Algorithmic Approaches for Playing and Solving Shannon Games

Rune Rasmussen
Bachelor of Information Technology in Software Engineering
Queensland University of Technology

A thesis submitted to the Faculty of Information Technology
at the Queensland University of Technology
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

Principal Supervisor: Dr Frederic Maire
Associate Supervisor: Dr Ross Hayward

Faculty of Information Technology
Queensland University of Technology
Brisbane, Queensland, 4000, AUSTRALIA

2007

STATEMENT OF OWNERSHIP

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signature: Rune Rasmussen

Date:

ACKNOWLEDGEMENTS

I would like to thank my supervisors Frederic Maire and Ross Hayward, who have been patient and helpful in providing this most valuable learning experience. In addition, I thank my wife Kerry and my children for their patience and trust in me, as I left a trade career as an Electrician to pursue and finish this journey for a more interesting career path and hopefully a more prosperous life.

ABSTRACT

The game of Hex is a board game that belongs to the family of *Shannon* games, which are connection-oriented games where players must secure certain connected components in graphs. The problem of solving Hex is a decision problem complete in PSPACE, which implies that the problem is NP-Hard. Although the Hex problem is difficult to solve, there are a number of problem reduction methods that allow solutions to be found for small Hex boards within practical search limits. The present work addresses two problems, the problem of solving the game of Hex for small board sizes and the problem of creating strong artificial Hex players for larger boards.

Recently, a leading Hex solving program has been shown to solve the 7x7 Hex board, but failed to solve 8x8 Hex within practical limits. This work investigates Hex-solving techniques and introduces a series of new search optimizations with the aim to develop a better Hex solver. The most significant of these new optimization techniques is a knowledge base approach that stores and reuses search information to prune Hex-solving searches. This technique involves a generalized form of transposition table that stores game features and uses such features to prove that certain board positions are winning. Experimental results demonstrate a knowledge base Hex solver that significantly speeds up the solving of 7x7 Hex.

The search optimization techniques for Hex solvers given here are highly specialized. This work reports on a search algorithm for artificial Hex players, called *Pattern Enhanced Alpha-Beta* search that can utilize these optimization techniques. On

large board positions, an artificial Hex player based on the *Pattern Enhanced Alpha-Beta* search can return moves in practical times if search depths are limited. Such a player can return a good move provided that the evaluated probabilities of winning on board positions at the depth cut-offs are accurate. Given a large database of Hex games, this work explores an apprenticeship learning approach that takes advantage of this database to derive board evaluation functions for strong Hex playing policies. This approach is compared against a temporal difference learning approach and local beam search approach. A contribution from this work is a method that can automatically generate good quality evaluation functions for Hex players.

Contents

List of Tables **xiii**
List of Figures **xiv**

CHAPTER

1 Introduction **1**

1.1 The Game of Hex 2
1.2 Research Question and Aim 4
1.3 Significance and Contribution of Research 4
1.4 Thesis Overview 6

2 Combinatorial Games and the Shannon Switching Games **10**

2.1 Combinatorial Games 11
2.2 Shannon Switching Games 12
2.3 The Shannon Games 17
2.4 A Trivial Hex Problem 19
2.5 Chapter Discussion 21

3 Search Techniques **22**

3.1 The game-tree 22
3.2 The Minimax Search Algorithm 24
3.3 The Alpha-Beta Pruning Algorithm 25
3.4 The Transposition Table Pruning Algorithm 27
3.5 Upper Confidence Tree Search 29
3.6 Chapter Discussion 33

4 Sub-game Deduction for Hex **35**

4.1 Sub-games and Deduction Rules 35

4.2 The H-Search Algorithm 37

4.3 The Must-play Region Deduction Rule 39

4.4 Chapter Discussion 46

Part I: Hex Solving Algorithms **47**

5 A Hex Solving Search Algorithm **48**

5.1 The Pattern Search Algorithm 49

5.1.1 Pseudo Code 53

5.1.2 Performance Tests and Results 54

5.1.3 Remarks 55

5.2 Pattern Search with a Dynamic Move Generating Algorithm . 55

5.2.1 Move Order with Alpha-Beta Pruning 56

5.2.2 The Pattern Search Cut-off Condition 56

5.2.3 A Dynamic Move Generator 58

5.2.4 Hex Solver 1 60

5.2.5 Performance Tests and Results 62

5.2.6 Remarks 63

5.3 Conclusion 63

6 Applications of the H-Search Algorithm for solving Hex **64**

6.1 Fine Tuning the H-Search Algorithm 64

6.1.1 An Effective Arrangement for Sub-Game Sets 65

6.1.2 An Optimization Technique for OR Deductions 66

6.1.3 Performance Tests and Results 70

6.1.4 Remarks on the Impact of H-Search in Hex Solvers . . 72

6.2 End-of-Game Pruning 73

6.2.1 Hex Solver 2 75

6.2.2 Performance Tests and Results 78
6.2.3 Remarks 79
6.3 Extracting H-Search Features for Move Ordering 79
6.3.1 Hex Solver 3 81
6.3.2 Performance Tests and Results 82
6.3.3 Remarks 83
6.4 Move and P-Triangle Domination 84
6.4.1 Hex Solver 4 85
6.4.2 Performance Tests and Results 88
6.4.3 Remarks 88
6.5 Additional Optimization Methods using H-Search 89
6.6 Conclusion 91

7 Applications of Threat Pattern Templates For Solving Hex 92
7.1 Multi-Target Sub-games and Templates 92
7.2 A Template Matching Table 94
7.3 Template Matching Tables in Pattern Search 95
7.3.1 A Method to Standardize Templates 97
7.3.2 Hex Solver 5 98
7.3.3 Performance Tests and Results 101
7.3.4 Remarks 101
7.4 Optimization Attempts using Template Matching Tables . . . 102
7.5 Conclusion 105

Part II: Machine Learning of Artificial Hex Players 106

8 A Hex Player Overview 107
8.1 The Pattern-enhanced Alpha-Beta Search 108
8.2 Evaluation 111
8.3 Discussion 112

9 Hex Evaluation Functions by Apprenticeship Learning 113

9.1 Introduction 114

9.2 Evaluation Functions for Games 116

9.3 Apprenticeship Learning 117

9.3.1 The Cross-Entropy Method 117

9.4 Experiments and Results 120

9.4.1 The Benchmark Player 128

9.4.2 Application of Apprenticeship Learning 128

9.4.3 Apprenticeship Learning Results 129

9.4.4 Application of Temporal Difference Learning 131

9.4.5 Temporal Difference Learning Results 132

9.4.6 Application of Stochastic Local Beam Search 133

9.4.7 Stochastic Local Beam Search Results 134

9.5 Conclusion 135

10 Conclusion and Future Work 137

10.1 Contributions 138

10.2 Future Work 139

GLOSSARY 146

Bibliography 150

List of Tables

TABLE **Page**

5.1.1 The number of nodes Pattern searches must visit to completely solve the 3x3 and 4x4 Hex boards. 54

5.2.2 The number of nodes the *Hex Solver 1* algorithm must visit to completely solve the 3x3 and 4x4 Hex boards and times in seconds in comparison to the Pattern search results. 62

6.2.1 The number of nodes the *Hex Solver 2* algorithm must visit to completely solve the 3x3, 4x4 and 5x5 Hex boards and the time in seconds in comparison to the *Hex Solver 1* results. 78

6.2.2 The worst case running costs for the *Hex Solver 2* algorithm as the number of nodes that could be visited for each Pattern search on the 3x3, 4x4 and 5x5 Hex boards if the worst case running costs for H-Search are taken into account. 79

6.3.3 The number of nodes the *Hex Solver 3* algorithm must visit to completely solve Hex boards inclusively in the range 3x3 to 6x6, and the time taken in comparison to the performance of *Hex Solver 2*. . . 83

6.4.4 The number of nodes the *Hex Solver 4* algorithm must visit and search times in seconds to completely solve Hex boards inclusively in the range 3x3 to 7x7, in comparison to the performances of *Hex Solver 3*. 88

7.3.1 The number of nodes the *Hex Solver 5* algorithm must visit and the time in seconds to completely solve Hex boards inclusively in the range 3x3 to 7x7, compared against the performances of *Hex Solver 4*.101

List of Figures

FIGURE **Page**

1.1.1 A 9x9 Hex board. 3

2.2.1 A graph labeled for a Shannon Switching game. 13

2.2.2 Left: The terminal position of a Shannon Switching game won by *Connect*. Right: The terminal position of a Shannon Switching game won by *Cut*. Edges that *Connect* has captured are displayed with thick lines. 13

2.2.3 Left: An edge that *Connect* has captured. Right: *Connect's* captured edge represented by an edge contraction. 15

2.2.4 Left: A Shannon Switching game graph G equal to the union of edge disjoint spanning trees S and T. Centre: The spanning tree S in the graph G. Right: The spanning tree T in the graph G that is edge disjoint to S. 15

2.2.5 Left: The resulting graph after *Cut* deleted the particular edge (X, Y) from T. Centre: S is a spanning tree in G_1. Right: T_1 is a subgraph of G_1 that is edge disjoint to S. 16

2.2.6 Left: A Shannon Switching game graph G_2 equal to the union of edge disjoint spanning trees S_1 and T_2. Centre: The spanning tree S_1 in the graph G_2. Right: The spanning tree T_2 in the graph G_2 that is edge disjoint to S_1. 17

2.3.7 Left: The end of a Shannon game won by *Connect*. Vertices that *Connect* has captured during play are described with a circle. *Connect's* winning path is X-a-b-Y. Right: The end of a Shannon game won by *Cut*. 17

2.3.8 Left: A Shannon game that represents a 9x9 Hex game where player Black is *Connect* and player White is *Cut*. A Shannon game that represents a 9x9 Hex game where player White is *Connect* and player Black is *Cut*. 18
2.4.9 The pairing strategy is a winning strategy for White, provided that White's moves are a mirror image of Black's moves over the cell labels. 20
2.4.10 These two paths each connect a side to a cell on the short diagonal and are the cell-pair image of each other. 20
3.1.1 The game-tree of 2x2 Hex boards. 23
3.2.2 An example of a minimax search. 25
3.3.3 An example of alpha-beta pruning algorithm. 26
3.4.4 Left: After searching the subtree under board positions b_i, the minimax search finds that Black has a winning strategy for b_i. Since the game theoretic value for a Black winning strategy is (-1), the search adds the mapping $(b_i, -1)$ to the transposition table T. Right: Following a different path, the same minimax search again arrives at board position b_i. The game theoretic value for b_i can be returned by the transposition table, in place of a search in the subtree of b_i. . 28
3.5.5 An example of the UCB1 algorithm in the context of finding an estimate v of the game theoretic value for board position b. 30
3.5.6 Left: A game played between *UCB1PlayerMin* (minimizing player) and *UCB1PlayerMax* (maximizing player) showing the values that the players used to select successors on the forward track. Right: The game reaches a terminal position and the search backtracks. As the search backtracks, the outcome of this game is used to update the average outcomes at positions D,C and B. 32
4.1.1 An example of a threat pattern for player Black. 36

4.2.2 The H-Search algorithm applies the AND rule to strong sub-games A and B. The OR deduction rule is triggered to operate on a set of weak sub-games $\{A_i\}_n$, if the AND rule deduces a weak sub-game that belongs in this set. 38
4.3.3 A H-Search execution deduces several weak sub-games with targets x and y, but is not able to deduce a strong sub-game (x, S, C, y). . . 40
4.3.4 Left:The must-play region M is the intersection of weak sub-games with targets x and y, which have been deduced using the H-Search algorithm. Right: *Cut* must move in the must-play region, otherwise *Connect* can immediately secure a connection. 41
4.3.5 Given $t_x \in X$ and $t_y \in Y$, *Connect* enumerates strong sub-games, where one target is either t_x or t_y and the other target is some empty cell t_j (highlighted with a small black dot). 42
4.3.6 Left: The strong sub-game associated with Connect's move on cell t_j and whose carrier is added to $\mathcal{C}$. Right: If an application of the OR rule does not yield a strong sub-game with targets t_j and t_y then there is a must-play region M that defines a move set for *Cut*. 43
4.3.7 Left: On this search path, the X and Y stone sets are virtually connected via a strong sub-game with targets 4 and b. Right: On this search path, the X and Y stone sets are virtually connected via a strong sub-game with targets 0 and c. 44
4.3.8 A form of OR deduction is applied to the weak sub-games on boards A and B to form the strong multi-target sub-game on board C. 45
5.0.1 The approximate sizes of Hex game-trees for 3×3 to 8×8 Hex according to Even and Tarjan in [23]. 49
5.1.2 The process in a Pattern search for constructing threat patterns. . . . 51
5.1.3 The Pattern search is in White mode and switches to Black mode because there is a strong threat pattern for Black at Y. 52

5.2.4 The alpha-beta pruning algorithm may be less effective if nodes J and F are not first. 56
5.2.5 A Pattern search cut-off condition. 57
5.2.6 Left: Each carrier cell has the number of times White moved on that cell to connect the targets. Right: Black's cutting move is the move White used most often to connect. 58
5.2.7 The accumulation of connection utilities in a must-play region reveals a good move for Black. 59
6.1.1 The H-Search algorithm applies the AND rule to strong sub-games A and B. The OR deduction rule is triggered to operate on a set of weak sub-games $\{A_i\}_m$, if the AND rule deduces a weak sub-game that belongs in this set. 66
6.1.2 Left: The Venn diagram that represents carriers C_1, C_2 and C_3 for sub-game path $A_1 - A_2 - A_3$ in an OR deduction search. Right: Since I is a subset of C_4, the tree rooted at sub-game $A_4 = (x, S, C_4, y)$ can be pruned from the search. 68
6.1.3 Top: The performance test results for the Anshelevich's OR deduction procedure given by Algorithm 6.1.3. Bottom: The performance test results for our improved OR deduction procedure given by Algorithm 6.1.4. In both plots m is the size of sub-game buckets. . . . 71
6.2.4 At position $\mathcal{B}_2$, a Pattern search executes a Black-H-Search subroutine that finds a strong threat pattern. The Pattern search can now treat position $\mathcal{B}_2$ as a terminal position and backtrack. 74

6.2.5 At position $\mathcal{B}_2$, a Pattern search executes a Black-H-Search subroutine that finds a set of weak threat pattern $\{A_1, A_2, A_3\}$. The Pattern search treats this set of threat patterns as though they were return values from Pattern searches on the successors of $\mathcal{B}_2$. The intersection of weak carriers C_1, C_2 and C_3 gives a must-play region M in the Pattern search. 75

6.3.6 The move generator traverses a single level game-tree rooted at a position where White has the turn. At each successor the move generator applies the H-Search algorithm 80

6.4.7 Left: A Black-Triangle where the tip is empty. Right: A Black-Triangle where the tip has a Black stone. 85

6.4.8 Blacks winning must-play region of opening moves for 7x7 Hex. . . 89

6.5.9 Left: A weak threat pattern found during a Pattern search by undoing a White move that breaks a White stone group in two about the empty cell x. Right: The empty cell y can be added to the carrier to construct a strong threat pattern. 90

7.1.1 Left: An arbitrary board position b. Right: A template that proves b is winning for Black. 93

7.2.2 Left: A game-tree search finds that Black has a winning strategy for b_i represented by template t_k. The search adds the mapping $(t_k, -1)$ to the template matching table T_T. Right: On a different path, the same search arrives at board position b_n. An application of Proposition 7.1.2 shows that template t_k matches b_n. The game theoretic value for b_n can be returned by the template matching table, in place of a search in the subtree of b_n. 95

9.3.1 Top-left: Given a population at $t = 0$ in a domain w and a performance function $R(w)$ the Cross-Entropy method selects an elite subpopulation whose mean individual is μ_0. Top-right and bottom-left: The mean individual of the elite is used to generate the next population according to a Gaussian random generator. The performance function $R(w)$ is used to find the new elite subpopulation. Bottom-right: Eventually the Cross-Entropy method converges and returns the mean individual μ_3 at the limit. 119
9.4.2 Left: The neighbourhood of empty cell x, which is not adjacent to any stones. Centre: The neighbourhood of empty cell x from Black's perspective, given x is adjacent to a Black stone group. Right: The neighbourhood of empty cell x from White's perspective, given x is adjacent to a Black stone group. 121
9.4.3 A simple circuit where an electric source delivers a work potential to a device, the device allows an electric current flow, which is determined by the resistance of the device. 124
9.4.4 A junction x in a resistor network, whose potential is V_x. 125
9.4.5 The tournament results of players P_j that was derived using Apprenticeship Learning on a 3GHz Intel Pentium 4 machine at iteration j and at time (minutes:seconds) in 20 games against the benchmark player P_R. 130
9.4.6 The performance of elite weights in the Cross Entropy method at iteration j in solving the optimization problem for this Apprenticeship Learning. 130
9.4.7 The tournament results of players P_j that was derived using Temporal Difference Learning on a 3GHz Intel Pentium 4 machine at iteration j and at time (hours:minutes), in 20 games against the benchmark player P_R. 133

9.4.8 The tournament results of players P_j that was derived using Stochastic Local Beam searches at iteration j and at time (hours:seconds), in 20 games against the benchmark player P_R. 135

Chapter 1

Introduction

Techniques for solving two-player discrete finite games have been important in the analysis of complex network problems. Calbert et al. in [14] show that the dynamics of many complex networks are adversarial and reducible to such games. In communication networks where an attacker might systematically disable network routers, a network administrator can apply game-solving techniques to maintain critical connections [33]. In voice over IP applications, techniques for solving two-player word games can guide the prediction of lost packets [30]. Hex is a kind of game that can be reduced to a canonical form, called an *LR game* [15]. In addition, the techniques for solving Hex may be modified to solve Hex in its *LR game* form. The implication is that such techniques would have applications in solving many other combinatorial games, because every combinatorial game can be reduced to an *LR game*. Techniques in this work for solving Hex could have applications in the computer games industry. Massively Multi-player Online Puzzle Games (MMOPGs) are game platforms based entirely on turn-based discrete finite games, where users may choose to learn how to play a game or to refine their game playing skills against artificial game players. Techniques for solving and playing Hex could be modified for artificial players of many other combinatorial games.

Algorithms for automatic game playing emerged circa 1950s and appeared in many pioneering works, such as the work done by Shannon in [43] on programming a computer to play Chess. In that paper, Shannon described a search that could be used to solve two-player board games. This search is now known as a

minimax search. An effective extension to the minimax search is the *alpha-beta pruning* algorithm. The alpha-beta pruning algorithm can prune branches from a minimax search and for many games it can extend the horizon of solved board positions [6][12]. Today the list of game solving algorithms is very diverse and many variations on the minimax search are available [37]. Although so many good algorithms are available, many two-player board games remain unsolved. The game of Hex is a very good example, as Hex has only been solved for the first seven board sizes [27]. The game of Hex belongs to a family of connection-oriented games called Shannon games, where the aim for the players is to secure or to cut certain paths that connect two designated vertices in finite graphs [25]. Hex is PSPACE-complete, which implies that Hex is NP-Hard [40][47]. However, algorithms do exist that can effectively reduce the problem of solving Hex, so that solutions for some small Hex boards can be found in *practical*[1] times.

1.1 The Game of Hex

The game of Hex is played on a tessellation of hexagonal cells that cover a rhombic board (see Figure 1.1.1)[25]. Each player has a cache of coloured stones. The goal of *Black*; who is the player with the black stones, is to connect the black sides of the board with an unbroken chain of stones. Similarly, the goal of *White*; who is the player with the white stones, is to connect the white sides of the board with an unbroken chain of stones. The initial board position is empty. Players take turns and place a single stone, from their respective cache, on an empty cell. The first player to connect their sides of the board is the winner. The game of Hex never ends in a draw [25].

[1] In this thesis, a Hex solving search will be deemed *practical* if it traverses less 25-million nodes, takes less than 1000 hours and has a total memory requirement that is less than 2 Gigabytes. For artificial Hex players, a search will be deemed *practical* if the player takes no longer than sixty-seconds to return a move.

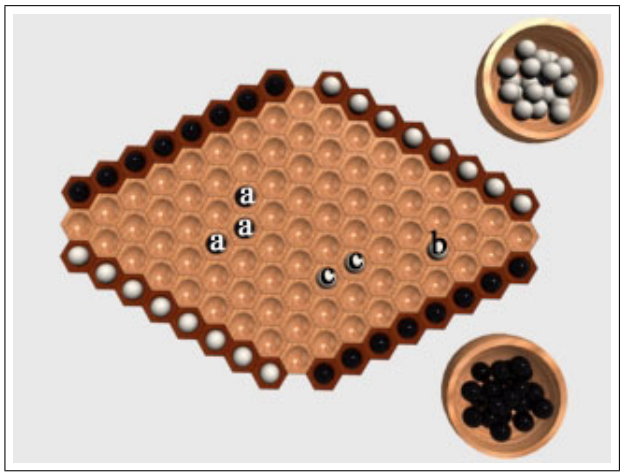

Figure 1.1.1: A 9x9 Hex board.

If a certain player's stones form an unbroken chain, not necessarily between that player's sides, then each single stone is said to be *connected* to itself and any two stones in that chain are said to be *connected* to each other. A *group*, is a maximal connected component of stones [3, 4]. Figure 1.1.1 shows seven groups where three of the seven groups have the labels 'a', 'b' and 'c', respectively. The four sides of the board also constitute four distinct groups. A player wins a game, when the opposite sides for that player are connected.

For many board games, investigating the outcome of play on subregions of board positions can reveal winning moves. For the game of Hex, a game that can be played on a subregion of a board position is called a *sub-game* [3, 4]. In a sub-game, the players are called *Cut* and *Connect*. Both *Cut* and *Connect* are restricted to play on a subset of the empty cells between disjoint targets, where a *target* is either an empty cell or one of *Connect's* groups. This subset of the empty cells that defines the playing region of a sub-game is called a *carrier*. The player's roles are not symmetric, as *Connect* moves to form a chain of stones connecting the two targets, while *Cut* moves to prevent *Connect* from forming any such chain of stones.

1.2 Research Question and Aim

Given the difficultly of solving Hex and recent advances in Hex solving approaches that use sub-game deduction rules, this work asks how can sub-game deduction be used to solve small Hex boards? Given, current Hex solving programs are bound by practical limits to solving small Hex boards, this work asks how Hex solving techniques combined with machine learning might be applied to improve the playing performance of artificial Hex players on large board?

This aim of this work was to solve Hex for small boards and to devise strong artificial Hex players for larger boards.

1.3 Significance and Contribution of Research

The game of Hex was first presented in Denmark in 1942 at the Niels Bohr Institute of Theoretical Physics as an interesting mathematical problem, by inventor and engineer Piet Hein. In 1949, John F. Nash proved the existence of a winning strategy for the opening player of Hex using a strategy stealing argument. Unfortunately, his proof does not render a winning strategy. In 1953, Claude Shannon and E. F. Moore of the Bell Telephone Laboratories devised the first automated Hex player. Their player was an electromechanical device where the electronic component was a network of resistors [25]. Computer models of resistor networks have been used by more recent computer based Hex players, such as the *Queen-bee* player in [45] and the *Hexy* player in [3, 4], to evaluate board positions. An additional feature of *Hexy* is that it used sub-games to more accurately evaluate board positions, which were deduced from elementary sub-games via the *H-Search* algorithm [3, 4]. Given the *Pattern Search* algorithm presented in [46], Hayward et al. applied a series of optimizations to devise a Hex solving program called *Solver* that has successfully solved the 7x7 Hex board for all opening positions [27]. The problem of solving

8x8 Hex remains open.

The game of Hex has made such a historical mark because it is a game that has very simple rules, but is very difficult to solve. Hex solutions can have wide applications, as they can be applied to optimization problems that can be cast to the problem of solving Hex. Even in the instances where Hex cannot be solved, strong artificial Hex players can have wide applications for the same reasons. This work makes the following contributions towards the problem of solving Hex for small boards and creating strong artificial Hex players for larger boards:

1. In Section 5.2, a move generating algorithm for Pattern searches that utilizes features from terminal board positions to evaluate the utility of cells on non-terminal positions.
2. In Section 6.1, a revised sub-game deduction procedure for the H-Search algorithm that speeds up H-Search executions.
3. In Section 6.4, a first independent confirmation of results reported by Hayward et al. for their *Solver* program in [27] via a Hex solving algorithm that identifies their move generating algorithm, which was critical to solving 7x7 Hex.
4. In Sections 7.1 and 7.2, a generalized form of transposition table, called a *template matching table* that uses sub-games to prove that certain board positions are winning.
5. In Section 7.3, a Hex solving algorithm that uses template matching tables to significantly reduce the time taken to solve 7x7 Hex.
6. In Chapter 9, an apprenticeship learning approach that automatically generates high quality evaluation functions for artificial Hex players from a database of games.

In addition to these contribution are the following publications:

- Rasmussen, Rune K. and Maire, Frederic D. and Hayward, Ross F. (2007) *A Template Matching Table for Speeding-up Game-Tree Searches for Hex.* In the Proceedings of AI 2007: The 20th ACS Australian Joint Conference on Artificial Intelligence, Gold Coast, Australia.

- Rasmussen, Rune K. and Maire, Frederic D. and Hayward, Ross F. (2006) *A Move Generating Algorithm for Hex Solvers.* In the Proceedings of AI 2006: The 19th ACS Australian Joint Conference on Artificial Intelligence 4304/2006, pages pp. 637-646, Hobart, Australia.

1.4 Thesis Overview

The problem of solving Hex is complete in PSPACE. This means that given unlimited time a Hex solving algorithm can solve Hex in polynomial space. Given that the canonical complete problem in PSPACE is NP-Hard, the problem of solving Hex is also NP-Hard. Although the problem of solving Hex is very difficult, many good problem reduction methods are available that allow Hex to be solved within practical limits for some small Hex boards. This thesis investigates new methods to further reduce the problem of solving Hex and the application of such method in artificial Hex players. The chapters of this thesis are as follows:

Chapter 2: Combinatorial Games and the Shannon Switching Games This chapter reviews the literature on a family of games called combinatorial games and discusses the complexity classes associated with solving these games. In addition, this chapter reviews a family of combinatorial games called *Shannon* games, which the game of Hex is a member. The problems of solving Shannon games are presented and the difficulty of solving Hex is placed into context.

Chapter 3: Search Techniques This chapter reviews the literature on game-tree structures and the minimax search algorithm for solving combinatorial games. In

addition, it gives an overview of the alpha-beta algorithm and transposition tables, which are used to prune searches. Finally a Monte Carlo approach called the *UCT* search is presented. The *UCT* search is able to search large numbers of random games and derive accurate evaluations of board positions.

Chapter 4: Sub-game Deduction for Hex This chapter reviews the literature on Hex sub-games and sub-game deduction rules. It reviews an algorithm called *H-Search*, which applies sub-game deduction rules to generate new sub-games. In addition it reviews a generalized game-tree search called *Must-play Deduction Search* for deducing sub-games.

Chapter 5: A Hex Solving Search Algorithm This chapter first reviews the literature on a Hex solving algorithm called the *Pattern Search* algorithm and then reports on a new move generating algorithm for optimizing Pattern searches. The *Pattern Search* algorithm provided a base for all Hex solving algorithms presented in this thesis. A contribution of this work is a new move generating algorithm to improve pruning in Pattern searches. This new move generating algorithm is demonstrated in the first of a series of Hex solving algorithms, called the *Hex Solver 1* algorithm.

Chapter 6: Applications of the H-Search Algorithm for solving Hex This chapter presents two contributions of this work. The first contribution is an optimization of the H-Search algorithm and the second contribution is the confirmation of results reported by Hayward et al. in [27] for their *Solver* program in solving 7x7 Hex. This chapter identifies the application of H-Search and the move generating algorithm used for optimization in the *Solver* program. In addition, this chapter gives a review of the literature on a property called move *domination* that can be used in a Pattern search optimization technique that eliminates moves. The applications of optimization techniques reported for the *Solver* program that are based

on H-Search, move generation and move domination are cumulatively identified in three successive Hex solving algorithms called *Hex Solver 2*, *Hex Solver 3* and *Hex Solver 4*. The *Hex Solver 4* algorithm employs all three optimizations and is used to confirm results reported for the *Solver* program in [27] on the problem of solving the 7x7 Hex board.

Chapter 7: Applications of Threat Pattern Templates For Solving Hex This chapter introduces two contributions of this work. The first is a generalized form of transposition table, called a *template matching table* that can be used to optimize Hex solving algorithms. The second contribution is a Hex solving algorithm called the *Hex Solver 5* algorithm that extends on *Hex Solver 4* with template matching tables and is found to perform significantly better than the *Solver* program in [27] on the problem of solving 7x7 Hex.

Chapter 8: A Hex Player Overview This chapter provides a bridge that relates the Hex solving techniques of previous chapters to the problem of creating artificial Hex players. This bridge is given in a review of the literature on a hybrid search called *Pattern-enhanced Alpha-Beta* search, which combines minimax search with alpha-beta pruning and the Pattern Search algorithm [46]. This review of the *Pattern-enhanced Alpha-Beta* search provides an introduction to the problems associated with creating artificial Hex players and provides a lead into the subject matter of Chapter 9.

Chapter 9: Hex Evaluation Functions by Apprenticeship Learning The application of apprenticeship learning is explored in this chapter in deriving strong artificial Hex players. This chapter presents a contribution of this works, which is an apprenticeship learning approach shown to rapidly generate good quality Hex playing policies given databases of Hex games between unevenly matched players.

Chapter 10: Conclusion and Future Work The contributions of this work are reiterated in this Chapter and a discussion about how this work could be directed in future research projects. This discussion briefly touches on ways that could assist in solving the 8x8 Hex board, the application of solving techniques in this work in other combinatorial games and to the UCT search reviewed in Chapter 3.

Chapter 2

Combinatorial Games and the Shannon Switching Games

The game of Hex belongs to a family of games called combinatorial games. Combinatorial games are discrete and finite two-player games. The problem of solving combinatorial games belong to the PSPACE class of decision problems. Problems in this class can be solved in polynomial space but some cannot be solved in polynomial time [20]. The game of Hex also belongs to the family of games called *Shannon* games. Shannon games are played on finite graphs where the players either secure or cut paths connecting designated vertices. Shannon games have very simple rules, but some Shannon games are very difficult to solve. The game of Hex is a form of Shannon game that is difficult to solve.

This chapter sets the context for Shannon games by exploring in Section 2.1, combinatorial games and problems in the PSPACE class associated with solving combinatorial games. Section 2.2 presents a family of Shannon Game called the *Shannon Switching games*. The Shannon Switching games are played on the edges of graphs that can be solved trivially and this section presents the general solution. Section 2.3 presents the Shannon Games played on the vertices of graphs, which in general cannot be solved trivially. The game of Hex is a Shannon Game played on the vertices of a graph. This section identifies the problem of solving Hex as NP-Hard; however a particular solvable case of Hex is presented in Section 2.4.

2.1 Combinatorial Games

The family of *combinatorial games* are those two-player games where:

1. there is no element of chance,
2. information about the game is not hidden from the players,
3. the players take alternate turns,
4. the outcome of a game is reached in a finite set of moves[1],
5. the outcome is either a win for one of the players or a draw [24].

The problem of solving combinatorial games shares many characteristics with the *Quantified Boolean Formula* problem, which is a generalized form of the *Boolean Satisfiability* problem [20]. The *Boolean Satisfiability* problem is a decision problem that asks for an assignment of Boolean values to the variables of a logical expression so that the return value is true. The Boolean Satisfiability problem is the first problem proved to be complete in the NP class, which is a class of decision problems solvable in polynomial time by a nondeterministic Turing machine [18]. The *Quantified Boolean Formula (QBF)* problem involves a quantified Boolean formula of the form: $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n F$, where Q_i are quantifiers, the x_i are Boolean variables and F is a well-formed formula in normal conjunctive form of variables $x_1, x_2, \ldots, x_n$. This problem asks for an assignment of Boolean values that satisfies the quantified formula. The QBF problem is the canonical complete problem for a class of decision problems known as PSPACE [23]. The PSPACE class is the set of decision problems that can be solved by a deterministic Turing machine given unlimited time but only using polynomial space.

The characteristics of solving combinatorial games are very similar to the QBF problem. A game where the first player has a winning strategy, the first player is

[1] It is possible for a game to enter an infinite loop over a finite set of board positions. Such loops can be detected after a finite set of moves. The outcome of a games with an infinite loop is a draw.

A and the second player is **B**, the optimization problem of solving this game asks for a set of board positions and moves that satisfies the condition: *There exists a move for **A**, such that for all moves by **B**, there exists a move for **A**, such that for all moves by **B**, ..., there exists a move for **A** such that **A** wins*. Similar strategies are required for games where the second player has a winning strategy or both players have a draw strategy. The decision problem associated with solving a game is PSPACE-complete if that decision problem is a QBF problem. The consequence for game problems that are PSPACE-complete is that two perfect players can search for a winning strategy using polynomial space, but it is extremely unlikely that a winning strategy can be found in polynomial time (unless P = PSPACE) [20]. Not all problems in PSPACE are in the worst case solvable in exponential time. For example, the *Shannon Switching games* are a set of combinatorial games that are in PSPACE, but are not PSPACE-complete. The *Shannon Switching games* can be solved in polynomial time [31][35].

