Nash. Personal communication, 1999.

[Rij00] J. van Rijswijck. Learning from perfection — A data mining approach to evaluation function learning in Awari. In *Second International Conference on Computers and Games*, to appear, 2000.

[SA77] D. J. Slate and L. R. Atkin. Chess 4.5 - The Northwestern University chess program. In P. W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, New York, 1977.

[Sam59] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.

[Sam67] A. Samuel. Some studies in machine learning using the game of checkers – II: Recent progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.

[Sch97] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers.* Springer Verlag, 1997.

[Sch00] J. Schaeffer. The games computers play. In M. Zelkowitz, editor, *Advances in Computers 50.* Academic Press, 2000.

[Sco69] J. Scott. A chess-playing program. In *Machine Intelligence 4*, pages 255–265, 1969.

[Sha53] C. E. Shannon. Computers and Automata. In *Proceedings of the Institute of Radio Engineers*, volume 41, pages 1234–1241, 1953.

[Sut88] R. S. Sutton. Learning to predict by the methods of temporal differences. In *Machine Learning 3*, pages 9–44. Kluwer, Boston, 1988.

[Tes95] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, March 1995.

[Tho82] K. Thompson. Chomputer Chess Strength. In M. R. B. Clarke, editor, *Advances in Computer Chess 3*, pages 55–56. Pergamon Press, Oxford, UK, 1982.

[UHA89] J. Uiterwijk, H. J. van den Herik, and L.V. Allis. A knowledge-based approach to Connect-Four — The game is over: White to move wins! In D. Levy and D. Beal, editors, *Heuristic Programming in Artificial Intelligence: The first computer olympiad*, pages 113–133. Ellis Horwood Ltd., Chichester, 1989.

[UP98] P. Utgoff and D. Precup. Constructive function approximation. In H. Liu and H. Motoda, editors, *Feature Extraction, Construction and Selection: A Data Mining Perspective*, volume 453 of *The Kluwer International Series in Engineering and Computer Science*, chapter 14. Kluwer Academic Publishers, 1998.

Index

Allis, Victor, 68
alpha-beta algorithm, 11
alternative move, 67
Anshelevich, Vadim, 5
aspiration search, 12
Atkin, Larry, 2
attack mobility, 39
Awari, 17

Backgammon, 2, 16
badness, move, 41
Baxter, Jonathan, 16
beam search, 45
beam width, 45
Björnsson, Yngvi, 17
board state, 7
borders, 20
branching factor, 3, 8
Brasa, Emanuele, 75
Bridge, 3, 67
Brouwer Fixed Point theorem, 21
brute force search, 3
budget, 46
Buro, Michael, 15–17

category, move, 47
chain, winning, 20
checkers, 3, 16, 17
chess, 4, 23
    Drosophila of AI, 1
    touchstone of intellect, 1
Chinook, 3
combinatorial game theory, 70
competitive Hex, 23
Con-Tac-Tix, 20
Connect-4, 3
connection
    virtual, 5
connection, virtual, 28
coordinate system, 27
Copenhagen, University of, 19
cost, 46
Cut, 24
cutoff, α-β, 11

Deep Blue, 3, 47, 51, 59
disjoint threats, 29
distance
    W- or B-, 38
double threat, 28, 40

edge extension, 48, 49
edge pattern, 28
edge pieces, 20
evaluation function, 9
    learning, 16
Even, S., 24
exploration vs. exploitation, 16
extension
    search, 13

fail high, 11
features, board, 16
first move, 20
forcing move, 32, 34
forcing move fight, 34
forward pruning, 47

Gale, David, 21
game, 7
game, two player, 7
games
    contribution to AI, 2
    imperfect information, 3
    stochastic, 3

game theoretic value, 8
game tree, 8
Gardner, Martin, 20, 49
ghost pieces, 48
Ginsberg, Matthew, 67
Go, 3, 4, 23, 34, 41
Go-Moku, 3, 68
Goethe, Johann Wolfgang von, 1
Greenblatt, Richard, 2

