Search and Learning Algorithms for Two-Player Games with Application to the Game of Hex

by

Chao Gao

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

Two-Player alternate-turn perfect-information zero-sum games have been suggested as a testbed for Artificial Intelligence research since Shannon in 1950s. In this thesis, we summarize and develop algorithms for this line of research. We focus on the game of Hex — a game created by Piet Hein in 1942 and popularized by Jonh Nash in 1949. We continue on previous research, further bringing the strength of machine learning techniques — specifically deep neural networks — to the game of Hex. We develop new methods and apply them to solving and playing Hex. We present state-of-the-art results for Hex and discuss relevant research on other domains. In particular, we develop solving and playing algorithm by combining search and deep learning methods. As a result of our algorithmic innovations, we produce stronger solver and player, winning gold medal at the Computer Olympiad Hex tournaments in 2017 and 2018.

What I believe is that if it takes 200 years to achieve artificial intelligence, and finally there is a textbook that explains how it is done; the hardest part of that textbook to write will be the part that explains why people didn't think of it 200 years ago, because we are really talking about how to make machines do things that are on the surface of our minds. It is just that our ability to observe our mental processes is not very good and has not been very good.

— John McCarthy

Acknowledgements

Many people deserve to be acknowledged for assisting me directly or indirectly in creating this thesis. First of all, I thank my supervisors Ryan Hayward and Martin Müller for supervising me. My initial interest in games were motivated by my keen interest in playing Chinese chess as a hobby. Even though I roughly know how computers play chess games, I was surprised to learn that super-human strength program does not exist for the game of Hex, especially when considering its simplest rules and straightforward goals. The other aspect of Hex that lures me is its close tie to combinatorics and graphs; these properties have enabled many aspects of Hex be studied in a rigorous mathematical manner rather than mostly empirical as in many other games; I was intrigued to learn more about them, even though I finally did not pursue these directions for my thesis.

My study of Hex with deep neural networks was inspired by the consecutive successes of using deep learning techniques as a general tool for bringing advancement in a number of areas, in particular the early breakthrough in playing the game of Go. By 2016, it becomes clear to me that the next step of research in Hex should be including deep learning to the game. Rather than straightforwardly adapting deep networks into existing Hex algorithms, I tried to understand the essence of each technique thus to invent new paradigms that should be more applicable to Hex and probably other challenging games as well, and this thesis is a reflection of my endeavour towards such an aspiration. In the process, I greatly appreciate the support of Martin Müller for his expertise in multiple domains; his critical questions on my work often pushed me to explore important things otherwise I would have neglected. I am greatly grateful to Ryan Hayward for buying two GPU machines to support my research; most experiments in this thesis were carried out on these machines. His attention on writing also influenced me; the *little* book he gave to me, *The Elements of Style*, which I originally thought of little use, turned out to be a treasure for providing valuable guidance to me.

I am also thankful to a number of people who have aided me. Previous researchers in Hex including Jakub Pawlewicz, Broderick Arneson, Philip Henderson and Aja Huang have provided me useful comments on codebase `benzene`. In particular, in 2018 summer, Jakub Pawlewicz helped me greatly for preparing the Computer Hex tournament. Thanks Siqi Yan, Xutong Zhao, Jiahao Li, Paul Banse, Joseph Meleshko, Wai Yi Low and Mengliao Wang for either doing tournament test for me or temporarily contributing their GPU computation for the testing. Nicolas Fabiano further discovered more pruning patterns in 2019 summer though these are not discussed in this thesis. Thanks Kenny Young for passing me the working code of MoHex 2.0. I am grateful to Noah Weninger for operating MoHex-CNN in 2017 and helping me produce some training games. Thanks Cybera.ca for providing cloud instances. Thanks Jing Yang for providing his pattern set and useful instructions for 9×9 Hex solution.

I thank the useful discussion on a variety of general topics with Farhad Haqiqat, Chenjun Xiao, Jincheng Mei, Gaojian Fan, Xuebin Qin, Chenyang Huang and Yunpeng Tang; researchers that I known of while I was interning at Borealis AI: Bilal Kartal, Pablo Hernandez Leal, and manager Matt Taylor; my manager at Huawei Edmonton research office Henshuai Yao.

Finally, I am deeply grateful to my family for being supportive of my pursuit — I am indebted to my parents, my brother, my sister, and in particular my wife Sitong Li for her great companion, unconditional support and priceless patience.

Contents

List of Tables

List of Figures

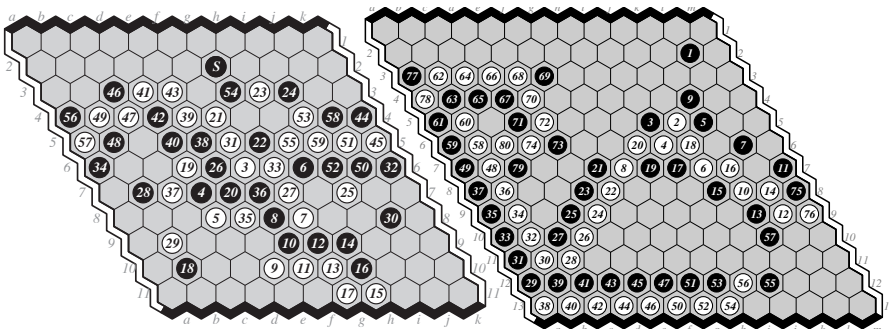
Chapter 1

Introduction

In this chapter, we first introduce the game of Hex, and discuss how it became a domain for scientific research. As the major interest of this thesis lies on algorithms for tackling the game from an artificial intelligence (AI) perspective, we overview general techniques and principles for problem-solving in AI research, with a particular focus on the role of knowledge in search for two-player games. We discuss the pioneering studies that use learning to harness knowledge in the game of checkers, and highlight two fundamental principles used in heuristic knowledge guided search for finding solutions in two-player games. Finally, we summarize our contributions and outline the content of this thesis.

1.1 The Game of Hex as a Benchmark for AI

Hex is a two-player game that can be played on any $N \times N$ hexagonal board. Figure 1.1b shows a game on 13×13 board. Players Black and White are each given a distinct pair of opposing borders. The game is played in an alternating fashion. Black starts first. On their turn, a player places a stone of their color on an empty hexagonal cell. The winner is the one who successfully forms a chain that connects the player's two sides of the board. In real play, to mitigate the first player's advantage, a *swap* rule is often used — in the first turn, the second player can either steal the first player's opening move or play a move in normal fashion. Figure 1.1a shows an example game where the *swap* move was played — the second player stole the opening move $g2$, then the game continued with the first player using white stones therefore actually playing second. The 11×11 board is commonly regarded as the regular board size. The 13×13, 14×14 and 19×19 board sizes are also used by human players.



(a) 11×11 Hex (b) 13×13 Hex

Figure 1.1: Two Hex games on 11×11 and 13×13 board sizes. Each move is labeled by a number, representing the playing order. White won both games by successfully forming a chain to connect his two sides. In the left subfigure, the second move *swaps* the color — S labeled move $g2$ indicates that the player using black stone was originally assigned as the second-player, effectively becoming the first-player after *swap*.

The game of Hex (originally called Polygon or Con-tac-tix) was created by Piet Hein [86], possibly motivated by his dissatisfaction with properties of many board games (e.g., chess), such as the existence of repetitions and the prevalence of draws. Hein contemplated several principles in the design of Hex: fair — both players shall have equal chance of winning; progressive — the same game position shall not reappear during play; final — each game shall end in a limited number of moves; comprehensible — beginners shall learn the rules quickly; tactical — the game shall be non-trivial for playing strategically; and decisive — the game shall have no draws. The game was later discovered and popularized by John Nash [137]; see [80] for a full account for the history of Hex. In 1957, Martin Gardner [71] introduced the game to the general public in an article to *Scientific American*, where he stated that

> "Hex may well become one of the most widely played and thoughtfully analyzed new mathematical games of the century."

Hex can be thought of as a game on a planar graph, thus it is not surprising that the game's no-draw property (any filled board will have a winning path for exactly one player) is related to a property that plays a key role in the proof of the *four-color theorem*, namely that any planar graph can have a clique of size at most four [80]. Gale [62] showed that the no-draw property of Hex is equivalent to Brouwer's fixed-point theorem in two dimensions. Nash [137] proved by *strategy stealing* [24] that a winning strategy exists for the first player, but the proof is non-constructive, i.e.,

an explicit winning strategy is unknown except on very small board sizes.

In a pioneering work for AI in the 1950s, Shannon [178] discussed the difficulty of creating an AI agent for Hex and proposed an electronic model for playing the game. As our goal is to apply state-of-the-art AI methods to the game of Hex, we note that, for two-player games, relevant AI research can be divided into two categories:

1. *Solving algorithms* aim to find the theoretic win/loss value of a game position under the assumption of perfect play on both sides. This is further defined on three hierarchical levels [4]; each higher level subsumes the lower one.

    - *Ultra-weakly solving* determines what is the best outcome that the first player can achieve from the initial position of the game (win, loss or draw), given that the opponent is perfect. However, the strategy for achieving such a goal is not required to be specified. Hex on any $N \times N$ board is ultra-weakly solved by the *strategy stealing* argument.
    - *Weakly solving* means to identify the winning strategy from the initial position of the game if there is one, to find all second-player's winning strategies after each possible opening move of the first player if the game is second-player win, or to specify the best strategies of both players that would force the game to a draw.
    - *Strongly solving* goes one step further by requiring to weakly-solve any position that could occur by the rules of the game.

2. *Playing algorithms* aim to provide a good move for any arbitrary position. How close the returned move is to being optimal can be difficult to specify with mathematical precision; evaluation of these algorithms thus relies on empirically competing against existing opponent players, such as strong human professionals. Most research in classic games is of this sort. It is clear that weakly solving a game implies perfect play.

Despite its simple rules, Hex presents significant challenges to AI research for both playing and solving, primarily because of the large and near-uniform branching factor, as well as the lack of reliable hand-crafted evaluation functions [204, 205, 206]. Devising an automated program for strongly solving arbitrary $N \times N$ Hex positions is PSPACE-complete [30, 162]. In practice, their versatile reasoning ability for

decomposing strategies and pruning irrelevant regions has allowed human players to find more efficient solutions than the best computer programs so far for board sizes up to 9×9 [214, 215].

1.2 Searching for Solutions with the Help of Knowledge

Early research in AI, such as the calculus solver by Slagle [186], the General Problem Solver (GPS) [141] and the Graph Traverser [53], were based on the plausible argument that intelligence is knowledge plus deduction, where *knowledge* refers to rules, facts and intuition that can be encoded by humans, and *deduction* is achieved by various search methods that operate on a structure exposed by the representation of the problem.

The description of knowledge can be heuristic or exact. In certain cases, exact knowledge is necessary for problem-solving. For example, to swiftly compute the exact length of the hypotenuse of an right-angled triangle — the side opposite to the right angle, exact knowledge of the Pythagorean theorem is needed. Indeed, even the knowledge about the Pythagorean theorem itself is a consequence of precisely applying logical deduction from a list of Euclidean postulates. However, apart from this, in real-world environments, a large body of human knowledge is imprecise, but it is ever-surprising to observe how much people can accomplish with that imprecise, unreliable information known as *intuition*. People drive cars without knowing how they function in detail and usually only have a vague picture of the road conditions ahead. A five-year old child can easily distinguish a cat from a dog, but it is hard to explain they do it. In computer science, imprecise yet helpful information content is called *heuristic knowledge* [153]. Heuristics represent a trade-off on the demand between theoretical and empirical problem-solving. In order to be practical, heuristics must meet the need of simplicity and intelligibility, and at the same time, they must be informative enough in discriminating bad and good choices.

The merits of heuristic and exact knowledge are often complementary in practical problem-solving. Heuristic knowledge by its nature tends to be vague in describing the quantity of interest; therefore, it may work well across a variety of domains, but it can be outperformed by accurate, precise domain knowledge. For example, in the task of finding the shortest path between a pair of *source* and *goal* nodes, the A* algorithm [77] guides its exploration by summing the cost so far and the estimated cost to goal. If for some nodes, the estimated cost to goal can be per-

fectly acquired, the search to goal will be accelerated. However, perfect information represents an exact description about the domain to solve, whose benefit disappears when transferring to other domains. Notable achievements such as solving Rubik's Cube [112] combine both heuristic and precise knowledge in their search, where the perfect knowledge is generated by a procedure [49] and stored in a database. Similar techniques were used in solving checkers [172] with checkers endgame databases. Other games such as Gomoku [3] and small board size Hex [82] were solved using domain-independent systematic search with the help of domain-specific algorithms for knowledge discovery, which were used for move pruning and early endgame detection.