2.2 Shannon Switching Games

Much of Shannon's earlier work explored the symbolic analysis of electric switching circuits consisting of switching devices known as relays [17]. A family of games that can characterize competitions in such switching circuits are the Shannon Switching games. A *Shannon Switching game* is a game played on the edges of a *Shannon graph*, which is a finite graph where two vertices have been designated for a special purpose . The two players in a Shannon Switching game can be called *Cut* and *Connect*. A move by player *Connect* captures an edge in the graph and a move by player *Cut* deletes an edge from the graph. Player *Cut* is **not** allowed to delete an edge that has already been captured by *Connect*. There are two vertices X and Y, called *targets*, which are preserved for a special purpose in these games (see Figure 2.2.1). Player *Connect* wins if *Connect* captures a path that connects target X to target Y (see left of Figure 2.2.2). Player *Cut* wins if *Cut* disconnects target

X from target Y (see right of Figure 2.2.2). A Shannon Switching game can never end in a draw [35].

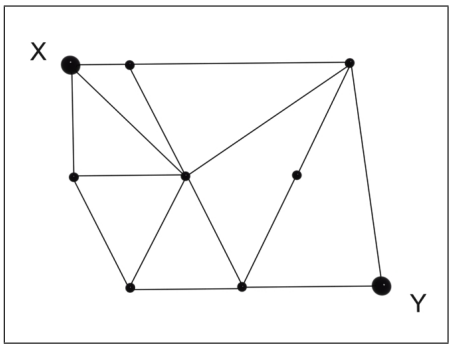

Figure 2.2.1: A graph labeled for a Shannon Switching game.

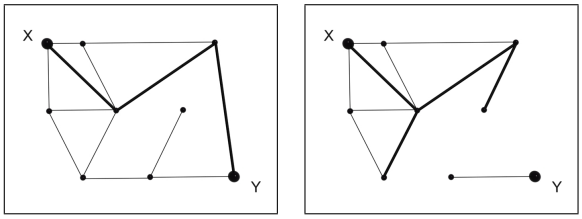

Figure 2.2.2: Left: The terminal position of a Shannon Switching game won by *Connect*. Right: The terminal position of a Shannon Switching game won by *Cut*. Edges that *Connect* has captured are displayed with thick lines.

To show that a Shannon Switching game can never end in a draw, let G_a be a terminal position where *Connect* has captured every edge and G_b be a terminal position where *Connect* has not captured every edge. The two distinct graphs have the following properties:

1. **In graph G_a:** Either, X is not connected to Y and *Cut* has won or X is connected to Y and *Connect* has won.
2. **In graph G_b:** Either *Connect* has won, *Cut* has won or the position is a draw case.

Graph G_a is never a draw case; however, graph G_b could be a draw case. Furthermore, the edges are consumed by the players, so, graph G_b could never be a draw case due to repeating positions in a cycle. If Shannon Switching games have a draw case, then only graph G_b can represent that case.

In graph G_b there is a maximum subgraph $G(V, E)$ where *Connect* has captured every edge. Since G has the same properties as G_a, G has the following two cases:

1. ***Connect* has won in G:** Implies $X, Y \in V$ and *Connect* has won in G_b.
2. ***Cut* has won in G:** But, this does not imply that *Cut* has won in G_b and G_b may still be a draw.

Assume that *Cut* has won in G and G_b is a draw. If *Cut* has won in G but not in G_b then X must be connected to Y in graph $G_b - G$ with non-captured edges, which implies G_b is not terminal since $G_b - G$ is not terminal. However, this is a contradiction because G_b is terminal. Therefore, *Cut* has won in G_b [2]. All possible terminal positions have been exhausted without a draw case. $\diamond$

If the graph of a Shannon Switching game has a subgraph connecting X to Y with two edge disjoint spanning trees, then *Connect* has a winning strategy and the game can be solved in polynomial time [31]. To show how this strategy works, it will be useful to take the view that *Connect* captures an edge by contracting it (see Figure 2.2.3). The consequence of this view is that the move of a game won by *Connect* contracts the edge (X, Y). According to Mansfield in [35], any Shannon game graph where a subgraph connects X to Y with two edge disjoint spanning trees is called *positive*.

[2]The graph G_b can only be terminal if either *Connect* or *Cut* has won in $G_b - G$. Since $G_b - G$ has no captured edges, only *Cut* could have won in $G_b - G$. In this particular case where G_b is terminal, if *Cut* has won G then *Cut* has won G_b and if *Connect* has won G then *Connect* has won G_b.

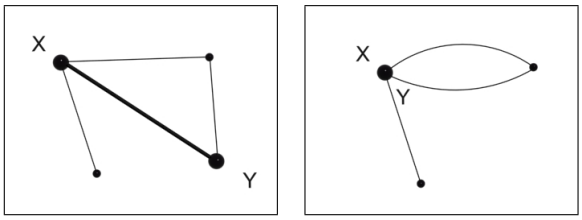

Figure 2.2.3: Left: An edge that *Connect* has captured. Right: *Connect's* captured edge represented by an edge contraction.

Consider a Shannon Switching game on the positive graph G given in Figure 2.2.4. Graph G is the union of edge disjoint spanning trees S and T. If *Cut's* first move deletes an edge from G that happens to also be in spanning tree T, then *Connect's* winning move is to contract an edge in G that is also in S, so that this *Connect* move reconnects the two components derived from T after *Cut's* move.

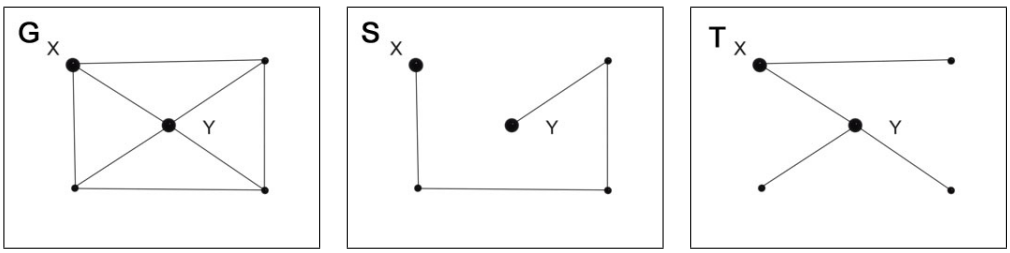

Figure 2.2.4: Left: A Shannon Switching game graph G equal to the union of edge disjoint spanning trees S and T. Centre: The spanning tree S in the graph G. Right: The spanning tree T in the graph G that is edge disjoint to S.

For example, assume *Cut* deletes the particular edge (X, Y) in T (see Figure 2.2.4). The result of *Cut's* move is graph G_1 in Figure 2.2.5, which is not positive because T_1 is not a spanning tree. The aim for *Connect* is to contract an edge in G_1 that results in a positive graph. In Figure 2.2.5, G_1 is the union of edge disjoint graphs S and T_1. Player *Connect* has to find a move in S that contracts G_1 to a positive graph. In this example, *Connect* can achieve this goal with a move that

contracts either edge (X, b), (a, Y) or (a, c).

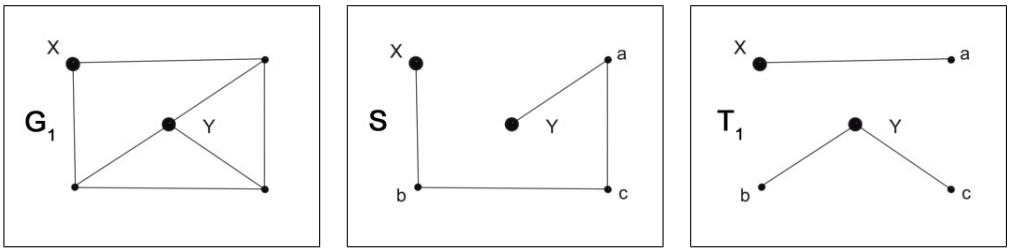

Figure 2.2.5: Left: The resulting graph after *Cut* deleted the particular edge (X, Y) from T. Centre: S is a spanning tree in G_1. Right: T_1 is a subgraph of G_1 that is edge disjoint to S.

Figure 2.2.6 shows the positive graphs if *Connect* chooses to contract edge (X, b). Graph G_2 is positive because it is the union of two edge disjoint spanning trees S_1 and T_2. The argument for *Connect's* winning strategy in G_2 is analogous to the argument for *Connect's* winning strategy in G. In [31], Lehman used Matroids of Shannon graphs to prove that for every positive graph and for every *Cut* move, *Connect* has a reply that results in either a win for *Connect* or a positive graph. In addition, Lehman applied the dual Matroid to prove *Cut*'s winning strategy in the dual graph. Mansfield gives a description of *Cut* and *Connect's* respective winning strategies in [35] without using Matroids. An algorithm for a perfect Shannon Switching game player was given in [16] and in [31].

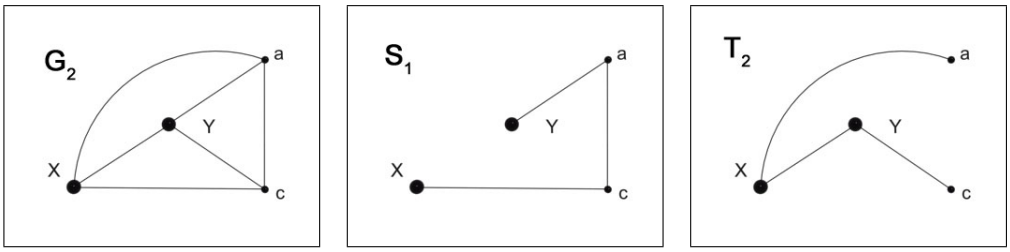

Figure 2.2.6: Left: A Shannon Switching game graph G_2 equal to the union of edge disjoint spanning trees S_1 and T_2. Centre: The spanning tree S_1 in the graph G_2. Right: The spanning tree T_2 in the graph G_2 that is edge disjoint to S_1.

2.3 The Shannon Games

A *Shannon game* is a combinatorial game where the players play on the vertices of a Shannon graph [9]. A *Connect* move captures a vertex in the graph and a *Cut* move deletes a vertex from the graph. Shannon games are the same as the Shannon Switching game, except that each player plays on the vertices instead of the edges and no player is allowed to delete or capture a target. A Shannon game can never end in a draw. Shannon games have in the past been called *Shannon Switching Games on the Vertices* [23]. Figure 2.3.7 shows two terminal Shannon games where on the left *Connect* has won and on the right *Cut* has won.

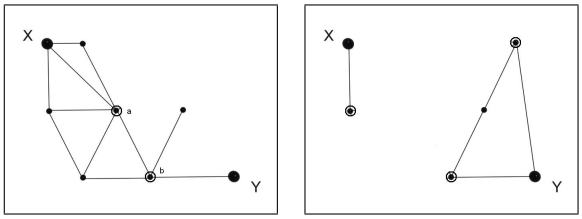

Figure 2.3.7: Left: The end of a Shannon game won by *Connect*. Vertices that *Connect* has captured during play are described with a circle. *Connect's* winning path is X-a-b-Y. Right: The end of a Shannon game won by *Cut*.

The family of Hex games are members of the Shannon games. Figure 2.3.8 shows two Shannon game graphs based on the cell adjacency graph of a 9x9 Hex board and augmented with target vertices X and Y. The Shannon game on both graphs is equivalent to 9x9 Hex. In the right of Figure 2.3.8, *Connect* represents the Black player, *Cut* represents the White player and target vertices X and Y represent Black's sides. In the right of Figure 2.3.8, *Connect* represents the White player, *Cut* represents the Black player and target vertices X and Y represent White's sides.

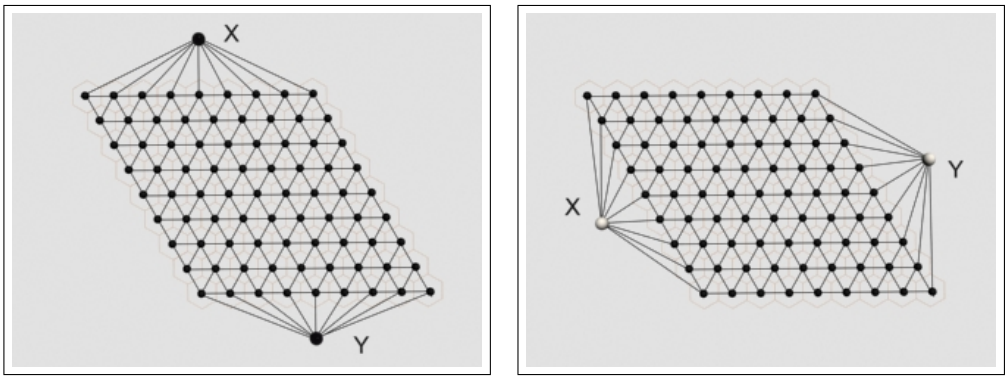

Figure 2.3.8: Left: A Shannon game that represents a 9x9 Hex game where player Black is *Connect* and player White is *Cut*. A Shannon game that represents a 9x9 Hex game where player White is *Connect* and player Black is *Cut*.

Even and Tarjan [23] show that the problem of determining a winning strategy in a general Shannon game is PSPACE-complete. In the extremely unlikely event that this problem is solvable in polynomial time, the consequence would be that every problem complete in PSPACE is solvable in polynomial time. Even for a game of Hex, which is played on a planar graph, Reisch proved in [40] that the problem of determining a winning strategy is PSPACE-complete. Their results show that the Shannon game problem and the special case of the Hex game problem are both NP-hard problems[3].

2.4 A Trivial Hex Problem

Although the Hex problem is difficult to solve, any Hex game on a board with dimensions $n \times (n-1)$ is solvable in polynomial time [10, 11]. Figure 2.4.9 shows a 9x8 Hex board where White has the winning strategy, even if white moves second. White's strategy is the mirror image of Black's strategy across the short diagonal of the board. White observes the label on the cell associated with each Black move and replies with a move on the empty cell that has the same label. This strategy is known as a *pairing strategy*.

In Figure 2.4.9, the White player has a pairing strategy. The board has labels such that the labels on cells in the *right triangle* region of the board correspond with labels on cells in the *left triangle* region. This labeling means that every path of cells that connect Black's sides has at least two cells with the same label.

In Figure 2.4.10, Black was the opening player and now has a chain of stones in the left triangle region up to the short diagonal. As a consequence of the pairing strategy, White now has a chain of stones in the right triangle region that is a mirror image of Black's stones. Black cannot form a winning chain by moving on cells

[3]It is widely suspected that PSPACE-complete is outside of NP, but this has not been proven.

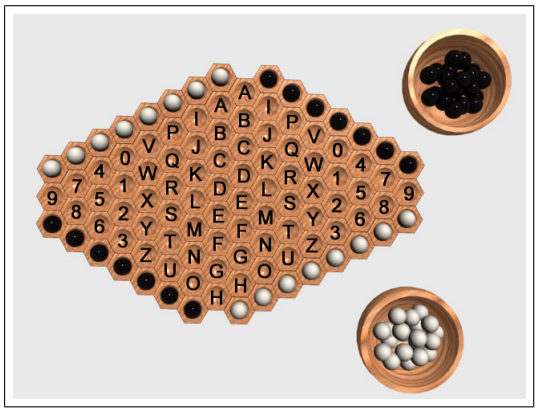

Figure 2.4.9: The pairing strategy is a winning strategy for White, provided that White's moves are a mirror image of Black's moves over the cell labels.

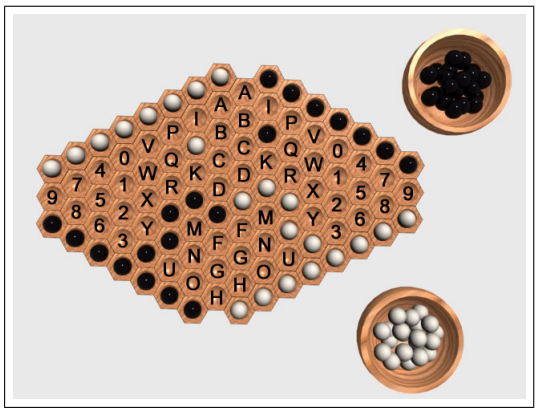

Figure 2.4.10: These two paths each connect a side to a cell on the short diagonal and are the cell-pair image of each other.

with the labels $\{F, G, H, M, N, O, U\}$, because White's chain of stones cuts all such paths.

If Black managed to win, then Black would have a path of stones that connects Black's sides. In addition, the cell labels in this path would be unique, because White would have applied the pairing strategy which would have mirrored Black's every move, as seen in Figure 2.4.10. However, a condition of the cell labeling for the pairing strategy is that all cell paths that connect Black's sides have two or more cells with the same label. So, Black could not possibly win.

2.5 Chapter Discussion

Combinatorial games are two-player, finite and discrete. They do not involve chance and the players have complete information. The players take alternate turns to move and a game ends after a finite set of moves with one player the winner or in a draw. The problem of solving combinatorial games have characteristics similar to the QBF problem, which is the canonical complete problem in PSPACE. Shannon switching games and Shannon games belong to the family of combinatorial games. Shannon switching games are played on the edges of a graph, while Shannon games are played on the vertices of a graph. The games of Hex on different board sizes is a subset of the Shannon games. The problem of solving a generalized Shannon game is PSPACE-complete and even the particular problem of solving Hex is PSPACE-complete, which implies that the problem of solving these games is NP-Hard. Although the problem of solving Hex is difficult, later chapters will show that there are many good problem reduction methods for solving Hex.

Chapter 3

Search Techniques

The approach for solving a combinatorial game generally involves a search that explores valid board positions and moves. The purpose of such a search is to determine if one player has a winning strategy or if both players have a draw strategy. The valid board positions and moves can be organized into a tree structure called a *game-tree*. To determine if one player has a winning strategy or if both players have a draw strategy, a search of a game-tree can be used to solve the problem posed in Section 2.1 for solving a given combinatorial game. Such a game-tree search traverses at least the sets of board positions and moves that satisfy this problem and deduces from this traversal the outcome of a game between perfect players.

Section 3.1 provides a detailed review of the game-tree structure. Section 3.2 gives a review of the *minimax* search, which is a game-tree search that can be used to create artificial game players and to solve many different combinatorial games. Section 3.3 reviews the *alpha-beta* pruning algorithm for minimax searches. A review of transpositions tables and their application in pruning minimax searches is given in Section 3.4.

3.1 The game-tree

The complete information of a combinatorial game can be represented in a hierarchy known as a *game-tree*. A *game-tree* is a rooted tree, whose nodes are valid board positions and whose edges are legal moves [45]. A game-tree represents the complete information of a combinatorial game as it has the complete set of valid

board positions. In a game-tree, a *game* is a single path beginning with the root node and ending with a leaf node. Board positions at the leaf nodes are called *terminal* board positions. The children of a board position are called *successors*. Figure 3.1.1 shows a game-tree representation for 2x2 Hex.

Combinatorial games are solved using algorithms that search game-trees. Because the decision problem for combinatorial games is in PSPACE, an objective for game solving algorithms is that they solve games using polynomial space. This objective can be achieved with algorithms that perform depth-first traversals of game-trees. At any instance in a depth-first traversal of a game-tree, it is clear that the number of board positions that the search algorithm stores is proportional to the height of the game-tree [23]. Given a board position the aim for a game-tree search is to return the winning player or a draw. In some cases, the search also aims to return the winning player's strategy or a draw strategy [48, 49].

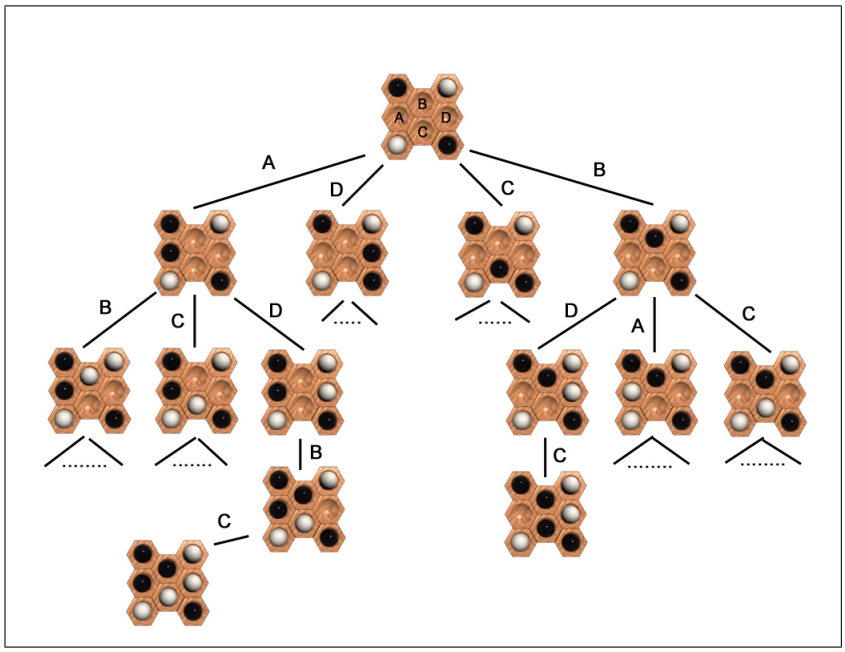

Figure 3.1.1: The game-tree of 2x2 Hex boards.

3.2 The Minimax Search Algorithm

On the problem of an artificial Chess player, Shannon devised a game-tree search that is now known as the *minimax* search [43]. A *minimax* search is an exhaustive depth-first search of a game-tree that returns a value called a *game theoretic* value for the root board position. The *game theoretic* value represents the outcome if both players play perfectly [45]. A minimax search has two phases called the *maximizing phase* and the *minimizing phase*, respectively. The maximizing phase occurs at all board positions where the first player has the turn and the minimizing phase occurs at all board positions where the second player has the turn. A search on a board position in the maximizing phase returns the largest game theoretic value assigned to the successors. A search on a board position in the minimizing phase returns the smallest value assigned to the successors. A minimax search backtracks values assigned to the terminal board positions. Terminal positions where the first player wins are assigned the largest game theoretic value and terminal positions where the second player wins are assigned the smallest value. A neutral value is assigned to terminal positions of games ending in a draw. In Figure 3.2.2, terminal board positions are assigned (1) if the first player is the winner, (-1) if the second player is the winner and (0) if the game ended in a draw. In addition, boxes represent the maximizing phase and circles represent the minimizing phase in a minimax search. As the search backtracks from terminal positions, board positions at the maximizing phase are assigned the largest value assigned to their successors and board positions at the minimizing phase are assigned the smallest value assigned to their successors. In this example, the search returns the value (1), which means that the first player has a winning strategy.

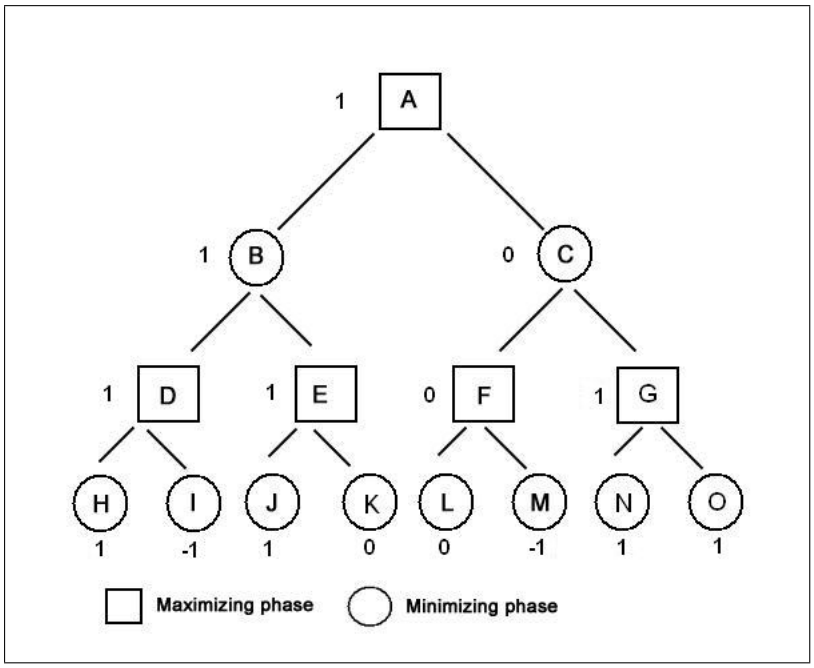

Figure 3.2.2: An example of a minimax search.

3.3 The Alpha-Beta Pruning Algorithm

The alpha-beta pruning algorithm is an extension for minimax searches that involves two values *alpha* and *beta*, which bound the game theoretic value at each board position [12][6]. The alpha value is a lower bound, which the search maximizes and the beta value is an upper bound, which the search minimizes. In Figure 3.3.3, boxes represent the maximizing phase and circles represent the minimizing phase in a minimax search. Pruning is triggered by two conditions known as alpha and beta cut-off conditions. A beta cut-off condition occurs while searching node E. Since the search is maximizing at node E and the value at node E is greater or equal to the beta value of node B, and because the search is minimizing at node B; the search at node E will never return a value smaller than the value at node B. Therefore, the remainder of the tree rooted at node E is eliminated from the search. Similarly, an alpha cut-off condition occurs while searching node C. Since

the search is minimizing at node C and the value at node C is smaller or equal to the alpha value of node A, and because the search is maximizing at node A; the search at node C will never return a value larger than the value at node A. Therefore, the remainder of the tree rooted at node C is eliminated from the search.

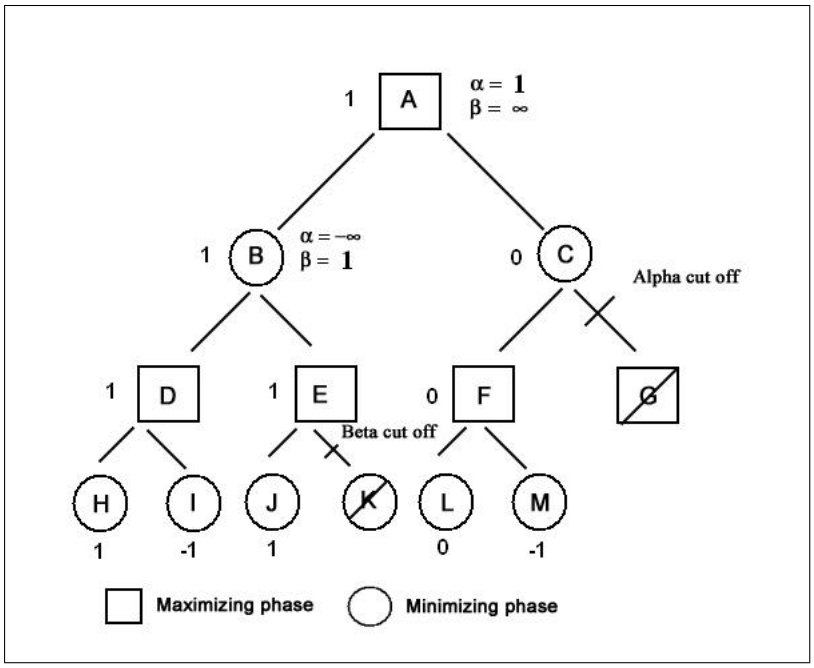

Figure 3.3.3: An example of alpha-beta pruning algorithm.

3.4 The Transposition Table Pruning Algorithm

Alpha-beta pruning is not the only form of pruning available. Many game-trees have repeating board position instances. That is, some board positions can be reached from the root via more than one path [13]. A *game-graph* is the reduction of a game-tree such that every board position instance is unique. The multiple paths that can lead from the root to a particular board position is called a set of *transpositions* [28]. The search can store board positions and their respective game theoretic values in a cache called a *transposition table* and reuse those values to prune the search.

A transposition table is a mapping $T : \mathcal{B} \rightarrow \mathcal{V}$, from a set of board positions $\mathcal{B}$ to a set of game theoretic values $\mathcal{V}$ [28]. A transposition table can be used to prune a minimax search. For example in Figure 3.4.4, a minimax search attempts to solve the Hex board position b_0. Game-theoretic values returned by this search are (-1) if a board position has a Black winning strategy and (+1) if a board position has a White winning strategy. In the left of Figure 3.4.4, the search traverses to board position b_i and proceeds to search the subtree rooted at b_i. After searching this subtree, the search returns (-1) for b_i and the transposition table T is updated with the mapping $(b_i, -1)$. In the right of Figure 3.4.4, the search traverses a different path and arrives again at board position b_i. Since the transposition table T has a mapping for b_i, the search uses the value $T(b_i)$ instead of searching the subtree rooted at b_i. The transposition table is used to prune the search, because the search can immediately backtrack from position b_i with value $T(b_i)$.

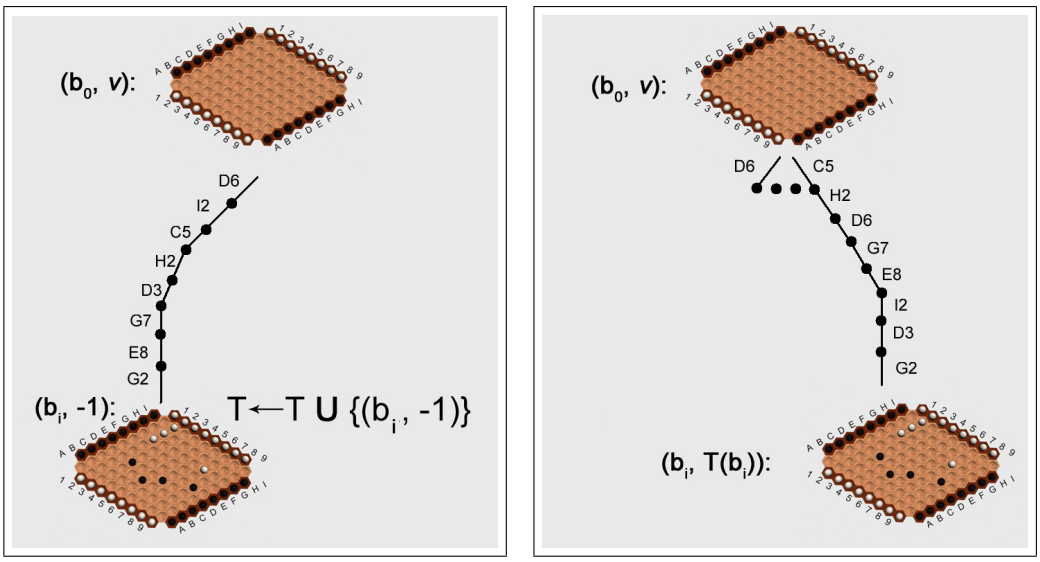

Figure 3.4.4: Left: After searching the subtree under board positions b_i, the minimax search finds that Black has a winning strategy for b_i. Since the game theoretic value for a Black winning strategy is (-1), the search adds the mapping $(b_i, -1)$ to the transposition table T. Right: Following a different path, the same minimax search again arrives at board position b_i. The game theoretic value for b_i can be returned by the transposition table, in place of a search in the subtree of b_i.

3.5 Upper Confidence Tree Search

An approach that can be used to solve combinatorial games is the *UCT* (Upper Confidence Tree) search [29]. The *UCT* search extends on a Monte-Carlo policy called the *UCB1* (Upper Confidence Bounds) algorithm reported in [5] for solving the multi-armed bandit problem. A *multi-armed bandit problem*, is a machine learning problem described with the analogy of a slot-machine that has more than one arm. A test of each arm returns a random reward according to some probability distribution. The objective of the the *UCB1* algorithm is to maximize the total expected reward through repeated tests.

The problem of estimating the game theoretic value for a given board position in a game-tree can be posed as a multi-armed bandit problem. Given a board position, a *test* of a particular successor returns the outcome of a random game, whose root position is that successor. A test of each successor is like a test of an arm on the slot-machine in the multi-armed bandit problem. In this case, the objective is to maximize the total expected outcome through repeated tests of the successors. An estimate of the game theoretic value can be found using the *UCB1* algorithm in [5].

In Figure 3.5.5, board positions b has a set of $K+1$ successors $\{s_j\}_{j=0}^{K}$. The problem for the *UCB1* algorithm is to find an estimate v of the game theoretic value for board position b by maximizing the total expected outcomes through repeated tests of its successors[1]. Let $\bar{x}_j$ be the real average outcome of n_j tests of successor s_j and let $n = \sum_{j=0}^{K} n_j$.

[1] We are assuming that the player who has the turn at b aims to maximize the value of b. A similar argument can be defined for the minimizing player.

The *UCB1* algorithm is as follows:

1. Test each successor once (this step initializes each $\bar{x}_j$ to the outcome of test j and assigns $n_j \leftarrow 1$),
2. Loop:
   (a) Select an index j with the largest $\left[\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}}\right]$ value and return $\bar{x}_j$ [2].

The return value is the estimate v.

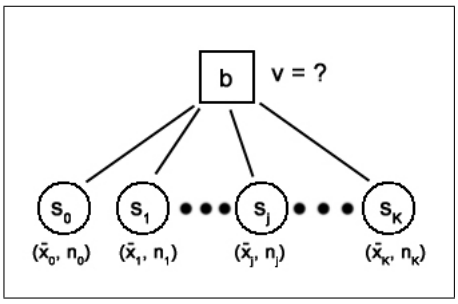

Figure 3.5.5: An example of the UCB1 algorithm in the context of finding an estimate v of the game theoretic value for board position b.