Hein, Piet, 19, 49
Hex
    competitive, 23
    first move, 20
    first player win, 22
    game theoretical value, 19
    history, 19
    Internet servers, 20
    notation, 27
    no draws, 21
    rules, 20
    strategy, 27
    ultra-weakly solved, 19
    unrestricted, 23
Hexy, 5, 53, 65, 75
horizontal addition, 67
horizon effect, 11
    for Sex search, 59
Hyatt, Robert, 15, 17

imperfect information games, 3
initial position, 7
interior node, 9
Internet, 53
Internet, Hex servers, 20
iterative deepening, 2, 12

Killer Bee, 75
Knuth, Donald, 11
Korf, Richard, 2
Kronrod, Alexander, 1

ladder, 30
leaf position, 7
learning
    evaluation function, 16
    opening book, 17
    reinforcement, 2
    search control, 17
    Temporal Difference, 2

Marsland, Tony, 15
McCarthy, John, 2
memory-assisted search, 2
minimal window search, 13
minimax algorithm, 8
Moore, R. W., 11
move
    alternative, 67
    badness, 41
    category, 47
    optimal, 8
    ordering, 11
    tension, 42

Nash, John, 19, 22
negamax algorithm, 8
neighbourhood
    W- or B-, 38
Nine Men's Morris, 3
node
    interior, 9
    leaf, 9
null move search, 50

odd/even effect, 11, 51
olympiad, computer, 20
one move equalization, 23
opening book
    automatic construction, 17
    horizontal addition, 67
    vertical addition, 67
opening moves, known wins, 34
optimal play, 8
optimal move, 8
optimal successor, 8
Othello, 3, 16
outpost, 31, 32

Parker Brothers, 20
partition search, 67
pattern database, 48

Playsite, 73
ply, 9
poker, 3
Polygon, 19
pondering, 13
position, 7
    initial, 7
    leaf, 7
    white or black, 8
position, terminal, 7
potential
    board, 39
    cell, 39
    total, 41
    W- or B-, 39
pre-evaluator, 48
Princeton, 19
principal variation, 13
principal variation search, 13
PSPACE-complete, 24
PVS/MWS, 13

QBF, 24
Queenbee, 1
    evaluation function, 37
    search algorithm, 45
    web page, 1

rating, 53
reduction
    search, 13
reinforcement learning, 2
root node, 8

Samuel, Arthur, 3
SAT, 25
Schaeffer, Jonathan, 15
score function, 7
Scott, J., 2
Scrabble, 3
search
    beam, 45
    brute force, 3
    memory-assisted, 2
    null move, 50
    partition, 67
    tapered, 45
    threat space, 68
search control
    learning, 17
search depth, 9
search extension, 13
search reduction, 13
search tree, 9
search window, 12
Sex horizon effect, 59
Sex search, 46
Shannon, Claude, 1, 5
Shannon switching game, 23
    complexity, 24
    on edges, 24
Short, 24
Slate, David, 2
stochastic games, 3
strategy stealing argument, 22
successor, optimal, 8
Sutton, Richard, 3
swap rule, 23, 34

tapered search, 45
tapering function, 46
Tarjan, R. E., 24
Temporal Difference
    algorithm, 16
    learning, 2
tension, move, 42
terminal position, 7
Tesauro, Gerald, 16
threat pattern, 68
threat space search, 68
transposition, 12
transposition table, 2, 12
tree
    game, 8
    search, 9
Tridgell, Andrew, 16
Turing, Alan, 1
two-bridge, 28, 40
two-distance, 37
two player game, 7

unrestricted Hex, 23

Utgoff, Paul, 17

variation, principal, 13
vertical addition, 67
virtual connection, 5, 28

Weaver, Lex, 16
window, search, 12

zugzwang, 11