Search with the aid of problem-independent or problem-specific knowledge has been extensively adopted in the field of operations research (OR) [160]. Similar to the role of action description languages [130] in heuristic search planning [29], combinatorial optimization problems are typically expressed via formal *integer linear programming* (ILP) models and solved by the help of relaxed models of *linear programming* (LP). The solving procedure *branch-and-bound* [116] works by repeatedly *branching* and *bounding* until the optimal solution is found or an external stopping criterion is satisfied. Clearly, selecting the variable on which to branch, how to branch and how to bound is crucial to practical problem-solving. Although some strategies work for all ILPs, when it comes to specific problem, e.g., the infamous Traveling Salesman Problem (TSP), problem-specific pruning (e.g., using domain-dependent cut planes [145]) and splitting rules can lead to drastic performance increase [10]. Essentially, those heuristic or exact information for guiding the search are domain-knowledge summarized by ingeniously identifying special properties of the problem model to solve. The difference between AI and OR algorithms is that in OR, branch-and-bound search is conducted by continuously *splitting* and *pruning* in the space of solutions while ensuring optimality. In AI, many algorithms [143] work by repeating exploring the space of states until a desired goal is achieved. For instance, in a single-agent game, A* variants model the problem as a *generative model*: starting from an initial condition, each action leads a new encoding called *state*. The goal is to find a series of actions to achieve the goal condition. A solution is generated incrementally during the process of search. Although the procedure is generate-and-test, *optimality* analysis has to be done by referring to split-and-prune [153].

Generalizing A* search for two-player games results in AO* [143]. Similarly, a game position to solve can be encoded as a state, such as a two-dimensional array to represent a Hex position. Solving requires repeated look-head by finding a move such that the opponent is unable to refute no matter which move is used to respond. Therefore, a *solution* for a game position is not a sequence of actions, but a graph-represented *strategy* which can be thought as a prescription for choosing actions in response to any adversarial moves (the formal definitions on how strategy could be represented in two-player games will be discussed in Chapter 2.1). In practice, due to the exponential growth rate of space requirement of look-ahead search, finding an exact solution is not always be feasible. Search with heuristics might only be able to find an approximate strategy for playing well. In either case, *knowledge* can help solving or playing by providing more efficient encoding of state, better state transitions (e.g., by heuristic or exact pruning of unpromising moves), and better ways to conduct the systematic search.

To illustrate the effect of *knowledge* in the context of solving the game of Hex, Figure 1.2 shows an exact *solution* to 3×3 Hex, where the only knowledge given to computer is this: a state is winning if there exists a move that joins the player's two sides. More than 30 nodes are used to in the solution. However, if one more piece of knowledge — *bridge pattern* ensures a safe connection (Figure 1.3) — is given, all nodes beyond the second level of Figure 1.2 can be pruned, and a computer program needs only two nodes to show the wining strategy.

1.2.1 Learning Heuristic Knowledge in Games

A fundamental question for an AI agent is where does the knowledge come from? Humans gain knowledge in two ways: (1) by summarizing past experience, (2) by deriving new knowledge through reasoning about already acquired knowledge. Earlier attempts to expert systems [78] tried to automate the process of (2) by encoding expert knowledge into rules and facts that computers can use to infer new knowledge about a domain. However, the acquisition, maintenance and representation of expert domain knowledge is difficult itself [19]. Another approach is to acquire knowledge automatically from experiences by mimicking (1) using *machine learning* [131], whose aim is to learn a general *concept* [134] from experiences (represented as *data*), which will lead to enhanced performance in solving subsequent problems.

The study of using *machine learning* to acquire knowledge in games was pio-

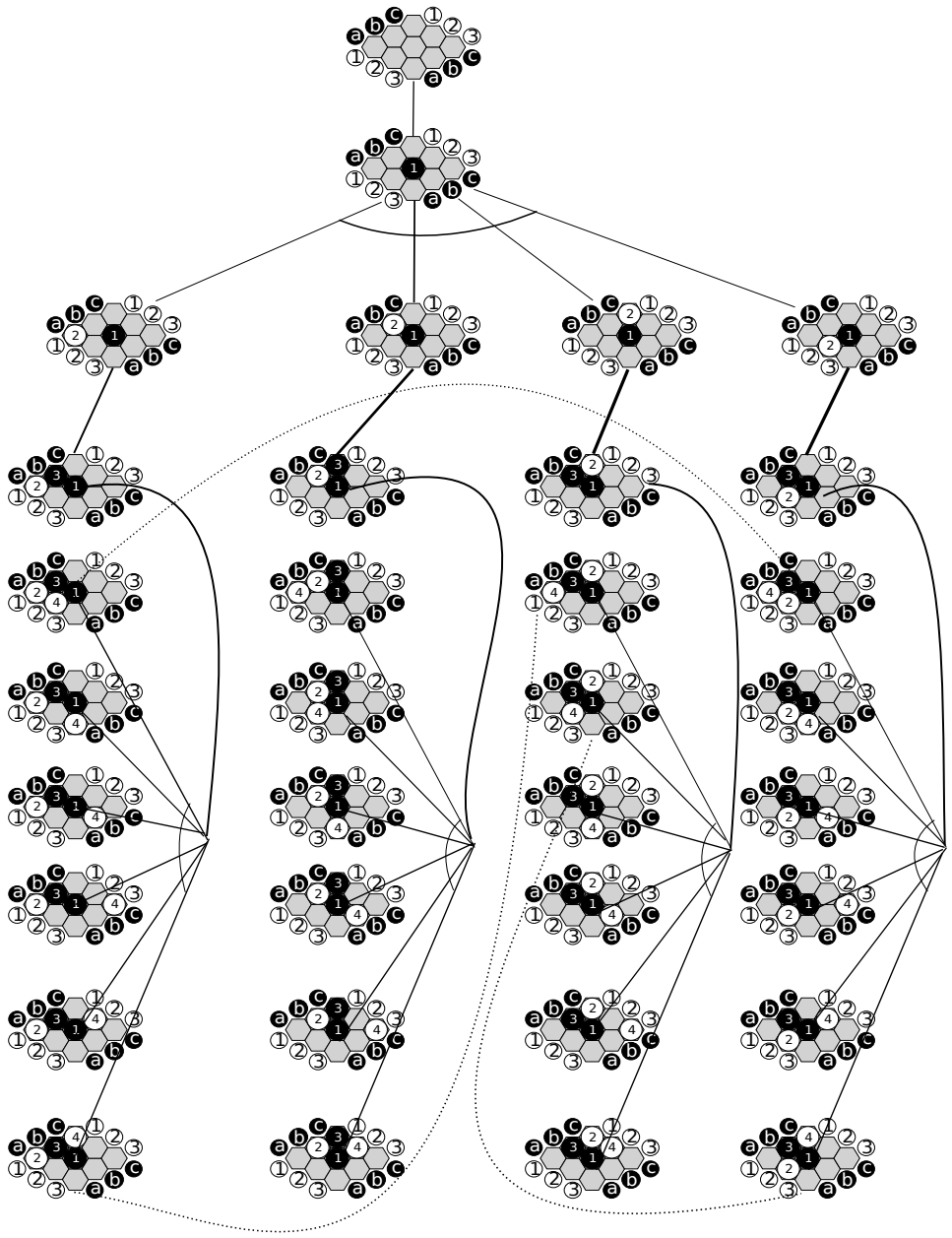


Figure 1.2: A 5-level solution for 3×3 Hex. A dotted line indicates that the connected two nodes can be merged into one as they are equivalent. Symmetric moves are ignored. This solution contains 34 nodes.

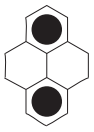


Figure 1.3: The bridge connection pattern in Hex. The two black stones can be connected via one of the two empty cells no matter if Black or White plays first — they are virtually connected.

neered by Samuel [168, 169] for checkers. Samuel argued that two sharply distinguished approaches exist to machine learning: (1) the general neural net approach that tries to simulate human behavior by a connected network, and (2) a more efficient problem-specific approach that is to produce a highly organized decision network for specific tasks. The neural net approach was infeasible in that time because the size of biological neural nets is far beyond those can be simulated by computers, so he chose the (2) to study machine learning. Samuel chose checkers for these reasons: large space — there was no existing algorithm guaranteeing win/loss; the goal and rules for playing are well-defined; knowledge is the major reason for well-playing performance; and the game is familiar to many people so that behavior of the computer program can be assessed by human professionals.

Samuel considered two learning paradigms: *rote learning* and *generalization learning*. The goal of rote learning is to memorize important search-produced evaluations for specific game positions and in the subsequent search, evaluations can be reused without further look-ahead. For example, in a situation where a node is reached after 3-ply look-ahead, if the evaluation of this node can be retrieved from storage that was saved as a result of 5-ply search, then the evaluation of the current position effectively simulates a 8-ply search. Samuel designed a number of schemes to for efficient information storage. A polynomial scoring function is used as the learning function. In generalization learning, feature weights are adjusted by referring to the game results of self-play or play against humans. In both learning methods, look-ahead search is restricted to at most 20-ply due to hardware limitation.

In later work [169], rote learning was improved by using an enhanced hash table for more efficient information storage. More advanced search techniques, such as alpha-beta pruning [109], were incorporated. Generalization learning was further improved by training on higher quality games played by master players. The

program failed to reach top-human playing strength. Samuel blamed the polynomial scoring function for not being expressive enough to approximate human expert evaluations on checkers.

In this thesis, we pay particular attention to the learning model of artificial neural networks [175] and show their usefulness in the game of Hex.

1.2.2 Principles for Heuristic Search in Games

To find the objective of interest with small effort, knowledge aided search procedures are often driven by optimizing a certain cost measurement, even though the quest is an arbitrary *feasible* solution, rather than the one with a minimum cost. For example, for Rubik's cube, the aim is to find any solution path — rather than a *shortest path* — to achieve the goal. However, in practice, to find the goal fast, the A* variant [112] searched the space by chasing a path that is likely to yield the minimum cost. The same is true for two-player games. Given a Hex position, in order to answer whether the it is a first-player win or not, search can be conducted by chasing a surrogate objective of finding the smallest proof. Such an idea is called *small-is-quick* principle. Yet, an exact measurement on which direction will lead to the real smallest proof is usually unknown, external knowledge has to be used for providing estimated proof sizes — such a selective exploration by treating guessed metric as a true measurement is called the *face-value* principle. All best-first search paradigms for single or two-player games adopt these two principles to guide the problem-solving [153].

In this thesis, we develop new search algorithms in the presence of heuristic knowledge from learned neural net models, and apply them for solving the game of Hex.

1.3 Contributions and Organization of This Thesis

The contributions of this thesis are as follows:

- We provide a summary on various general/specific learning and search techniques — these include informed best-first search [153], reinforcement learning [192] and deep learning [75] — related to solving and playing the game of Hex. We discuss the limitation and usefulness of each, and highlight the necessity for combining them together for better practical algorithms.

- We present experimental studies on supervised learning and model-free reinforcement learning with deep neural networks for Hex.
- We achieve state-of-the-art playing and solving results for the game of Hex by devising novel algorithms that combine deep neural nets and search — these include the developments of a three-head neural network architecture for Monte Carlo tree search for better learning and search, and incorporating deep networks to proof number search for more efficient solving of Hex states.

The rest of this thesis is organized as follows:

- Chapter 2 reviews and discusses general as well as specific techniques for solving and playing games, including an overview of important algorithms and techniques that have been developed upon more general problem-settings, and discussions on how they can be harnessed together for tackling the game of Hex.
- Chapter 3 presents studies of using deep neural nets in supervised and model-free reinforcement learning for Hex. We show that high move prediction accuracy can be achieved using deep networks as the learning model, and the trained model can be used to achieve better playing strength of the search. We then develop a new reinforcement learning procedure and show that it outperforms the vanilla methods in the game of Hex.
- Chapter 4 presents a three-head network architecture, and its practical merit in Monte Carlo tree search and search-based reinforcement learning. Using such an architecture, we develop a new Hex player called MoHex-3HNN, which became the champion player in 2018 Computer Hex competition.
- Chapter 5 presents a study of incorporating deep network for more efficiently solving Hex. Specifically, we show that by incorporating neural networks to proof number search, faster solving can be achieved for Hex. We then discuss the limitation of current search paradigms, and outline a new search method for solving large board size Hex positions.
- Chapter 6 concludes the thesis by pointing a few promising future directions for the game of Hex.

Publications

First-authored publications created during my PhD studies are as follows:

- Chao Gao, Ryan Hayward, and Martin Müller. "Move prediction using deep convolutional neural networks in Hex." IEEE Transactions on Games 10.4 (2017): 336-343. See [63].
- Chao Gao, Martin Müller, and Ryan Hayward. Adversarial policy gradient for Alternating Markov Games. 2018 International Conference on Learning and Representation workshop track. See [64].
- Chao Gao, Martin Müller, and Ryan Hayward. "Focused Depth-first Proof Number Search using Convolutional Neural Networks for the Game of Hex." IJCAI 2017. See [65].
- Chao Gao, Martin Müller, and Ryan Hayward. "Three-Head Neural Network Architecture for Monte Carlo Tree Search." IJCAI 2018. See [66].
- Chao Gao, Kei Takada, and Ryan Hayward. "Hex 2018: MoHex3HNN over DeepEzo." ICGA Journal 41.1 (2019): 39-42. See [67].
- Chao Gao, Siqi Yan, Ryan Hayward and Martin Müller. "A transferable neural network for Hex". ICGA Journal 40.3 (2019): 224-233. See [69].

First-authored publications created during my PhD time but outside of the scope of this thesis are

- Chao Gao, Guanzhou Lu, Xin Yao and Jinlong Li. "An iterative pseudo-gap enumeration approach for the Multidimensional Multiple-choice Knapsack Problem." European Journal of Operational Research 260.1 (2017): 1-11. See [68].
- Chao Gao, Bilal Kartal, Pablo Hernandez-Leal, and Matthew E. Taylor. "On Hard Exploration for Reinforcement Learning: A Case Study in Pommerman." AIIDE 2019. See [70]. Although content of this paper is not in this thesis, its discussion on the limitation of model-free reinforcement learning is related to those in Chapter 3.2.