The *UCT* search is an extension of the *UCB1* algorithm to a minimax search [29]. In a *UCT* search, the *UCB1* algorithm is used in a player policy to generate a move at each internal board position. For clarity, this review will discuss the application of the *UCB1* algorithm using two player policies, *UCB1PlayerMax* and *UCB1PlayerMin*. The *UCB1PlayerMax* player involves a maximizing *UCB1* algorithm and generates moves for positions where the maximizing player has the turn. The *UCB1PlayerMin* player involves a minimizing *UCB1* algorithm and generates moves for positions where the minimizing player has the turn.

[2] Initially, n is small and the value of n_j has significant influence over the value of the term: $\sqrt{\frac{2 \ln n}{n_j}}$ ensuring that each successor is tested at least once. However, as n grows large the term $\sqrt{\frac{2 \ln n}{n_j}}$ tends towards zero and the choice of index j depends more on the average outcome $\bar{x}_j$.

Given board position b in the previous example the *UCB1PlayerMax* player is as follows:

1. Test each successor once (updates the value of $\bar{x}_j$, n_j and n),
2. Loop:
   (a) Select an index j with the largest $\left[\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}}\right]$ value and return $(s_j, \bar{x}_j, n)$.

Similarly, given board position b in the previous example the *UCB1PlayerMin* player is as follows:

1. Test each successor once (updates the value of $\bar{x}_j$, n_j and n),
2. Loop:
   (a) Select an index j with the smallest $\left[\bar{x}_j - \sqrt{\frac{2 \ln n}{n_j}}\right]$ value and return $(s_j, \bar{x}_j, n)$.

A *UCT* search traverses a game-tree through a series of games played out between *UCB1PlayerMin* and *UCB1PlayerMax*. In addition, the *UCT* search uses a table and for each position b in its traversal it stores an entry (b, v, n), where v is a value for b and n is the total number of tests that a *UCB1* player has performed on the successors of b. The *UCB1* players make use of this table to decide moves. The *UCT* search treats each board position as an independent multi-armed bandit problem.

In the left of Figure 3.5.6, a *UCT* search begins at position A where the maximizing player has the turn and applies the *UCB1PlayerMax* player. The *UCB1PlayerMax* player tests both successors of A and chooses a move that maximizes the policy value for *UCB1PlayerMax*, which happens to lead to successor B. The search adds an entry for position A to its table. At position B, the search applies the *UCB1PlayerMin* player. The *UCB1PlayerMin* player tests both successors of B and chooses a move that minimizes the policy value for *UCB1PlayerMin*, which

happens to lead to successor C. The search adds an entry for position B to its table. Both players continue this process until the game terminates. In the right of Figure 3.5.6, the search backtracks from a terminal position and updates the average outcome for each position entry in its table, starting with the outcome of the terminal position.

The *UCT* search will apply *UCB1PlayerMin* and *UCB1PlayerMax* over many games starting at position A until a stopping criteria has been satisfied. For example, if the *UCT* search is used in an artificial game player then the search may have a designated period before it is terminated. When the search in this example terminates, the value for position A is given by the highest average outcome recorded for the successors of A.

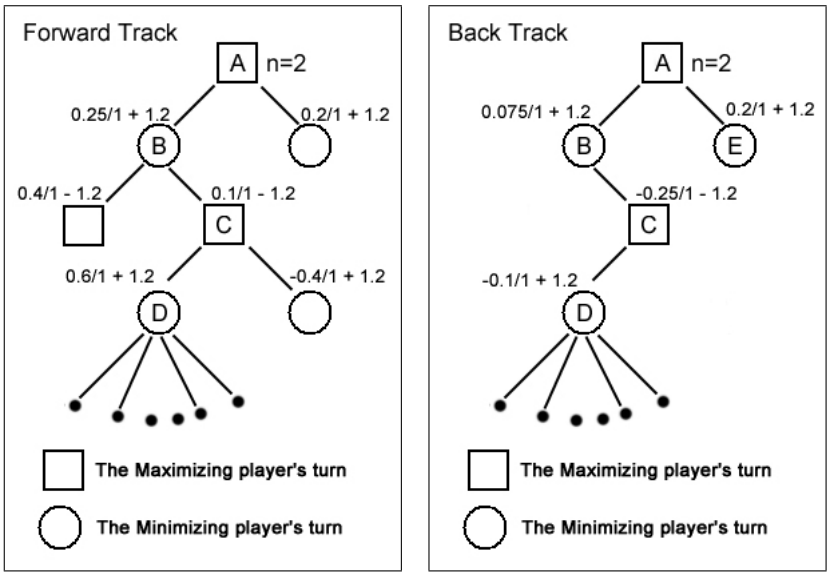

Figure 3.5.6: Left: A game played between *UCB1PlayerMin* (minimizing player) and *UCB1PlayerMax* (maximizing player) showing the values that the players used to select successors on the forward track. Right: The game reaches a terminal position and the search backtracks. As the search backtracks, the outcome of this game is used to update the average outcomes at positions D,C and B.

The advantage of the *UCT* search is that it can be stopped at any time and will probably return a good value. However, there is no guarantee that the UCT search will not miss some vital or unique moves as it is not a complete search. In contrast, stopping a minimax search at any time will probably mean that most of the root successors have not been searched and the return value is not good. The advantage of a minimax search is that it only uses polynomial space, whereas the space used by a *UCT* search could grow exponentially. But for many problems, a *UCT* search can use space economically, because the *UCB1* algorithm provides good balance between exploration and exploitation.

In the particular case of Hex, the search techniques applied in artificial Hex players such as *Hexy* in [3, 4] and *Six* in [36] are particularly effective. These players apply standard alpha-beta searches and utilize evaluation techniques through the deduction of sub-games and a particular board evaluation function, called a *resistor network* function. An early objective in this work was to extend on such artificial Hex players with Hex solving techniques based on sub-game deductive searches. This objective lead to research of Hex solving techniques that was later found to be incompatible with the random game-tree traversal approach of the *UCT* search, and so the *UCT* search was not explored any further in this work.

3.6 Chapter Discussion

The depth-first traversal of minimax searches demonstrate that combinatorial games can be solved in polynomial space. With the addition of pruning algorithms such as alpha-beta pruning, minimax search times can be reduced. Pruning algorithms present the opportunity to further reduce search times through good move orderings. Transposition tables provide another approach to pruning game-tree searches and demonstrate how game-trees can be reduced to a more compact game-graph. Finally, the problems of game-tree searching can be treated as a stochastic

search problem and solved using the *UCT* search. A *UCT* search can return a good value for positions in games where the evaluation of positions is difficult.

Chapter 4

Sub-game Deduction for Hex

Some game-tree searches such as the minimax search are *top-down*, because they begin at a root position and traverse down towards terminal positions. Alternatively, other game-tree searches are *bottom-up*, because they begin from terminal positions and traverses up towards a root position. The *H-Search* algorithm is an example of a bottom-up game-tree search for Hex [3, 4]. This search starts with a set of sub-games that have adjacent targets and empty carriers, called *elementary* sub-games. Elementary sub-games can represent winning chains on the terminal positions. The H-Search algorithm applies a set of sub-game deduction rules in procedures that can generate new sub-games from the elementary sub-games.

Section 4.1 gives definitions to sub-games and the sub-game deduction rules used in H-Search. The H-Search algorithm in Section 4.2 performs a search on sub-game sets to deduce sub-games that represent the games on board positions towards the end of games. Section 4.3 presents a top-down search that uses existing sub-games to deduce new sub-games.

4.1 Sub-games and Deduction Rules

For the analysis of board positions, Anshelevich introduces the concept of a *sub-game* [3, 4]. In a sub-game, the players are called *Cut* and *Connect*. Both *Cut* and *Connect* play on a subset of the empty cells between two disjoint targets. In a sub-game, a *target* is either an empty cell or one of *Connect's* groups. The player's roles are not symmetric, as *Connect* moves to form a chain of stones connecting the two

targets, while *Cut* moves to prevent *Connect* from forming any such chain of stones. Sub-games have been generalized here so as to include a subset of *Connect's* stones, this was also done in [39].

Definition 4.1.1 (sub-game)**.** *A sub-game is a four-tuple* (x, S, C, y) *where* x *and* y *are targets. The set* S*, is a set of cells with Connect's stones and the set* C *is a set of empty cells. Finally,* x*,* y*,* S *and* C *are all disjoint.*

The set S is called the *support* and the set C is called the *carrier*. Anshelevich's sub-game definition is equivalent to the one given here when S is the empty set [3, 4]. A sub-game is a *virtual connection* if *Connect* can win this sub-game against a perfect *Cut* player. A virtual connection is *weak* if *Connect* must play first to win the sub-game. In addition, a virtual connection is *strong* if *Connect* can play second and still win the sub-game. Yang et al. define a *threat pattern* as a virtual connection, where the targets are two opposite sides of the board [48]. In Figure 4.1.1, lightly shaded cells represent the carrier of a threat pattern and the Black stones mid-board form the support set.

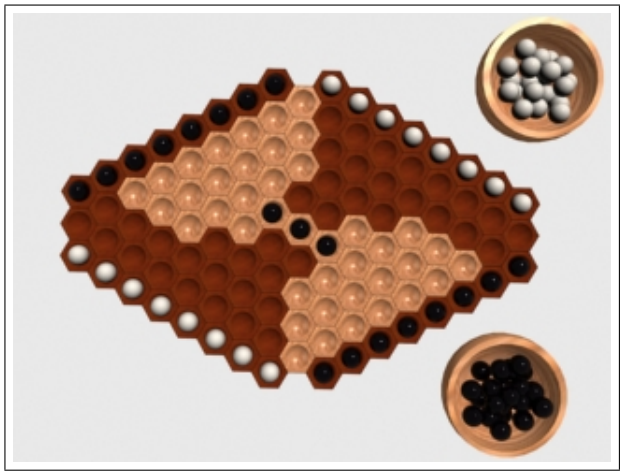

Figure 4.1.1: An example of a threat pattern for player Black.

Given a set of sub-games, new sub-games can be found by applying deduction rules. Two common deduction rules are the AND and the OR deduction rule. Proofs were provided by Anshelevich for the AND and the OR deduction rule theorems in [3, 4] for the case where $S = \emptyset$. The AND and OR deduction rules generalize trivially to the case where $S \neq \emptyset$.

Theorem 4.1.2 (AND Deduction Rule)**.** *Let sub-games* (x, S_a, C_a, u) *and* (u, S_b, C_b, y) *be strong with the common target* u*. Target* u *is the particular feature both sub-games must have in common and all other features must be disjoint, except* S_a *and* S_b*.*

- *If* u *is not an empty cell then the sub-game* $(x, (S_a \cup u \cup S_b), (C_a \cup C_b), y)$ *is a strong sub-game.*
- *If* u *is an empty cell then the sub-game* $(x, (S_a \cup S_b), (C_a \cup u \cup C_b), y)$ *is a weak sub-game.*

Theorem 4.1.3 (OR Deduction Rule)**.** *Let* $\{(x, S_i, C_i, y)\}_{i=1}^{n}$ *be a set of* n *weak sub-games with common targets* x *and* y*. If* $\bigcap_{i=1}^{n} C_i = \emptyset$*, then the sub-game* $(x, \bigcup_{i=1}^{n} S_i, \bigcup_{i=1}^{n} C_i, y)$ *is a strong sub-game.*

With respect to the OR deduction rule, let $M = \bigcap_{i=1}^{n} C_i$. If $M \neq \emptyset$ then the OR rule has failed to deduce a strong sub-game. Player *Cut* must move on a cell in M, otherwise one of the weak sub-games represent a winning strategy for *Connect*. Hayward et al. call the set M, the *must-play region* [27].

4.2 The H-Search Algorithm

The *H-Search* algorithm is a hierarchical search algorithm, described by Anshelevich in [4], that recursively applies the AND and OR deduction rules to generate sub-games. To avoid any confusion caused by the unfortunate verb used in the name

of this algorithm, the H-Search algorithm in execution will be called a *H-Search execution*. The input and the output of the search are a set of weak and a set of strong sub-games. Strong sub-games of the form $(x, \emptyset, \emptyset, y)$, where one target is a stone group or both targets are empty cells, are called *elementary* sub-games. If the aim of the search is to generate sub-games for a particular board position, then the input sub-games are board features, which includes the complete set of elementary sub-games on that position.

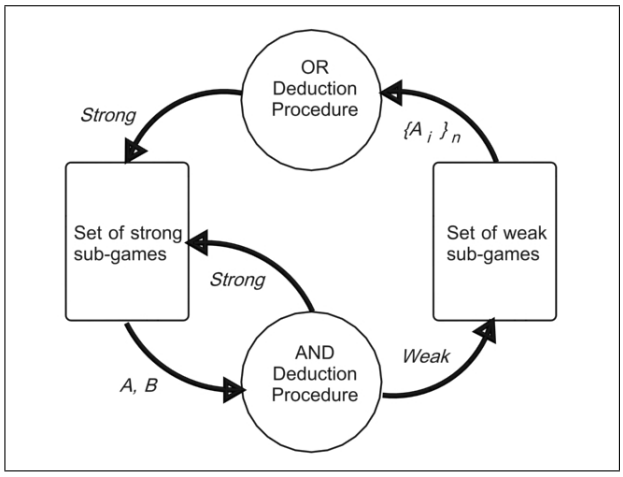

Figure 4.2.2: The H-Search algorithm applies the AND rule to strong sub-games A and B. The OR deduction rule is triggered to operate on a set of weak sub-games $\{A_i\}_n$, if the AND rule deduces a weak sub-game that belongs in this set.

Figure 4.2.2 gives an overview of a H-Search execution. The search begins by applying the AND rule to each pair of strong sub-games A and B that satisfy the conditions for AND deduction. The search updates the strong sub-game set, if it applies the AND rule and derives a strong sub-game. The search updates the weak sub-game set and triggers an application of the OR rule, if it applies the AND rule and derives a weak sub-game. In the example, the search applies the AND rule and adds to the set $\{A_i\}_{i=1}^{n}$ a weak sub-game, which triggers OR deduction on that particular set. The search updates the strong sub-game set, if OR deduction yields

a strong sub-game. After every update of the strong sub-game set or after every application of the OR rule, the search applies the AND rule. The search terminates when sub-games satisfying AND and OR deduction have been exhausted. The result is a set of weak and a set of strong sub-games. The winning strategy for each sub-game can be derived by retracing the AND and OR deductions between sub-games in the two sets.

The H-Search algorithm is unable to solve opening positions for boards larger than the 5x5 board and is most effective for solving board positions towards the end of games. In addition, the sub-game sets can be used to improve the evaluation of arbitrary board positions. Anshelevich described an application for the H-Search algorithm in an artificial Hex player, where sub-games were used in pruning the player's search and in providing more accurate evaluations of board positions [3, 4]. The H-Search algorithm was also used by Hayward et al. to prune the search space for their Hex-solving program, called *Solver* [27].

4.3 The Must-play Region Deduction Rule

Given that H-Search executions seldom solve Hex boards completely, effective measures can be taken to make such searches more complete. Rasmussen and Maire described in [38] a deduction rule that can extend the search results returned by the H-Search algorithm. The *must-play deduction rule* is a specialized game-tree search that exploits must-play regions to rapidly deduce sub-games. Given weak and strong sub-game sets returned by the H-Search algorithm, the must-play deduction rule enumerates the non-empty weak sub-game sets of the form $\{(x, S_i, C_i, y)\}_{i=1}^{n}$, where the OR deduction rule failed to deduce strong sub-games. For each of these weak sub-game sets, the must-play deduction rule begins a game-tree search that attempts to derive a strong sub-game of the form (x, S, C, y). To show how this game-tree search proceeds, consider the problem shown in Figure 4.3.3, where a

H-Search execution returns several weak sub-games with targets x and y, but fails to deduce a strong sub-game that features these targets. The game-tree search for the must-play deduction rule aims to return a strong sub-game involving group x and group y as targets.

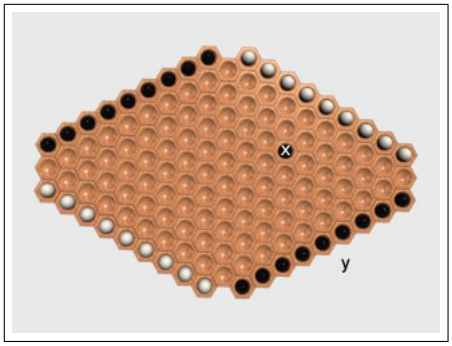

Figure 4.3.3: A H-Search execution deduces several weak sub-games with targets x and y, but is not able to deduce a strong sub-game (x, S, C, y).

The two players in this game-tree search are *Cut* and *Connect.* The game-tree search for the must-play deduction rule always begins with a *Cut* move. In this example, *Cut* is the White player and *Connect* is the Black player. Given that the weak sub-games with targets x and y form the set $\{(x, S_i, C_i, y)\}_{i=1}^{n}$, one can assume *Cut* moves on the must-play region $M = \bigcap_{i=1}^{n} C_i$, otherwise *Connect* can immediately secure a connection. Left of Figure 4.3.4 shows the must-play region with lightly shaded cells for *Cut's* move set. Let us assume that the search proceeded to the position with *Cut* move shown in the right of Figure 4.3.4.

In this game-tree search, *Connect's* objective is very different from *Cut's*, because *Connect's* objective is to prove a strong sub-game carrier that connects targets x to y. *Connect* tries to prove this carrier by assuming that certain sets of cells belong to this carrier and by applying the AND and OR rule during the search to test these assumptions. A set of empty cells that *Connect* assumes belongs to the

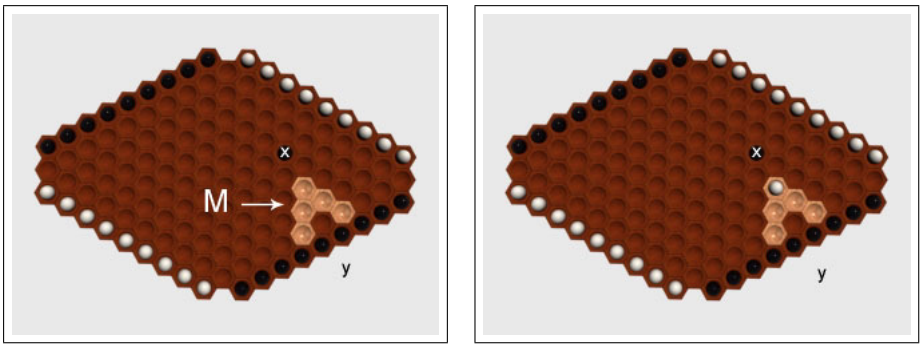

Figure 4.3.4: Left:The must-play region M is the intersection of weak sub-games with targets x and y, which have been deduced using the H-Search algorithm. Right: *Cut* must move in the must-play region, otherwise *Connect* can immediately secure a connection.

said carrier is called a *provisional* set. In the search, $\mathcal{C}$ is *Connect's* provisional set. At the beginning of the search, $\mathcal{C}$ is the empty set. Another feature of the search is that *Connect* stones are partitioned into sets, the set X of stones that are strongly connected to the target x and the set Y of stones that are strongly connected to the target y. These sets are placed in temporal order of when the stone was played. In this case, the sides are treated as single stones. At the beginning of the search, $X = \{x\}$ and $Y = \{y\}$.

Given the *Cut* move and position shown in the right of Figure 4.3.4, *Connect* must move so as to reestablish weak connections between targets x and y. In Figure 4.3.5, *Connect* uses $t_x \in X$ and $t_y \in Y$, where t_x and t_y were the last stones added to each set, as reference targets to enumerate a set of strong sub-games $\{G_j\}_{j=1}^{m}$ such that for the carrier in each sub-game, $C_j \cap \mathcal{C} = \emptyset$, one target in each sub-game is either t_x or t_y and the other target is an empty cell t_j not in $\mathcal{C}$. In Figure 4.3.5, the strong sub-games can be distinguished by the lightly shaded cells on each board. An additional constraint is that, if sub-game G_j, whose carrier is C_j, has target t_x then there must be at least one weak sub-game (t_j, S_k, C_k, z) where $z \in Y$ and $C_k \cap (C_j \cup \mathcal{C}) = \emptyset$ and if sub-game G_j has target t_y then there must be at

least one weak sub-game (t_j, S_k, C_k, z) where $z \in X$ and $C_k \cap (C_j \cup \mathcal{C}) = \emptyset$. If this constraint cannot be satisfied then the scope of the search is exhausted and the search must backtrack. If there is a strong sub-game where one target is in X, the other target is in Y and the carrier of this strong sub-game is disjoint from $\mathcal{C}$, then *Connect* wins and the search backtracks. Both search backtrack cases will be covered in more detail shortly.

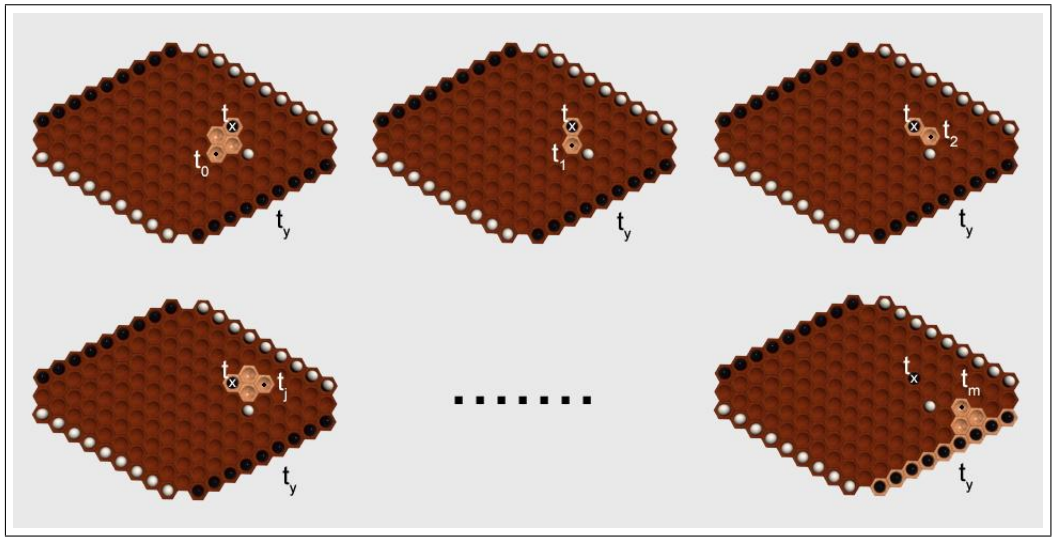

Figure 4.3.5: Given $t_x \in X$ and $t_y \in Y$, *Connect* enumerates strong sub-games, where one target is either t_x or t_y and the other target is some empty cell t_j (highlighted with a small black dot).

Player *Connect's* moves are limited to the set of cells $\{t_0, t_1, \ldots, t_j, \ldots, t_m\}$. *Connect* moves on a cell t_j, which is a target in the strong sub-game shown on the left board of Figure 4.3.6. Let that strong sub-game be $(t_x, \emptyset, C_0, t_j)$. In connection with *Connect's* move on cell t_j, the search updates the provisional set such that $\mathcal{C} \leftarrow \mathcal{C} \cup C_0$ and updates X such that $X \leftarrow \{t_j\}$. An alternate stone set update would have occurred if t_j and a stone in Y were the targets of a strong sub-game, and in that case t_j would have been added to Y. The problem for *Connect* on this branch of the search is reduced to the problem of finding a strong sub-game with

targets t_j and t_y and with carrier disjoint from $\mathcal{C}$. The search applies the OR rule to the set of weak sub-games, where one target is t_j, the other target is t_y and the carriers are disjoint from $\mathcal{C}$. If this application of the OR rules yields a strong sub-game then *Connect* can win and the search backtracks. Otherwise, the problem resembles the initial problem (see right of Figure 4.3.6) and is solved by repeating this procedure.

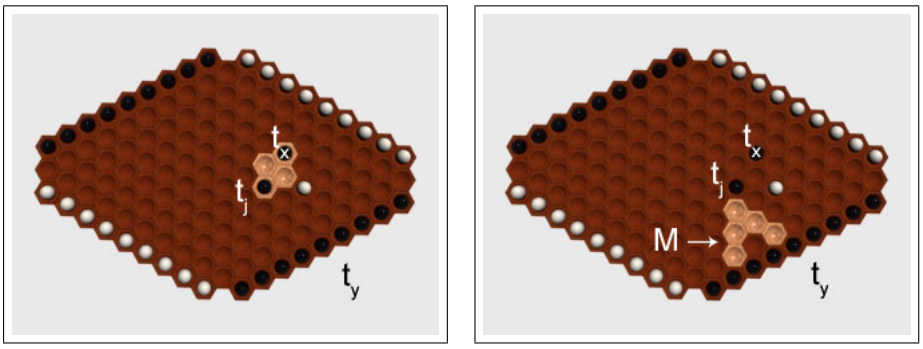

Figure 4.3.6: Left: The strong sub-game associated with Connect's move on cell t_j and whose carrier is added to $\mathcal{C}$. Right: If an application of the OR rule does not yield a strong sub-game with targets t_j and t_y then there is a must-play region M that defines a move set for *Cut*.

At some point, the search will arrive at *terminal* positions where strong sub-games can be used to determine the winning player. Such terminal positions put the search into one of two modes, *Connect Win* mode or *Cut Win* mode. If the search is in *Connect Win* mode, then the last terminal position in the search was one where a strong sub-game connected a stone in X to a stone in Y, such that its carrier was disjoint from $\mathcal{C}$. If the search is in *Cut Win* mode, then the last terminal position in the search was one where no such strong sub-game could be found and *Connect's* move set is empty. Figure 4.3.7 shows two terminal positions found on different search paths where the search has switched to *Connect Win* mode. The Black stones have been relabelled so that $X = \{0, 1, 2, 3, 4\}$ and $Y = \{a, b\}$ in the left, and $X = \{0, 1, 2, 3\}$ and $Y = \{a, b, c\}$ in the right of Figure 4.3.7. In addition,

lightly shaded cells highlight the strong sub-games that have triggered *Connect Win* mode.

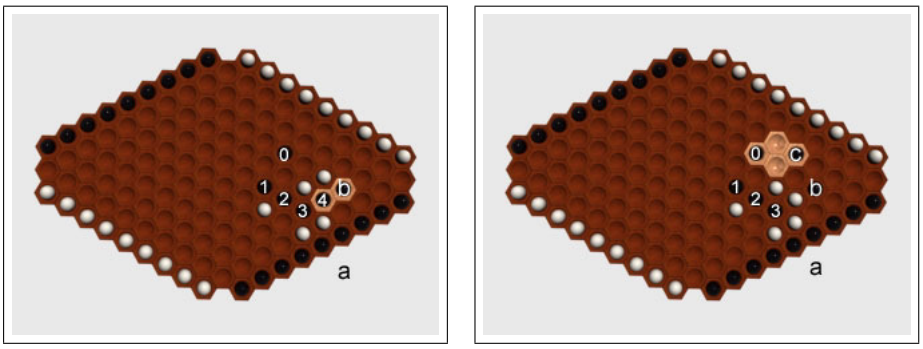

Figure 4.3.7: Left: On this search path, the X and Y stone sets are virtually connected via a strong sub-game with targets 4 and b. Right: On this search path, the X and Y stone sets are virtually connected via a strong sub-game with targets 0 and c.

In Figure 4.3.8, board position A is the parent of the position in the left of Figure 4.3.7 and board position B is the parent of the position in the right of Figure 4.3.7. As the search backtracks from the board in the left of Figure 4.3.7, *Connect's* move denoted by stone 4 in X is undone, 4 is now an empty cell and $X \leftarrow X - \{4\}$ is the X update. At board position A in figure 4.3.8, the search has two strong sub-games, the strong sub-game $(3, \emptyset, K, 4)$, where K was the strong sub-game carrier that had been added to the provisional set $\mathcal{C}$ on the forward track and the sub-game $(4, \emptyset, \emptyset, b)$ from its child. The carrier K will generally not be the empty set, as is the case in this example. Because all carriers were disjoint to the provisional set on the forward track, the search can apply the AND rule, where the common target is the empty cell 4, to deduce a weak sub-game $(3, \emptyset, K \cup \{4\}, b)$ from the strong sub-games $(4, \emptyset, \emptyset, b)$ and $(3, \emptyset, K, 4)$ at board position A. The final step is to update the provisional set so that $\mathcal{C} \leftarrow \mathcal{C} - K$. The same process occurs on a different search that backtracks from the positions in the right of Figure 4.3.7 to board position B, in Figure 4.3.8.

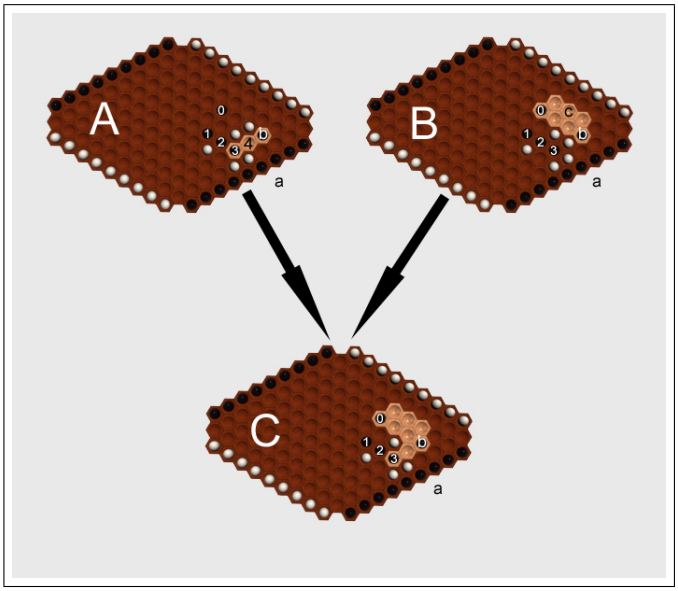

Figure 4.3.8: A form of OR deduction is applied to the weak sub-games on boards A and B to form the strong multi-target sub-game on board C.

When the search is in *Connect Win* mode, the search applies the OR rule at board positions where *Cut's* has the turn. At these board positions, the OR rule is applied to the weak sub-games found at the successors and to weak sub-games from the initial H-Search execution. In this example, the search at position C applies the OR rule to the weak sub-games found first at positions A and then at position B to deduce a strong sub-game, however, this application of the OR rule is not standard. In this case, the weak sub-games have one target in X and the other in Y. This application of the OR rule is valid because stones exclusive to each set are virtually connected via strong sub-games, whose carriers are disjoint from the weak carriers because those carriers are subsets of the provisional set $\mathcal{C}$. The sub-game highlighted with lightly shaded cells on board position C is the non-standard strong sub-game that results from this OR deduction, connects X to Y and is disjoint from the provisional set. If the carrier of this non-standard sub-game is O, then this sub-game can be viewed as part of a strong sub-game $(x, X \cup Y, \mathcal{C} \cup O, y)$. The search

backtracks from C as it did from positions in figure 4.3.7 and the process to here is repeated. If this OR deduction fails then the search switches to *Cut Win* mode and backtracks.

In *Cut Win* mode, the search backtracks until it reaches a node where *Connect* has the turn and *Connect's* move set is not empty. At such nodes the search forward tracks by exploring *Connect's* move set. If the search terminates in *Cut Win* mode then the search fails to deduces a strong sub-game carrier that connects x to y. If the search terminates in *Connect Win* mode then the final application of the OR rule generates a strong sub-game (x, S, C, y).

4.4 Chapter Discussion

Sub-games provide an efficient way to represent winning strategies in Hex. In addition, if threat patterns can be deduced using the sub-game deduction rules then the deduction of must-play regions will reduce move sets in game-tree searches. In a game-tree search, the H-Search algorithm could provide threat patterns toward the end of games so that the search can be pruned at a shallow depth. Finally, sub-games or threat patterns could be stored and reused for pruning elsewhere in a game-tree search.

Part I

Hex Solving Algorithms

Chapter 5

A Hex Solving Search Algorithm

The problem of solving the game of Hex for arbitrary board sizes is PSPACE-complete, which implies the problem is NP-Hard. The search space for solving Hex compounds with increasing board sizes. Figure 5.0.1, shows a plot of upper bounds for Hex game-trees over a range of small boards, according to a Hex game-tree size formula presented in [23]. Although this plot shows that the search space for solving Hex is very large, there are many effective problem reduction methods. The *Solver* program developed by Hayward et al. and presented in [27], demonstrates many good problem reduction methods and is the first program strong enough to completely solve the 7x7 Hex board for all opening moves. The *Solver* program builds on a particular search algorithm that will be covered in this chapter. Over the following chapters a succession of Hex solving algorithms will be described such that each algorithm builds upon its predecessor. Each Hex solving algorithm is performance tested and the results from these tests are compared. The aim of this investigation is to develop Hex solving algorithms that approach practical search times for board sizes larger than 7x7. In addition, an objective in this investigation is to arrive at a Hex solving algorithm that can independently confirm the results reported by Hayward et al. in applying their *Solver* algorithm to solve the 7x7 Hex board.