- Chao Gao, Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. "Skynet: A Top Deep RL Agent in the Inaugural Pommerman Team Competition." RLDM 2019.

Further collaborated papers outside of the scope of this thesis are

- Xuebin Gao, Zichen Zhang, Chenyang Huang, Chao Gao, Masood Dehghan, and Martin Jagersand. "BASNet: Boundary-Aware Salient Object Detection." In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 7479-7489. 2019.

- Bilal Kartal, Pablo Hernandez-Leal, Chao Gao, and Matthew E. Taylor. "Safer Deep RL with Shallow MCTS: A Case Study in Pommerman." Adaptive and Learning Agents Workshop at AAMAS 2019.

Chapter 2

General and Specific Techniques for Solving and Playing Games

Before presenting our new algorithms for the game of Hex, we first give necessary background. In this chapter, Section 2.1 reviews how strategy is represented. Section 2.2 discusses two formulations used in search and learning algorithms. Section 2.3 reviews and summarizes general search and learning techniques in the literature, how they are connected, how they can help to find the desired strategies in two-player alternate-turn games. Finally, in Section 2.4, we survey the large body of previous work on Hex and explain how some Hex-specific algorithms can be improved by combining with general artificial intelligence methods.

Our improved Hex algorithms — presented in latter chapters 3–5 — rely on the content in this chapter. Understanding these general methods, as well as their commonalities and differences, helps understand how we came to our improved algorithms. We hope that our comments here on the interconnections of various methodologies will also be of use to researchers working on other AI problems.

2.1 Strategy Representation

We begin by reviewing strategy representations, examining their assumptions, and discussing their advantages and shortcomings.

2.1.1 Minimax Strategy and Solution Graphs

Figure 2.1 shows the directed acyclic graph of all possible continuations of a game. Two players, MAX and MIN, appear alternately in the layered graph; each node represents a game state; each edge stands for a move. The first layer contains only one node belonging to MAX. It represents the start of the game. Each leaf node

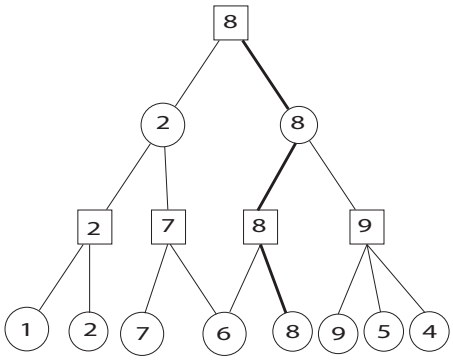


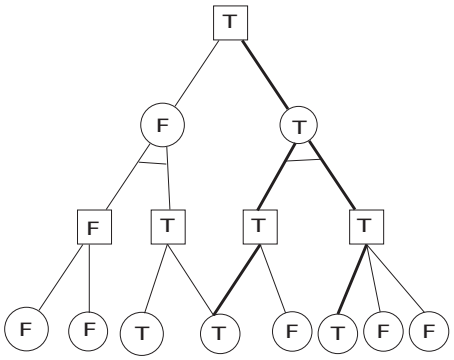
Figure 2.1: The directed acyclic graph (edges are directed downwards) of a game. Each node corresponds to a state, and is labeled with that state's minimax value score for player MAX

has a utility score with respect to the MAX player; for any other node s, its score is decided by maximizing (for MAX nodes) or minimizing (for MIN nodes) among the scores from child nodes, as expressed in Eq. (2.1).

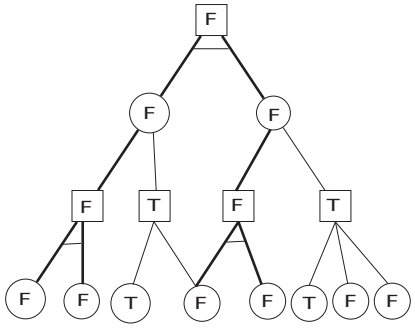
$$v(s) = \begin{cases} eval(s) & \text{if } s \text{ is leaf} \\ \max_{s'} v(s') & \text{if } s \text{ is MAX node} \\ \min_{s'} v(s') & \text{if } s \text{ is MIN node} \end{cases} \tag{2.1}$$

In Figure 2.1, bold edges (representing best actions) indicate an optimal move sequence to achieve the best score for both players. An optimal minimax strategy sequence is referred to as a *principal variation*, where the optimality is built on the assumption that evaluation of leaf nodes is exact.

Rather than optimizing on seemingly correct score function, a player might only want to find a strategy to win if it is possible or to prove that there is no winning strategy at all. In such case, a *solution-graph* can be used to represent the sought target. Similar to a *principal variation*, a solution-graph is identified from a directed acyclic graph of a game, where each node has a value of either true (T) or false (F). Figure 2.2 shows two examples of solution-graphs: subfigure 2.2a contains a winning strategy for the first player, marked by bold edges; subfigure 2.2b has a winning strategy for the second player — no matter which move the root player chooses to play, the opponent can always refute it. Similar to Eq. (2.1), the value at each node can be computed recursively using and ($\wedge$) and or ($\vee$) logical operations, as in Eq. (2.2).



(a) Bold edge shows a first-player winning strategy.



(b) Bold edge marks a second-player winning strategy.

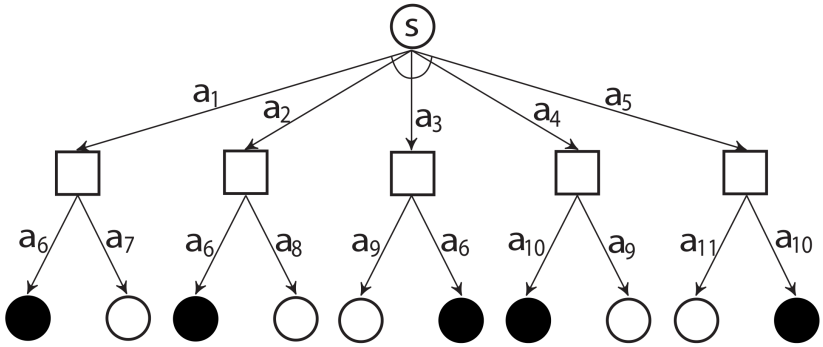
Figure 2.2: Example solution graphs for the first and second players. Each node represents a game state; each edge stands for a legal action from its upward game state. Value at each node is labeled with respect to the player at root, i.e., the first-player.

$$v(s) = \begin{cases} eval(s) & \text{if } s \text{ is terminal} \\ \wedge_{s'} v(s') & \text{if } s \text{ belongs to first-player} \\ \vee_{s'} v(s') & \text{if } s \text{ belongs to second-player} \end{cases} \tag{2.2}$$

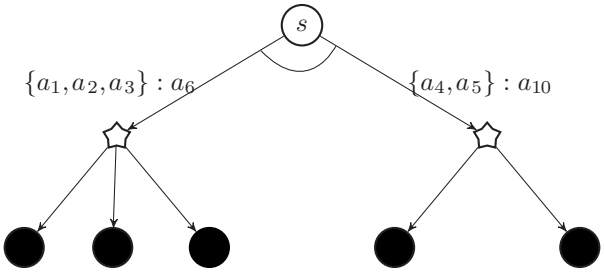
2.1.2 Strategy Decomposition

The solution graph defined in the previous section is identified on a graph with reference to the optimal value of each game state. The drawback of this is that even representing an arbitrary solution graph could be expensive: assume that on average the branching factor is b and depth is d, the space required to represent a winning strategy is $\Omega(b^{d/2})$. A natural question arises: is there any more space efficient encoding for direct strategy representation in two-player alternate-turn games?

Recall that in single-agent path-finding problems, a *solution* (*strategy* or *plan*) can be represented as a series of action choices. In two-player games, however, due to the existence of the adversarial opponent, each action has to be assessed by taking into account all possible opponent responses. In other words, the fundamental difficulty comes from the fact that to prove a game state s is losing, all legal moves from s have to be evaluated individually. Thus, a more clever algorithm would identify the relationships between substrategies of different moves and reuse them across moves. This observation can help to reduce the size of a proof that a state s is losing: instead of treating each move independently, we can divide the legal moves



(a) s is losing because every child of s has a winning action.



(b) All actions of s and the corresponding response are grouped

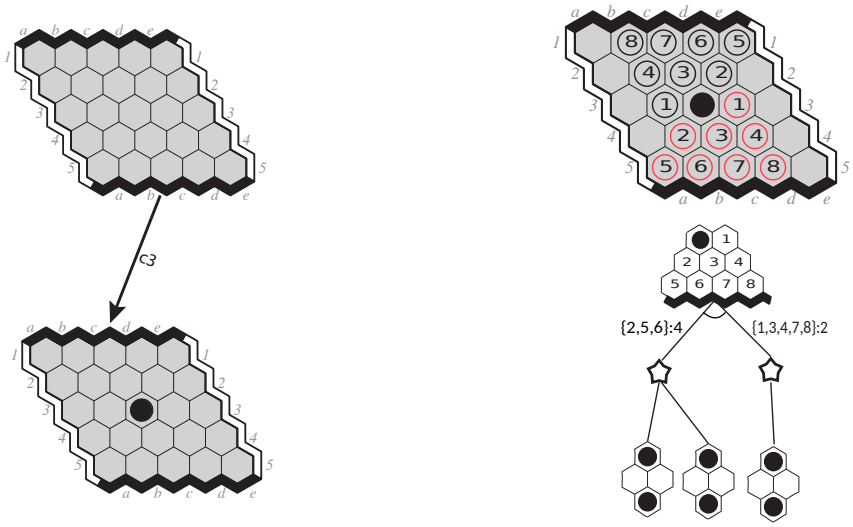
Figure 2.3: Grouping moves when proving s is losing

of s into different groups if we observe that among all branches only a few distinct *counter* moves exist, i.e., moves leading to terminal nodes that fulfill the goal that player at s is losing.

Figure 2.3a shows an example game: In order to prove that s is losing, all its five actions a_1 to a_5 have to be evaluated. For each child node of s, there exists an action leading to a terminal state (filled with black) where the player at s is losing. However, the successful responses from child nodes of s have only two distinct forms: a_6 and a_{10}. Thus, we can abstract actions at s and their responses into two groups, resulting in Figure 2.3b.

The idea in Figure 2.3b can be further developed: if treating state s a second player winning strategy, we see that s has been represented by subgame second player winning strategies from two branches. Figure 2.3b is a decomposition-oriented strategy representation. The number of filled black nodes in 2.3b may be further reduced by merging game states sharing the same subgame strategy.

We illustrate these ideas using 5×5 Hex. To start the game, Black plays at the center cell $c3$. It remains to prove that Black has a second-player winning strategy from then on. Black tentatively plays a move at the center cell c_3, as shown in



(a) Initially, the board is empty; Black wants to prove that c_3 is the winning move.

(b) Subgame strategy for 4-3-2 pattern constructed using strategies for the bridge pattern.

Figure 2.4: In the right figure, by symmetry, the winning strategy for Black can be summarized by two identical and independent subgame strategies, i.e., cells encircled with numbers 1 to 8 plus the black stone. This subgame is called the 4-3-2 pattern in Hex; it can be composed from bridge patterns. The starred nodes are dummy whose purpose is to make the exposition of the decomposition clear.

Figure 2.4a. Since the board is symmetric, to achieve the goal of connecting Black's two sides, it is enough to find a strategy to connect c_3 to one side of the board using only the lower half of the empty cells, e.g., $d_3, e_3, a_4, b_4, \ldots, e_4, a_5, b_5, \ldots, e_5$. We can represent the strategy using mappings $f_2 \triangleq \{a_4, b_4, a_5, b_5\} \to d_4$ and $f_3 \triangleq \{d_3, e_3, c_4, d_4, e_4, c_5, d_5, e_5\} \to b_4$, where for each mapping, the *argument* is a set of possible White moves and the *output* is a Black response. For each possible play from these two mappings, the induced subgame can always be solved using the bridge pattern strategy. Here, White moves e_3, e_4, e_5, a_4 are ignored, because they can be put in either f_2 and f_3 or even responded by random Black move; see Figure 2.4b for the illustration. From this example we see that, given only knowledge about the *bridge pattern*, winning strategy for proving that the center move c_3 in 5×5 Hex can be represented less than 10 nodes, while using a solution-graph as in Figure 1.2, a larger number of nodes would be required. Figure 2.5 further demonstrates that a solution from such decomposition can also be viewed as a solution-graph; each circular node represents a game strategy; each square F node stands for a collection of mappings whose argument sets in union is the set of all White moves. In G

of Figure 2.5, it is Black's turn, so opponent moves can be regarded as $\emptyset$. Using this representation, an explicit winning strategy for 9×9 Hex identified by human player and Go expert Jing Yang [214, 215] contains only around 700 nodes, while with game state representation, even with extensive knowledge computation, the state-of-the-art solver [149] finds a solution with around 1 million nodes [1].

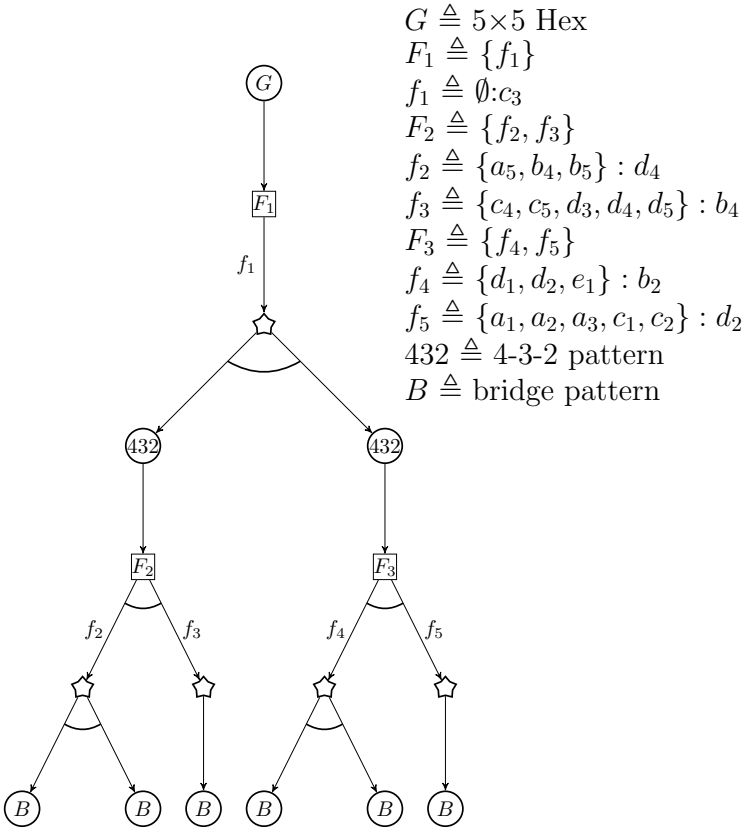


Figure 2.5: Solution for 5×5 Hex by strategy decomposition. Each f edge is a mapping whose argument is a set of moves and output is a single response move. The graph can be further simplified by discarding all squared nodes and merging all subgame nodes sharing the same substrategy into one. Starred nodes are called branch nodes in [214].

2.2 Search and Learning Formulations

In this section, we review and discuss two types of formulations related to solving and playing games. We start with state-space and problem-space graphs, then go on to Markov Decision Processes (MDPs) and Alternating Markov Games (AMGs). The former ones are used in search methods [153]. The latter are adopted in learning

[1] Obtained using Benzene: https://github.com/cgao3/benzene-vanilla-cmake

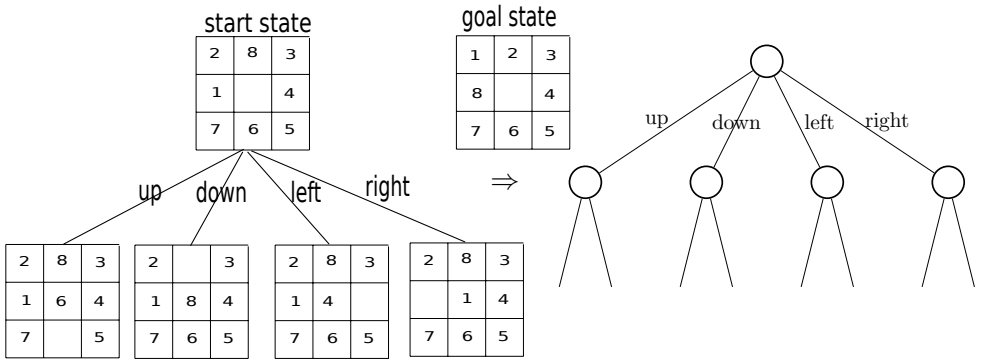


Figure 2.6: The 8-puzzle sliding tile problem [153]: Given an initial configuration and rules for moving tiles, the objective is to find a sequence of moves to reach the goal state. Such a puzzle can be directly translated into a *state-space graph*, which is an OR graph where every vertex is a decision node for the problem-solver.

algorithms [123, 192].

2.2.1 State and Problem Space Graphs

A game of interest, for example, a single-player puzzle problem as in Figure 2.6 or the game of Hex, is described by (1) starting state, (2) rules of change, i.e., permitted actions; and (3) desired goals. This description can be translated, in a straightforward manner, into a graph formally known as a *state-space graph* [153].

Each *state* refers to a *complete* description of a configuration of the game. The whole state-space graph can be too large to be explicitly represented. An efficient problem-solver therefore aspires to generate only a small portion of the whole state-space graph to find the desired solution. In Figure 2.6, one important characteristic is that every state is a decision point for the problem-solver, and each edge from a state results in another state that is also a decision point for the problem-solver; state-space graphs possessing only one decision maker are called *OR Graphs* [153].

For two-player alternate-turn games such as the game of Hex, not all nodes belong to the same decision-maker, as shown in Figure 2.7. In this case, the effect of an action for a problem-solver must consider all opponent responses. Such graphs are referred to as *AND/OR Graphs* [153]. For OR graphs, the pursuit solution-object is a *solution path* which represents a sequence of actions. For example, a solution to the puzzle in Figure 2.6 is a sequence of moves to achieve the goal state. For AND/OR graphs, a solution for the player at root node is an AND/OR *solution graph* $\mathcal{S}$ satisfying the following properties: (1) $\mathcal{S}$ must contain the root node; (2) all terminal nodes of $\mathcal{S}$ (those without any successor) are known and the same; (3)

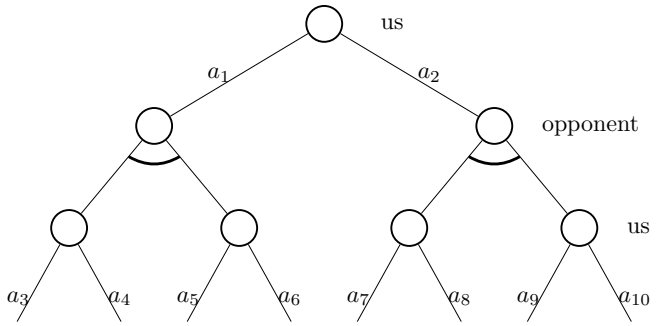


Figure 2.7: The state-space graph for two-player alternate-turn games is an AND/OR graph, where AND nodes are decision points controlled by an adversary opponent.

for any AND node s in $\mathcal{S}$, all successors of s are in $\mathcal{S}$; (4) for any OR node s in $\mathcal{S}$, one successor of s must be in $\mathcal{S}$.

We have seen in Section 2.1 that it could be more space efficient if a node is used to represent a solution to a subgame, not just a single state. AND/OR graphs can express not just state-space, but also *problem-space* graphs. The underlying problem-solving methodology is viewing solving as a series of problem-reduction steps. In this setting, each node of an AND/OR graph represents a problem or subproblem. An AND node is a subproblem containing multiple subproblems that must all be solved, whereas an OR node can be solved by solving any successor.

For example, solution to the Tower of Hanoi problem can be found by successively reducing the problem into a conjunction of three subproblems. Such a reduction oriented solution is a pure AND graph; it is identified from a problem space graph from an AND/OR graph shown in Figure 2.8. Problem-solver that excels at such a reasoning (i.e., by means-ends-analysis [141]) will eventually choose the leftmost branch as this is the only feasible solution graph exhibited in Figure 2.8. On the other hand, being a single-player puzzle game, Tower of Hanoi may also be directly modeled into a state-space OR graph — the root node will contain two edges representing that two actions choices: moving disk D_1 to D_2 or D_3. This formulation, however, fail to leverage the well-ordered dependencies between subgoals, and finding a solution in this OR graph thus becomes more difficult [153].

For two-player games, recall that in *state-space graph*, each enumerating option is an edge-represented legal move from its corresponding game state, in *problem-space graph*, each node G represents a strategy, and each edge represents one step reduction for G. For example, suppose for a Hex state s there is a Black winning strategy

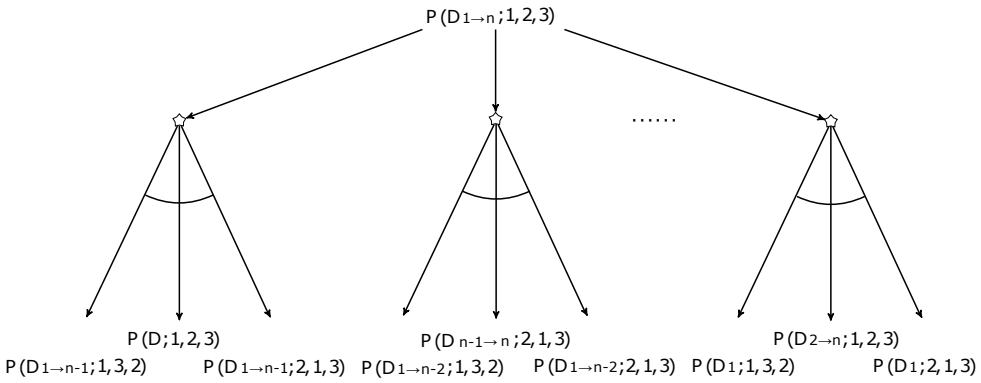


Figure 2.8: Tower of Hanoi problem space graph. $P(D_{1\to n}; 1, 2, 3)$ represents the problem to move n disks from tower 1 to 3 using 2. Leftmost branch is actually a recursive solution to the problem. Identifying such a solution relies on the availability of well-ordered subgoals and purposeful identification on the relations between parent goal and subgoals; such a process is known as means-ends analysis [141]. It is also possible to formulate this problem as a state-space OR graph; however, finding a solution becomes more difficult. Solution graph from the leftmost branch is actually a tree that grows exponentially; this is simply because for n disks, no solution required less than $O(2^n)$ steps exists.

G, each edge from G can be a pair (D, c), where D is subset legal White moves at G, and c is a Black counter move for all moves in D. Figure 2.9 schematically illustrates one step strategy decomposition. G_0 is the original game. There are n mappings $f_1, f_2, \ldots, f_n$, where $f : 2^U \to C$. That is, U is the set of legal moves in G_0 and C is the set of counter moves. The total number of such mappings, i.e., n, can be $2^{|U|} \cdot |C|$. However, most of these edges may be ignored by the problem-solver, and from the state-space graph we know, it is certainly true that there exists a solution by exploring at most $|U| \times |C|$ edges. The practical merit of searching in the problem-space as defined in Figure 2.9, similar to that of Figure 2.7, depends critically on how to ensure promising edge are preferentially explored.

2.2.2 Markov Decision Processes and Alternating Markov Games

A finite Markov Decision Process(MDP) [20, 158] can be expressed by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ a set of actions, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ a reward function, and $\mathcal{P}$ denotes the probabilistic transitions among states. The Markov property states that the agent at time t is independent from all other previous states and actions, i.e., $\Pr(s_{t+1}|s_1, a_1, \ldots, s_t, a_t) = \Pr(s_{t+1}|s_t, a_t)$. The envi-

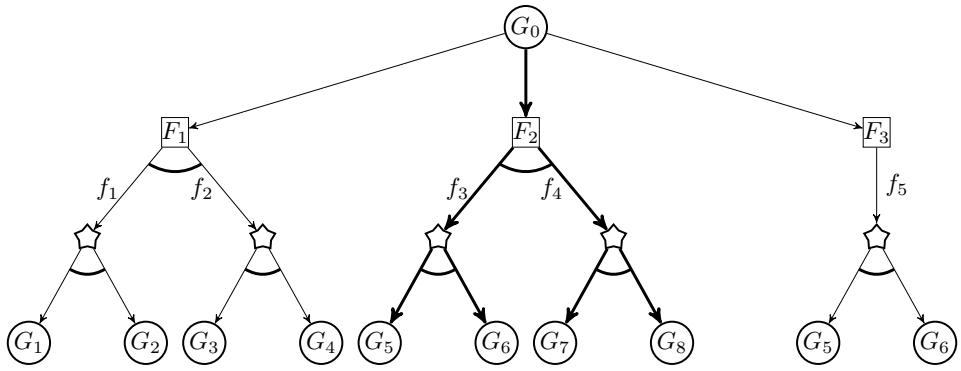


Figure 2.9: An illustration of searching for solution graph from a problem-space graph. For G_0. F_1, F_2 and F_3 are alternative options; each of them represents a collection of functions with $\cup_i argument(f_i) = U$, i.e., covers all legal opponent moves. Thus, each F node represents a solution to a set covering problem, and it is unnecessary to explicitly list all F nodes during search. Note that G_0 could also represent a first-player winning strategy by letting the argument set of each function f be empty.

ronment is fully described by transition probabilities and a reward function:

$$\mathcal{P}(s'|s,a) = \Pr(s_{t+1} = s'|s_t = s, a_t = a)$$

$$\mathcal{R}(s'|s,a) = \mathbb{E}\left[r_{t+1}|s_t = s, s_{t+1} = s', a_t = a\right]$$

The objective is to *learn* a policy to maximize the expected discounted cumulative reward:

$$\mathbb{E}\sum_{t=k}^{\infty}\gamma^{t-k}R_{t-k+1},$$

where γ is a discounting factor $0 < \gamma \leq 1$ that controls the contribution of long-term and short-term rewards. A policy $\pi(a|s)$ is a function that maps any state s to a probability distribution over all available actions at s, $\pi(a|s) = \Pr(a|s) \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$. Many single agent games, such as the 8-puzzle in Figure 2.6, can be viewed as an MDP where transitions are deterministic. For this reason, analogous to shortest path, finding an optimal policy for an arbitrary MDP is sometimes called *stochastic shortest-path problem* [25].