Section 5.1 provides a high level overview of a game-tree search algorithm called the *Pattern* search algorithm, which gives the base search for the *Solver* program [46]. The *Pattern Search* algorithm is presented in pseudo code and details of a

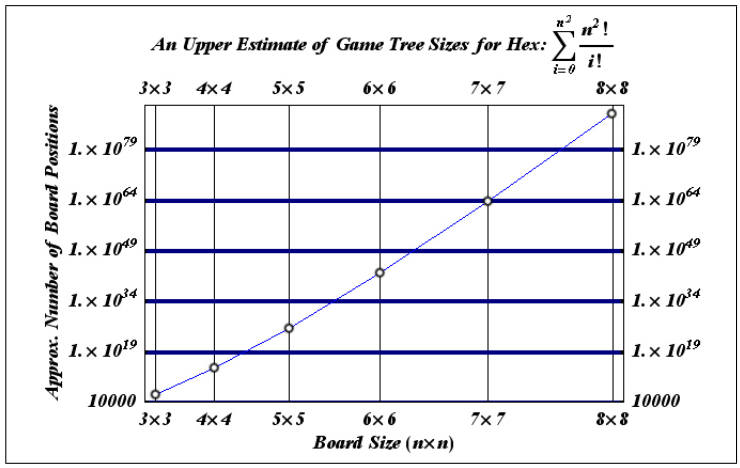

Figure 5.0.1: The approximate sizes of Hex game-trees for 3×3 to 8×8 Hex according to Even and Tarjan in [23].

performance test for an implementation of this algorithm are also provided. Section 5.2 reports on a cell utility measure called *Connection Utility* used in a heuristic for a new move ordering algorithm that successfully increases pruning in Pattern searches. The pseudo code for the *Pattern Search* algorithm is extended with this move generating algorithm and the resulting algorithm, called *Hex Solver 1*, is presented. Finally, the performance of *Hex Solver 1* is compared against the results for the *Pattern Search* algorithm. Section 5.3 gives an overview of the chapter and looks at some of the consequences of the move generating algorithm presented herein.

5.1 The Pattern Search Algorithm

Virtual connection deduction has long been recognized as an effective tool for solving Hex. In [48], Yang et al. showed that threat pattern deduction can be used to solve Hex for small board sizes. Yang applied an informal threat pattern deduction approach that relies largely on human intuition to solve particular opening positions on the 7x7 Hex board. This approach is called the *pattern decomposition* method.

Remarkably, Yang was able to apply the pattern decomposition method to solve some 7x7 positions without the aid of a computer. Yang's success has enticed the development of game-tree search algorithms that can deduce threat patterns, and to various degrees emulate the pattern decomposition method.

The *Pattern Search* algorithm, by van Rijswijck, is a game-tree search which deduces threat patterns [46]. Hayward et al. extend on the *Pattern Search* algorithm in their *Solver* algorithm [27]. A Pattern search is a depth-first traversal of the game-tree that deduces threat patterns as it backtracks from terminal board positions. The search switches between two modes, *Black mode* and *White mode*. In Black mode, the search tries to prove threat patterns for player Black and in White mode the search tries to prove threat patterns for player White.

Figure 5.1.2 gives an example of a search on a 3x3 Hex board in White mode. Player White is *Connect* and player Black is *Cut*. The diagram displays the carrier and the support on lightly coloured cells. The numbers on the stones indicate the order of the player's moves. On the left branch, the search visits terminal board position D where *Connect* is the winner. As the search backtracks from this terminal board position it removes *Connect's* move. At board position B, *Connect* has a weak threat pattern. The search backtracks once more and removes *Cut's* move. The search does a similar traversal of the right branch and deduces a weak threat pattern at board position C. At board position A, the search applies the OR deduction rule on the weak threat patterns found at B and C to deduce a strong threat pattern. On deducing that strong threat pattern, the search can backtrack and deal with board position A as it did with terminal board positions D and E.

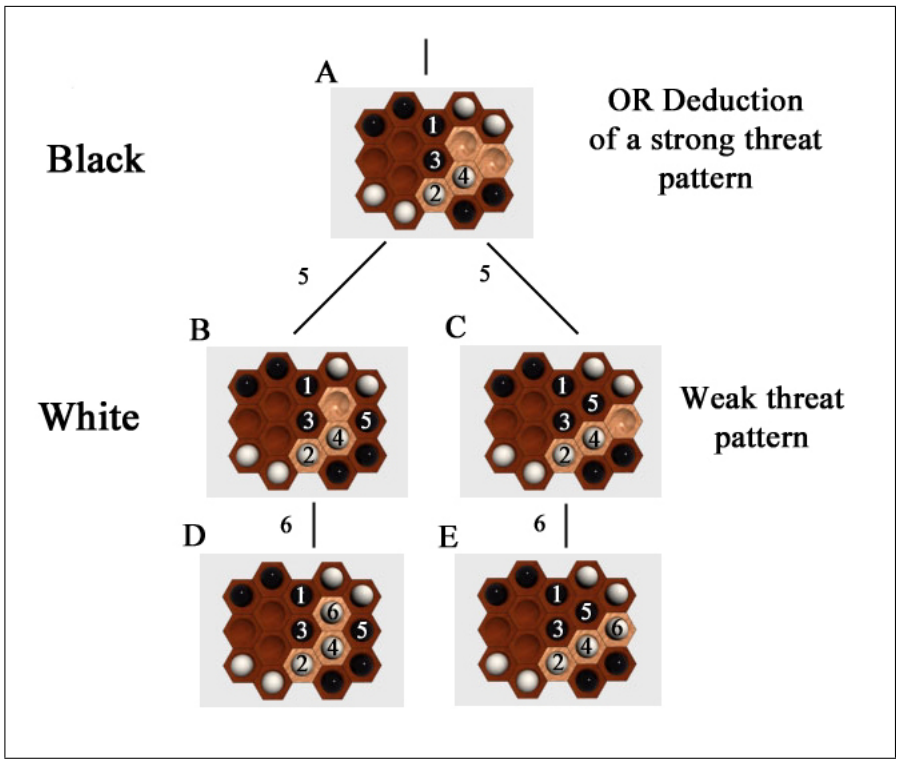

Figure 5.1.2: The process in a Pattern search for constructing threat patterns.

A cut-off condition and a switch of mode colour can occur if Black has a winning move elsewhere in this search. Figure 5.1.3 follows on from the previous Pattern search example where the search is in White mode. The search backtracks from board position A, where it removes *Connect's* move. Board position X has a weak threat pattern. The search backtracks to board position Z and removes Black's move. At Z the search tries another move, which happens to be a winning move for player Black. The result is a strong threat pattern for Black at Y, which indicates a search cut-off condition and a switch of mode colour. The search abandons White's threat pattern at Z and switches to Black mode. The search deals with Y as it did with terminal board positions D and E. The switch of mode colour means player Black is now *Connect* and player White is now *Cut*.

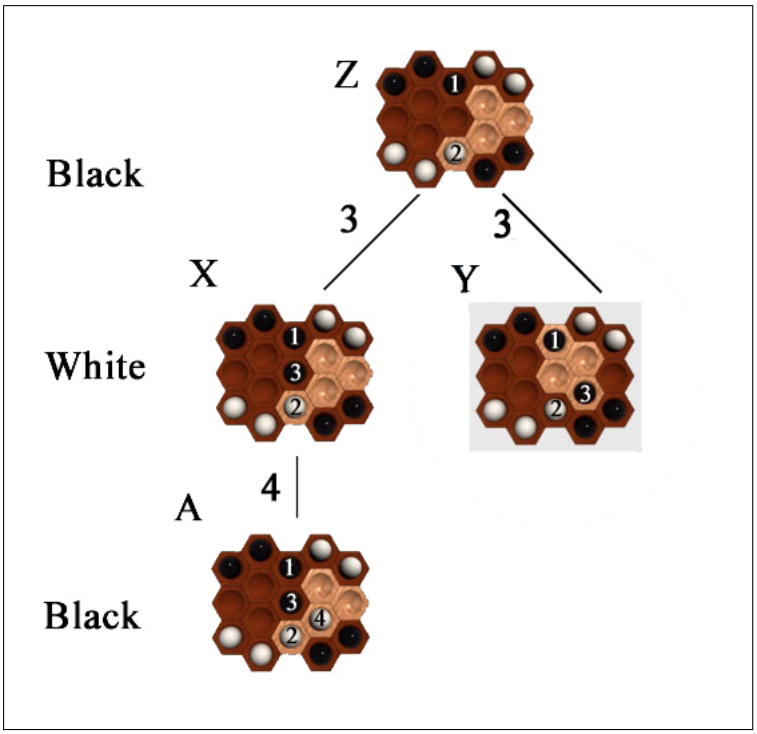

Figure 5.1.3: The Pattern search is in White mode and switches to Black mode because there is a strong threat pattern for Black at Y.

5.1.1 Pseudo Code

Algorithm 5.1.1 describes the pseudo code for the *Pattern Search* algorithm. This pseudo code is equivalent to the pseudo code van Rijswijck presented for the *Pattern Search* algorithm in [46]. The difference in the pseudo code of Algorithm 5.1.1 and the pseudo code presented by van Rijswijck, is that the mode colour in Algorithm 5.1.1 is used to trigger weak or strong threat pattern deductions in the search nodes.

Algorithm 5.1.1 Pattern_Search($board$) **Returns** (Carrier, Player colour)

```
 1: carrier ← ∅
 2: modeColour ← Black {Can start the search in Black mode}
 3:
 4: if board.isTerminal then
 5:    modeColour ← board.winningPlayer
 6:    return(carrier, modeColour)
 7: end if
 8:
 9: moves ← board.emptyCells {The initial must-play region.}
10:
11: for m ∈ moves do
12:    board.playMove(m)
13:    (C, winColour) ← Pattern_Search(board)
14:    {The mode colour changes if winColour ≠ modeColour.}
15:    modeColour ← winColour
16:    board.undoMove(m)
17:
18:    if modeColour = board.turn then
19:       {m was a winning move.}
20:       carrier ← {m} ∪ C {carrier, is now a weak threat pattern carrier.}
21:       return(carrier, modeColour)
22:    else
23:       {m was not winning, apply the OR deduction rule.}
24:       carrier ← carrier ∪ C {The union of carriers.}
25:       moves ← moves ∩ C {Update the must-play region.}
26:    end if
27: end for
28: return(carrier, modeColour) {Successful OR deduction.}
```

5.1.2 Performance Tests and Results

Pattern search is an example of a game-tree search that uses threat pattern deduction to solve Hex positions. This search involves many of the deduction rules that Yang et al. apply in their pattern decomposition method. Although the *Pattern Search* algorithm approximates the pattern decomposition method, how well Pattern searches perform in practice is not well documented. The following experiment involves an implementation of the *Pattern Search* algorithm. Beginning with the 3x3 Hex board and for each increasing board size, this search implementation takes as its input an empty board position and executes an exhaustive search. The measure of performance to be used in each search is the number of nodes in the game-tree visited for a complete solution.

Table 5.1.1 shows the performance test results for the *Pattern Search* algorithm in solving the 3x3 and 4x4 Hex boards. An attempt at solving the 5x5 Hex board with this algorithm failed to return after several days of execution on a 3GHz Pentium 4 class computer.

Search	*Pattern Search*
Nodes Visited 3x3	9,613
Nodes Visited 4x4	790,811,318

Table 5.1.1

The number of nodes Pattern searches must visit to completely solve the 3x3 and 4x4 Hex boards.

5.1.3 Remarks

The *Pattern Search* algorithm provides a good starting point for an investigation of Hex solving algorithms, given that Hex game-tree sizes are in the order of $\sum_{i=0}^{n^2} \frac{n^2!}{i!}$ according to Even and Tarjan in [23]. The *Pattern Search* algorithm has very effective pruning mechanisms through cut-off events and the deduction of must-play regions. Although Pattern searches are limited to board sizes less than 5x5, this algorithm exposes a number of features that can be exploited to improve efficiency. Firstly, the *Pattern Search* algorithm does not make use of any move ordering algorithms that might maximize cut-off events in the search. Secondly, this algorithm deduces threat patterns, which could be stored and reused elsewhere in the search. Finally, threat patterns found towards the end of games may be deduced using more efficient search algorithms, such as the H-Search algorithm.

5.2 Pattern Search with a Dynamic Move Generating Algorithm

The previous investigation of the *Pattern Search* algorithm exposed a number of features that one could exploit for better search times. In this section, an investigation of move order and the generation of move orderings is given. Algorithms that order moves are known as *move generators* and they place moves in an order which maximize pruning. Move generators which generate moves as a function of a single board position are called *static*, while *dynamic* move generators order moves as a function of a game-tree [2]. Such move generators sample some or all of the board positions in a game-tree to evaluate moves at the root board position. The objective of this investigation is to extend the *Pattern Search* algorithm with a dynamic move generator to improve search times over those reported in the previous section.

5.2.1 Move Order with Alpha-Beta Pruning

Game-tree searches that employ a pruning algorithm often perform better if their moves are placed in some appropriate order. To demonstrate how move order can change the performance of a game-tree search, consider the alpha-beta pruning algorithm example from Section 3.3. In Figure 5.2.4, if node J is not first then the search may explore additional nodes succeeding node E before satisfying a beta cut-off condition. In addition, if node F is not first then the search may explore additional nodes succeeding node C before satisfying an alpha cut-off condition. In both cases, poor move order increases the size of the search.

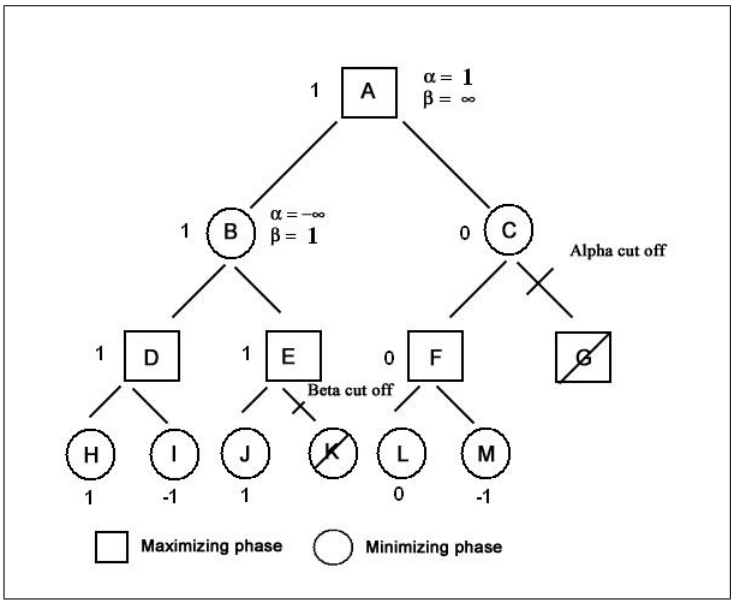

Figure 5.2.4: The alpha-beta pruning algorithm may be less effective if nodes J and F are not first.

5.2.2 The Pattern Search Cut-off Condition

The *Pattern Search* algorithm employs a similar pruning mechanism sensitive to move order. Assume that a Pattern search is in White mode (see Figure 5.2.5), the search completes a traversal of the subtree rooted at board position X and it

backtracks to position Z. At position Z the search explores another move, which is a winning move for player Black. The strong threat pattern at board position Y satisfies a cut-off condition that triggers a change of mode colour. The search abandons White's threat pattern at Z and deals with Y as though it was terminal board position. The switch of mode colour means player Black is now *Connect* and player White is now *Cut*. In this example, if the move that leads to board position Y comes before the move that leads to position X then the search never explores the subtree rooted at X. That is, the sub-tree rooted at X would be pruned from the search.

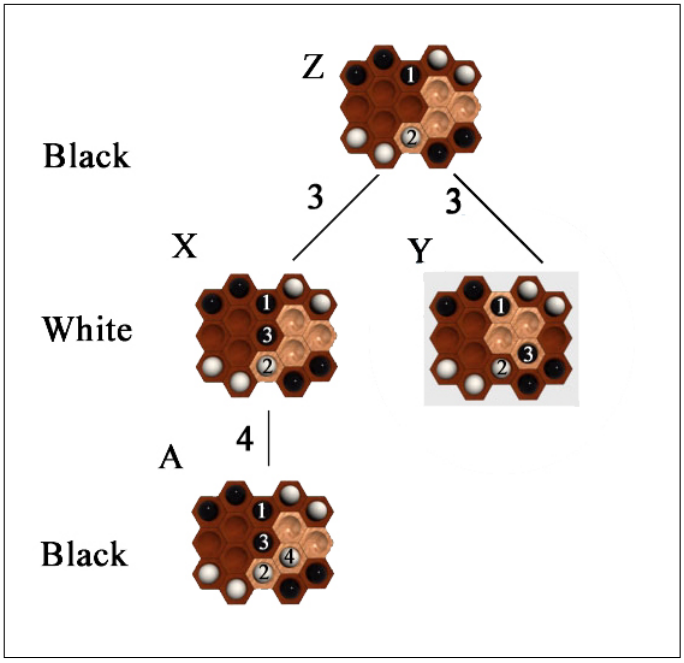

Figure 5.2.5: A Pattern search cut-off condition.

Since Black is the player who moves at board position Z and Black begins this search as *Cut*, the problem of generating moves in a Pattern search, so as to exploit a Pattern search cut-off condition, is restricted to generating good move orderings for player *Cut*.

5.2.3 A Dynamic Move Generator

The dynamic move generator presented herein makes use of features from terminal board positions to order moves for the internal board positions in a search tree [39]. Board position Z (from Figure 5.2.5) with *Connect's* weak threat pattern is shown on the left of Figure 5.2.6. Each empty cell in the carrier is labeled with the number of terminal board positions where that cell had a *Connect* stone and *Connect* wins. Right of Figure 5.2.6 gives board position Y (also from Figure 5.2.5), where *Cut* has moved on that empty cell with the largest number, successfully cutting *Connect's* weak threat pattern.

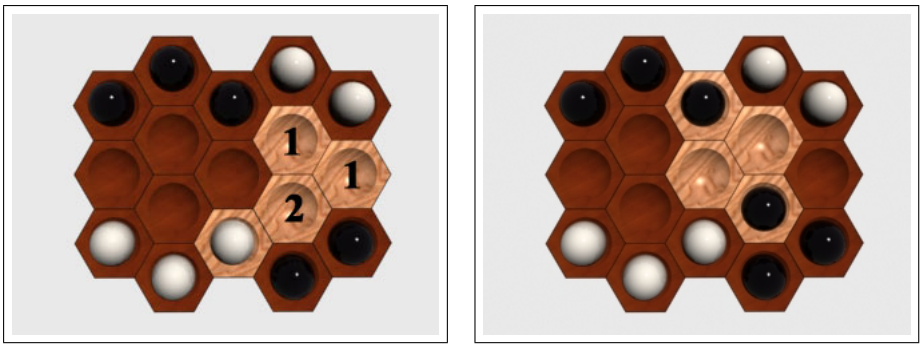

Figure 5.2.6: Left: Each carrier cell has the number of times White moved on that cell to connect the targets. Right: Black's cutting move is the move White used most often to connect.

In the search for a weak threat pattern, the *connection utility* of a cell in the carrier of that weak threat pattern, is the number of terminal board positions where that cell had a *Connect* stone and *Connect* wins. The move generator being considered here assumes that the goodness of a move for *Cut* on a cell is related to the connection utility of that cell. This move generator is based on the following hypothesis.

Hypothesis 5.2.1. *The greater the connection utility the better is the move for Cut.*

The heuristic for this move generating algorithm is to order moves so that a move with the highest connection utility has the highest priority. When a search traverses a subtree from a losing *Cut* move, it returns with a weak threat pattern and applies the OR deduction rule. In applying the Or deduction rule, the search updates must-play regions. Our move generator must derive the connection utilities for cells in must-play regions. Figure 5.2.7 shows how connection utilities are derived for a must-play region. In this example, player White is *Connect*, putting the search in White mode and player Black is *Cut*. The number on each carrier cell is its connection utility. The connection utilities on cells at board position A is the sum of connection utilities found at both board positions B and C and is restricted to the must-play region.

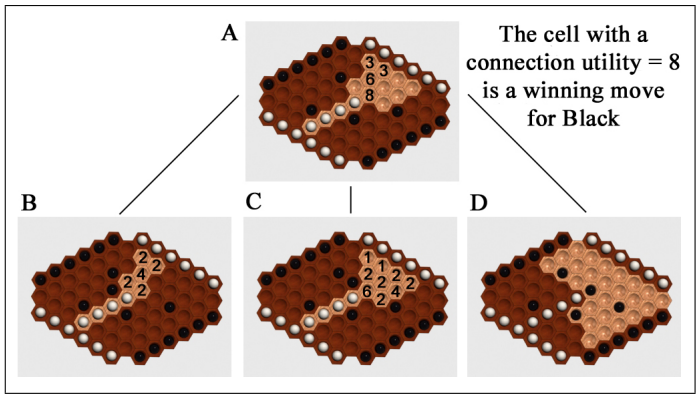

Figure 5.2.7: The accumulation of connection utilities in a must-play region reveals a good move for Black.

Figure 5.2.7 gives another example that validates Hypothesis 5.2.1 as connection utilities are used to predict a winning move for player *Cut*. From board position A, the search makes a move for Black on the cell with the largest connection utility. Board position D gives Black's winning threat pattern which causes a switch from White mode to Black mode. Board position D is treated as a terminal board position. Hypothesis 5.2.1 fails in the case where the must-play region is empty.

5.2.4 Hex Solver 1

Algorithm 5.2.2 extends the *Pattern Search* algorithm with the dynamic move generator described in the previous section. This algorithm describes the first Hex solver in a series and is called the *Hex Solver 1* algorithm. Some of the features appearing in the pseudo code for the *Pattern Search* algorithm have been suppressed from the pseudo code for *Hex Solver 1* and replaced with the symbol (. . .). In Algorithm 5.2.2, pseudo code for the move generating feature begins at line 1 where a vector of integer used to represent connection utilities is initialized to a zero vector. Lines $5, 6$ and 7 sets the value of the connection utility vector from terminal board positions. For every cell with a *Connect* stone on a terminal board position, the value one is assigned to the corresponding element in the connection utility vector. At Line 25, the connection utility vector returned from the search is added to the connection utility vector in the current search node. This sum occurs for each new weak carrier returned and corresponds with the application of the OR deduction rule. At line 26, the moves are reordered according to the recently updated vector of connection utilities.

Algorithm 5.2.2 Hex_Solver_1($board$) **Returns** (Carrier, Vector of utility values, Player colour)

```
1: carrier ← ∅
2: utilities[1 : board.size] ← 0;
3: ...
4: if board.isTerminal then
5:    modeColour ← board.winningPlayer
6:    for all i ∈ board.winnersStones do
7:       utilities[i] ← 1 {Set the cells in Connect's winning path.}
8:    end for
9:    return(carrier, utilities, modeColour)
10: end if
11:
12: moves ← board.emptyCells {The initial must-play region.}
13:
14: while moves ≠ ∅ do
15:    m ← popFirst(moves)
16:    board.playMove(m)
17:    (C, Util, winColour) ← Hex_Solver_1(board)
18:    ...
19:    board.undoMove(m)
20:
21:    if modeColour = board.turn then
22:       ...
23:       return(carrier, Util, modeColour)
24:    else
25:       ...
26:       utilities ← utilities + Util {vector operation}
27:       moves ← SortDescending(moves, utilities) {Order moves according
          to Hypothesis 5.2.1.}
28:    end if
29: end while
30: return(carrier, utilities, modeColour) {Successful OR deduction.}
```

5.2.5 Performance Tests and Results

The following experiment involves an implementation of *Hex Solver 1*. Beginning with the 3x3 Hex board and for each increasing board size, this search implementation takes as its input an empty board position and executes an exhaustive search. The measure of performance to be used in each search is the number of nodes in the game-tree visited for a complete solution and the time taken on a 3GHz Intel Pentium 4 class computer. In addition, the performance results from this experiment are compared against the performance results for standard Pattern searches.

Table 5.2.2 shows the performance results of *Hex Solver 1* on the problem of solving the 3x3 and 4x4 Hex boards in comparison to the performances measured for the standard *Pattern Search* implementation. In addition to these results, *Hex Solver 1* was also found to be an impractical solution for the 5x5 Hex board.

Search	*Pattern Search*	*Hex Solver 1*
Nodes Visited (Time) 3x3	9,613 ($<$ 1 sec)	2,811 ($<$ 1 sec)
Nodes Visited (Time) 4x4	790,811,318 (3876 sec)	4,906,570 (35 sec)

Table 5.2.2

The number of nodes the *Hex Solver 1* algorithm must visit to completely solve the 3x3 and 4x4 Hex boards and times in seconds in comparison to the Pattern search results.

5.2.6 Remarks

The results for *Hex Solver 1* demonstrate the potential to dramatically improve search times over standard Pattern searches. The dynamic move generator presented in this chapter was active on the backtrack phases of each search. Effective forward track move ordering was not solved by this algorithm and is open for further explorations.

5.3 Conclusion

This chapter began by looking at the *Pattern Search* algorithm, which provided an efficient search algorithm for creating Hex solving programs. A dynamic move generating algorithm that used *Connection Utilities* in a move ordering heuristic was also presented and performance tests show that this move generator is an effective enhancement to the Pattern Search algorithm. An interesting feature concerning this dynamic mover generator is that it only generates move orderings on the backtrack of searches. In response, a very strong forward track move generator will be presented in the following chapter. Effective move generation will always be an important problem to solve for Pattern searches. Since Hex has not been solved for arbitrary board sizes, the problem of optimal move generation for Pattern searches remains open. The way forward beyond devising better move generators, is to explore search pruning using sub-games and sub-game deduction. Towards this aim, applications of the H-Search algorithm in minimizing Pattern search times will be covered in the following chapter.

Chapter 6

Applications of the H-Search Algorithm for solving Hex

In order to solve larger Hex board sizes, Pattern searches will need to employ better pruning and move ordering algorithms. A possible set of search enhancements might come by applying the H-Search algorithm towards pruning and move ordering. Although the H-Search algorithm has some search inefficiencies, which will be addressed in this chapter, effective applications of H-Search in Pattern searches could affect conditions whereby some combinations give superior search times over the Hex solvers given in previous chapters.

Section 6.1 looks at the H-Search algorithm in more detail and describes techniques to minimize computational cost for this algorithm. This section also presents some performance characteristics of the H-Search algorithm that justify its application in Hex solving programs. In Section 6.2, the H-Search algorithm is presented as a tool for generating sub-games that can be used to replace threat patterns in Pattern searches. This application of H-Search is effectively a pruning algorithm that prunes subtrees towards the end of games. In Section 6.3, a move generator is presented that exploits H-Search executions by utilizing sub-game features. Section 6.4 presents techniques for move elimination by using sub-games to prove that certain moves do not change the outcome of a given game.

6.1 Fine Tuning the H-Search Algorithm

A high level description of the H-Search algorithm was given in Section 4.2 that showed the general search processes for H-Search executions. In this section, the H-

Search algorithm will be reviewed with greater detail. In addition, the arrangement of sub-game sets and optimizations used in applying the H-Search algorithm in the following experiments are covered in this chapter.

6.1.1 An Effective Arrangement for Sub-Game Sets

Figure 6.1.1 shows the high level view of H-Search that was presented in Section 4.2. This view shows two sets of sub-games. The first set is the set of strong sub-games and the other set is the set of weak sub-games [38]. In practice, each sub-game set was partitioned so that the sub-games in each partition have the same targets. Such a partition in the sub-game sets is called a *sub-game bucket*. Since the number of targets on any $n \times n$ board position is either less than or equal to n^2 plus the number of sides (4), targets can always be enumerated so that their target index i is in the domain $0 \leq i < n^2 + 4$. In the $Strong$ sub-game set, each sub-game bucket was indexed using target indices i and j, such that $Strong(i, j) = \{A_1, A_2, \ldots, A_m\}$ is the sub-game bucket where each sub-game member has one target at index i and the other target at index j. In addition, the access function had the property $Strong(i, j) = Strong(j, i)$. The $Weak$ sub-game set was equivalently partitioned.

In practice, the size of a sub-game bucket m was bound by some maximum limit K so that $m \leq K$. There is some trade off in choosing a value for K. H-Search executions on small sub-game buckets may terminate quickly, but with very few sub-games. Whereas, H-Search executions on larger sub-game buckets will generate more sub-games but will take longer to terminate. For the experiments in the following sections, it was found that $K = 35$ gave a reasonable balance between run times and sub-game yields. Finally, the access functions to the sub-game buckets can be implemented by using data structures such as square arrays or hash maps, which both have very efficient data access.

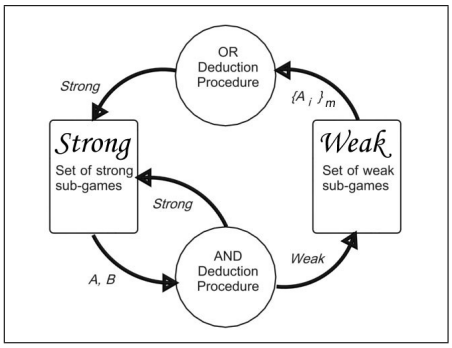

Figure 6.1.1: The H-Search algorithm applies the AND rule to strong sub-games A and B. The OR deduction rule is triggered to operate on a set of weak sub-games $\{A_i\}_m$, if the AND rule deduces a weak sub-game that belongs in this set.

6.1.2 An Optimization Technique for OR Deductions

A feature of the H-Search algorithm that needs to be covered in more detail is the procedure that applies the OR deduction rule. Recall that the OR rule involves a set of weak sub-games $\{(x, S_j, C_j, y)\}_{j=1}^{k}$ and states that if the intersection of the carriers in this set is empty, then the union of these carriers is a strong sub-game carrier. In the layout for H-Search shown in Figure 6.1.1, the OR deduction procedure is presented with a sub-game bucket of weak sub-games $\{A_i\}_{i=1}^{m}$, where their respective carriers form a set $\{C_i\}_{i\in I}$ ($I = \{1, 2, \ldots, m\}$). Given $\cap_{i\in I} C_i = \emptyset$ we want to find every minimal subset $J \subseteq I$ such that $\cap_{j\in J} C_j = \emptyset$. The aim is to apply the OR rule to every such subset $\{A_j\}_{j\in J}$, so as to maximize the number and variety of strong sub-games in the output. That is, an effective OR deduction procedure must first search the sub-game bucket $\{A_i\}_{i=1}^{m}$ for such subsets before applying the OR rule .

The approach that Anshelevich gives in [4] is a brute-force search that examines every possible sub-game combination in $\{A_i\}_{i=1}^{m}$. Algorithm 6.1.3 is equivalent to Anshelevich's OR deduction procedure, but has been modified to deal with the extended sub-games defined in Section 4.1. This algorithm performs a search that

traverses the entire forest, where each node is a weak sub-game and the sub-game sequence in each root-to-leaf path is a unique combination of weak sub-games in $\{A_i\}_{i=1}^{m}$.

Algorithm 6.1.3 Anshelevichs_OR_Deduction($\mathcal{W}$, $\mathcal{S}$, U_S, U_C, M)

Require: Initialize cell sets: $U_S \leftarrow \emptyset$ $U_C \leftarrow \emptyset$ and $M \leftarrow \mathcal{U}$ (the universal set of empty cells).
$\{\mathcal{W} = \{A_i\}_{i=1}^{m}$ is the set of weak sub-games with common target pairs x and $y.\}$
$\{\mathcal{S}$ is the set of strong sub-games with common target pairs x and $y.\}$

1: **for all** $(x, S, C, y) \in \mathcal{W}$ **do**
2: $\Delta \leftarrow U_S \cup S$ {Update support stones.}
3: $\Theta \leftarrow U_C \cup C$ {Update carrier union.}
4: $\Lambda \leftarrow M \cap C$ {Update carrier intersection.}
5: **if** $(\Lambda = \emptyset)$ **then**
6: $\mathcal{S} \leftarrow \mathcal{S} \cup \{(x, \Delta, \Theta, y)\}$
7: **else**
8: Anshelevichs_OR_Deduction($\mathcal{S}$, $\mathcal{W} - \{(x, S, C, y)\}$, Δ, Θ, Λ)
9: **end if**
10: **end for**

Two modifications were made to Anshelevich's OR deduction procedure for improved performance. The first modification was to change the search so that the sub-game sequence in each root-to-leaf path of a search tree, was a unique subset of $\{A_i\}_{i=1}^{m}$. Consider the two paths of sub-games $A_1 - A_2 - A_3$ and $A_3 - A_1 - A_2$, which are valid and unique search paths in an Algorithm 6.1.3 search. Although the sub-game sequence in each path is a unique combination, both sub-game sequences are the same set. Since, the OR rule applies to weak sub-game sets, a search that explores every sub-game combination in $\{A_i\}_{i=1}^{m}$ will inefficiently explore redundant subsets. The applied solution was an indexing rule that caused the search to traverse root-to-leaf paths provided the sub-game indices in each path was strictly increasing.