MDPs are single-agent games, while we are primarily interested in two-player games. Extending ideas from MDPs to two-player zero-sum alternate-turn games results in Alternating Markov Games (AMGs) [123]. An alternating Markov game is a tuple $(\mathcal{S}_1, \mathcal{S}_2, \mathcal{A}_1, \mathcal{A}_2, \mathcal{R}, \mathcal{P}, \gamma)$ where $\mathcal{S}_1$ and $\mathcal{S}_2$ are respectively *this* and *other* agent states, and $\mathcal{A}_1$ and $\mathcal{A}_2$ are the actions at each player's states. MDPs can be viewed as a special case of AMGs where $|\mathcal{S}_2| = 0$.

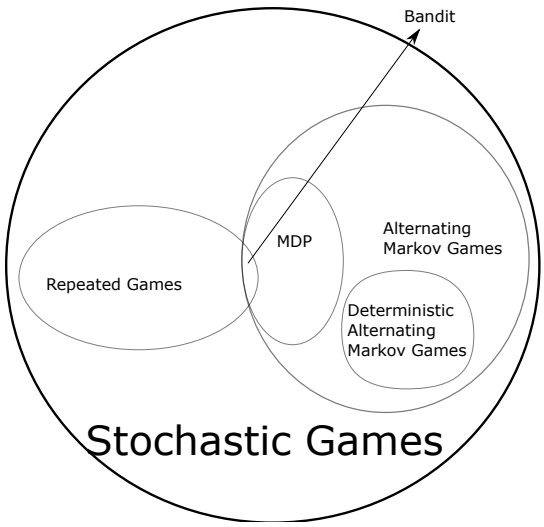


Figure 2.10: MDPs and AMGs are all special cases of Stochastic Games [179]. Repeated Games [188] contain one state, whereas MDPs contain one player. The game of Hex belongs to Deterministic Alternating Markov Games as for each action, the next resulting state is deterministic. Bandit games [117] are the intersection of MDPs and Repeated Games as they contain single player with one state but with repeated trials.

Figure 2.10 shows the relations between various learning formulations. Although some reinforcement learning algorithms developed for MDPs can be applied to two-player games [123], strictly speaking, the game of Hex has two players but no stochastic state transitions, thus it should be classified as a Deterministic Alternating Markov Game (DAMG).

2.3 Techniques for Strategy Discovery

In this section we review best first search algorithms for AND/OR graphs and discuss the connection between algorithmic variants. We then review reinforcement learning and deep neural networks. Finally, we discuss algorithms that combine learning and search, with a particular focus on Monte Carlo Tree Search. In these discussions, we comment on how these methods could be applied to the game of Hex and how they can be helpful to our goal of devising better playing and solving Hex algorithms.

2.3.1 Informed Best-first Search

The state-space graph can be too large to be explicitly represented. For example, 11×11 Hex contains about 2.38×10^{56} states [35], which is far beyond the capacity of computer storage. Instead of creating the entire graph, rules of the game are given

for creating the graph incrementally. The hidden, complete graph is conventionally refereed to as the *implicit graph*; the transparent, partial subgraph that an algorithm works on is called the *explicit graph*; they are noted as G and G' respectively. Leaf nodes with no successors in G' are called *frontier* or *tip* nodes. A tip node whose value is immediately known is called terminal and can be *solvable* or *unsolvable* — in the context of cost minimization, they are assigned values 0 and ∞, respectively. The process of generating successors for a non-terminal tip node is called *expansion*.

Starting from a single node, a search paradigm enlarges G' gradually by a sequence of node expansions. Suppose the merit of each node in G' can be assessed by a value function. To find a solution faster, it is natural to select the node with the most promising evaluation for expansion, and after each node expansion, relevant assessments must be updated. Following such an idea, a general best-first search (GBFS) has been depicted in [153] for searching solution-graph in general AND/OR graphs. Yet, one regularity for the AND/OR graph of a two-player alternate-turn game is that AND and OR nodes appear alternately in layers. Therefore, for a given recursive cost-scheme Ψ, it is possible to define a pair of intermingled value functions $\{\phi(n), \delta(n)\}$ recursively, as in Eq. (2.3).

$$\phi(n) = \begin{cases} h(n) & n \text{ is non-terminal tip node} \\ 0 & n \text{ is terminal winning state} \\ \infty & n \text{ is terminal losing state} \\ \min\limits_{n_j \in successor(n)} (c(n, n_j) + \delta(n_j)) & n \text{ is AND node} \end{cases}$$
$$\delta(n) = \begin{cases} \bar{h}(n) & n \text{ is non-terminal tip node} \\ \infty & n \text{ is terminal winning state} \\ 0 & n \text{ is terminal losing state} \\ \Psi_{n_j \in successor(n)}(c(n, n_j) + \phi(n_j)) & n \text{ is OR node} \end{cases} \tag{2.3}$$

If we interpret $h(n)$ and $\bar{h}(n)$ as the difficulty of proving n is winning and losing respectively (with respect to the player to play at n), using $\sum$ for Ψ, letting $h = \bar{h} = 1$ and all edge cost $c(n, n_j) = 0$, GBFS with Eq. (2.3) becomes proof number search (PNS) [2], except that PNS was originally defined on trees. Figure 2.11 shows an example for PNS, where node j is to be selected for expansion. After that, the ancestor nodes of j will be updated due to the change in j.

If replacing Ψ as *max* in Equation (2.3), the goal becomes a minimax strategy. To indicate that a game is zero-sum, it is natural to let $h(n)$ and $\bar{h}(n)$ to be additive

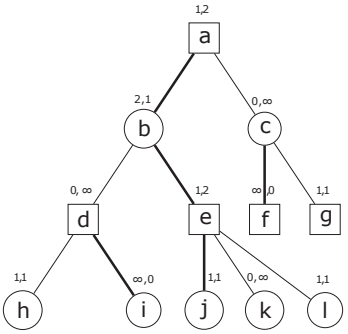


Figure 2.11: PNS example graph. Each node has a pair of evaluations (ϕ, δ), computed bottom up. A bold edge indicates a link where the minimum selection is made at each node. In PNS, all edge costs are 0, and a recursive sum-cost is used. PNS often uses $\{1, 1\}$ for h and $\bar{h}$, therefore $\phi(n)$ and $\delta(n)$ can be interpreted as the minimum number of leaf nodes needed to *prove* and *disprove* node n (with respective to the player at n), respectively.

inverses by restricting $h(n) + \bar{h}(n) = 0$. In games literature, such a realization of GBFS is called *best-first minimax*, and its practical performance in game-playing has been studied in Korf *et al*. [113].

Given limited partial exploration of the implicit graph G', goodness of root evaluation depends on the quality of evaluations by h. If further assuming that after depth d, evaluations given by h must agree with true value of the game state (i.e., winning is always associated with a value larger than 0) and all edge costs are 0, the remaining task is how to discover "optimal" root value by ignoring parts of the graph without bypassing the optimal. Algorithms like $\alpha\beta$ pruning [109] conduct a depth-first search while maintaining two bounds α and β to prune further branches if it is guaranteed that better values will never be discovered from these lines of search. SSS* [156, 164, 190] achieves a similar goal but using a best-first mechanism that runs multiple traversals to obtain sharper bounds for pruning. However, the principal drawback of depth-first search is that decisions made by problem solver are *irrevocable*, *i.e.*, a second alternative will never be checked until work on the first move is done. Such a feature makes the practical merits of $\alpha\beta$ pruning depend on the quality of move ordering [1, 170]. Iterative deepening modifies the depth-first behavior by doing multiple depth-limited search. Other enhancements of $\alpha\beta$ pruning reexamine the assumption that evaluation below depth d is exact, proposing a number of techniques, *e.g.*, forward pruning [52], quiescence search [17], to improve the accuracy of leaf evaluations. Algorithms following similar lines of research have produced superhuman playing programs in various games like Othello [135],

chess [40] and checkers [171], but less successful in some other games (*e.g.*, Go and Hex) where simple and reliable evaluation function is difficult to construct [32]. Indeed, because of the heuristic nature of evaluation function h, in some domains, $\alpha\beta$ based minimax search may even produce pathological behavior, *i.e.*, deeper exploration produces worse evaluations [138, 139, 140]. Pathology can occur because optimizing upon heuristic evaluation could only compound estimation error [152]; in this sense, the high branching factors in Go and Hex also make them less suited to moderately accurate evaluation based minimax search.

The development of PNS [4] was originally inspired by conspiracy number search [129], with the motivation to design algorithms specialized to solve games, especially in the presence of deep and narrow plays before reaching terminal nodes — where techniques based on $\alpha\beta$ pruning fixed-depth search become inadequate. PNS in its naive form initializes $\{h(n), \bar{h}(n)\}$ as $(1, 1)$ and assumes all edge costs are 0. Other enhancements try to establish more informative initialization of proof and disproof numbers at each newly created node [33, 34, 212]. Depth-first proof number search (DFPN) [136] is a PNS variant that adopts two thresholds to avoid unnecessary traversal of the search tree — it conducts Multiple Iterative Deepening (MID) until these thresholds are violated. DFPN has the same behavior as PNS in AND/OR trees, but exhibits lower memory footprint at the expense of re-expansion; it is also complete in directed acyclic graphs [107]. DFPN is often more applicable than PNS, and can often be improved by incorporating various general or game-dependent techniques. Yoshizoe *et al.* [216] introduced λ search to DFPN to solve the capturing problems in Go; threshold controlling and source node detection [105] were introduced to DFPN to deal with a variety of issues from Tsume-Shogi.

We show the relations between several representative algorithms in a hierarchical diagram of Figure 2.12. How they are connected is explained in Appendix A.4. These algorithms assume the AND/OR graph is a DAG, i.e., no directed loops exist. For general single agent finite MDPs, because of the existence of uncertainty nodes, the underlying state-space can also be modeled as an AND/OR graph, though these graphs may contain directed loops. A variant of AO*, namely LAO* [76], has been proposed for such cases: it sidesteps the problem caused by directed loop using a *value iteration* [158] procedure that will be discussed in the next section.

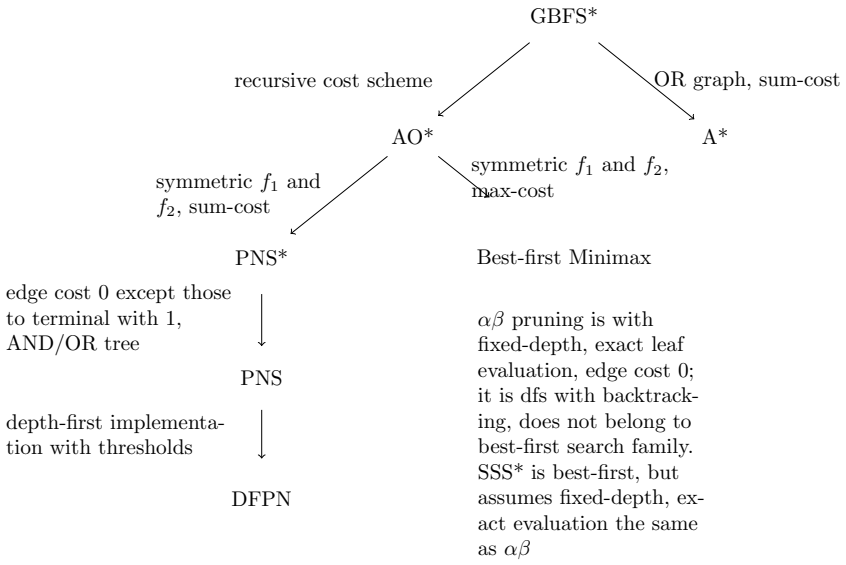


Figure 2.12: Relation of several search algorithms. AO* is a variant of GBFS in AND/OR graphs, while A* is for OR graph.

2.3.2 Reinforcement Learning

Unlike search algorithms that aspire to explore a small portion of a space graph, reinforcement learning (RL) methods aim to learn the true value of each node from experience. In MDPs, the basis for RL is a set of *Bellman Equations*. For a given policy π, let the value for state s under π be $v_\pi(s)$. The Bellman equation is defined as follows.

$$v_\pi(s) = \sum_a \pi(s,a) \sum_{s'} \Pr(s'|s,a)(r(s,a,s') + \gamma v_\pi(s')), \tag{2.4}$$

where $\Pr(s'|s,a)$ is the transition function and $r(s,a,s')$ is the reward of taking a at s, leading to s'. It is also popular to use an action-value function, defined likewise:

$$q_\pi(s,a) = \sum_{s'} \Pr(s'|s,a)(r(s,a,s') + \gamma v_\pi(s')) \tag{2.5}$$

These Bellman equations are the basis for *policy evaluation* [158], which computes the true value function for a given π. The *Optimal Bellman Equation* is a recursive relation for the optimal policy:

$$v_*(s) = \max_a \sum_{s'} \Pr(s'|s,a)\Big(r(s,a,s') + \gamma v_*(s')\Big), \tag{2.6}$$

$$q_*(s,a) = \sum_{s'} \Pr(s'|s,a)(r(s,a,s') + \gamma \max_{a'} q_*(s',a')). \tag{2.7}$$

Given v explicitly stores the value of each state, a value iteration procedure derived from the optimal Bellman equation converges to optimal [20]. Alternatively, it is possible to define a *policy iteration* [27, 94] by combining *policy evaluation* by Equation (2.4) and *policy improvement* due to (2.6).

In Alternating Markov games, for RL algorithms, because there are two players, the *policy evaluation* must involve two policies, i.e., π_1 and π_2. Bellman equations for policy evaluation are as follows [123]:

$$\begin{cases} v_{\pi_1}(s) = \sum_a \pi_1(a|s) \sum_{s'} \Pr(s'|s,a)(r(s,a,s') + \gamma v_{\pi_2}(s')), s \in \mathcal{S}_1 \text{ and } s' \in \mathcal{S}_2 \\ v_{\pi_2}(s) = \sum_a \pi_2(a|s) \sum_{s'} \Pr(s'|s,a)(r(s,a,s') + \gamma v_{\pi_1}(s')), s \in \mathcal{S}_2 \text{ and } s' \in \mathcal{S}_1 \end{cases} \tag{2.8}$$

Action-value functions can also be defined in the same fashion [123]:

$$\begin{cases} q_{\pi_1}(s,a) = \sum_{s'} \Pr(s'|s,a)(r(s,a,s') + \gamma \sum_{a'} \pi_2(a'|s') q_{\pi_2}(s',a')), s \in \mathcal{S}_1 \text{ and } s' \in \mathcal{S}_2 \\ q_{\pi_2}(s,a) = \sum_{s'} \Pr(s'|s,a)(r(s,a,s') + \gamma \sum_{a'} \pi_1(a'|s') q_{\pi_1}(s',a')), s \in \mathcal{S}_2 \text{ and } s' \in \mathcal{S}_1 \end{cases} \tag{2.9}$$

Let π_1 be the *max* player and π_2 be the *min* player. Assuming π_2 is an optimal *counter policy* with respect to π_1, we may rewrite the above equation as follows:

$$\begin{aligned} q_{\pi_1}(s,a) = \sum_{s'} \Pr(s'|s,a) \Bigg\{ & r(s,a,s') + \gamma \min_{a'} \sum_{s''} \Pr(s''|s',a') \\ & \left[r(s',a',s'') + \sum_{a''} \pi_1(a''|s'') q_{\pi_1}(s'',a'') \right] \Bigg\}, \\ & \text{where } s \in \mathcal{S}_1, s' \in \mathcal{S}_2 \text{ and } s'' \in \mathcal{S}_1. \end{aligned} \tag{2.10}$$

π_2 is replaced with a min operator, because π_1 is fixed and the problem reduces to a single agent MDP where an agent tries to minimize the received rewards. Assuming that states in $\mathcal{S}_1$ belong to the *max* player, optimal Bellman equations can be expressed as:

$$\begin{cases} v^*(s) = \max_a \sum_{s'} p(s'|s,a)(r(s,a,s') + \gamma v^*(s')), s \in \mathcal{S}_1 \text{ and } s' \in \mathcal{S}_2, \\ v^*(s) = \min_a \sum_{s'} p(s'|s,a)(r(s,a,s') + \gamma v^*(s')), s \in \mathcal{S}_2 \text{ and } s' \in \mathcal{S}_1. \end{cases} \tag{2.11}$$

A *value iteration* algorithm according to this minimax recursion converges to optimal [46]. However, because the *policy evaluation* in AMGs consists of two policies π_1, π_2, the policy iteration, which alternates between policy evaluation and policy improvement, can have four formats [45, 123]. The difference between AMGs and MDPs is reflected in finding solutions using learning programming (LP): a finite MDP can be solved by LP formulation and then computing optimal solution in

polynomial time w.r.t the number of states [123], whereas no LP solution is known for AMGs [46].

In the earlier literature, the policy and value iterations above, either for MDPs or AMGs, are called Dynamic Programming (DP) [94] algorithms. Implementation of these methods requires an explicit knowledge of $\Pr(s'|s,a)$. By contrast, most RL algorithms learn by assuming that the precise environment model is unknown, thus learning has to be done by repeatedly *exploring* the environment. Such algorithms interleave *policy evaluation* and *policy improvement*, and can be viewed as generalized policy iteration [192].

To make RL algorithms practically useful, instead of explicitly representing every state, a parameterized value function v_θ or q_θ can be used to approximate the tabular function and generalize across states. In this way, RL algorithm such as Q-learning and SARSA can be extended to work with parameterized approximated functions (called *function approximators*), though they may diverge [192]. A second approach is to directly use function approximator to represent policies and then optimize parameter weights by following the observed reward signals. For example, policy gradient methods [193] learn by iteratively adjusting a parameterized policy with respect to an estimated gradient. These methods have stable convergence in practice [155].

We provide a brief taxonomy of RL algorithms as follows. Depending on whether the agent is aware of the transition model or not, RL algorithms can be divided into *model-free* and *model-based*; the former can be further classified into three categories: policy-based, value-based, and actor-critic [192]. A simple policy gradient algorithm is REINFORCE [211]. Model-based RL can be divided into three types. DP algorithms require an *explicit* model with exact representation for every possible state. Search-based RL only requires rules for generating the next state after each action; they can rely on a function approximator for generalization. The third type tries *learn an approximately correct model* from experience; one example of such an approach is Dyna [191].

Both value and policy-based RL are well-studied in MDPs. Two player alternate turn zero-sum games are AMGs not MDPs [123]. In the literature, with a notion of *self-play* policy, standard reinforcement learning methods developed in MDPs have been applied to AMGs [180, 198] by simply negating the reward signal of the opponent's turn. However, since they do have fundamental differences, a careful

examination of these lead to better RL methods for two-player games, as we will see in the next chapter.

2.3.3 Deep Neural Networks

We have seen from the previous section that, when the state-space is large, practically useful RL algorithms have to rely on effective function approximator for generalizing across *all* states. Exploring only a subset of examples for grasping a *generally useful concept* is exactly the promise of machine learning, whose challenge is to find, by a process of training on some data, an approximately correct target function that is able to generalize across unseen inputs.

A fundamental problem is how to represent inputs in a convenient format that is easy to learn. Using neural networks for representation learning or feature learning [22] allows a system to automatically discover appropriate features from raw data. In particular, deep learning with many layered neural networks [21, 118] has been shown to be effective in practice; they exploit the composition hierarchies of natural signals. The study of neural networks dates back to 1940s–1960s [175]. Convolutional neural networks (CNNs) have been shown to be particularly effective for learning. The core idea of CNNs is to use *filters* (or *kernels*) for transforming information from one layer to next. Such an idea was proposed in 1970s [61], and became well-known after LeCun [119]. Figure 2.13 illustrates a convolution operator for a 2D input. Each filter (kernel) contains a vector of weights, applying which step by step horizontally and vertically using the same filter transforms the original picture into a new image. Here the filter is with size 3×3; the input image has an extra border padding with 0, therefore the resultant image remains the same size. More generally, for a $W \times W$ image, suppose the padding is P, convolution size is $F \times F$, stride is S, then the resulting image has dimension $W' \times W'$ where $W' = \frac{W+2P-F}{S} + 1$.

A fundamental problem for deep neural networks is how to efficiently update the connection weights, given the observed difference between the predicted output and the *labelled true result*. Backpropagation [37, 54, 55, 103, 122, 166, 210] has become a standard algorithm for training feedforward networks.

Due to computational and data acquisition limits, training many-layered networks was infeasible before the use of GPUs [144]. In 2012, the record-breaking winner [114] (AlexNet) of the ImageNet classification contest [51] popularized the

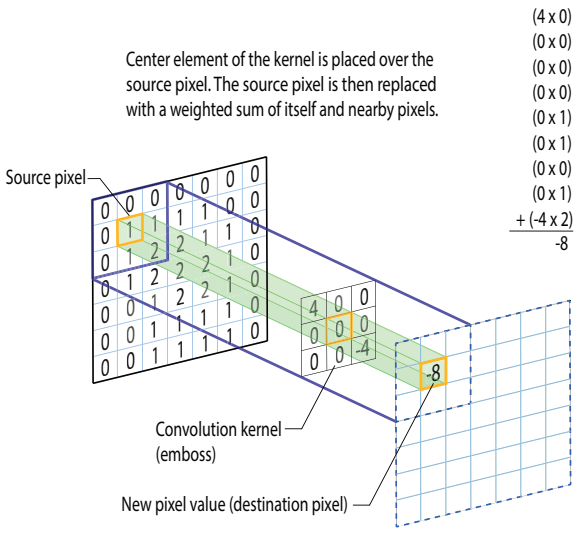


Figure 2.13: Illustration of convolution operation in 2D image.

use of deep neural nets implemented with GPUs in the computer vision community. The empirical success suggests that neural network depth is critical for achieving high performance. The accompanied problem is that network training becomes more difficult with increasing depth due to the phenomena of exploding and vanishing gradients. He et al. [84] proposed Residual Net (ResNet) that uses *short-cut* connections to tackle the gradient vanishing/exploding problem, making it possible to train neural nets with over 100 layers. In summary, research in deep learning are mostly concentrated on the following aspects:

- Architecture. The choice of a neural net architecture can be critical to deep learning applications. For a given application, a suitable CNN architecture may be found by many trials and novel design.

- Optimization. In practice, backpropagation is usually implemented with Stochastic Gradient Descent (SGD) [31] because the dataset is often too large to be loaded as a whole.

In RL, there is a long history of using neural nets as function approximators [27, 198], because practical RL relies on expressive function approximation to generalize learned knowledge to unseen states and many layered neural nets can approximate arbitrary continuous functions on compact subsets of $\mathbb{R}^n$ [93]. Revived interest in RL methods with neural nets as function approximators is partly due to faster training methods on GPUs. Deep Reinforcement Learning (DRL) combines the progress in

deep learning and reinforcement learning, leading to improvements in playing Atari games [132, 133, 173] and many continuous control applications [120, 176, 177, 183, 208].

In the game of Hex, another goal for neural architecture design is to let a model obtained on some board size $N \times N$ transfer to other board sizes. Transfer Learning [146, 161, 217] studies how to *transfer* learned knowledge from a source domain $\mathcal{D}_S$ to a target domain $\mathcal{D}_T$. Given a neural net model trained source domain data, common transfer learning scenarios [217] are as follows:

- *Frozen.* Trained neurons from a source task are copied to the target network and kept fixed when training on the new dataset for the target task.
- *Fine-tuning.* The target network copies neurons from the source task, then the whole network is optimized as usual.

We investigate transfer learning for Hex in Chapter 4.5.

2.3.4 Combining Learning and Search: Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [47] has led impressive progress in Go [58, 180], Hex [11, 96], and many sequential decision making problems [36]. Like other best-search algorithms, MCTS grows a selective tree by concentrating on nodes with better value estimates. Each search iteration consists of four distinct phases. (1) The *in-tree* phase traverses the current tree from the root until a leaf is reached. Child nodes are selected by a function such as UCT [111], which balances *exploration* and *exploitation*. (2) The *expansion* phase expands a leaf node, typically after its visit count has reached an expansion threshold. (3) Leaf nodes are evaluated, for example by randomized rollouts. (4) Results are backpropagated in the tree. While MCTS works without any game specific knowledge, its performance can often be drastically improved by incorporating domain knowledge [73], better child node *selection* and leaf evaluation.

Many improvements of MCTS has been proposed. UCT search [111] uses upper confidence bound in the selection phase, striking a balance between exploration and exploitation. Gelly *et al.* [74] propose Rapid Action Value Estimation (RAVE) heuristic, which leads to a faster accumulation of statistics and better performance in games where move values are only slightly influenced by their playing order.

MCTS based program Fuego [58] became the first computer player winning a game against a 9-dan professional in 9×9 Go.

In the game of Go, several research groups showed that deep learning can indeed provide high quality expert move predictions [44, 125, 200]. This strength was then used in conjunction with MCTS, leading to the Policy Value MCTS (PV-MCTS) [180, 184] that uses a deep policy net to set prior move probabilities and a value network for estimating the value of leaf nodes. The resulting AlphaGo program [180] convincingly beat top human professional players [180] in 19×19 Go. AlphaGo Zero [184] adopts the same search framework, but totally abandoned roll-out; it yielded stronger playing strength as because better quality neural nets were obtained by an iterated training that optimizes the neural net based on data generated by PV-MCTS with previous neural net models. These successes were achieved on advanced hardware TPU [100] for fast neural net inference and large scale parallel computation for data generation: AlphaZero used 5000 TPUs during training [181].

It is a research question how to further improve the efficiency of PV-MCTS on regular hardware and obtain large number of high quality expert games with limited computation resources. We present our result in improving AlphaZero style learning in Chapter 4.

2.4 Hex Specific Research

In this section we review past research that are specific to the game of Hex.

2.4.1 Complexity of Hex

Determining the winner for arbitrary Hex position has been shown to be PSPACE-complete [60, 162]; therefore, developing an automatic algorithm for solving arbitrary Hex position with polynomial time complexity is unlikely unless P=PSPACE, consequently P=NP. In practice, what we are mostly interested in is the number of existing states for a given board size.