The second modification was to apply a pruning algorithm so that the sub-game sequences in most root-to-leaf paths were minimal sub-game sets that satisfy the OR rule. To demonstrate this pruning algorithm, consider an OR deduction search that has traversed a sub-game path $A_1 - A_2 - A_3$, whose carriers are respectively C_1, C_2 and C_3. The left of Figure 6.1.2 shows a Venn diagram representation of these three carriers with non-empty intersection I. The intersection I is the must-play region at this point in the OR deduction search. Recall that the *must-play* region is a set of cells where if *Cut's* move is not a stone on a cell in the must-play region then *Connect* has a move that is winning. If the next sub-game in the search is $A_4 = (x, S, C_4, y)$, such that the must-play region I is a subset of C_4 (see Right of Figure 6.1.2), then for any set of sub-games $\{A_1, A_2, A_3, A_4, A_5, \ldots, A_s\}$ that satisfies the OR rule, the set $\{A_1, A_2, A_3, A_5, \ldots, A_s\}$ also satisfies the OR rule. This substitution is valid because the exclusion of A_4 from OR deduction after the search node A_3 does not affect *Cut's* must-play region I. The subtree rooted at A_4 can be pruned from this part of the search using the pruning test $I \subseteq C_4$. Given these two improvements, Algorithm 6.1.4 defines a new OR deduction procedure.

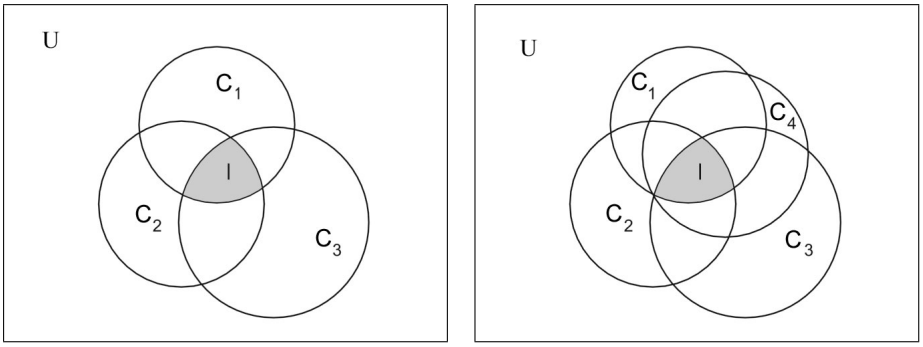

Figure 6.1.2: Left: The Venn diagram that represents carriers C_1, C_2 and C_3 for sub-game path $A_1 - A_2 - A_3$ in an OR deduction search. Right: Since I is a subset of C_4, the tree rooted at sub-game $A_4 = (x, S, C_4, y)$ can be pruned from the search.

Algorithm 6.1.4 Improved_OR_Deduction($\mathcal{S}$, $\mathcal{W}$, U_S, U_C, M, $i = 1$)

Require: Initialize cell sets: $U_S \leftarrow \emptyset$ $U_C \leftarrow \emptyset$ and $M \leftarrow \mathcal{U}$ (the universal set of empty cells).
$\{\mathcal{W} = \{A_i\}_m$ is the set of weak sub-games with common target pairs x and $y.\}$
$\{\mathcal{S}$ is the set of strong sub-games with common target pairs x and $y.\}$

1: **for** $j = i$ **to** m **do**
2: $(x, S_j, C_j, y) \leftarrow A_j$ {Recall $\mathcal{W} = \{A_i\}_m$.}
3: **if** $(M \nsubseteq C_j)$ {The search pruning condition.} **then**
4: $\Delta \leftarrow U_S \cup S_j$ {Update support stones.}
5: $\Theta \leftarrow U_C \cup C_j$ {Update carrier union.}
6: $\Lambda \leftarrow M \cap C_j$ {Update carrier intersection.}
7: **if** $(\Lambda = \emptyset)$ **then**
8: $\mathcal{S} \leftarrow \mathcal{S} \cup \{(x, \Delta, \Theta, y)\}$
9: **else**
10: **if** $j < m$ **then**
11: Improved_OR_Deduction($\mathcal{S}$, $\mathcal{W}$, Δ, Θ, Λ, $j + 1$)
{The value $j + 1$ is the starting index for the next search node, which ensures that the indices of sub-games in each search path is strictly increasing.}
12: **end if**
13: **end if**
14: **end if**
15: **end for**

6.1.3 Performance Tests and Results

The first test compares the performance of Algorithm 6.1.4 against the performance of Algorithm 6.1.3. The implementation of each procedure was given a diverse set of inputs of various sizes during executions of *Hex Solver 2* in attempts to solve 7x7 Hex. The performance of each procedure, for each possible input size, was given as the average number of nodes in a search traversal. The input sizes were limited to $m \leq (K = 35)$, however neither procedure received sub-game buckets greater than 29 sub-games in size during this experiment, which indicated that neither H-Search execution was limited by the size of sub-game buckets.

The top plot of Figure 6.1.3 shows the search results for Algorithm 6.1.3 and the bottom plot shows the search results for Algorithm 6.1.4. The average numbers of nodes visited in both tests are displayed on a logarithmic scale. The results show that both algorithms have similar performance characteristics when the sub-game bucket size (m) is small. In addition, the average search nodes at $m = 5$ in both tests was a fraction, because in practice the procedures counted all executions and could return immediately in the case where input sub-games buckets had non-empty must-play regions. The results diverge significantly as m approaches $K = 35$, showing Algorithm 6.1.4 as the best performer. Applications of Algorithm 6.1.4 in H-Search executions provide dramatic search time improvements. Several full H-Search executions on random 11x11 boards using $K = 35$ on a 3GHz Intel Pentium class machine were also done and these tests show that H-Search executions using Algorithm 6.1.3 took on average sixteen seconds to terminate, where H-Search executions using Algorithm 6.1.4 took on average twenty-five milliseconds to terminate.

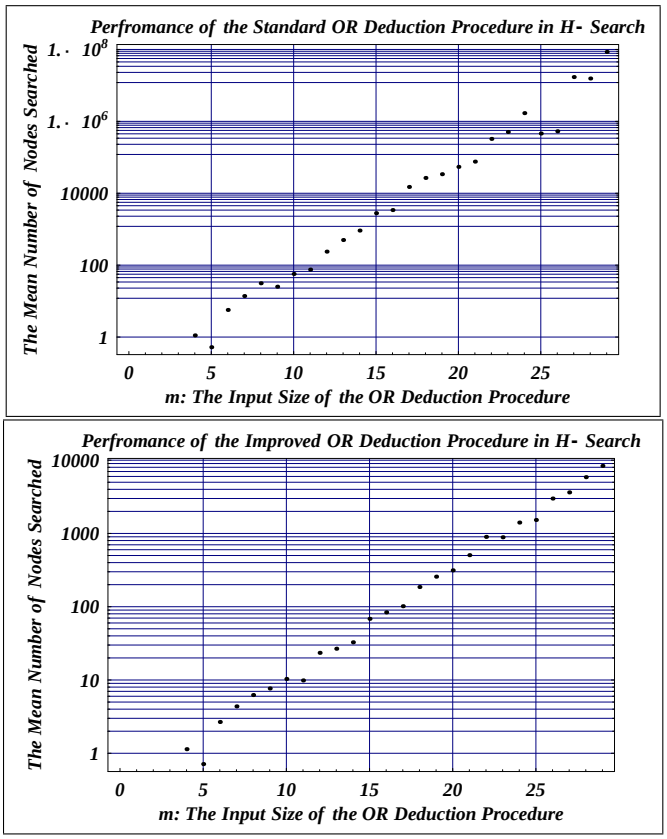

Figure 6.1.3: Top: The performance test results for the Anshelevich's OR deduction procedure given by Algorithm 6.1.3. Bottom: The performance test results for our improved OR deduction procedure given by Algorithm 6.1.4. In both plots m is the size of sub-game buckets.

6.1.4 Remarks on the Impact of H-Search in Hex Solvers

The OR deduction procedure has a very expensive running cost. Even with the performance improvements available with Algorithm 6.1.4, the plots in Figures 6.1.3 empirically show that the expected running time for both OR deduction searches is $O(e^m)$. Looking at the other procedures in the H-Search algorithm, the AND deduction procedure has an upper bound running time of only $O(m)$ as the procedure attempts to match a single sub-game to sub-games in a given sub-game bucket. When H-Search is applied to $n \times n$ boards, a traversal over the sub-game buckets has an upper bound running time of only $O(n^4)$, since such traversals use two target indices to index the sub-game buckets in the sub-game sets and any single target index i is in the domain $0 \leq i < n^2 + 4$. Taking all of these features into account, the expected running time for the H-Search algorithm is $O(e^m \times n^4)$, where m is the average sub-game bucket size.

In the following sections, sub-game bucket sizes are bound such that $m \leq K$ and K is a constant. This effectively reduces the complete upper running time for the H-Search algorithm to $O(n^4)$, for $(n \times n)$ boards. The problem with K bound sub-game bucket sizes, is that sub-game buckets may not be large enough to support useful H-Search executions on the larger boards. This problem can be addressed by applying Algorithm 6.1.4 to the H-Search algorithm, as sub-game buckets of a larger size can be processed for the same running time. Another problem with fixed upper limits is that a best K needs to be determined for a set of applications. In the applications presented in the following sections, $K = 35$ was determined empirically so that no bucket fills completely.

6.2 End-of-Game Pruning

In Section 4.1, a *threat pattern* was defined as a virtual connection between opposite sides of a board position. It happens that if there are enough *Connect* stones on a board position and the H-Search algorithm is allowed to index targets that are side groups, then the H-Search algorithm gives a search that can generate threat patterns. In Section 5.1, A *pattern* search is defined as a game-tree search that deduces threat patterns. A hybrid of these two algorithms is possible, as the H-Search algorithm can be used to generate threat patterns that can directly replace return values in Pattern searches. An advantage in making this replacement would be to prune sub-trees that occur towards the end of games in Pattern searches.

In investigating this idea, the *Hex Solver 1* algorithm (Algorithm 5.2.2) was extended so that a H-Search execution would generate sub-games at each search node in a Pattern search. Two distinct instances of the H-Search algorithm were defined. One instance of the H-Search algorithm would generate those sub-games where player Black is *Connect*, while the other instance would generate those sub-games where player White is *Connect*. For this discussion, the two instances of H-Search will be distinguished with the prefixes *Black-* and *White-*. A Black-H-Search execution took place on board positions where it was the White player's turn and a White-H-Search execution took place for board positions where it was the Black player's turn. In this arrangement of H-Search executions, strong threat patterns can prove if the move at the previous search node was winning.

To demonstrate how this approach works, consider a Pattern search on the board position $\mathcal{B}_1$, shown in Figure 6.2.4. At this point in the search, player Black has the turn at board position $\mathcal{B}_1$, which means player White has the turn at position $\mathcal{B}_2$. The White-H-Search performs a H-Search execution at position $\mathcal{B}_1$, but fails to find a strong threat pattern for White. At position $\mathcal{B}_1$ player Black makes a move m,

which puts position $\mathcal{B}_2$ at the input of a Pattern search. At position $\mathcal{B}_2$, the Black-H-Search performs a H-Search execution and finds a threat pattern for Black, whose carrier is shown as lightly shaded cells on board $\mathcal{B}_2$. At this point, the Pattern search can treat position $\mathcal{B}_2$ as a terminal position and backtrack by returning this threat pattern.

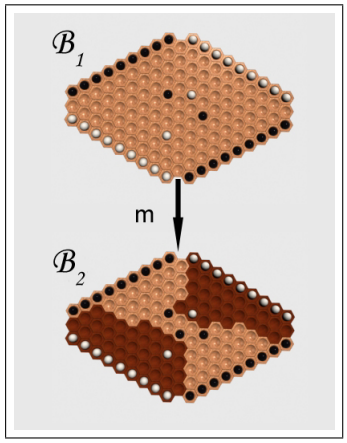

Figure 6.2.4: At position $\mathcal{B}_2$, a Pattern search executes a Black-H-Search subroutine that finds a strong threat pattern. The Pattern search can now treat position $\mathcal{B}_2$ as a terminal position and backtrack.

If a H-Search execution fails to find a strong threat pattern then pruning can be done with weak threat patterns. In Figure 6.2.5, instead of finding a strong threat pattern at board position $\mathcal{B}_2$, the Black-H-Search performs a H-Search execution that finds a set of weak threat patterns $\{A_1, A_2, A_3\}$ with carriers C_1, C_2 and C_3, which are shown as lightly shaded cells. At position $\mathcal{B}_2$, the Pattern search treats this set of weak threat patterns $\{A_1, A_2, A_3\}$ as though they were return values from Pattern searches on the successors of $\mathcal{B}_2$. In addition, the intersection of carriers C_1, C_2 and C_3 is the must-play region M for player White, which is shown as lightly shaded cells on position $\mathcal{B}_2$.

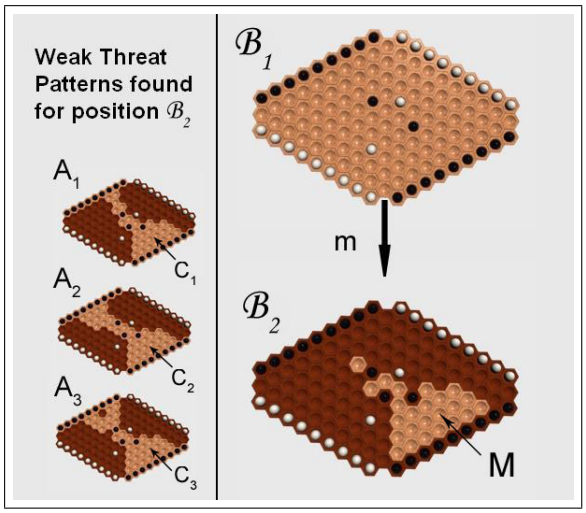

Figure 6.2.5: At position $\mathcal{B}_2$, a Pattern search executes a Black-H-Search subroutine that finds a set of weak threat pattern $\{A_1, A_2, A_3\}$. The Pattern search treats this set of threat patterns as though they were return values from Pattern searches on the successors of $\mathcal{B}_2$. The intersection of weak carriers C_1, C_2 and C_3 gives a must-play region M in the Pattern search.

6.2.1 Hex Solver 2

The pseudo code presented in Algorithm 6.2.5 extends on the pseudo code for the *Hex Solver 1* algorithm (Algorithm 5.2.2) with the application of H-Search just given. Unrelated features from Algorithm 5.2.2 have been suppressed with the symbol (. . .). Lines 11 to 30 are the additional lines of pseudo code in Algorithm 6.2.5 that are not in Algorithm 5.2.2. In the condition block that begins at line 11 and ends at line 17, the appropriate H-Search instance is selected for the turn order, a H-Search execution takes place and the mode colour is set. In the condition block that begins at line 19 and ends at line 25, if the H-Search instance can generate a strong threat pattern for the current board position then that threat pattern is retrieved, all of the cells in the carrier of that threat pattern are given a connection utility value of one and the Pattern search returns. Lines 27 to 30 handle the case where no strong threat patterns can be found for the current board position. In

this case, the union of the carriers over the available weak threat patterns partially defines the return carrier and the intersection of these carriers defines a must-play region for the move set.

Algorithm 6.2.5 Hex_Solver_2($board$) **Returns** (Carrier, Vector of utility values, Player colour)

```
1: carrier ← ∅
2: utilities[1 : board.size] ← 0;
3: ...
4: if board.isTerminal then
5:    ...
6:    return(carrier, utilities, modeColour)
7: end if
8:
9: moves ← board.emptyCells {The initial must-play region.}
10:
11: if board.turn = White then
12:    HSearch ← BlackHSearch
13:    modeColour ← Black
14: else
15:    HSearch ← WhiteHSearch
16:    modeColour ← White
17: end if
18:
19: if HSearch.hasStrongThreatPattern then
20:    (x, S, C, y) ← HSearch.getStrongThreatPattern()
21:    for all i ∈ C do
22:       utilities[i] ← 1 {Handle utility values for carrier cells.}
23:    end for
24:    return(C, utilities, modeColour)
25: end if
26:
27: for all (x, S, C, y) ∈ HSearch.WeakThreatPatternSet() do
28:    carrier ← carrier ∪ C
29:    moves ← moves ∩ C
30: end for
31:
32: while moves ≠ ∅ do
33:    ...
34: end while
35: return(carrier, utilities, modeColour) {Successful OR deduction.}
```

6.2.2 Performance Tests and Results

The following experiment involves an implementation of *Hex Solver 2* and contrasts the performance of *Hex Solver 2* against *Hex Solver 1* in solving Hex. The experiment begins with the 3x3 Hex board and requires that both algorithms solve Hex for increasing board sizes. The measure of performance to be used in each search is the number of nodes in the game-tree visited and the time taken on a 3GHz Intel Pentium 4 class computer.

Table 6.2.1 shows the performance results for *Hex Solver 2* on the problem of solving the 3x3, 4x4 and 5x5 Hex boards in comparison to the performance measured for *Hex Solver 1*. The *Hex Solver 2* algorithm was not able to solve the 6x6 Hex board in a practical time. Table 6.2.2 gives the upper bound running costs for the *Hex Solver 2* algorithm, which takes the upper running costs for the H-Search algorithm into account.

Search	*Hex Solver 1*	*Hex Solver 2*
Nodes Visited (Time) 3x3	2,811 (< 1 sec)	5 (< 1 sec)
Nodes Visited (Time) 4x4	4,906,570 (35 sec)	22 (< 1 sec)
Nodes Visited (Time) 5x5	undefined (∞ sec)	210,018 (973 sec)

Table 6.2.1

The number of nodes the *Hex Solver 2* algorithm must visit to completely solve the 3x3, 4x4 and 5x5 Hex boards and the time in seconds in comparison to the *Hex Solver 1* results.

Search	Nodes 3x3	Nodes 4x4	Nodes 5x5
Hex Solver 2	486	7,168	131,261,250

Table 6.2.2

The worst case running costs for the *Hex Solver 2* algorithm as the number of nodes that could be visited for each Pattern search on the 3x3, 4x4 and 5x5 Hex boards if the worst case running costs for H-Search are taken into account.

6.2.3 Remarks

The *Hex Solver 2* algorithm uses a simple application of the H-Search algorithm to massively improve Pattern search times in comparison to the *Hex Solver 1* search results. The upper bound running times for the *Hex Solver 2* algorithm are also significantly better in comparison, which robustly demonstrate that this application of H-Search is successful in reducing Pattern search times.

6.3 Extracting H-Search Features for Move Ordering

In the previous section, the H-Search algorithm was used to generate replacement threat patterns in Pattern searches with the affect of pruning subtrees that occur towards the end of games. A possible method for maximizing this pruning is to guide Pattern searches towards board positions where H-Search executions will return threat patterns. A possible approach might use a single level game-tree search for a given board position and uses H-Search results at the successors to prioritize moves[1]. In this section, a move generating algorithm based on this description will be applied in solving Hex.

[1]This idea was provided by Ryan Hayward in a personal communication to describe the move generator for the *Solver* program.

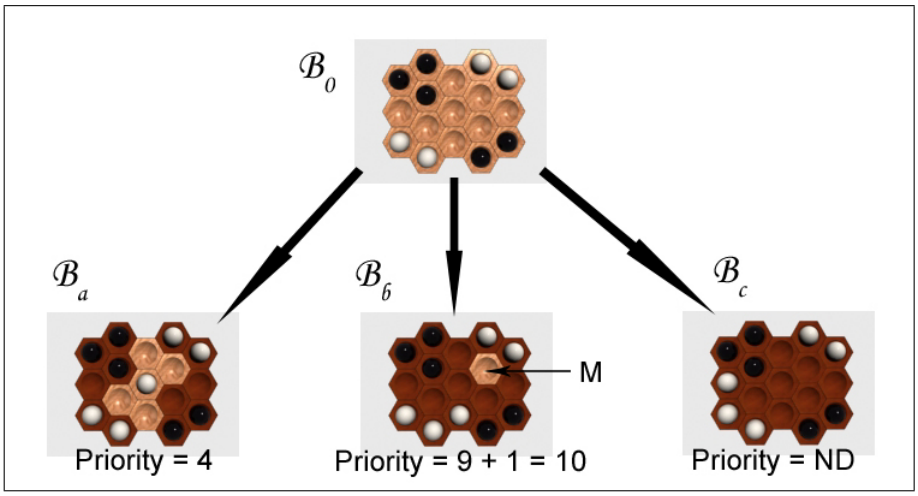

Figure 6.3.6: The move generator traverses a single level game-tree rooted at a position where White has the turn. At each successor the move generator applies the H-Search algorithm

Figure 6.3.6 shows a single level in a game-tree, where it is the White player's turn at the root. To generate moves for the root position, the move generator triggers a H-Search for each successor with the objective to find threat patterns where White is *Connect*. If for a given successor $\mathcal{B}_a$, a H-Search execution finds a strong threat pattern then the priority value of that successor is equal to the size of the smallest strong threat pattern carrier. If at another successor $\mathcal{B}_b$ only weak threat patterns can be found then the priority value is equal to the board size plus the size of the must-play region. In the final case $\mathcal{B}_c$, where the H-Search execution fails to find any such threat patterns, the priority value is not defined (ND). For board positions where Black has the turn, the move generator performs an equivalent search where Black is *Connect*.

The proposed move generator orders moves so that successors with the lowest priority value have the greatest priority. In addition, this move generator omits moves that lead to successors with undefined priority values. Let this move gener-

ator be called the *Pattern Feature* move generator. To ensure that all moves were placed in some order, both the *Connection Utility* move generator from Section 5.2 and this *Pattern Feature* move generator were applied such that the *Pattern Feature* move generator had precedence over the *Connection Utility* move generator. The role of the *Connection Utility* move generator was to order those moves that the *Pattern Feature* move generator had omitted.

6.3.1 Hex Solver 3

The pseudo code presented in Algorithm 6.3.6 extends on the pseudo code for *Hex Solver 2* (Algorithm 6.2.5) with the move generating algorithm just described. In Algorithm 6.3.6, the proposed move generator is represented by the following function:

$$\mathcal{M} = \text{PatternFeatureMoveGenerator}(board) \tag{1}$$

Where $board$ is a board position and $\mathcal{M}$ is an ordered set of cells that correspond with moves and their priority. With this move generator, the set $\mathcal{M}$ is not necessarily the complete set of empty cells and may be a subset of must-play regions. Code features from Algorithm 6.2.5 that do not provide information on implementing the *Pattern Feature* move generator have been suppressed with the symbol (...). In Algorithm 6.3.6, there are two sets of moves $moves$ and $\mathcal{M}$. At line 9 the $moves$ set is initialized with the complete set of empty cells, which is then refined to the must-play region in the statement block of lines 11 to 13. At line 15 the *Pattern Feature* move generator from Equation (1) was applied to return moves that intersected with must-play regions. Line 16 calculates a cell set $moves$ that is disjoint from $\mathcal{M}$, which means that the union of these two sets is the must-play region. Line 18 has been changed from the previous Hex Solver pseudo code, so that if no pattern search cut-off condition occurs then the exploration of moves does not terminate unless both $moves$ and $\mathcal{M}$ are the empty set. The condition block at lines 19 to 23 ensures that the moves on cells in $\mathcal{M}$ take precedence over moves on

cells in $moves$. The $moves$ set is ordered by the dynamic *Connection Utility* move generator, whose details have been suppressed in Algorithm 6.3.6.

Algorithm 6.3.6 Hex_Solver_3($board$) **Returns** (Carrier, Vector of utility values, Player colour)

```
 1: carrier ← ∅
 2: utilities[1 : board.size] ← 0;
 3: ...
 4: if board.isTerminal then
 5:    ...
 6:    return(carrier, utilities, modeColour)
 7: end if
 8:
 9: moves ← board.emptyCells {The initial must-play region.}
10: ...
11: for all (x, S, C, y) ∈ HSearch.WeakThreatPatternSet() do
12:    ...
13: end for
14:
15: M ← PatternFeatureMoveGenerator(board) ∩ moves {Get ordered moves in
    the must-play region.}
16: moves = moves − M {Make sure move sets are disjoint.}
17:
18: while moves ≠ ∅ ∨ M ≠ ∅ do
19:    if M ≠ ∅ then
20:       m ← popFirst(M) {Pattern Feature moves first.}
21:    else
22:       m ← popFirst(moves) {Connection Utility moves next.}
23:    end if
24:    ...
25: end while
26: return(carrier, utilities, modeColour) {Successful OR deduction.}
```

6.3.2 Performance Tests and Results

The following experiment compares the performance of *Hex Solver 3* against *Hex Solver 2* in solving Hex. Beginning with the 3x3 Hex board the solver algorithms solve Hex for each increasing board size. The measure of performance to be used in

each search is the number of nodes in the game-tree visited for a complete solution and the time taken on a 3GHz Intel Pentium 4 class computer.

Table 6.3.3 shows the performance results of *Hex Solver 3* on the problem of solving Hex boards inclusively in the size range 3x3 to 6x6, in comparison to the performances measured for *Hex Solver 2*. In addition to these results, *Hex Solver 3* was found to be an impractical solution for the 7x7 Hex board.

Search	*Hex Solver 2*	*Hex Solver 3*
Nodes Visited (Time) 3x3	5 ($<$ 1 sec)	5 ($<$ 1 sec)
Nodes Visited (Time) 4x4	22 ($<$ 1 sec)	16 ($<$ 1 sec)
Nodes Visited (Time) 5x5	210018 (973 sec)	530 (15 sec)
Nodes Visited (Time) 6x6	Undefined (∞ sec)	149859 (9376 sec)

Table 6.3.3

The number of nodes the *Hex Solver 3* algorithm must visit to completely solve Hex boards inclusively in the range 3x3 to 6x6, and the time taken in comparison to the performance of *Hex Solver 2*.

6.3.3 Remarks

These results show that this *Pattern Feature* move generator was able to guide *Hex Solver 3* towards positions that could be solved using H-Search and they indicate that significant end-of-game pruning took place. The details of this move generating method are critical if one aims to reproduce the results Hayward et al. report for their Hex Solver in [27].

6.4 Move and P-Triangle Domination

The previous experiments give empirical evidence that show how search times in Hex game-trees compound quickly with increasing board sizes. A feature that is problematic for Hex Solvers is the large branching factors in Hex game-trees. Thus far, the H-Search algorithm has been applied to deduce must-play regions for board positions, which effectively eliminate moves from the search and greatly reduce search times. One would expect that other methods to find and eliminate moves would also be effective.

In [27], Hayward et al. applied a board feature template to decide if moves could be omitted from Hex game-tree searches without affecting the return value. Their method relied on a principle of move domination. Given a subset of the moves D on a board position in a game-tree search where player P has the turn, if there exists another move $m_x \notin D$ such that m_x is a winning move for P whenever each move in D is a winning move for P, then move m_x is said to *dominate* D. In [26], Hayward gave a proof for move domination on cells that form certain patterns called *P-Triangles*.

Theorem 6.4.1 (P-Triangle)**.** *Given a player P, a P-Triangle is a set of of three adjacent cells $\{x_1, x_2, t\}$ such that:*

1. *the cells x_1 and x_2 (called the* base*) are adjacent cells where both are empty or one has a P stone,*
2. *the cell t (called the* tip*) is either empty or has a P stone,*
3. *cells x_1 and x_2 are both adjacent to a P coloured side,*
4. *both x_1 and x_2 are adjacent to t.*

Figure 6.4.7 shows two Black-Triangles (P = Black) in lightly shaded cells each with different tips. In the left of Figure 6.4.7, a move on the tip cell t dominates

moves on cells x_1 and x_2. In [9], Björnsson et al. show that certain moves are equal to missing a turn. In the right of Figure 6.4.7, if White has the turn then a White move on either cell x_1 or x_2 is equal to missing a turn. P-Triangles can be categorized into two types. Given a P-Triangle $\{x_1, x_2, t\}$, if t is an empty cell then t is said to *dominate* the base and the base cells are called *P-Dominated*. Otherwise if if t has a P stone then t is said to *capture* the base and the base cells are called *P-Captured*.

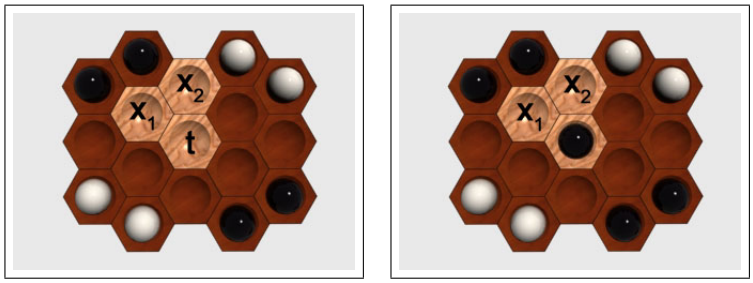

Figure 6.4.7: Left: A Black-Triangle where the tip is empty. Right: A Black-Triangle where the tip has a Black stone.

In a game-tree search, move domination can be used to omit moves without changing the return value of the search. Let player P be Black and player Q be White. On board positions where Black has the turn, moves on Black-Dominated and White-Captured cells can be omitted. Alternatively on board positions where White has the turn, moves White-Dominated and Black-Captured cells can be omitted. A justification for this cell omission process is given in the following subsection.

6.4.1 Hex Solver 4

The pseudo code presented in Algorithm 6.4.7 extends on the pseudo code for *Hex Solver 3* (Algorithm 6.3.6) with an application of P-Triangle templates for

move elimination. This application of P-Triangle templates is expressed in a function that takes as input a board position $board$, a player P and returns a set of cells D:

$$\mathcal{D} = \text{GetEliminatedCells}(board, P) \tag{2}$$

If P is player Black then $\mathcal{D}$ is a set of Black-Dominated and White-Captured cells. Otherwise, if P is player White then $\mathcal{D}$ is a set of White-Dominated and Black-Captured cells. Code features from Algorithm 6.3.6 that do not relate to move domination and capture have been suppressed with the symbol (...). At line 15, Equation (2) is applied to get a set of cells $\mathcal{D}$ that will be used to eliminate dominated and captured moves. At line 16, the set $\mathcal{D}$ is subtracted from $moves$, which is initially the must-play region. In the condition block 17 to 23, if every cell in the must-play region was also in $\mathcal{D}$ then the current carrier, which is the union of weak threat pattern carriers found via H-Search, is strong.

To justify the previous claim, consider a board position where player P has the turn and Q is the opponent of P. At this position, a H-Search execution provides a set of weak threat patterns where Q is *Connect* and a non-empty must-play region for P. Let the cells in this must-play region be either P-Dominated or Q-Captured. A P move on a Q-Captured cell is equal to missing a turn so removing these cells from the must-play region is trivially valid. If the remaining must-play region consists of P-Dominated bases then the tips for these bases sit outside of the must-play region. However, no move on a cell outside of a must-play region can possibly be a winning move. Therefore, no move on a P-Dominated cell inside this must-play region is winning. The union of these weak threat pattern carriers is a strong threat pattern carrier.

Another possibility is that Q-Captured cells form the carrier of a strong threat pattern where player Q is *Connect*. Given that P-Triangles are very simple, H-Search executions can be set to always find such threat patterns.

Algorithm 6.4.7 Hex_Solver_4($board$) **Returns** (Carrier, Vector of utility values, Player colour)

```
1: carrier ← ∅
2: utilities[1 : board.size] ← 0;
3: ...
4: if board.isTerminal then
5:    ...
6:    return(carrier, utilities, modeColour)
7: end if
8:
9: moves ← board.emptyCells {The initial must-play region.}
10: ...
11: for all (x, S, C, y) ∈ HSearch.WeakThreatPatternSet() do
12:    carrier ← carrier ∪ C
13:    moves ← moves ∩ C
14: end for
15: 𝒟 ← GetEliminatedCells(board, board.turn) {Get board.turn-dominated and
    board.turn-captured cells.}
16: moves ← (moves − 𝒟) {Omit board.turn-dominated and board.turn-captured
    cells.}
17: if moves = ∅ then
18:    {By exhausting all moves, the current carrier must be strong.}
19:    for all c ∈ carrier do
20:       utilities[c] ← 1 {Set the Connection Utility for the carrier cell.}
21:    end for
22:    return(carrier, utilities, modeColour)
23: end if
24: ℳ ← PatternFeatureMoveGenerator(board) ∩ moves
25: ...
26: while moves ≠ ∅ ∨ ℳ ≠ ∅ do
27:    ...
28: end while
29: return(carrier, utilities, modeColour) {Successful OR deduction.}
```

6.4.2 Performance Tests and Results

The following experiment compares the performance of *Hex Solver 4* against the performance of *Hex Solver 3* in solving Hex. Beginning with the 3x3 Hex board both solver algorithms must solve Hex for each increasing board size. The measure of performance to be used in each search is the number of nodes in the game-tree visited for a complete solution and the time taken on a 3GHz Intel Pentium 4 class computer.

Table 6.4.4 shows the performance results of *Hex Solver 4* on the problem of solving Hex boards inclusively in the size range 3x3 to 7x7, in comparison to the performances measured for *Hex Solver 3*. In addition to these results, *Hex Solver 4* was found to be an impractical solution for the 8x8 Hex board.

Search	*Hex Solver 3*	*Hex Solver 4*
Nodes Visited (Time) 3x3	5 ($<$ 1 sec)	5 ($<$ 1 sec)
Nodes Visited (Time) 4x4	16 ($<$ 1 sec)	16 (1 sec)
Nodes Visited (Time) 5x5	530 (15 sec)	157 (8 sec)
Nodes Visited (Time) 6x6	149,859 (9,376 sec)	14,451 (1759 sec)
Nodes Visited (Time) 7x7	Undefined	21,192,028 (2,420,021 sec)

Table 6.4.4

The number of nodes the *Hex Solver 4* algorithm must visit and search times in seconds to completely solve Hex boards inclusively in the range 3x3 to 7x7, in comparison to the performances of *Hex Solver 3*.

6.4.3 Remarks

These results show that the elimination of moves based on move domination via P-Triangles is a significant pruning strategy for Hex solvers. However, the pruning

for this method is not of the level achieved using the *Pattern Feature* move generator in *Hex Solver 3*, which guided searches towards positions where threat patterns could be deduced using H-Search. The performance results given in Table 6.4.4 show that the necessary optimization techniques for solving 7x7 Hex have been successfully identified in *Hex Solver 4*. The *Hex Solver 4* employs a combination of *Connection Utility* and *Pattern Feature* move generation along with a high performance H-Search. With the addition of search optimization techniques involving move domination reported in [27], the performance of *Hex Solver 4* in solving the 7x7 Hex board is comparable to the results Hayward et al. report for their *Solver* algorithm. That is, the *Hex Solver 4* results independently confirm the results that Hayward et al. [27] report. In addition, the *Hex Solver 4* finds the same pattern of winning opening moves for 7x7 Hex, which is shown in Figure 6.4.8 as the cells with Black stones for Black's winning opening moves.

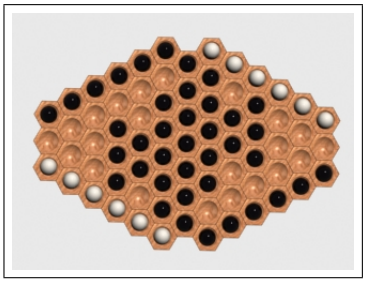

Figure 6.4.8: Blacks winning must-play region of opening moves for 7x7 Hex.

6.5 Additional Optimization Methods using H-Search

During the course of this investigation, two other optimization methods were devised. The first method involves weak threat patterns that Pattern searches can deduce by undoing moves that break stone chains in two. In the left of Figure 6.5.9 a move has been undone such that a White stone group is broken in two about the empty cell x. The Pattern search will treat the set of lightly shaded cell as the

must-play region and will search moves on any lightly shaded cell. This must-play region is not minimal because a strong threat pattern can be deduced from this weak threat pattern by treating the set $\{x\}$ as the weak carrier and the two adjacent White stone groups as targets. In the right of Figure 6.5.9, another weak carrier $\{y\}$ is introduced such that the intersection $\{x\} \cap \{y\}$ is empty, which means $\{x\} \cup \{y\}$ is a strong carrier. This deduction is valid because both White stone groups are strongly connected to their respective White side. The result of this carrier repair method is that the search can return the resulting strong carrier immediately, thus pruning the search.

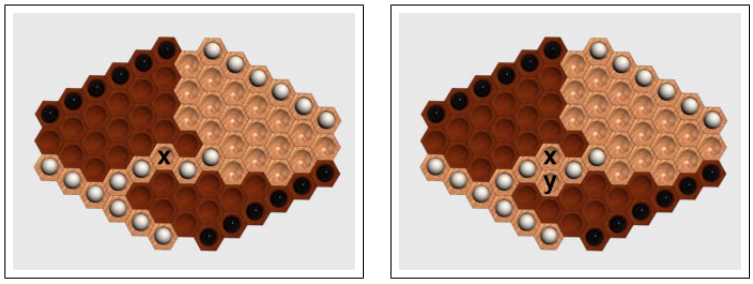

Figure 6.5.9: Left: A weak threat pattern found during a Pattern search by undoing a White move that breaks a White stone group in two about the empty cell x. Right: The empty cell y can be added to the carrier to construct a strong threat pattern.

The second optimization method involved an OR deduction procedure, modified from Algorithm 6.1.4, to deduce from each list of weak threat patterns found during Pattern search a strong threat pattern with a minimal carrier. This modification simply requires a condition in the search of the OR deduction procedure that tests the size of each strong threat pattern carrier deduced and maintains a return reference to the threat pattern with the smallest carrier.

Of these two methods, the first method was the most effective. However, neither method is effective in a Hex solver that applies the *Pattern Feature* move gener-

ator from Section 6.3. This move generator is very successful at guiding Pattern searches to board positions where compact threat patterns can be found and contiguous carriers can be deduced. The effectiveness of these additional methods in Hex solvers applied to boards larger that 7x7 remains unclear. One would expect that the *Pattern Feature* move generator will become less effective for Hex solvers applied to larger boards, which implies that these additional optimization method could be more effective for larger boards.

6.6 Conclusion

This chapter addressed issues of applying the H-Search algorithm to a Pattern search. In the issue of H-Search performance, the solution presented was a more efficient OR deduction procedure. This improved OR deduction procedure meant that larger boards could be processed with H-Search for the same running time. Given an efficient H-Search method, the deduction of threat patterns via H-Search were applied in a very effective end-of-game pruning in Pattern searches. The results showed that search times had improved regardless of the additional running costs of H-Search. Based on an idea by Ryan Hayward, H-Search deduced threat patterns were used in a single level game-tree search to order moves. This move generating algorithm was very successful in guiding Pattern searches towards positions where H-Search could deduce threat patterns and maximize end-of-game pruning. Finally, the concept and application of move domination was explored in the *Hex Solver 4* algorithm. The significance of *Hex Solver 4* was that its search results gave evidence that support the results reported by Hayward et al. in [27] for their *Solver* algorithm.

Chapter 7

Applications of Threat Pattern Templates For Solving Hex

From the search results of Hex Solver 4 in Section 6.4, it is clear that a Hex solving search on the 7x7 Hex board can be small enough so that a standard computer can store an entire search trace in physical memory. This chapter presents an approach for tracing a search and using the trace data to further prune the search space for Hex solvers. This approach prunes searches in a manner similar to a transposition table (in Section 3.4), but instead of caching board positions, it caches a compact form of sub-game called a *template*. Templates are the features of this approach that will be used to prove that certain board positions are winning in pruning searches.

Section 7.1 will introduce an extended form of sub-game called a *multi-target sub-game*, which is the super class for templates. A method will be presented that uses templates to prove winning positions. Section 7.2 will introduce an approach for caching templates called a *Template Matching Table* that can be used to prune pattern searches for Hex solvers in a manner similar to transposition tables. Section 7.3 will show how to apply template matching tables to Pattern searches. This section will give details about the lookup procedure for template matching tables and describe a method for efficient template matching lookups.

7.1 Multi-Target Sub-games and Templates

In Section 4.1, a sub-game is a four-tuple that involves a carrier, a support set and two distinct targets. However, more complex sub-games can occur on arbitrary

board positions, where player *Connect* has the option of connecting one of many target pairs. A *multi-target sub-game* is a game played by *Cut* and *Connect* on empty cells between two sets of targets X and Y. The role for *Connect* is to form a chain of stones connecting at least one target in X to at least one target in Y, while *Cut* moves to prevent *Connect* from forming any such chain of stones. A more formal definition of a multi-target sub-game is given in the following definition:

Definition 7.1.1 (multi-target sub-game)**.** *A multi-target sub-game is a four-tuple* (X, S, C, Y) *where* X *and* Y *are* target sets. *The set* S*, is a set of cells with Connect's stones called* support *and the set* C *is a set of empty cells called the* carrier. *Finally,* X*,* Y*,* S *and* C *are all disjoint.*

A multi-target sub-game is a *virtual connection*, if *Connect* can win this sub-game against a perfect *Cut* player. In addition, the terms *weak* and *strong* apply. A multi-target sub-game that is a virtual connection is called a *template*.

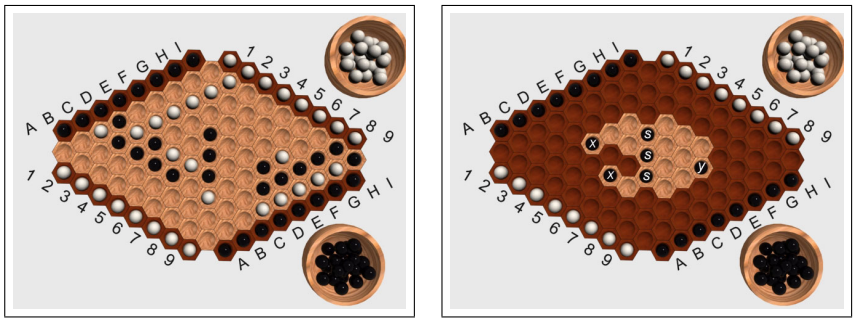

Figure 7.1.1: Left: An arbitrary board position b. Right: A template that proves b is winning for Black.

A template can be used to prove a board position is winning for one of the players. For example the left of Figure 7.1.1, shows an arbitrary board position b. For board position b, let B_X be the set of stones that form Black's top-left side group and let B_Y be the set of stones that form Black's bottom-right side group. In addition,

let B_S be the remaining set of Black stones and let B_C be the set of empty cells. The right of Figure 7.1.1 shows a template (X, S, C, Y), where Black is *Connect*, $X = \{C5, D3\}$, $Y = \{F7\}$, $S = \{D6, E5, F4\}$ and the carrier C is represented by the set of empty cells with a lighter shade. This template *proves* b is winning for Black because, $X \subseteq B_X$, $Y \subseteq B_Y$, $S \subseteq B_S$ and $C \subseteq B_C$, which means that every chain of stones that Black can form to win this template will also connect Black's sides on b.

Proposition 7.1.2 (Template Matching)**.** *Let (X, S, C, Y) be a template where player P is Connect. Given a board position b, let B_X be the set of stones in one side group for player P, B_Y be the set of stones in the other side group for player P, B_S be the set of P stones not in B_X or B_Y and B_C be the set of empty cells. If $(X \subseteq B_X) \wedge (Y \subseteq B_Y) \wedge (S \subseteq B_S) \wedge (C \subseteq B_C)$, template (X, S, C, Y) proves b is winning for player P.*

7.2 A Template Matching Table

A *template matching* table is a mapping $T_T : t \rightarrow v$, from a set of templates t to a set of game theoretic values v. The application of a template matching table is similar to the application of a transposition table (see Section 3.4). In Figure 7.2.2, a game-tree search that employs a template generating algorithm attempts to solve the Hex board position b_0. The game-theoretic values are (-1) and (+1) for Black and White's winning positions, respectively. In the left of Figure 7.2.2, the search traverses to board position b_j where a template generating algorithm derives template t_k. Since the Black player is the *Connect* player in t_k, the template matching table T_T is updated with the mapping $(t_k, -1)$. In the right of Figure 7.2.2, the search traverses a different path to board position b_n. A lookup procedure applies Proposition 7.1.2 to each template in T_T and finds that template t_k is a match for position b_n. The game-tree search uses the value $T_T(t_k)$ instead of searching the subtree rooted at b_n.

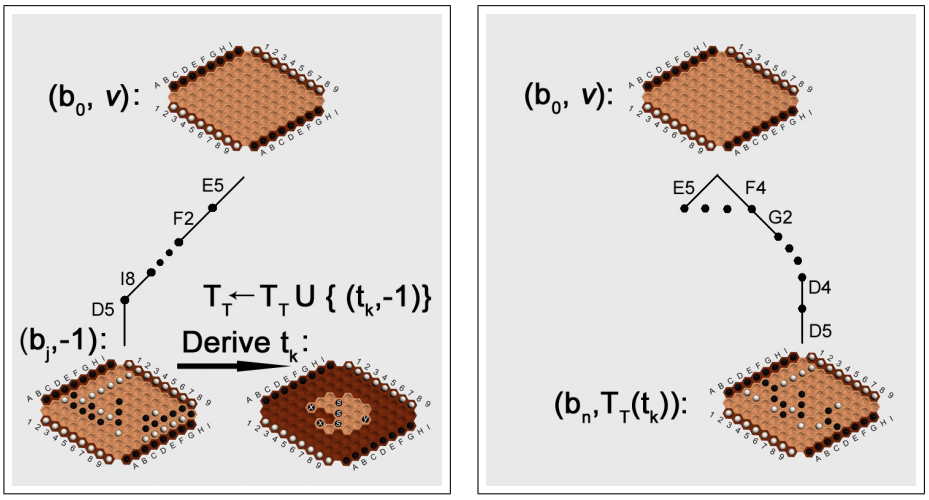

Figure 7.2.2: Left: A game-tree search finds that Black has a winning strategy for b_i represented by template t_k. The search adds the mapping $(t_k, -1)$ to the template matching table T_T. Right: On a different path, the same search arrives at board position b_n. An application of Proposition 7.1.2 shows that template t_k matches b_n. The game theoretic value for b_n can be returned by the template matching table, in place of a search in the subtree of b_n.

7.3 Template Matching Tables in Pattern Search

The Pattern Search algorithm can be defined as a game-tree search that returns either a weak or strong threat pattern carrier. In applying a template matching table in a Hex solver that performs Pattern searches, the table has to return a carrier and not a game theoretic value. In this case, the lookup procedure of a template matching table can be used to find matching templates and carriers of those templates can be returned in Pattern searches. The following template matching lookup procedure takes a board position b and a template matching table T_T and returns a set of those templates $\mathcal{T}$ in T_T that prove b is a winning position:

$$\mathcal{T} = \text{templateLookUp}(b, T_T) \tag{1}$$

where $\mathcal{T}$ is either the empty set or a singleton.

The lookup procedure in Equation (1) applies Proposition 7.1.2 to the template in each template-value pair in T_T in a sequential search for a match. If the search terminates without a match then $\mathcal{T}$ is empty. Otherwise, the lookup procedure returns a singleton $\mathcal{T}$, whose element is the first matching template.

A rule can be applied to eliminate some of the template-value pairs from T_T prior to lookups. Given board position feature sets B_X, B_S, B_C and B_Y described in Section 7.1, let (B_X, B_S, B_C, B_Y) be a *board-template*. Each feature set in a board-template has a maximum of n^2 elements for an $n \times n$ board. In addition, a *bit-string* can represent sets on a computer. A bit-string $(\alpha_1, \alpha_2, \ldots, \alpha_{n^2})$ can be used to represent a board feature set B_F when the set of all features possible in B_F has some ordering $\{f_1, f_2, \ldots, f_{n^2}\}$ and if $\alpha_j = 1$ then $f_j \in B_F$ otherwise $f_j \notin B_F$. Consequentially, A bit-string $(\beta_1, \beta_2, \ldots, \beta_{4.n^2})$ has enough bits to represent a board-template. In addition, $(\beta_1, \beta_2, \ldots, \beta_{4.n^2})$ also has enough bits to represent a template, since a template is also a four-tuple of board feature sets.

Every finite bit-string has a natural mapping to the integers, which means that both templates and board-templates can be mapped to the integers via bit-string representations. Bit-strings of $4.n^2$ bits can be mapped to the integers using a sum of the form:

$$x = \sum_{j=1}^{4.n^2} 2^j \beta_j \tag{2}$$

However, for the board feature set B_F, there are many permutations of the features $\{f_1, f_2, \ldots, f_{n^2}\}$. Assume that for every board feature set, a feature permutation has been selected as the conventional feature ordering. Given such a convention and Equation (2), there follows a conventional mapping:

$$Z(t) : t \rightarrow I \tag{3}$$

where t is either a board-template or a template and $I \subset \mathbb{Z}$.

An integer mapping of board positions and templates can be used to efficiently determine if a given template **does not** match a given board position. When a template does not match a given board position then this is called a *template mismatch*. If template mismatches can be found efficiently, then they can be used to prune template table searches. An efficient rule for template mismatching is given in the following proposition:

Proposition 7.3.1 (A Template Mismatch Rule)**.** *Given an $n \times n$ board position b, let $\breve{b}$ be the board-template of b and t_j be a template in a template matching table T_T. Given a mapping of the form Equation (3), if $Z(t_j) > Z(\breve{b})$ then t_j does **not** match b.*

Let bit-string $(\beta_1, \beta_2, \ldots, \beta_{4.n^2})$ represent t_j and bit-string $(\gamma_1, \gamma_2, \ldots, \gamma_{4.n^2})$ represent $\breve{b}$. The explanation for Proposition 7.3.1 is that if $Z(t_j) > Z(\breve{b})$ then there is at least one bit $\beta_i \neq \gamma_i$. That is, t_j has a feature that $\breve{b}$ does not have and therefore t_j is not a match for position b.

Let T_T be an ordered template matching table where for consecutive elements (t_j, v_j) and (t_{j+1}, v_{j+1}) the template are such that $Z(t_j) < Z(t_{j+1})$. Given a board position b and board-template $\breve{b}$, a binary search can be applied using the test $Z(t_j) > Z(\breve{b})$ from Proposition 7.3.1 to find the largest k such that $Z(t_k) \leq Z(\breve{b})$. The lookup procedure in Equation (1) can search the table T_T in the index interval $[1, k]$. This method effectively prunes templates at indices greater than k from the lookup search. Although this pruning approach is effective in reducing search times, there can be cases where a search must visit every template in the table.

7.3.1 A Method to Standardize Templates

In practice, a problem was found in representing templates efficiently. The problem was that several templates could appear in a template table with slightly differ-

ent X, S and Y sets but identical carriers. That is, all of these templates represented the same sub-game but had slight variations. For example, the hexagonal shape of cells meant that the same cell in a given carrier could be adjacent to many different singleton X sets. This meant that some templates could potentially match a given board position, but had features that prevented the lookup procedure from a successful match. To overcome this problem, an extended form of template was applied.

Definition 7.3.2 (extended template)**.** *An extended template is a template representation of the form $(C_X, \mathcal{S}, C, C_Y)$, where C is a carrier, C_X is a subset of C containing cells adjacent to one of Connect's side groups, C_Y is a subset of C containing cells adjacent to Connect's opposite side group and $\mathcal{S}$ is a set of sets such that each element $C_g \in \mathcal{S}$ is a subset of C of cells adjacent to a Connect stone group g that is **not** a Connect side group.*

The extended template was used to represent templates in a way that removed details about how stones were arranged about the neighborhood of carriers. In practice, all templates were first converted to extended templates and then converted back to ordinary templates by using a standard set of stone patterns to reconstruct the X, S and Y sets. This process standardized templates and minimized this inefficiency in representing templates.

7.3.2 Hex Solver 5

Detailed pseudo code describing how template matching tables were applied in an actual Hex solver is far too lengthy to present in Algorithm 7.3.8. Instead, high level function statements will be given to describe in general what should take place in the application of template matching tables. The following pseudo code involves two template matching tables called *WeakTemplates* and *StrongTemplates*. Each table represents templates for both players by flipping templates about the

short diagonal of the board to change the intended *Connect* player. For example, given a board position $board$ and player P, table *StrongTemplates* had a function lookupTemplate$(board, P)$ that applied the lookup procedure described by Equation (1) with template flipping to return either the empty set or a singleton whose element was a template where P was *Connect*. The *WeakTemplates* table had an extended lookup function lookupAllTemplate$(board, P)$ that could return a set of every matching weak template in the table, where P was *Connect*. Finally, given a carrier C, both template tables had a function insertTemplate$(board, C, P)$, that could deduce a template from these parameters and insert this template into its respective table.

The pseudo code presented in Algorithm 7.3.8 extends the pseudo code for *Hex Solver 4* (Algorithm 6.4.7). Unrelated features from Algorithm 6.4.7 have been suppressed with the $(\ldots)$ symbol. In addition, some feature from previous Hex Solver pseudo code has been given again here. In the statement block lines 6 to 11, if there is a strong template then Connection Utility values are assigned to cells in the template carrier and the search can return with this strong carrier. Otherwise, in the statement block line 12 to 15, weak template carriers are found that partially define a carrier and completely define a must-play region. At line 24, $carrier$ is weak, so the insertTemplate$(board, C, P)$ function derives a weak template and inserts it into the weak template table. At line 30, $carrier$ is strong, so the insertTemplate$(board, C, P)$ function derives a strong template and inserts it into the strong template table. Finally, the move generator defined in Section 6.3 was extended to also make use of templates. This extension is trivial as weak and strong templates were treated in exactly the same manner as prescribed for weak and strong threat patterns in Section 6.3.

Algorithm 7.3.8 Hex_Solver_5($board$) **Returns** (Carrier, Vector of utility values, Player colour)

```
1: carrier ← ∅
2: utilities[1 : board.size] ← 0;
3: ...
4: moves ← board.emptyCells {The initial must-play region.}
5: ...
6: for all (X, S, C, Y) ∈ StrongTemplates.lookupTemplate(board, board.notTurn)
   do
7:    for all i ∈ C do
8:       utilities[i] ← 1 {Handle utility values for carrier cells.}
9:    end for
10:   return(C, utilities, modeColour)
11: end for
12: for all (X, S, C, Y) ∈ WeakTemplates.lookupAllTemplate(board, board.notTurn)
    do
13:   carrier ← carrier ∪ C
14:   moves ← moves ∩ C
15: end for
16: for all (x, S, C, y) ∈ HSearch.WeakThreatPatternSet() do
17:   ...
18: end for
19: ...
20: while moves ≠ ∅ ∨ M ≠ ∅ do
21:   ...
22:   if modeColour = board.turn then
23:      ...
24:      WeakTemplates.insertTemplate(board, carrier, board.notTurn)
25:      return(carrier, Util, modeColour)
26:   else
27:      ...
28:   end if
29: end while
30: StrongTemplates.insertTemplate(board, carrier, board.turn)
31: return(carrier, utilities, modeColour) {Successful OR deduction.}
```

7.3.3 Performance Tests and Results

The following experiment compares the performance *Hex Solver 5* against the performance of *Hex Solver 5* for solving Hex. Beginning with the 3x3 Hex board the solver algorithms are compared for each increasing board size. The measure of performance to be used in each search is the number of nodes in the game-tree visited for a complete solution and the time taken on a 3GHz Intel Pentium 4 class computer.

Table 7.3.1 shows the performance results of *Hex Solver 5* on the problem of solving Hex boards inclusively in the size range 3x3 to 7x7, in comparison to the performances measured for *Hex Solver 4*. In addition to these results, *Hex Solver 5* was found to be an impractical search on the 8x8 Hex board.

Search	*Hex Solver 4*	*Hex Solver 5*
Nodes Visited (Time) 3x3	5 ($<$ 1 sec)	5 ($<$ 1 sec)
Nodes Visited (Time) 4x4	16 (1 sec)	12 ($<$ 1 sec)
Nodes Visited (Time) 5x5	157 (8 sec)	73 (4 sec)
Nodes Visited (Time) 6x6	14,451 (1759 sec)	3,211 (552 sec)
Nodes Visited (Time) 7x7	21,192,028 (2,420,021 sec)	369,717 (220,081 sec)

Table 7.3.1

The number of nodes the *Hex Solver 5* algorithm must visit and the time in seconds to completely solve Hex boards inclusively in the range 3x3 to 7x7, compared against the performances of *Hex Solver 4*.

7.3.4 Remarks

The performance results for *Hex Solver 5* show that 7x7 Hex can be solved in a search many times smaller than Hayward et al. reported for their *Solver* algorithm

in [27]. Given that *Hex Solver 5* extends on *Hex Solver 4* with a set of template matching tables and that *Hex Solver 4* does not perform quite as well as Hayward's *Solver* algorithm, one would anticipate that further improvements can yet be made to the *Hex Solver 5* algorithm.

An estimate of running cost for solving the 8x8 Hex board is needed to determine if such an experiment is feasible. Given that the last three running times were reliably measured and the running cost of solving an $n \times n$ board when $n = 0$ is zero seconds, the following growth equation was used to estimate the running cost in seconds:

$$f(n) = e^{P(n)} - 1, \text{ where } P(n) = An + Bn^2 + Cn^3 \tag{4}$$

Through a process of Gaussian Elimination the coefficients were found to be: $A = -3.719044503215379$, $B = 0.8727816943961655$ and $C = 0.012919055451145134$. The roots of $P(n)$ were found to be $\{0, 4.57033, 62.9874\}$, which are well outside of the domain that will be used in this estimation. According to Equation (4), an estimated time to solve 8x8 Hex would be about 2.9×10^8 seconds on a 3GHz Intel Pentium 4 class computer. The running time would be about 3360 days or 9 years. An attempt to solve some of the opening moves on the 8x8 board ran for more than a month, but there was no evidence that the search was close to solving any opening moves.

7.4 Optimization Attempts using Template Matching Tables

Given the performance of the *Hex Solver 5* algorithm and its simple application of template matching tables, the following series of additional applications of template matching tables were proposed.

1. *Apply iterative deepening in Hex solvers that use template matching tables for pruning.*

2. *Seed template matching tables for searches on $n \times n$ boards using templates found on $(n-1) \times (n-1)$ boards.*
3. *Use templates found on $(n-1) \times (n-1)$ boards in a move generator for searches on $n \times n$ boards.*

Apply iterative deepening in Hex solvers that use template matching tables for pruning: For this method, the *Hex Solver 5* algorithm was modified to perform depth limited searches. This modified search procedure, called *Depth Limit Hex Solver*, took as inputs a board position $board$ and a search depth limit $depth$ and had the same return value as *Hex Solver 5*. In addition, the *Depth Limit Hex Solver* algorithm was applied in an iterative deepening search algorithm [42]. Algorithm 7.4.9 gives a high level pseudo code for this iterative deepening Hex Solver.

Algorithm 7.4.9 Iterative_Deepening_Hex_Solver($board$) **Returns** (Carrier, Vector of utility values, Player colour)

```
1: for depth = 1 to board.size do
2:    (C, U, modeColour) ← Depth_Limit_Hex_Solver(board, depth)
3:    if modeColour ∈ {Black, White} then
4:       return(C, U, modeColour)
5:    end if
6: end for
```

The Hex Solver in Algorithm 7.4.9 takes advantage of the fact that threat patterns are found for board positions at various depth levels in game-trees. Given that Pattern searches are depth-first searches that rely on good move orderings, threat patterns for some board positions may be found using deeper than necessary searches. The Hex Solver in Algorithm 7.4.9 will find threat patterns for board positions using the smallest depth possible. Since the *Depth Limit Hex Solver* search procedure applies template matching tables, any threat patterns found with a search at depth d are reused in the search at depth $d+1$ to prune the search. That is, at each iteration Algorithm 7.4.9 will only search moves in must-play regions.

Algorithm 7.4.9 had performance results comparable to *Hex Solver 5* in solving the 6x6 and 7x7 boards. However, Algorithm 7.4.9 was also found to perform an impractical search on the 8x8 Hex board on a 3GHz Pentium 4 class computer.

Seed template matching tables for searches on $n \times n$ ***boards using templates found on*** $(n-1) \times (n-1)$ ***boards:*** Given a set of templates found by applying *Hex Solver 5* to an $(n-1) \times (n-1)$ board, each template was mapped to a new set of templates, called *seed templates*, that are placed in juxtaposition over the $n \times n$ board. These new templates were added to the template matching tables of a *Hex Solver 5* search to solve the $n \times n$ board. The aim was to prune subtree in searching $n \times n$ board positions given templates from searches on smaller board sizes.

This approach gave no improvement to the performance of *Hex Solver 5*. Additional testing found that the seed templates rarely matched $n \times n$ board positions. There are two reasonable explanations for this result. Firstly, the size of template matching tables grow exponentially with respect to board sizes and the number of seed templates were too few to effectively seed template matching tables. Secondly, the seed templates were highly specialized patterns for $(n-1) \times (n-1)$ boards and were unlikely to represent threat patterns on $n \times n$ board positions. It remains unclear as to how effective this approach will be for larger boards. However, as n becomes larger the ratio $n/(n-1)$ approaches closer to one, which implies that $(n-1) \times (n-1)$ boards are more likely to have similar features to $n \times n$ boards for large n. That is, one can reasonably suggest that this approach could be more effective for larger board positions.

Use templates found on $(n-1) \times (n-1)$ ***boards in a move generator for searches on*** $n \times n$ ***boards:*** For this approach, templates found by applying *Hex Solver 5* to a $(n-1) \times (n-1)$ board were used in an oracle to advise how moves should be ordered in searching $n \times n$ boards. For each $n \times n$ board position in a *Hex Solver 5*

search where all of the stones could be contained within an $(n-1) \times (n-1)$ sub-board, if a $(n-1) \times (n-1)$ template could be found that proved a move was winning on this sub-board then that move was given priority over moves not winning on the $(n-1) \times (n-1)$ sub-board and not already ordered using the *Pattern Feature* move generator from Section 6.3.

The application of this approach resulted in a lower performing Hex solver in comparison to the *Hex Solver 5*. That this approach was not effective further supports the hypothesis that templates matching $(n-1) \times (n-1)$ board positions are highly specialized and rarely match $n \times n$ board positions. Once again, one can reasonably suggest that this approach could be more effective for larger boards.

7.5 Conclusion

This chapter presented the application of template matching tables in pruning Pattern searches. In addition, it presented the *Hex Solver 5* algorithm along with its performance results for solving various board sizes. The performance test for *Hex Solver 5* showed that the application of template matching tables in a Hex solver could significantly reduce search times. The *Hex Solver 5* was able to solve 7x7 Hex in $369,717$ nodes, which is a significantly better result than the performance of $14,210,662$ nodes reported by Hayward et al. for their *Solver* algorithm in [27]. Despite many attempts to improve on the *Hex Solver 5* results, the problem of solving the 8x8 Hex board remains open. The frontier challenge for future Hex solving attempts will be to develop a Hex solving algorithm that can solve the 8x8 Hex board.

Part II

Machine Learning of Artificial Hex Players

Chapter 8

A Hex Player Overview

Thus far, the thesis has concentrated on the problem of solving Hex. A different problem to solving Hex is the problem of creating automatic Hex players. The object in solving this problem is to create players that possess a high probability of winning games. Hex players such as *Hexy* in [3, 4] and *Six* in [36] are strong players that apply alpha-beta searches. The competitive strength of *Hexy* and *Six* show that the application of alpha-beta searches in automatic Hex players is very practical. In addition, it would be illogical not to devise Hex players that utilize the methods that were effective in solving small Hex boards. The problem however, is that these methods apply to the Pattern Search algorithm and operate on special features that are unique to Pattern searches. A solution to this problem is available in [46], in the form of a Pattern Search algorithm with alpha-beta pruning, called the *Pattern-enhanced Alpha-Beta* search.

This chapter provides a literature review that relates the previous Hex solving techniques to a search called the *Pattern-enhanced Alpha-Beta* search that can be used by artificial Hex players. In addition, this chapter provides a lead into the problem of applying machine learning to derive good Hex playing policies, which is presented in the next chapter. Section 8.1 gives a pseudo code definition for the Pattern-enhanced Alpha-Beta search algorithm. In addition, this section provides some details on how to apply the present Hex solving techniques to this search. Section 8.3 discusses the consequence of applying template matching tables to this search and gives a lead on to the topic of the following chapter.

8.1 The Pattern-enhanced Alpha-Beta Search

A pseudo code equivalent to Rijswijck's *Pattern-enhanced Alpha-Beta* search algorithm is given in Algorithm 8.1.10. If lines 8 to 11 and lines 27 to 34 are removed from Algorithm 8.1.10, then the resulting algorithm will perform the same Pattern searches as the Pattern Search algorithm (Algorithm 5.1.1). Since previous chapters present details that describe how to extend the Pattern Search algorithm with search optimization techniques, there is no need to repeat those details here.

There are however some attributes relating to alpha-beta pruning in this algorithm that differ significantly from those same attributes in the Pattern Search algorithm. The attribute $modeColour$ is no longer a value restricted to the set $\{Black, White\}$. In fact, the $modeColour$ attribute is now a real number. Assuming $Black = -1$ and $White = 1$. If a threat pattern can be found for a given board position then $modeColour$ is either $Black = -1$ or $White = 1$, which means $modeColour$ has a game theoretic value. However, if no threat pattern can be found then $-1 < modeColour < 1$ is only the expected value of a given position.

The details associated with applying a *Connection Utility* move generator to Algorithm 8.1.10 are a little more complex than the details given for the *Hex_Solver_1* algorithm in Section 5.2.3. In Algorithm 8.1.10, the search returns at lines 10, 29 and 33 without a threat pattern and only an expected value of the given position. To apply a *Connection Utility* move generator to Algorithm 8.1.10, the returned tuples must also contain a *Connection Utility* vector and a static move generator will be needed to generate a vector of approximate *Connection Utility* values just before lines 10, 29 and 33. Method for generating values that can approximate *Connection Utility* values will be presented in the following chapter. All other details associated with the application of a *Connection Utility* move generator correspond with the details for the *Hex_Solver_1* algorithm in Section 5.2.3.

At lines 8 to 11 in Algorithm 8.1.10, if the search reaches its depth limit then the $Evaluate(board)$ function calculates the expected value of position $board$, such that $-1 < Evaluate(board) < 1$. Details about such evaluation functions will also be covered in the following chapter.