For $N \times N$ Hex, the total number of reachable states is upper bounded by $\sum_{x+y+z=N^2 \wedge (x=y \vee x=y+1)} \binom{N^2}{x,y,z}$, i.e., enumerating all possibilities by fillin the board with x black stones, y white stones, and z empty cells (by the rule of Hex $x = y+1$ or $x = y$ always holds assuming Black plays first). Consider also that each board configuration has a symmetry by 180-degree rotation, the upper bound should be halved. Therefore, when $N = 11$, such an estimation gives $\approxeq 2.3835 \times 10^{56}$ [35].

However, the estimation method used above would count some states where a win/loss is already decided many times, whereas to form a winning chain at least $2N-1$ stones have to played. This gives a lower bound on DAG

$$\sum_{x+y+z=N^2 \wedge (x=y \vee x=y+1) \wedge x+y \leq 2N-1} \binom{N^2}{x,y,z}$$

. For $N = 11$ we obtain about 3.06×10^{29}. The estimation from [38] shows that for $N \times N$ Hex, the minimum number of steps required to win is at least $N + \lceil N/4 \rceil$. This provides us a way to estimate the lower bound for minimum size of state-space where there is a solution graph: 1.39×10^{21} for $N = 11$. Table 2.1 summarizes the results for $N = 9, 10$, 11 and 13. These estimates imply that techniques for pruning the state-space must be employed, otherwise solving Hex by exhaustive search even on small board sizes such as 9×9 can be unachievable. As in Chapter 2.1, if regarding the state-space is a tree of size b^d, then a proof tree is of size $b^{d/2}$. The numbers in Table 2.1 further imply that the representing a solution can also be difficult as the board size increases if no state pruning or state-aggregation techniques are employed.

Table 2.1: Approximate number of estimated states for $N \times N$ Hex, $N = 9, 10, 11, 13$.

	9×9	10×10	11×11	13×13
Upper Bound	10^{37}	10^{46}	10^{56}	10^{79}
Lower Bound	10^{22}	10^{25}	10^{29}	10^{37}
Lower Bound (Sol)	10^{14}	10^{17}	10^{21}	10^{25}

2.4.2 Graph Properties and Inferior Cell Analysis

A graph can be constructed from either Black or White's perspective. For instance, a Black Hex graph is created by treating two black borders as two distinct vertices, every uncolored cell is a vertex with edges connecting neighboring cells; a black stone on a location makes all pairs of neighboring cells become adjacent after the vertex on which Black just played; such a procedure is in short called *vertex implosion* — a white stone to a location deletes the corresponding vertex and all its edges. Black wins if the two black border vertices become adjacent; conversely, White wins if the two black border nodes are in two independent connected components. A White graph can be built in the same fashion. Figure 2.14 demonstrates an example from [206]. In Black's Hex graph, Black is called the *Short* player and White the

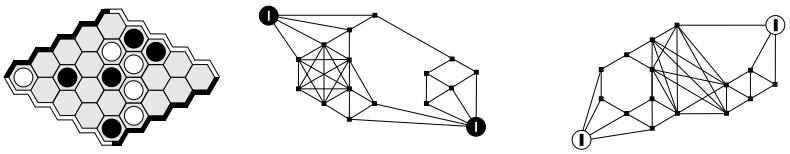


Figure 2.14: An example Hex position and the corresponding graphs for Black and White players. Image from [206]

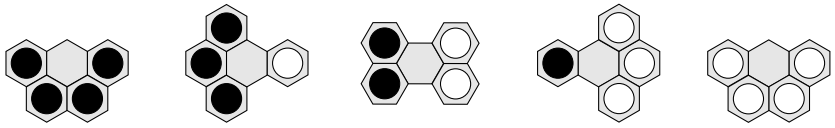


Figure 2.15: Dead cell patterns in Hex from [87]. Each unoccupied cell is dead.

Cut player; in White's graph, their roles are exchanged. If generalizing the concept to arbitrary graph, the resulting game is called *Shannon vertex-switching game* or *Generalized Hex* [15, 60, 99].

As suggested by Berge [23, 81], it is possible to prune some moves from a given game state by analyzing the graphical properties of Hex. Techniques for proving that a move can be discarded without changing the optimal value of a game state are collectively called Inferior Cell Analysis (ICA). Depending on how they prune cells, certain formal terminologies on a cell or a set of cells can defined, e.g., *dead*,*vulnerable captured vulnerable-by-capture* and *capture-domination* cells. See [87]. Figure 2.15 shows several examples of dead cells.

While deciding an unoccupied cell is dead or not in arbitrary graph is NP-complete [28], identifying and proving local dead-cell patterns can be useful in automated playing and solving Hex. See Figure 2.15 for some examples. A dead cell can be filled-in with any color without altering game theoretic value of a Hex position; similarly, a Black (or White) captured set C can also be filled-in with Black (or White) stones. The process of artificially filling the board with either black or white stones while ensuring game-theoretic value unchanged is called *fillin* in Hex. Besides dead and captured cells, another type of fillin by Henderson *et al.* [87] is called *permanently-inferior*.

Pruning by fillin is strong in the sense that filled cells are removed from consideration by any continuation of a game position. Indeed, following a set of dead,

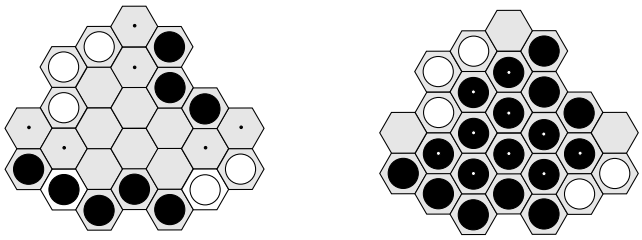


Figure 2.16: An example of Black captured-region-by-chain-decomposition from [87]; cells inside of such a region can be filled-in.

captured and permanently-inferior patterns, Henderson et al [91] described an explicit winning strategy using $\lceil \frac{N+1}{6} \rceil$ handicap cells on any $N \times N$ board. The strategy works by placing black stones in the row one cell away to border and apply fillin to fill the row next to boarder permanently, then the resulting irregular Hex board can be solved via a pairing strategy from Shannon [72].

A weaker type of pruning is to remove cells from given game state if a provable better one exists. A *reversible* move can be *bypassed* while a *dominated* move can be *pruned* given at least one dominating move retains [24]. Using these concepts, captured-reversible and captured-domination have been defined [87, 90]. Other domination classes explored in [87] include *induced-path* and *neighborhood* domination.

Additionally, board decomposition [87] are useful for fillin or pruning, given that a connection strategy in the relevant region is known. Figure 2.16 shows an example of region fillin as a result of chain composition. For any given board position, it seems that many cells can be either filled-in or pruned by chain or star decomposition. However, the practical feasibility of these pruning relies on known explicit strategy for constructing *interboundary connection*. For instance, for $N \times N$ Hex where $N \geq 3$ we have known that black acute corner opening is a loss [12, 18], it follows that the remaining board cells together with four borders form a chain decomposition with R containing all unoccupied cells, thus White captured-region-by-chain-decomposition. However, white fillin the whole board is meaningless as explicit strategy for preventing all black interboundary connection is missing. A bottom-up connection strategy computation can be used for carrying out the pruning or fillin of such kind in certain cases.

2.4.3 Bottom Up Connection Strategy Computation

We have seen in Chapter 2.1 that strategy for a game can be decomposed and represented by subgame strategies. Anshelevich [6, 7] discovered that some connection strategies for local subgames of Hex can be computed in a bottom-up manner. The algorithm *H-search* was proposed for building these strategies, which became a critical component in the 2000 ICGA champion player Hexy [5]. Two types of connection strategies are defined:

- Virtual Connection (VC) — given two endpoints x, y and a set of empty cells A, (x, A, y) is a Black VC if there is a way to connect x and y by only playing inside of A even if let White play first. That is, VCs represent second-player connection strategies. A is called *carrier* of the VC.
- Semi-virtual Connection (SC) — given two endpoints, x, y, a set of empty cells A, (x, A, y) is Black SC if there exists a $k \in A$ Black plays at k first, then $(x, A\backslash\{k\}, y)$ is a Black VC. That is, SCs stand for first-player connection strategies; the first move k fulfilling a SC is called the *key* for that SC.

In a Hex graph, because a connected group of same-colored stones can be abstracted into one point, for any $N \times N$ Hex from the empty board, there must be a Black SC since it has been proved that there exists a first-player winning strategy. The PSPACE-complete complexity result, however, implies that finding such SCs for arbitrary Hex board size is difficult. Nevertheless, using the particular property of Hex we can compute some connection strategies in a bottom-up manner. To achieve this, H-Search uses some base cases and two combination rules.

1. Base Cases: any adjacent pair of same-colored endpoints form a base VC with carrier $\emptyset$, i.e., ●——●; any pair of same-colored endpoints with one empty cell between them forms a base SC, i.e., ●——○——●.
2. AND-rule: Two VCs (x_1, A_1, x_2) and (x_2, A_2, x_3), with $(\{x_1\} \cup A_1) \cap (\{x_3\} \cup A_2) = \emptyset$, can be combined to form a new SC if x_2 is an empty cell or a VC if x_2 has the same color as x_1 and x_3.
3. OR-rule: Two SCs (x_1, A_1, x_2) and (x_1, A_2, x_2), with $A_1 \cap A_2 = \emptyset, A_1 \neq \emptyset, A_2 \neq \emptyset$, can be combined to form a new VC. The bridge pattern shown in Figure 1.3 can be obtained by ORing the two base SCs sharing the same pair of endpoints. This rule also applies to more than two SCs sharing the same

pair of endpoints, provided that their carrier intersection is the empty set and none of them is the empty set.

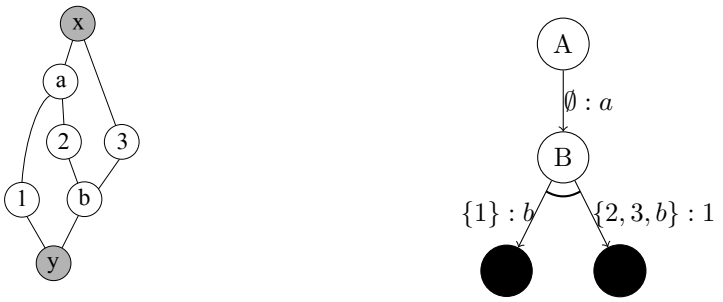
Given a set of manually created base cases, H-Search automatically and repeatedly applies the above OR- and AND- rules until no more connections can be found, resembling to automatic theorem proving [163]. The `Apply_OR_Rule` of H-Search is recursively defined [7] and can be expensive in practice since it tries to find new VCs by enumerating all combinations of semi-virtual connections sharing the same two endpoints. As noted in [87], a number of ways could be used to make H-search either find more connections or become faster in practice:

1. Enlarge the set of base connections.
2. Abandon superset carriers.
3. Allowing border to be midpoints.
4. ORing all SCs check: if ORing all SCs does not produce a VC indicate that ORing any subset of them cannot produce a new VC.
5. When appplying the OR-Rule, backtrack immediately if the intersection does not shrink (since it implies this SC is redundant).
6. Only check at most k combinations when applying the OR rule.
7. Limit the number of SCs and VCs stored for each pair of endpoints.

The last two schemes can reduce the number of connections that H-search can find, but, arguably, they bring more benefit than harm due to the improved speed [87]. Indeed, even without these, H-search does not guarantee to find all connection strategies, i.e., H-search is incomplete [7]. One simple example that H-search is incapable to solve is called *braid*, illustrated in Figure 2.17 (left). A solution by strategy decomposition is depicted in Figure 2.17 (right).

The reason that H-search cannot find a SC between x and y via AND-rule is that all the VCs and SCs are intervened. The attainable VCs are $(x, a, \emptyset)$, $(x, 3, \emptyset)$, $(y, 1, \emptyset)$, and $(y, b, \emptyset)$; SCs are $(a, b, key = 2, \emptyset)$, $(x, b, key = 3, \emptyset)$, $(y, a, key = 1, \emptyset)$, $(a, b, key = 1, \emptyset), (a, b, key = 3, \emptyset)$. Applying AND-rule does not produce a SC between x and y simply because there is no two VCs to concatenate. By defining *Partition Chains* that can be computed in parallel with AND/OR rules, a *Crossing Rule* is proposed in [87] to solve braid-type cases.

Although the improvement may lead H-search to find more connections, they do not make H-search complete [87]. Pawlewicz et al. [151] observed that computational



(a) The braid example

(b) Solve braid by decomposition

Figure 2.17: The braid example that H-search fails to discover a SC connecting to x and y (left). Either a or b could be the *key* to a SC. The right subfigure shows a decomposition represented solution: after playing at cell a, move 1 can be responded with b while any move in $\{2, 3, b\}$ can be answered by move 1.

efficiency is a major bottleneck for H-search, then described a more efficient data structure for storing and updating VCs and SCs. In particular, the OR rule is exponentially expensive with respect to the number of SCs between two endpoints, while hard-limiting it to small (e.g., 3 or 4) prevents H-search from finding many useful connections, Pawlewicz et al. [151] proposed *semi-combiner* and *fast-semi-combiner* for producing *critical* subsets of VCs.