Algorithm 8.1.10 Pattern_$\alpha\beta$_Search($board$, α, β, $depth$) **Returns** (Carrier, Player colour)

```
1: carrier ← ∅
2: modeColour ← board.notTurn
3:
4: if board.isTerminal then
5:    modeColour ← board.winningPlayer
6:    return (carrier, modeColour)
7: end if
8: if depth = 0 then
9:    modeColour ← Evaluate(board)
10:   return (∅, modeColour)
11: end if
12:
13: moves ← board.emptyCells {The initial must-play region.}
14:
15: for m ∈ moves do
16:    board.playMove(m)
17:    (C, modeColour) ← Pattern_αβ_Search(board, −β, −α, depth − 1)
18:    board.undoMove(m)
19:
20:    if modeColour = board.turn then
21:       carrier ← {m} ∪ C {carrier, is now a weak threat pattern carrier.}
22:       return(carrier, modeColour)
23:    else if modeColour = board.notTurn then
24:       carrier ← carrier ∪ C {The union of carriers.}
25:       moves ← moves ∩ C {Update the must-play region.}
26:    end if
27:    α ← max(α, −modeColour)
28:    if −modeColour ≥ β then
29:       return (∅, α)
30:    end if
31: end for
32: if moves ≠ ∅ then
33:    return (∅, α)
34: end if
35: return(carrier, modeColour) {Successful OR deduction.}
```

8.2 Evaluation

The Pattern-enhanced Alpha-Beta search operates in one of two modes, *Pattern Search* mode and *Alpha-Beta Search* mode. When this search is in *Pattern Search* mode, the search is solving Hex using the Pattern Search algorithm, however, when this search is in *Alpha-Beta Search* mode the search is estimating the value of board positions using a standard alpha-beta search. The Pattern-enhanced Alpha-Beta search always begins in *Pattern Search* mode, but will switch to *Alpha-Beta Search* mode if the search reaches the depth cut-off point before it reaches a terminal position. In addition, the search will switch back to *Pattern Search* mode if it reaches a terminal position. The evaluation of the Pattern-enhanced Alpha-Beta search algorithm can be done in two parts, the evaluation of searches in the *Pattern Search* mode and the evaluation of searches in the *Alpha-Beta Search* mode.

Evaluation of Pattern Search mode The Pattern-enhanced Alpha-Beta search in *Pattern Search* mode is equal to a Pattern Search that is solving Hex. Values found in *Alpha-Beta Search* mode are never used by the search while in *Pattern Search* mode. This means that the performance results provided for the Pattern Search algorithm and its search optimization techniques in the previous chapters, also apply to this mode of the Pattern-enhanced Alpha-Beta search.

Evaluation of Alpha-Beta Search mode The Pattern-enhanced Alpha-Beta search in *Alpha-Beta Search* mode is equal to a standard alpha-beta search. Values found in *Pattern Search* mode are indeed used in the *Alpha-Beta Search* mode, but this is an advantage because the *Pattern Search* mode returns the game theoretic value of board positions. In the context of a standard alpha-beta search, a detailed evaluation of Hex players based on alpha-beta search is given in the following chapter.

8.3 Discussion

One can define the *Pattern-enhanced Alpha-Beta* search algorithm, given in Algorithm 8.1.10, as a knowledge base player should template matching tables with permanent store be applied to this algorithm. An interesting consequence in such a application is that the Pattern Search feature of Algorithm 8.1.10 will deduce new threat patterns during searches that can be added to the player's knowledge base and reused in future searches. Since a template matching table can store threat patterns for both players, a player that applies Algorithm 8.1.10 with template matching will be able to deduce new must-play regions in consecutive games against the same opponent. This means that the player will eventually deduce enough templates to avoid losing moves.

Since game-trees of the game of Hex have very large branching factors close to the root and threat patterns are easier to deduce towards the end of games, the challenge to create strong players based on Algorithm 8.1.10 will rely on the creation of functions that can accurately evaluate board positions. That is, the Pattern search feature of Algorithm 8.1.10 will not be very effective at the beginning of games. Rather, searches at the beginning of games will most likely be alpha-beta searches that rely on good board position evaluations to make strong moves. Therefore, a method is needed that can generate good evaluation functions for Hex players.

Chapter 9

Hex Evaluation Functions by Apprenticeship Learning

Artificial Hex players generally employ a depth-first search algorithm, such as a minimax search with alpha-beta pruning or the Pattern-enhanced Alpha-Beta search in Section 8.1. Such searches can be made to return a move within some practical period by setting search depth limits. The player is able to return good moves if the search can evaluate the expected probabilities of winning board positions at the search limits. This chapter explores an apprenticeship learning approach that takes advantage of a database of Hex games to derive functions that evaluate board positions for strong Hex players. This database contains games played between unevenly matched players where the stronger player was the winner. A reasonable conjecture is that the game theoretic value of positions in each game would be mostly increasing in favour of the stronger player. The apprenticeship learning approach presented here is able to show that this conjecture is plausible and can rapidly generate good quality evaluation functions.

Section 9.1 gives an introduction to the problem of estimating game theoretic values. In addition, relevant machine learning techniques are introduced and a hypothesis, about how game theoretic values map to board positions in certain games, is given. Section 9.2 gives an overview of game-tree searching and value functions for estimating game theoretic values. Section 9.3 gives a review on the *cross entropy* methods, which is a search method devised by Rubinstein in [41] and used here to solve the optimization for apprenticeship learning. Section 9.4 presents a detailed method and results for an experiment. In this experiment the present ap-

prenticeship learning method is compared against a temporal difference learning approach and local beam search approach.

9.1 Introduction

Artificial players of combinatorial games often employ minimax searches that are bound by practical search times. Instead of returning the game theoretic value for an input board position, these searches return an approximate value. The limits for such minimax searches are set at designated cut-off points where the search may employ a value function. Such value functions estimate the game theoretic value of board positions, in place of complete minimax searches [2]. In such minimax searches, the accuracy of return values will largely depend on the accuracy of their value functions [12][6]. Therefore, methods that can be used to derive accurate value functions for minimax searches are essential for creating strong players.

Players have complete information about board positions during games, however many artificial game players have fixed policies and do not utilize this information effectively. In contrast, a player that learns better move policies from past board positions makes more effective use of the available information. For example, a player based on minimax searching could learn the parameters for a value function from past positions and become a stronger player. The problem for such players is that without feedback concerning past board positions, the player will have no training on how to move in the future.

Reinforcement learning refers to processes where positive rewards are granted for solving a given problem, and a series of reward feedbacks are applied to learn a problem solving policy that maximizes the total expected reward [42]. A *policy* is a mapping from state to action and a policy will be called a *player policy* if it maps board positions to moves. A property of game playing processes is that players

provide a solution to a sequential decision problem with complete information. If players can be appropriately rewarded for visiting board positions, then a Markov Decision Process (MDP) can model the game playing process. A problem that can be associated with MDPs is to find an *optimal policy*, which is a policy that maximizes the total expected reward. Since reinforcement learning can be used to learn optimal policies for MDPs, given an appropriate reward system, reinforcement learning can also be used to learn player policies.

Apprenticeship learning is a form of reinforcement learning, where the rewards derived from the actions of an expert agent are used to find policies that mimic the expert [1]. This chapter explores apprenticeship learning in training an artificial game player from a database of Hex games, which was an idea suggested by Maire and Bulitko in [34]. Given a database of games played between unevenly matched players where the stronger player was the winner, a reasonable conjecture is that the game theoretic value of positions in each game would be mostly increasing in favour of the stronger player. Let $V^*(b)$ be the game theoretic value of position b and $(b_1, b_2, \ldots, b_m)$ be those board positions where the stronger player has the turn in a game from the given database. Our hypothesis is that $(b_1, b_2, \ldots, b_m)$ will satisfy $V^*(b_i) \leq V^*(b_{i+1})$ for most i, because the stronger player will be more successful at choosing winning moves. To test this hypothesis, apprenticeship learning will be used to find a player policy π, whose value function V^π has a maximum number of instances where it satisfies the condition $V^\pi(b_i) \leq V^\pi(b_{i+1})$ in the database of games. That is, V^π mimics the assumed behavior of V^*. An application of the *cross-entropy* method described in [19], will be used to solve the optimization for apprenticeship learning. Details of the cross-entropy method are given in Section 9.3.1.

9.2 Evaluation Functions for Games

Evaluation functions have been classified as either type-A or type-B [43]. A type-A evaluation function is a game-tree search that returns an estimate of the game theoretic value for an input board position. A type-B evaluation function makes use of board position features to deduce values without a game-tree search. For this discussion, a type-A function will be called a *dynamic* evaluation function and a type-B function will be called a *static* evaluation function.

A widely used dynamic evaluation function is the minimax search with alpha-beta pruning, called an *alpha-beta search* [12][6]. Although an alpha-beta search employs a pruning algorithm, a timely search in most game-trees can only be achieved by also restricting the search to a fixed depth. For those board positions that occur at the depth cut-off point, the search procedure has to employ a static evaluation function to estimate the value that would have been returned by the alpha-beta search beyond the depth cut-off point [2].

For many combinatorial games, one can derive a set of static evaluation functions, called *feature functions*, that each exploit a different board position feature. A weighted linear combination of feature functions is called a *committee* [22]. In such a committee, each feature function $a_i : \mathcal{B} \rightarrow \mathbb{R}$, maps the set of valid board positions to the real numbers and is called an *advisor*. Given a board position $b \in \mathcal{B}$, a vector of advisors $\mathbf{v} = \langle a_0, a_1, \ldots a_n \rangle$ and a weight vector $\mathbf{w}$, an evaluation function $V(b)$ is a committee of the form:

$$V(b) = \mathbf{w}^T \mathbf{v} \tag{1}$$

The value that a committee returns is called a *consensus*. The problem is to learn a weight vector $\mathbf{w}$ so that the consensus of $V(b)$ is an accurate estimate of the value that would have been returned by a full alpha-beta search from a board position b.

9.3 Apprenticeship Learning

Apprenticeship learning is the novel approach used in this experiment to find artificial game players. The apprenticeship learning approach used involves a *k*-vector of advisors $\mathbf{v}$ so that the evaluation function for a policy π is defined as $V^{\pi}(b) = \mathbf{w}_{\pi}^{T}\mathbf{v}$. Since the advisor vector $\mathbf{v}$ is fixed, the value function for policy π is described by the weight vector $\mathbf{w}_{\pi}$ in $\mathbb{R}^{k}$. Let G be the database of games between unevenly matched players, where the stronger player was the winner. For each game $g_i \in G$, let $B_i = \langle (b_1, b_2), (b_2, b_3), \ldots, (b_j, b_{j+1}), \ldots, (b_{m-1}, b_m) \rangle$ be the sequence of board position pairs from the sequence of board positions where it was the stronger player's turn. A weight vector $\mathbf{w}_{\pi}$ scores a reward point on each pair (b_j, b_{j+1}) if the value function $V^{\pi}(b) = \mathbf{w}_{\pi}^{T}\mathbf{v}$ satisfies the condition $V^{\pi}(b_j) \leq V^{\pi}(b_{j+1})$. Given the games in G and their corresponding sequences B_i, let B be the concatenation of all sequences B_i. In addition, let I be the Boolean indicator function (such that $I(true) = 1$ and $I(false) = 0$). We want to find a weight vector $\mathbf{w}$:

$$\arg\max_{\mathbf{w} \in \mathbb{R}^{k}} \sum_{(b_j, b_{j+1}) \in B} I(\mathbf{w}^{T}\mathbf{v}(b_j) \leq \mathbf{w}^{T}\mathbf{v}(b_{j+1})) \tag{2}$$

9.3.1 The Cross-Entropy Method

The cross-entropy method is a general purpose, population based search that can solve continuous multi-extremal problems [41]. The optimization problem given in Equation (2) will be solved by this method. Derived from Equation (2), the cross-entropy method applied to this problem makes use of Equation (3) as a performance function for weight vectors.

$$S(\mathbf{w}) = \sum_{(b_j, b_{j+1}) \in B} I(\mathbf{w}^{T}\mathbf{v}(b_j) \leq \mathbf{w}^{T}\mathbf{v}(b_{j+1})) \tag{3}$$

At the beginning of every iteration the search generates a random population W of n weight vectors having a multivariate normal distribution (see Algorithm 9.3.11, line 5). Where the performance function $S(\mathbf{w})$ is defined by Equation (3), W is ordered such that $S(\mathbf{w}_j) \geq S(\mathbf{w}_{j+1})$ (see line 6). The *elite* population is a proper subpopulation of W of the best performing individuals. Given a proportion factor r of the population to be selected as elite, the elite population is $E = \{\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{\lfloor nr \rfloor}\}$ (see line 7). From the elite, the search finds a mean weight vector $\boldsymbol{\mu}_t$ and covariance matrix $\boldsymbol{cov}$ (see lines 9 and 10). On the next iteration, the search uses this mean weight vector and its covariance matrix to generate the next population W of n weight vectors, which also has a multivariate normal distribution. The search iterates this procedure until, at some iteration t where $\|\boldsymbol{\mu}_t - \boldsymbol{\mu}_{t-1}\| < \epsilon$, the search terminates and returns the weight vector $\boldsymbol{\mu}_t$. Figure 9.3.1 gives a pictorial overview of how the Cross-Entropy method might solve such an optimization problem.

Algorithm 9.3.11 : $\boldsymbol{\mu}_t =$ Cross-Entropy-Search(n, r)

Require: Input n is the search population size and input r is a proportion in $(0, 1]$ of the population that will be selected as *elite* on each iteration.

1: $\boldsymbol{\mu}_{t-1} \leftarrow$ ones(k) {Initialize with ones so that the condition at line 4 is satisfied.}
2: $\boldsymbol{\mu}_t \leftarrow$ ones(k)$\times\epsilon$ {Initialize very close to a zero vector.}
3: $\boldsymbol{cov} \leftarrow 4 \times I_k$ {Initialize the covariance matrix. I_k is the $k \times k$ identity matrix.}
4: **while** $\|\boldsymbol{\mu}_t - \boldsymbol{\mu}_{t-1}\| \geq \epsilon$ **do**
5: $W \leftarrow$ mvnrnd($\boldsymbol{\mu}_t, \boldsymbol{cov}, n$) {A random number generator is used to create a population of n weight vectors in $\mathbb{R}^k$, having a multivariate normal distribution.}
6: $W \leftarrow$ sort($W, (S(\mathbf{w}_j) \geq S(\mathbf{w}_{j+1}))$ {Sort W descending with respect to the reward function defined by Equation (3).}
7: $E \leftarrow \{\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{\lfloor nr \rfloor}\}$ {Define the *elite* population E as the first $\lfloor nr \rfloor$ individuals of the ordered population W.}
8: $\boldsymbol{\mu}_{t-1} \leftarrow \boldsymbol{\mu}_t$
9: $\boldsymbol{\mu}_t \leftarrow$ mean(E) {Find the new mean weight vector in E.}
10: $\boldsymbol{cov} \leftarrow$ cov(E) {Find the new covariance matrix from E}
11: **end while**

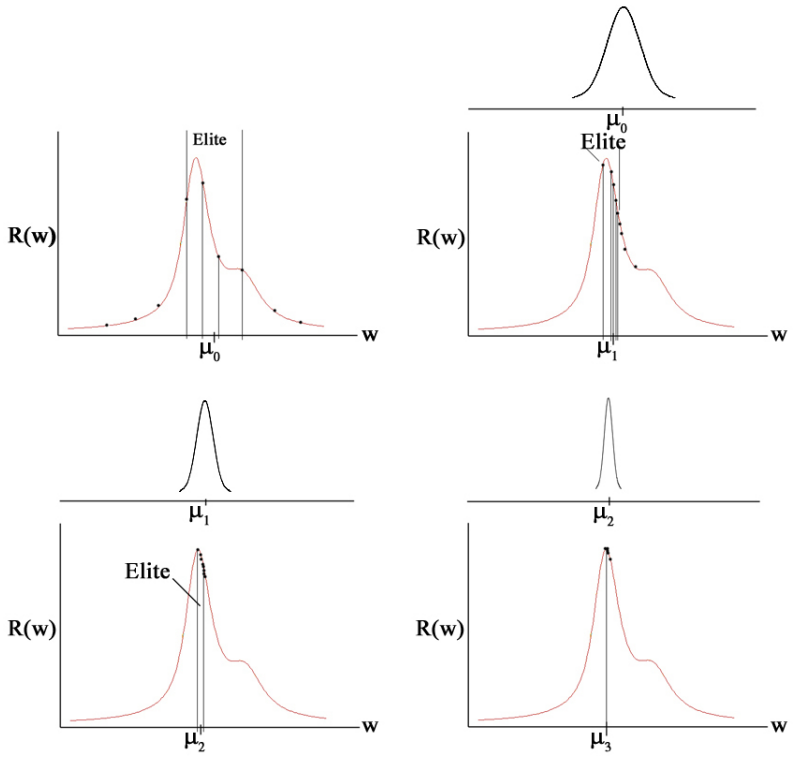

Figure 9.3.1: Top-left: Given a population at $t = 0$ in a domain w and a performance function $R(w)$ the Cross-Entropy method selects an elite subpopulation whose mean individual is μ_0. Top-right and bottom-left: The mean individual of the elite is used to generate the next population according to a Gaussian random generator. The performance function $R(w)$ is used to find the new elite subpopulation. Bottom-right: Eventually the Cross-Entropy method converges and returns the mean individual μ_3 at the limit.

9.4 Experiments and Results

The aim of these experiments is to test the assumption that the game theoretic value of board positions will be mostly increasing in favour of the stronger player in games played between unevenly matched players where the stronger player was the winner. In making this assessment, the apprenticeship learning method will be used to learn evaluation functions for a set of artificial Hex players. The objectives will be to compare the performances of these players against the performances of Hex players derived by Temporal Difference learning and by a Stochastic Local Beam search. The performances of these players will be determined via tournaments against a benchmark player. The evaluation functions for each player will be a consensus of sixteen advisors that have been categorized as one of the following types, *Local Mobility*, *Distance*, *Flow*, *Shortest Path*.

Local Mobility Advisors

For Hex, *local mobility* refers to the number of empty cells in the neighbourhood of a given cell. The neighbourhoods of cells will differ for each player, because stone groups affect cell adjacency and each player has a different arrangement of stones. If a cell x is not adjacent to any stones then both players will define the neighbourhood of that cell as the immediate neighboring cells. The left of Figure 9.4.2 shows the neighbourhood of empty cell x with lightly shaded cells.

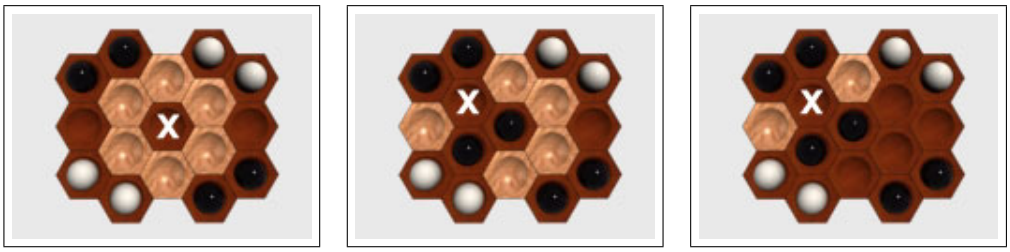

Figure 9.4.2: Left: The neighbourhood of empty cell x, which is not adjacent to any stones. Centre: The neighbourhood of empty cell x from Black's perspective, given x is adjacent to a Black stone group. Right: The neighbourhood of empty cell x from White's perspective, given x is adjacent to a Black stone group.

If x is an empty cell that is adjacent to a Black stone group (see centre of Figure 9.4.2), then from Black's perspective the neighbourhood of x is the set union of the empty-cell neighbourhood of the Black stone group and the empty cells adjacent to x. But from White's perspective, the neighbourhood of x is just the empty cells adjacent to x (see right of Figure 9.4.2). These neighbourhood rules can be applied on the alternate coloured positions by swapping the player's roles.

Given this approach for deciding group neighbourhoods, let $N_B(x)$ be the function that returns the neighbourhood of a cell x from the Black player's perspective and $N_W(x)$ be the function that returns the neighbourhood of a cell x from the White player's perspective. In addition, if x is a Black side then $N_B(x)$ returns the neighbourhood of x, but $N_W(x)$ returns the empty set. if x is a White side then $N_W(x)$ returns the neighbourhood of x, but $N_B(x)$ returns the empty set. Given C is the set of empty cells on given board positions b, four advisors were defined:

1. $a_1(b)$: The advisor that returned: $\sum_{c \in C} |N_B(c)|$.
2. $a_2(b)$: The advisor that returned: $\sum_{c \in C} |N_W(c)|$.
3. $a_3(b)$: The advisor that returned the number of empty cells that scored: $\max_{c \in C} |N_B(c)|$.
4. $a_4(b)$: The advisor that returned the number of empty cells that scored:

$\max_{c \in C} |N_W(c)|$.

Distance Advisors

For Hex, *distance* was treated as a metric evaluation of how far a cell was to the sides of a board position. The distance based advisors extracted Shannon graphs from given board positions and used a distance metric called *Queen-Bee* distance to measure the length of cell-to-target paths [45]. For each board position, two Shannon graphs were defined such that Black was *Connect* in one and White was *Connect* in the other. In the Shannon graphs where Black was *Connect*, the vertices were either empty cells or Black side groups, such that the side groups were designated as targets. Let any such Shannon graph where Black was *Connect* be $G_B(V_B, E_B)$. A graph $G_B(V_B, E_B)$ was defined such that the adjacency function for this graph was the neighbourhood function $N_B(v)$, where $v \in V_B$ (see *Local Mobility Advisors* for $N_B(v)$). Any Shannon graph $G_W(V_W, E_W)$, where White was *Connect* was similarly defined using White's neighbourhood function $N_W(v)$. In this experiment, the *Queen-Bee* distance was used measure the length of cell-to-target paths in the graphs $G_B(V_B, E_B)$ and $G_W(V_W, E_W)$. .

The Queen-bee distance function relies on the assumption that an opponent will always move so as to maximize the player's connecting distance, which implies that the player should look for strategies using second best moves. The Queen-bee distance function explores a series of second best moves to determine how far two vertices are apart. In a Shannon graph $G_P(V_P, E_P)$ where $P \in \{B, W\}$, the *Queen-bee distance* function between vertices $v_a, v_b \in V_P$ is:

$$QB_P(v_a, v_b) = \begin{cases} 0, & \text{If } v_a = v_b; \\ QB_P(v_x, v_b) + 1, & \text{Otherwise.} \end{cases} \quad (4)$$

Where, $QB_P(v_x, v_b)$ is the second smallest or only element in $\{QB_P(v_i, v_b) \mid v_i \in N_P(v_a)\}$.

Given X and Y are the targets vertices in $G_P(V_P, E_P)$, two distance metrics were used in this experiment. The first distance metric was $QB_P(v, X)$ and the second was $QB_P(v, Y)$, where $v \in V_P$. From these two distance metrics, the *Queen-bee distance utility* function $U_P(v)$ was defined as:

$$U_P(v) = QB_P(v, X) + QB_P(v, Y) \tag{5}$$

The function $U_P(v)$ measures how far v is away from both of the P player's targets. In addition, the *Potential* function in this experiment was given as:

$$\mathcal{P}(V_P) = \min_{v \in V_P} U_P(v) \tag{6}$$

The following advisors converted given board positions b to Shannon graphs $G_B(V_B, E_B)$ and $G_W(V_W, E_W)$:

1. $a_5(b)$: The advisor that returned $\mathcal{P}(V_B)$.
2. $a_6(b)$: The advisor that returned $\mathcal{P}(V_W)$.
3. $a_7(b)$: The advisor that returned the number of vertices in V_B whose Queen-bee distance utility is equal to $\mathcal{P}(V_B)$.
4. $a_8(b)$: The advisor that returned the number of vertices in V_W whose Queen-bee distance utility is equal to $\mathcal{P}(V_W)$.

Flow Advisors

Flow networks can be used to evaluate the Shannon graph of Hex board positions. A *flow network* is a directed graph where each edge has a flow capacity. If flow networks are treated as electronic circuits then flow in these networks can be defined with standard electronic formulas and quantities [21].

Three fundamental quantities in electronic circuits are: *Current* (I), *Potential* ($|\Delta V|$) and *Resistance* (R). The relationship for these quantities is given by Ohm's law:

$$I = \frac{|\Delta V|}{R} \tag{7}$$

An electronic circuit generally involves a source of electricity connected to an electric device by two terminals (see Figure 9.4.3). The purpose of the source is to deliver an electric *Potential* ($|\Delta V|$) to do work. At one terminal the source delivers a Potential V_+ and at the other terminal it delivers the Potential V_-. Since the circuit is a closed system, the device only reacts to the Potential difference $|\Delta V| = |V_+ - V_-|$. Given this Potential difference, the device allows electric charge to flow from one terminal to the other. In the circuit of Figure 9.4.3, *Current* (I) is the flow of electric charge that the device permits, given that the device has a level of *Resistance* (R) opposing the flow.

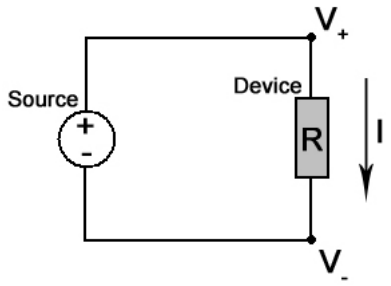

Figure 9.4.3: A simple circuit where an electric source delivers a work potential to a device, the device allows an electric current flow, which is determined by the resistance of the device.

A *resistor* is a two-terminal device that has a preset Resistance to the flow of electric charge. Resistors connected in a circuit form a *resistor network*, which is a graph where the edges are resistors and the vertices are junctions. Given a

resistor network where the Resistance at each edge is known, a problem is to find the total Resistance between two junctions, called *terminals*. This problem can be solved with a sequence of steps that begins by applying a source Potential to the terminals of the network. Resistor networks where at least two terminals have source Potentials are flow networks. In such flow networks, Current defines flow and Kirchoff's Current law can be applied [21].

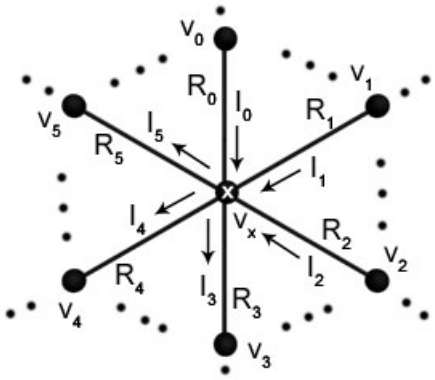

Figure 9.4.4: A junction x in a resistor network, whose potential is V_x.

Given junction x in Figure 9.4.4 and Currents $I_0, \ldots, I_5$ flowing in and out of junction x, Kirchoff's Current law is:

$$\sum_{i=0}^{5} I_i = 0 \tag{8}$$

Provided that the terminals in the network are set to different Potentials, the Potential at every other junction can be found [32]. Ohm's law can be substituted into equation (8) to derive an equation of unknown Potentials:

$$V_x = \frac{1}{K} \sum_{i=0}^{5} \frac{V_i}{R_i} \text{ where } K = \sum_{i=0}^{5} \frac{1}{R_i} \tag{9}$$

A similar equation can be found at each junction. The complete set of such equations forms a system of linear equations that can be solved using Gaussian elimination[32]. Once a Potential for each junction is known, Kirchoff's Current law can be used to find the Current flow at any one of the two terminals. Given the Current flow at a terminal and the source Potential, Ohm's law can be applied to give the total Resistance of the network.

In this experiment, the advisors based on flow converted given board positions to Shannon graphs and extended these Shannon graphs to resistor networks. For each board position b, two Shannon graphs $G_B(V_B, E_B)$ and $G_W(V_W, E_W)$ were defined according to the method used by the distance advisors. The graphs were respectively extended to resistor networks $\vec{G_B}(\vec{V_B}, \vec{E_B})$ and $\vec{G_W}(\vec{V_W}, \vec{E_W})$, such that every edge had the same Resistance and the targets in each graph were the terminals. For each graph, a source Potential was applied to the terminals. The two flow advisors were:

1. $a_9(b)$: The advisor that returned the total network Resistance of $\vec{G_B}$.
2. $a_{10}(b)$: The advisor that returned the total network Resistance of $\vec{G_W}$.

Shortest Path Advisors

In a Shannon game where *Connect* wins, *Connect's* winning path will be a shortest target-to-target path on the penultimate position and possibly in previous positions. This property suggests that a good policy for a *Connect* player should aim at maximizing the number of shortest target-to-target paths throughout games, while a good policy for a *Cut* player should aim at minimizing the number of such paths.

For each board position b, Shannon graphs $G_B(V_B, E_B)$ and $G_W(V_W, E_W)$ were defined according to the method used by the distance advisors. For a Shannon graph

$G_P(V_P, E_P)$, where $P \in \{B, W\}$, the advisors extract the set of shortest target-to-target paths $\mathcal{T}_P$ and for each non-target vertex $v \in V_P$ calculate a *shortest path utility*, which is the number of paths in $\mathcal{T}_P$ where v is a member. The shortest path utility of $v \in V_P$ is:

$$SPU_P(v) = |\{t \mid t \in \mathcal{T}_P, v \in t\}| \tag{10}$$

In addition, the *shortest path potential* is given as:

$$SPP(V_P) = \max_{v \in V_P} SPU_P(v) \tag{11}$$

The four advisors based on shortest paths were:

1. $a_{11}(b)$: The advisor that returned the shortest target-to-target path length in G_B.
2. $a_{12}(b)$: The advisor that returned the shortest target-to-target path length in G_W.
3. $a_{13}(b)$: The advisor that returned $SPP(V_B)$.
4. $a_{14}(b)$: The advisor that returned $SPP(V_W)$.
5. $a_{15}(b)$: The advisor that returned the number of vertices whose shortest path utility is equal to $SPP(V_B)$.
6. $a_{16}(b)$: The advisor that returned the number of vertices whose shortest path utility is equal to $SPP(V_W)$.

The sixteen advisors presented hereto were used to form a vector of advisors:

$$\mathbf{v} = \langle a_1, a_2, \ldots a_{16} \rangle \tag{12}$$

For each set of weight vectors found using Apprenticeship learning, temporal difference learning and by Stochastic Local Beam Search, an evaluation function

was defined according to Equation (1) and the vector of advisors $\mathbf{v}$. For each evaluation function, a player was defined that performs single level alpha-beta searches with end-of-game pruning via H-Search and applies the evaluation function at depth cut-off points.

9.4.1 The Benchmark Player

The benchmark player for this experiment used a single level minimax search to decide best moves. A Resistor network evaluation function was applied at the depth cut-off positions in this search. Given the Resistor network advisors $R_B(b) = a_9(b)$ and $R_W(b) = a_{10}(b)$, the evaluation function used by the benchmark player was:

$$V_R(b) = \ln(R_B(b)/R_W(b)) \tag{13}$$

This particular treatment of resistance values in evaluating Hex board positions has also been applied in strong Hex players, such as *Hexy* and *Six* [3, 4][36]. Let the benchmark player be P_R.

9.4.2 Application of Apprenticeship Learning

In this experiment, A database of 237 games was generated from a large set of unevenly matched artificial Hex players in tournament. These uneven Hex players were created using modified versions of the open source code for Metis' *Six* player, which is the player that won first place for Hex at the ICGA computer Olympiad [36]. The *Six* player uses an alpha-beta search algorithm that applies a resistor network evaluation function at depth cut-off points. Different versions of the Six player were created by applying different depth cut-offs and different move set sizes. Player versions that were allowed shallow depth cut-offs and small move sets were treated as weaker than those player versions that were allowed both deeper depth cut-offs and larger move sets. A set of five Hex players were defined and in each match a small random variable was added to board evaluations made

by the weakest player so that the game played would probably be unique. The Apprenticeship leaning method described in Section 9.3 was used to derive a list of weights $\mathcal{W}_{ARL}$ from this database of games, such that $\mathbf{w}_j \in \mathcal{W}_{ARL}$ was found at iteration j in the Cross-Entropy search. The Cross-Entropy parameters were a total population size of $n = 20000$ weights and an elite population size defined as a proportion $r = 0.1$ of the total population. This experiment took a total of 9 minutes to terminate after 50 iterations on a 3GHz Intel Pentium 4 class machine.

9.4.3 Apprenticeship Learning Results

Given the vector of advisors $\mathbf{v}$ from Equation (12), Equation (1) was used to define a committee $V_j(b)$ for each weight vector $\mathbf{w}_j \in \mathcal{W}_{ARL}$. Each $V_j(b)$ was used to evaluate depth cut-off board positions in single level minimax searches, which were the searches player P_j used to decide best moves. For each fifth iteration in $1 \leq j \leq 50$, the benchmark player P_R played 20 games against player P_j beginning from random positions. Figure 9.4.5 shows the results for this experiment in a stacked bar graph, where each dark bar represents the number of games won by players P_j.

Figure 9.4.6 shows the performance of the best performing elite weight at each iteration j. The performance is given as percentages of perfect performance. This plot reveals that the Cross Entropy method converges too quickly. Slightly better performances can be achieved by multiplying the covariance matrix in Algorithm 9.3.11 with a random value close to but larger than one. A comparison of Figure 9.4.5 with Figure 9.4.6 shows that the tournament results have a pattern similar to the convergence of the Cross Entropy method. This implies that the optimization formula in this Apprenticeship Learning method extracts a critical feature in representing the strength of Hex players.

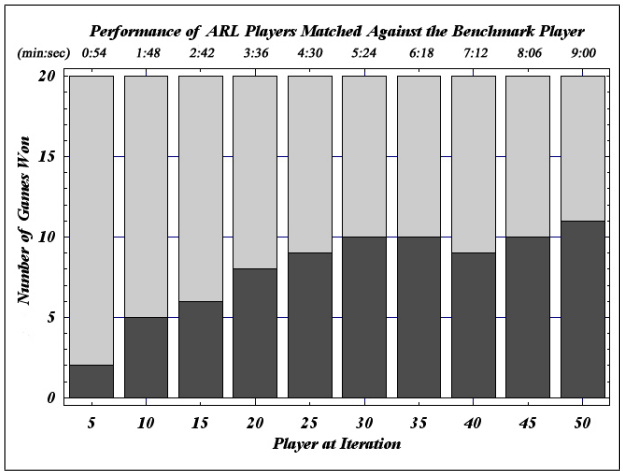

Figure 9.4.5: The tournament results of players P_j that was derived using Apprenticeship Learning on a 3GHz Intel Pentium 4 machine at iteration j and at time (minutes:seconds) in 20 games against the benchmark player P_R.

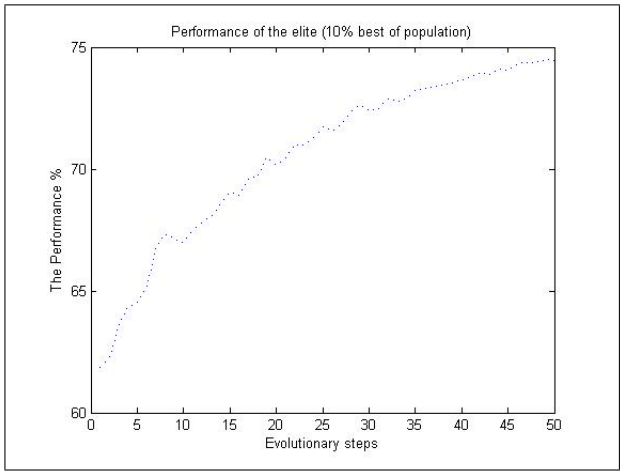

Figure 9.4.6: The performance of elite weights in the Cross Entropy method at iteration j in solving the optimization problem for this Apprenticeship Learning.

9.4.4 Application of Temporal Difference Learning

An objective in this experiment is to compare the results of this Apprenticeship Learning approach with the performance of other inductive learning methods on the same problem. The two other leaning method used in this experiment are Temporal difference learning and Stochastic Local Beam search. Temporal difference learning addresses the problem of predicting future outcomes from past events [44]. A *temporal difference learning* method predicts the outcome in choosing a state, from past prediction errors for that state and its successors.

The Family of Temporal Difference Learning Procedures: $\mathbf{TD}(\lambda)$

Temporal difference learning procedures can be expressed as rules for updating a weight vector $\mathbf{w}$ [44]. Predictions are made from sequences of observations of the form $o_1, o_2, \ldots, o_m, R$, where each o_t is a observation made at time t and R is the sequence outcome. The predictions $P_1, P_2, \ldots, P_m$, where P_t is a prediction about observation o_t, are each an estimate of R. Predictions are weighted as a function of time since those predictions were made. Predictions about observations made k iterations in the past are weighted λ^k, where $0 \leq \lambda \leq 1$. Given α is the learning rate and $\nabla_{\mathbf{w}} P_k$ is a vector of partial derivatives of P_k with respect to $\mathbf{w}$, the temporal difference learning procedure $TD(\lambda)$ is:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(P_{t+1} - P_t) \sum_{k-1}^{t} \lambda^{t-k} \nabla_{\mathbf{w}} P_k \tag{14}$$

$\mathbf{TD}(\mathbf{0})$ in Learning Evaluation Functions for Hex

For this experiment a $TD(0)$ learning procedure was used to learn a set of weight vectors for evaluating Hex board positions. The procedure for $TD(0)$ was:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(P_{t+1} - P_t) \nabla_{\mathbf{w}} P_t \text{ where } \mathbf{w} \in \mathbb{R}^{16} \tag{15}$$

Baxter et al. showed that $TD(\lambda)$ learning procedures could be coupled to game-tree searches with an algorithm they called TD-Leaf(λ) [7][8]. For this experiment, an algorithm based on the TD-Leaf(λ) algorithm was used to apply the $TD(0)$ learning procedure from Equation (15) to Hex. In applying $TD(0)$ to Hex, predictions were made about positions in games of the form $b_1, b_2, \ldots, b_m, R$, where b_1 was the root position in the game-tree, b_t was a board position at time t and R was the game theoretic value at a terminal position b_m. Given the vector of advisors $\mathbf{v}$ (see Equation (12)) and a weight vector $\bar{\mathbf{w}}$ found at the last update, Equation (1) was used to define an evaluation function $\bar{V}(b)$. The prediction for each board position b_t was $P_t = \bar{V}(b_t)$. The players in the TD-Leaf algorithm each performed single level minimax searches and invoked the most recent evaluation function $\bar{V}(b)$ at the cut-off positions. The $TD(0)$ procedure was used to update $\mathbf{w}$ at each board position in games between these two players.

In this experiment, the players in the TD-Leaf algorithm made a total of 47000 moves over 3500 games. At each move, Equation (15) was applied to update the weight vector $\mathbf{w}$ and this new weight vector was added to a list of weight vectors $\mathcal{W}_{TD}$. The learning rate α was set $\alpha = 0.05$. This experiment took 793 hours to generate the results on a 3GHz Intel Pentium 4 class machine.

9.4.5 Temporal Difference Learning Results

Given the vector of advisors $\mathbf{v}$ from Equation (12), Equation (1) was used to define a committee $V_j(b)$ for each weight vector $\mathbf{w}_j \in \mathcal{W}_{TD}$. Each $V_j(b)$ was used to evaluate depth cut-off board positions in single level minimax searches, which were the searches player P_j used to decide best moves. For each 4700^{th} iteration in $1 \leq j \leq 47000$, the benchmark player P_R played 20 games against player P_j beginning from random positions. Figure 9.4.7 shows the results for this experiment in a stacked bar graph, where each dark bar represents the number of games won

by players P_j.

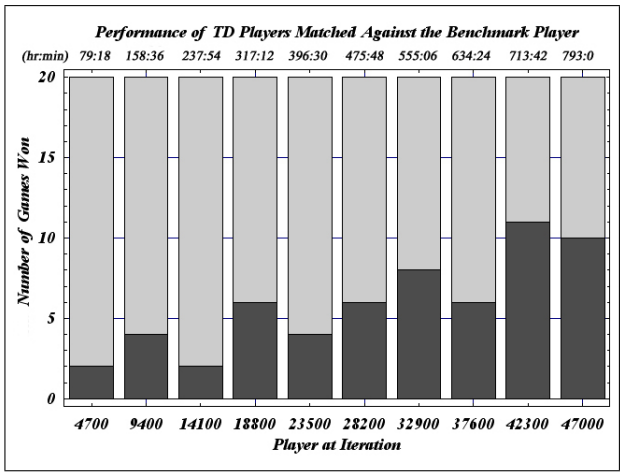

Figure 9.4.7: The tournament results of players P_j that was derived using Temporal Difference Learning on a 3GHz Intel Pentium 4 machine at iteration j and at time (hours:minutes), in 20 games against the benchmark player P_R.

9.4.6 Application of Stochastic Local Beam Search

The apprenticeship Learning approach presented herein was also compared against a Stochastic Local Beam search. A *Local Beam* search is a form of stochastic hill climbing that begins with k random states and generates the complete set of successors. If any one of the successors is the goal state then the algorithm terminates, otherwise it repeats this process on the k best performing successors [42]. A *Stochastic Local Beam* search is a Local Beam search where each of the k successors are chosen at random, with a probability proportional to that successor's performance value.

The search used for this experiment was a specialized form of Stochastic Local Beam search. The aim of the search was to find weight vectors of the form $\mathbf{w} \in \mathbb{R}^{16}$ that could be used to define effective evaluation functions for artificial Hex players. The performance value of a weight vector $\mathbf{w}$ was defined as the rating of the player,

whose evaluation function was defined by this $\mathbf{w}$. The ratings of players were determined via tournaments. The search began with $t = 1$ and a set of weight vectors $\mathcal{W} = \{\mathbf{w}_1\}$, where $\mathbf{w}_1$ was selected at random:

1. With the best performing weight vector $\mathbf{w}_q \in \mathcal{W}$, the search generated a weight vector $\mathbf{w}_t = \mathbf{w}_q + \mathbf{z}$, where $\mathbf{z}$ was a vector of small random values in an interval of reals $[-z, z]$. In this experiment, a good z was found empirically to be $z = 0.15$.
2. From $\mathbf{w}_t$ and advisor vector $\mathbf{v}$, Equation (1) was used to define an evaluation function $V(s)_t$.
3. This evaluation function was used by a Hex player p_t that performed single level minimax searches and invoked $V(b)_t$ at depth cut-offs. A set of players $\mathcal{P}_t$ was similarly defined for the weight vectors in $\mathcal{W}$.
4. If the player p_t won a game of Hex against every player in $\mathcal{P}_t$, then $\mathbf{w}_t$ was added to $\mathcal{W}$ as the highest performing weight in that set.
5. $t \leftarrow t + 1$ and go to 1.

In this experiment, the search was terminated after a total of $t = 575$ iterations. At each iteration the best performing weight vector in $\mathcal{W}$ was added to a list of weight vectors $\mathcal{W}_{LBS}$. This experiment took a total of 1344 hours to generate the following results on a 3GHz Intel Pentium 4 class machine.

9.4.7 Stochastic Local Beam Search Results

Given the vector of advisors $\mathbf{v}$ from Equation (12), Equation (1) was used to define a committee $V_j(b)$ for each weight vector $\mathbf{w}_j \in \mathcal{W}_{LBS}$. Each $V_j(b)$ was used to evaluate depth cut-off board positions in single level minimax searches, which player P_j used to decide best moves. For each 28^{th} iteration in $1 \leq j \leq 575$, the benchmark player P_R played 20 games against player P_j beginning from random

positions. Figure 9.4.8 shows the results for this experiment in a stacked bar graph, where each dark bar represents the number of games won by players P_j.

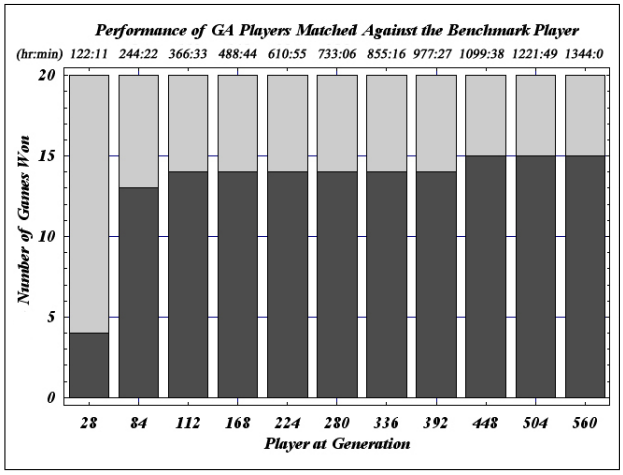

Figure 9.4.8: The tournament results of players P_j that was derived using Stochastic Local Beam searches at iteration j and at time (hours:seconds), in 20 games against the benchmark player P_R.

9.5 Conclusion

The problem of solving the game of Hex for arbitrary board sizes is intractable and any general rules concerning how game theoretic values can be mapped to board positions are valuable for creating Hex playing policies. Given a database of games played between unevenly matched players where the stronger player was the winner, the hypothesis that the game theoretic value of positions in each game would be mostly increasing in favour of the stronger player, was made. There was no prerequisite that the players be strong by human standards, only that the players were unevenly matched. To test this hypothesis, apprenticeship learning was applied to learn a set of Hex playing policies, where the premise for learning was the given hypothesis. If the method for apprenticeship learning failed to yield a set of Hex playing policies with increasing performance then this hypothesis about values

of board positions in the chosen database of games could be falsified. However, the results in Figure 9.4.5 shows that the method for apprenticeship learning did yield a set of Hex playing policies with increasing performance in games played against a benchmark player. The experiment supports the hypothesis for the chosen database of games.

This experiment also shows that players derived through the given apprenticeship learning method have comparable performances to players derived using Temporal Difference Learning and Stochastic Local Beam search. With the exception that the Stochastic Local Beam search did generate some players that performed better. The main advantage of the apprenticeship learning method was the rapid generation of weights. The application of the Cross-Entropy method to solve the optimization for apprenticeship learning took only 9 minutes to terminate in comparison to 793 hours for temporal difference learning and 1344 hours for the stochastic local beam search. The present apprenticeship learning method takes advantage of large game databases that are readily available from Internet game sites. The deliverable from this investigation is a rapid approach for generating good quality evaluation functions for artificial Hex players.

Chapter 10

Conclusion and Future Work

The game of Hex has very simple rules, but is very difficult to solve. The problem of solving Hex was proven in [40] to be PSPACE-complete, which implies that this problem is NP-Hard. Although Hex is difficult to solve, there are many good problem reduction methods. This work is the result of an investigation into a number of problem reduction methods for solving Hex in the application of Hex solving programs and artificial Hex players.

A view taken in this research, was that random-turn Hex games could model competitions in a communication network where an *attacker* systematically disables network routers and a network *administrator* would select and secure routers so as to maintain certain critical connections. the search techniques presented in this thesis could be applied in a strategy for the administrator. This view is supported by the application of two-player game theory to communication networks in [33] and the reduction and analysis of competitions in complex networks to discrete finite two-player games in [14].

In addition, Hex is a combinatorial game that can be reduced to a canonical form, called an *LR game* [15]. The Hex solving techniques in this work can also be modified to solve Hex in its *LR game* form. The implication is that the solving techniques presented in this work can be modified and applied effectively to many other combinatorial games, because every combinatorial game can be reduced to an *LR game*. Given the strong prospect that the solving techniques in this thesis

can be applied to other combinatorial games, they would have an application in the game industry. In particular, MMOPGs (*Massively Multi-player Online Puzzle Games*) offer networking environments where many users subscribe to play puzzle and board games. In such environments, users may opt to compete against an artificial game player. The techniques presented in this work could be modified and applied in such artificial game players.

Section 10.1 gives a summary of the contributions made in this thesis and Section 10.2 discusses some directions for future research.

10.1 Contributions

The aims of this work was to solve Hex for small board sizes and create artificial Hex players for larger board sizes. On the problem of solving Hex for small board sizes, this work investigated sub-game deduction using the H-Search, *Pattern Search* and sub-game reuse through template matching tables and made a number of contributions that fulfilled the first research aim:

1. In Section 5.2, a move generating algorithm that utilized features from terminal board positions in Pattern searches to evaluate the utility of cells on non-terminal positions.
2. In Section 6.1, a revised sub-game deduction procedure for the H-Search algorithm that significantly reduced the running cost of the H-Search algorithm.
3. In Section 6.4, a first independent confirmation of results reported by Hayward et al. for their *Solver* program in [27]. A Hex solving algorithm that successfully identified a combination of search optimization methods necessary for solving the 7x7 Hex board.
4. In Sections 7.1 and 7.2, a generalized form of transposition table, called a *template matching table* that made use of multi-target sub-games called templates to prove that certain board positions were winning.

5. In Section 7.3, a Hex solving algorithm that uses template matching tables to significantly reduce the time taken to solve 7x7 Hex. An estimate running time for solving the 8x8 Hex board on a 3GHz Intel P4 Computer.

For the problem of creating strong artificial Hex players for larger boards, this work investigated machine learning approaches in the derivation of Hex playing policies. The contribution of this investigation is presented in Chapter 9. This contribution is an apprenticeship learning approach that rapidly generated high quality evaluation functions for artificial Hex players from a database of games.

10.2 Future Work

An obvious extension to this research is to further investigate Hex solving techniques and attempt to solve the 8x8 Hex board. There is still more work that can be done to reduce the problem of solving Hex. A possible optimization for the Pattern Search algorithm might explore ways to incorporate the AND deduction rule into the threat pattern deduction process of the Pattern Search algorithm. Given a Pattern Search with template matching tables, the search might routinely attempt to generate new templates by applying AND deduction on existing templates in a table. An objective will be to find efficient approaches to minimize the running cost of such AND deductions.

Another interesting direction for future research is to explore the solving techniques from this work in the context of *LR games* [15]. In particular, applied to the problem of solving games that have been random generated using a game description logic. The application of solving techniques in this work to *LR games* could contribute to a general tool kit for solving combinatorial games.

Chapter 8 gives an overview of the *Pattern-enhanced Alpha-Beta* search. This search is a hybrid Pattern Search and alpha-beta search. The search performs a Pat-

tern search when it is able to solve board position, but switches to alpha-beta search when it can only approximate the value of positions. Another possible hybrid for future work might incorporate a Pattern Search and the *UCT* search from Section 3.5. If this problem can be solved, then the solving techniques presented in this thesis could be applied to Hex players that apply *UCT* searches.

Glossary

alpha-beta pruning: A pruning algorithm that uses a comparison of values to eliminate moves that do not change the return value of minimax searches.

AND rule: A sub-game deduction rule that involves two sub-games with disjoint carriers but a single common target.

apprenticeship learning: A form of reinforcement learning, where the rewards derived from the actions of an expert agent are used to find policies that mimic the expert.

base (P-Triangle) The two cells of a P-Triangle adjacent to a player P side of the board.

Black player: The player in a game of Hex who has the black stones.

Black-H-Search: A H-Search method that only finds sub-games where Black is *Connect*.

Boolean Satisfiability problem: A decision problem that involves a logic expression and asks for an assignment of Boolean values to the variables of that expression rendering the expression true.

carrier: The subset of the empty cells on a Hex board that defines the playing region of a sub-game.

Combinatorial game: A two player game with perfect information and no element of chance, such that the players take alternate turns and the game ends in either a win for one player or a draw after a finite set of moves.

connected: Any two stones in a chain of same coloured stones on a Hex board are said to be *connected*.

connection utility: A value assigned to each cell in a must-play region and is the number of terminal board positions where that cell had a *Connect* stone and *Connect* wins.

cross-entropy method: A search that can solve continuous multi-extremal optimization problems .

elementary sub-game: A strong sub-game where at least one target is an empty cell and the carrier is empty.

evaluation function: A function the estimates the game theoretic value for board positions in games.

flow network: A directed graph where each edge has a flow capacity.

game theoretic value: A value that represents the winning player or a draw for a given board position.

game-graph: The graph that results by reducing a game-tree such that every board position is unique.

game-tree: A rooted tree, whose nodes are valid board positions and whose edges are legal moves.

game: A root to terminal position path in a game tree.

group: A *group* on a Hex board, is a maximal connected component of stones.

H-Search algorithm: A bottom-up search on sub-game sets that applies the AND and OR deduction rules to deduce new sub-games.

Markov Decision Process (MDP): The specification for a fully observable environment with a Markov transition model and additive rewards.

minimax search: An exhaustive depth-first search of a game-tree that either returns the game theoretic value or an estimate value of a given board position.

move domination A move m_x is said to *dominate* a set of moves D, if m_x is a winning move whenever each move in D is a winning move.

move generator An algorithm that places moves in an order that aims to maximize pruning in game-tree searches.

multi-target sub-game: A sub-game that involves a carrier and two disjoint sets of cells X and Y, where *Connect* aims to form a chain of stones connecting at least one cell in X to at least one cell in Y, while *Cut* moves to prevent *Connect* from forming any such chain.

optimal policy: A policy that yields the highest expected reward in an Markov Decision Process.

OR rule: A sub-game deduction rule that involves a set of sub-games with common targets, such that the intersection of carriers is empty.

P-Captured: Given a P-Triangle, if the tip has a player P stone then a move by the opponent of P on a cell in the base is equal to missing a turn and the base is said to be P-Captured.

P-Dominated: The base of a P-Triangle is P-Dominated because a move on the tip is winning for player P whenever a move on a cell in the base is winning.

P-Triangle: A set of of three adjacent cells $\{x_1, x_2, t\}$ used as a template to find P-Dominated and P-Captured cells.

pattern decomposition: An informal approach of threat pattern deduction that relies largely on human intuition to solve small Hex boards.

Pattern search: A Hex game-tree search that deduces threat patterns as the search backtracks from terminal board positions.

Player Connect (Hex): The player of a Hex sub-game who has the role of forming a chain of stones that connect the targets.

Player Connect (Shannon Switching game): The player of a Shannon Switching game who has the role of securing a path that connects target X to target Y.

Player Cut (Hex): The player of a Hex sub-game who has the role of preventing *Connect* from forming a chain of stones that connect the targets.

Player Cut (Shannon Switching game): The player of a Shannon Switching game who has the role of deleting edges in paths that connects target X to target Y.

player policy: A mapping from board positions to moves.

policy: A mapping from state to action.

PSPACE class: The set of decision problems that can be solved by a Turing machine using a polynomial amount of memory and unlimited time.

Quantified Boolean Formula (QBF) problem: A generalization of the Boolean Satisfiability problem in which both existential and universal quantifiers can be applied to each variable.

reinforcement learning: A processes where positive rewards are granted for solving a given problem, and a series of reward feedbacks are applied to learn a problem solving policy that maximizes the total expected reward.

strong sub-game: A virtual connection where *Connect* can win even, if *Connect* moves second.

sub-game: A game that can be played on a subregion of a Hex board position.

successors: The children of board positions in a game tree.

target: A target is a feature of a sub-game that is either an empty cell or one of *Connect's* groups.

template A multi-target sub-game that is a virtual connection.

template matching A rule for matching a template to a given board position to prove that the *Connect* player has a winning strategy for that position.

template matching table A table of pairs (t, v) where t is a template and v is a game theoretic value, used in pruning game-tree searches in a manner similar to the application of transposition tables.

terminal position: A leaf board position in a game tree.

threat pattern: A virtual connection whose targets are *Connect's* two opposite sides of a Hex board.

tip (P-Triangle) The single cell of a P-Triangle not adjacent to a player P side of the board.

transposition table: A cache of board position and value pairs used to prune game-tree searches.

transpositions: The multiple paths that can lead from the root to a particular board position in a game-tree.

virtual connection: A sub-game where *Connect* can win against a perfect *Cut* player.

weak sub-game: A virtual connection where *Connect* can win, but only if *Connect* moves first.

White player: The player in a game of Hex who has the white stones.

White-H-Search: A H-Search method that only finds sub-games where White is *Connect*.

Index

artificial player
knowledge-base Hex player, 112
artificial players, 114
Hex player, 107
minimax search, *see* minimax search

Boolean Satisfiability problem, 11
Quantified Boolean Formula, 11

cell domination
P-Captured, 85
P-Dominated, 85
P-Triangles, 84
base, 84
tip, 84
Combinatorial game, 11
Connection Utility search, *see* Hex Solver One *under* Pattern search
cross-entropy method, 115, 117

evaluation function, 116
advisor, 116
committee, 116
consensus, 116
dynamic, 116
feature function, 116
distance, 122
local mobility, 120
Queen-bee distance, 122
Queen-bee distance potential, 123
Queen-bee distance utility, 123
resistor network, 123, 124
shortest path, 126
shortest path potential, 127
shortest path utility, 127
resistor network, 128
static, 116

flow network, 123

game theoretic value, *see* minimax search
game-tree, 22
game, 23
game-graph, 27
successors, 23
terminal position, 23
transpositions, 27

H-Search, 4, 35, 37, 73
An improved OR deduction procedure, 68
AND rule, 37
Anshelevich's OR deduction procedure, 67

Black-H-Search, 73
OR deduction procedure, 66
OR rule, 37
sub-game bucket, 65
sub-game sets, 38, 65
White-H-Search, 73
Hex, 2, 18
Black player, 2
connected, 3
group, 3
multi-target sub-game, 93
extended template, 98
template, 92, 93
template matching, 94
template mismatch rule, 97
sub-game, 3, 35
AND rule, *see* H-Search
carrier, 3, 36
Connect player, 3, 35
Cut player, 3, 35
elementary, 38
must-play region, 37, 59, 68, 74, 80
OR rule, *see* H-Search
strong, 36
support, 36
target, 3, 35
threat pattern, 36, 73
virtual connection, 36
weak, 36
White player, 2
Hex Solver algorithms 1 to 5, *see* Pattern search

local beam search, 133
stochastic local beam search, 133

Markov Decision Process (MDP), 115
minimax search, 24, 114
alpha-beta pruning, 25, 116
search cut-off, 56
game theoretic value, 24, 108, 114
maximizing phase, 24
minimizing phase, 24
move generator, *see* move generator
move domination
capture, 85
dominate, 84
move generator, 55
dynamic, 55, 58
static, 55
must-play deduction rule, 39

NP-Hard, 19

pattern decomposition, 49
Pattern search, 50
Black mode, 50
connection utility, 58

Hex Solver Five, 95
iterative deepening search, 103
pseudo code, 98
Hex Solver Four, 84
pseudo code, 86
Hex Solver One, 58
pseudo code, 60
Hex Solver Three, 79
pseudo code, 81
Hex Solver Two, 73
pseudo code, 75
move generators, 55
connection utility, 58
Pattern Feature, 79
must-play region, 37, 59
pattern-enhanced alpha-beta, 107
pseudo code, 108
pseudo code, 53
search cut-off, 56
Solver, 4, 48
threat pattern, *see* Hex
White mode, 50
policy
optimal policy, 115
player policy, 114
PSPACE, 11
PSPACE-complete, 19
PSPACE class
PSPACE-complete, 12
reinforcement learning, 114
apprenticeship learning, 115, 117, 128
temporal difference learning, 131
Shannon games
Hex, 18
pairing strategy, 19
Shannon games, 17
Connect player, 17
Cut player, 17
Shannon graph, 12
positive, 14
Shannon Switching game, 12
Connect player, 12
Cut player, 13
sub-game, *see* Hex
template, *see* Hex *under* multi-target sub-game
template matching table, 92, 94, 112
lookup procedure, 95
Standardized template, 98
template elimination, 96, 97
template matching, 94
template mismatch rule, 97
threat pattern, *see* Hex
transposition table, 27
virtual connection, *see* Hex

Bibliography

[1] Pieter Abbeel and Andrew Y. Ng. Exploration and apprenticeship learning in reinforcement learning. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 1–8, New York, NY, USA, 2005. ACM Press.

[2] Bruce Abramson. Control strategies for two player games. *ACM Computing Survey*, Volume 21(2):137–161, 1989.

[3] Vadim V. Anshelevich. An automatic theorem proving approach to game programming. In *Proceedings of the Seventh National Conference of Artificial Intelligence*, pages 198–194, Menlo Park, California, 2000. AAAI Press.

[4] Vadim V. Anshelevich. A hierarchical approach to computer hex. *Artificial Intelligence*, 134:101–120, 2002.

[5] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[6] Gérard M. Baudet. An analysis of the full alpha-beta pruning algorithm. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 296–313, New York, NY, USA, 1978. ACM Press.

[7] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Tdleaf(lambda): Combining temporal difference learning with game-tree search. *Australian Journal of Intelligent Information Processing Systems*, 5(1):39–43, 1998.

[8] Jonathan Baxter, Andrew Trigdell, and Lex Weaver. Knightcap: a chess program that learns by combining TD(λ) with game-tree search. In *Proc. 15th*

International Conf. on Machine Learning, pages 28–36. Morgan Kaufmann, San Francisco, CA, 1998.

[9] Yngvi Björnsson, Ryan Hayward, Michael Johanson, and Jack van Rijswijck. Dead cell analysis in hex and the shannon game. In *Graph Theory in Paris*, Trends in Mathematics. Birkhäuser Basel, 2007.

[10] Cameron Browne. *Hex Strategy:Making the Right Connections*. A. K. Peters, Natick, 2000.

[11] Cameron Browne. *Connection Games Variations on a Theme*. A. K. Peters, Wellesley, 2005.

[12] A. L. Brudno. Bounds and valuations for abridging the search of estimates. *Problems of Cybernetics*, 10:225–241, 1963.

[13] Arie de Bruin and Wim Pijls. Trends in game-tree search. In *Conference on Current Trends in Theory and Practice of Informatics*, pages 255–274. 1996.

[14] Gregory Calbert, Peter Smet, Jason B. Scholz, and Hing-Wah Kwok. Dynamic games to assess network value and performance. In *Australian Conference on Artificial Intelligence*, pages 1038–1050, 2003.

[15] Dan Calistrate. The reduced canonical form of a game. In R. J. Nowakowski, editor, *Games of No Chance*, volume 29 of *Lecture Notes in Computer Science*, pages 409–416. Cambridge University Press, 1996.

[16] Stephen M. Chase. An implemented graph algorithm for winning shannon switching games. *Commun. ACM*, 15(4):253–256, 1972.

[17] Shannon Claude. *Symbolic Analysis of Relay and Switching Circuits*. Masters thesis, Massachusetts Institute of Technology, 1940.

[18] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[19] P. T. de Boer, D. P. Kroese, S. Mannor, and R.Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of Operations Research*, 134(1):19–67, 2005.

[20] Erik D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *Proceedings of the 26th Symposium on Mathematical Foundations in Computer Science (MFCS 2001)*, volume 2136 of *Lecture Notes in Computer Science*, pages 18–32, Marianske Lazne, Czech Republic, August 27–31 2001.

[21] W. J. Duffin. *Electricity and Magnetism*. McGraw-Hill, London, 4th edition, 1990.

[22] Susan L. Epstein, Jack J. Gelfand, and Joanna Lesniak. Pattern-based learning and spatially-oriented concept formation in a multi-agent, decision-making expert. *Computational Intelligence*, 12(1):199–221, 1996.

[23] S. Even and R. E. Tarjan. A combinatorial problem which is complete in polynomial space. *Journal of the Association for Computing Machinery*, 23(4):710–719, 1976.

[24] A. Fraenkel. Combinatorial games : Selected bibliography with a succinct gourmet introduction, 1996.

[25] Martin Gardener. The game of hex. In *The Scientific American Book of Mathematical Puzzles and Diversions*. Simon and Schuster, New York, 1959.

[26] Ryan Hayward. A note on domination in hex, 2004.

[27] Ryan Hayward, Y. Björnsson, M.Johanson, M. Kan, N. Po, and Jack Van Rijswijck. *Advances in Computer Games: Solving 7x7 HEX: Virtual Connections and Game-State Reduction*, volume 263 of *IFIP International Federation of Information Processing*. Kluwer Achedemic Publishers, Boston, 2004.

[28] Akihiro Kishimoto and Martin Müller. A general solution to the graph history interaction problem. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 644–649. AAAI Press / The MIT Press, 2004.

[29] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[30] M. Lee, I. Zitouni, and Q. Zhou. Prediction-based packet loss concealment for voice over ip: a statistical n-gram approach. In *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, volume 4, pages 2308–2312, Dallas Texas USA, 2004. IEEE.

[31] Alfred Lehman. A solution of the shannon switching game. *Journal of the Society for Industrial and Applied Mathematics*, 12(4):687–725, 1964.

[32] V. Litovski and M. Zwolinski. *VLSI Circuit Simulation and Optimization*. Springer, Berlin, 1996.

[33] K. Lye and J. Wing. Game strategies in network security. In *FLoC'02:The Workshop on Foundations of Computer Security*, 2002.

[34] Frederic Maire and Vadim Bulitko. Apprenticeship learning for initial value functions in reinforcement learning. In *International Joint Conference on Artificial Intelligence (IJCAI), Workshop on Planning and Learning in A Priori Unknown or Dynamic Domains*, pages 28–36, Edinburgh, 2005.

[35] Richard Mansfield. Strategies for the Shannon switching game. *American Mathematical Monthly*, 103(3):250–252, 1996.

[36] Gabor Metis and Ryan Hayward. Hex gold at graz: Six defeats mongoose. *International Computer Games Association Journal*, 26(4):281–282, 2003.

[37] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. *Nearly Optimal Minimax Search Tree*. Technical report, University of Alberta, 1994.

[38] Rune Rasmussen and Frederic Maire. An extension of the h-search algorithm for artificial hex players. In *Australian Conference on Artificial Intelligence*, pages 646–657. Springer, 2004.

[39] Rune Rasmussen, Frederic Maire, and Ross Hayward. A move generating algorithm for hex solvers. In Abdul Sattar Kang and Byeong-Ho, editors, *AI 2006 Advances in Artificial Intelligence: 19th Australian Joint Conference on Artificial Intelligence*, volume 4304, pages 637–646, Hobart, 2006. Springer.

[40] Stefan Reisch. Hex is PSPACE-complete. *Acta Informatica*, 15(2):167–191, 1981.

[41] Reuven Y. Rubinstein. The cross-entropy metod for combinatorial and continuous optimization. *Methodology and Computing in Applied Probability*, 1:127–190, 1999.

[42] S. Russell and P. Norvig. *Artificial Intelligence a Modern Approach*. Prentice Hall Series In Artificial Intelligence. Pearson Education, Upper Saddle River, second edition, 2003.

[43] Claude Shannon. Programming a computer for playing chess. *Philosophical Magazine*, Series 7, 41(314), 1950.

[44] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[45] Jack van Rijswijck. *Computer Hex: Are Bees Better than Fruitflies?* Master of science thesis, University of Alberta, 2000.

[46] Jack van Rijswijck. *Search and Evaluation in Hex*. Technical report, University of Alberta, 2002.

[47] Jack van Rijswijck. *Set Colouring Games*. Doctor of philosophy, University of Alberta, 2006.

[48] Jing Yang, Simon Liao, and Miroslaw Pawlak. *On a Decomposition Method for Finding Winning Strategies in Hex game*. Technical, University of Manitoba, 2001.

[49] Jing Yang, Simon Liao, and Miroslaw Pawlak. New winning and losing positions for 7x7 hex. *Lecture Notes in Computer Science*, 2883(2003):230–248, 2003.