The source of incompleteness of H-search comes from the fact that its bottom-up combination rules are just a special treatment to achieve the goal of solving subgames by strategy decomposition. For the braid example, as in shown Figure 2.17b, the SC between x and y can be formed by two VCs ($x \to a$ and $a \to y$), but the AND rule of H-Search failed to compute the SC. In general, as demonstrated in Figure 2.18, if we want to prove that there is a SC in position A for player P, by decomposition, it is equivalent to finding a move that will lead to a position where there is an VC. To prove position B has a VC for player P is to decompose the VC into several smaller VCs. Rather than using H-Search, in Chapter 5.4, we discuss the possibility of searching for decomposition-based solutions with the help of deep neural networks.

2.4.4 Iterative Knowledge Computation

Not only inferior cell analysis can help H-search [87, 151], connection strategies from H-search can also help to identify more fillin or inferior cells [87]. Therefore, for a given Hex game state, combining these two types of knowledge computation results an iterative procedure, sketched in Algorithm 1 [87].

Upon the termination of Algorithm 1, two possibilities exist: (1) the state-value

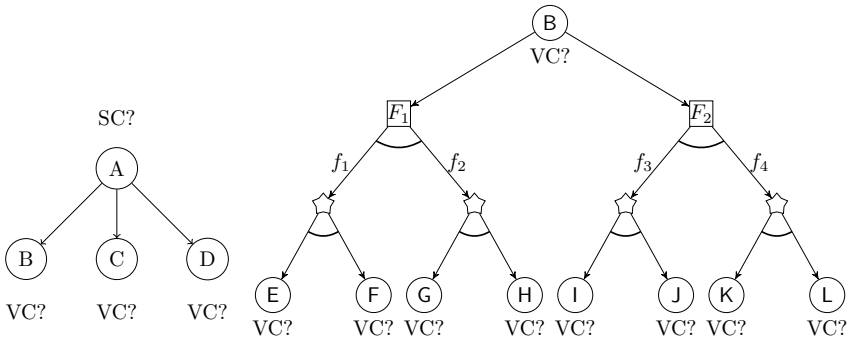


Figure 2.18: Searching for SC and VC by strategy decomposition. Finding there is a SC for player P at position A equals to finding a P move that will lead to a child position where there is a VC for P.

Algorithm 1: Iterative Knowledge Computation [87]

Input: A board position s
Result: Processed board position s'

1 For both players, compute dead, captured, permanently-inferior cells for fillin; do this step repeatedly until no new fillin can be found;
2 For the player to play, compute dead-reversible, captured-reversible and various domination cells for pruning;
3 For both players, run H-search on the fillin-reduced position;
4 Apply deduced connections, compute captured-region-by-chain-decomposition, dominated-region-by-chain-decomposition and dominated-set-by-star-decomposition; if new fillin is produced, go to step 1, otherwise exit;

is decided, (2) the state-value is unknown but certain moves can be pruned. Such an iterative knowledge computation at each game state has been implemented in Benzene, available from [2], with visualization using HexGUI [3]. See Figure 2.19. We will use this platform to conduct our experiments in later chapters.

2.4.5 Automated Player and Solver

In Hex, a seminal work for designing a computer player is due to Shannon [178], who proposed an electric-resistance based evaluation for playing the game. Research for other games (e.g., DeepBlue for chess [40]) have produced human-level playing machines. However, it has been shown that designing effective heuristic evaluations for Hex is difficult: Concepts used in many other games for feature extraction, such as material, balance, and mobility, are meaningless for Hex [205]. Evaluations

[2] https://github.com/cgao3/benzene-vanilla-cmake
[3] https://github.com/ryanbhayward/hexgui

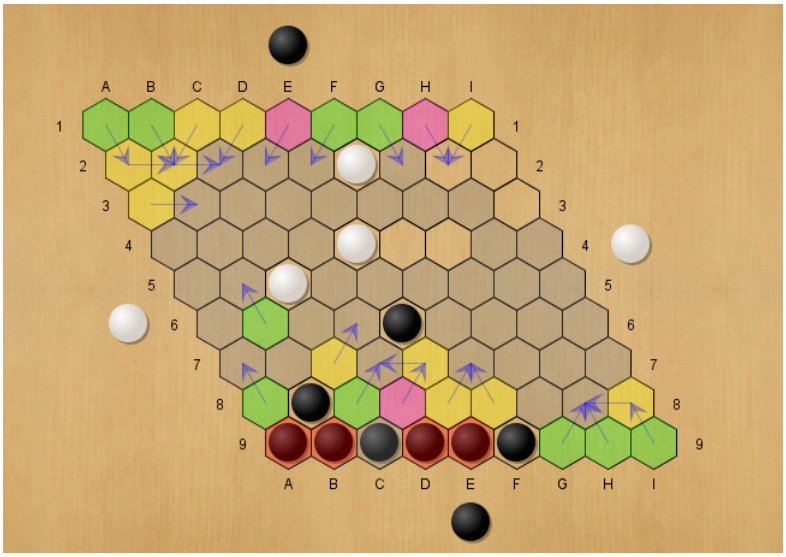


Figure 2.19: Knowledge computation visualized via HexGUI. Black played stones: $b8$, $f9$ and $e6$. White played stones: $c5$, $e4$ and $f2$. Black to play. $a9, b9, c9, d9$ and $e9$ are Black fillin. Gray shaded cells are pruned due to mustplay; black filled cells with pink shading are captured; gray filled with shading are permanently-inferior; green: vulnerable; magenta: captured-reversible satisfying independence condition; yellow: dominated by various domination patterns. Only five cells remain to be considered after knowledge computation.

were based on either measuring properties of the game graph, such as network resistance [178] and graph distance [204].

Table 2.2 summarizes some major computer programs developed for playing Hex. Equipped with graph distance evaluation, together with an advanced implementation of $\alpha\beta$ search, Queenbee [204] became the first program achieving novice level strength. Inspired by Shannon's model, using electronic circuit resistance as an evaluation function in minimax search was used by Anshelevich [6, 7]; the resulting program Hexy [5] defeated Queenbee. Improved programs based on a similar idea include Six [79] and Wolve [13]. In all these programs, virtual connection computation is vital to the quality of evaluation, and they all use a form of H-search [7] for computing VCs. The $\alpha\beta$ player Wolve combines both ICA and VC to improve its heuristic evaluation function.

Although the resistance-based evaluation can be improved by adding VC and ICA information, it is still frequently pathological, as it tends to favor fillin and dominated cells [87]. This problem was sidestepped by the development of Monte Carlo Tree Search for Hex [11], which evaluates a game position by simulating random playouts. The success of MCTS is due to its superior ability to focus the

Table 2.2: Evolution of Computer Programs for Playing Hex.

Year	Program	Brief Description
2000	Queenbee	$\alpha\beta$ with two-distance
2000	Hexy	$\alpha\beta$ with circuit resistance plus VC computation
2003	Six	$\alpha\beta$ with circuit resistance plus VC computation
2008	Wolve	$\alpha\beta$ with circuit resistance plus VC and ICA
2010	MoHex	parallel MCTS with VC and ICA
2013	MoHex 2.0	parallel MCTS with stronger VC and ICA plus pattern-based playout

search on promising nodes in the tree, by dynamically backing up the evaluations from random sequences of self-play. Enhancements of MCTS increase the quality of such biases by incorporating learned off-line prior knowledge [73]. MoHex 2.0 by Huang et al. [96] uses preferences learned from expert game records. The program is 300 Elo stronger than MoHex [96] on the 13×13 board size. Besides MoHex 2.0 and Wolve, recent strong computer players, such as DeepHex [150] and Ezo [196], all use the VC and ICA computation available from the Benzene codebase.

As in Table 2.2, the discovery of H-search marked the beginning of successful application of $\alpha\beta$ search. The evaluation deficiency is only partially addressed by VC computation. There are also work trying to improve the evaluation function using more complicated network models [196], the resulting player failed to defeat MoHex 2.0 [83]. Despite the playing strength of the recent programs in Table 2.2, a mathematical guarantee on measuring the quality of their returned moves is lacking. While statistical models, such Elo rating [56], have been developed for ranking players, these models do not provide explanation how close a player's move is to optimal play, and in practice using a finite number of self-players do suffer from problems such as *Elo inflation* [159].

Instead of heuristically playing well, the other research direction is developing programs for solving Hex. Combining some ICA and VC computation, Hayward et al. [82] showed that a DFS on state-space graph can solve arbitrary 7×7 Hex openings in reasonable time. After extending inferior cell analysis and enhancing VC computation, all 8x8 openings were solved by Henderson et al. [89] via DFS. Switching the search to PNS made the solving all 8×8 openings at least twice faster [87]. To date, all 9×9 openings have been solved using parallel PNS with stronger VC [149, 151].

Given the success of deep neural networks for playing, a natural research direction is to further improve MoHex 2.0 by using deep neural networks for improving prior knowledge. Similarly, a PNS-based solving algorithm could incorporate deep neural networks for better node expansion. We shall present the results of improving PNS using deep neural networks in Chapter 5.

Chapter 3

Supervised Learning and Policy Gradient Reinforcement Learning in Hex

This chapter contains content from "Move prediction using Deep CNNs in Hex" [63] and "Adversarial Policy Gradient for Alternating Markov Games" [64].

3.1 Supervised Learning with Deep CNNs for Move Prediction

3.1.1 Background

The breakthrough [44, 125, 180, 200] in computer Go has shown that using deep convolutional neural networks (CNN) [114] for representation and learning game knowledge works better than earlier approaches — they showed deep CNNs can produce very high prediction accuracy of *expert moves* after training on professional game records. Motivated by the success in Go, we investigate how to use deep convolutional neural nets to represent and learn knowledge in Hex. The goal is use this knowledge for better for computer Hex programs— a task that poses the following challenges:

- Representation of input. Compact representation is essential to any learning model. Typically, feeding more input features to a neural net yields better prediction accuracy. However, since trained neural nets will eventually be used in search, evaluation needs to be sufficiently fast. Deep networks usually have better learning capacity [84], but the development of a successful architecture often requires many trials.

- Sophistication. Training data obtained from either human or computer game records is usually imperfect. Appropriately dealing with imperfect training data is essential.

- Search. Combining neural net with search efficiently is non-trivial. A challenge is that the speed of neural nets on regular hardware is too slow to be intensively used in the search.

The above challenges need to be solved one by one. In this section, focusing on 13×13 Hex, we conduct an empirical study of move prediction using deep neural networks in Hex, and investigate the possibility of improving the search in MoHex 2.0 given that an accurate move predictor is obtained.

3.1.2 Input Features

Let s denote an arbitrary Hex position to be preprocessed into feature planes as input to a neural net. Since the goal of Hex is to connect two sides of the board, to represent this goal, we use extra two borders to pad the Hex board. See Figure 3.1 for the padding on a 13×13 Hex empty board. Note that we use two, rather than one, border padding to highlight the importance of border stones for Hex. It is usually desirable to encode some useful features to the feature planes. We encode the common *bridge pattern* (see Figure 1.3) to the input. Table 3.1 lists the feature representation in each plane. Bridge endpoints are those occupied cells where there is a bridge pattern (as in Figure 1.3, the two black stones); a save bridge point is an empty cell by playing where saves the bridge (so in Figure 1.3, if White plays on one empty cell, the other one becomes a save bridge point for Black); similarly, a make-connection point is an empty cell by playing where it enables a connection of its neighbor cells of the same color. The input features consist of 9 planes with only binary values. We tried to add history information as [200], but experimental results show that it does not seem to help in Hex, perhaps because Hex is fully described by the current board position plus the player to play and the playing order of each stone is irrelevant to state evaluation.

3.1.3 Architecture

Following on previous work in Go [44, 125, 200], the neural network we designed for Hex is a straightforward stack of multiple convolution and non-linear activation

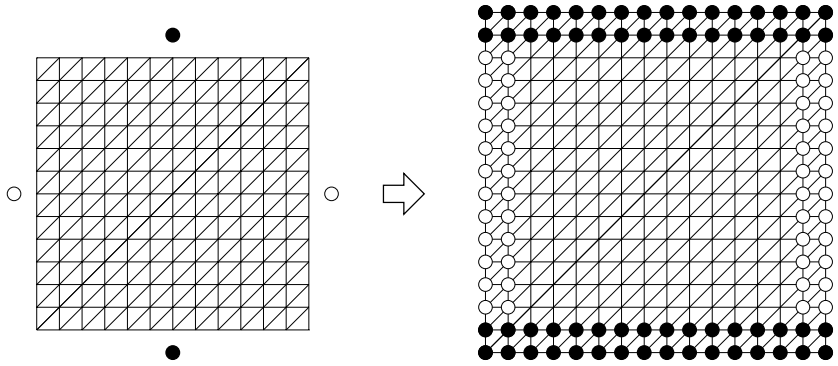


Figure 3.1: Hexagonal board mapped as a square grid with moves played at intersections (left). Two extra rows of padding at each side used as input to neural net (right).

Table 3.1: Input features; *form bridge* are empty cells that playing there forms a bridge pattern. Similarly, an empty cell is *save bridge* if it can be played as a response to the opponent's move in the bridge carrier.

Plane	Description	Plane	Description
0	Black played stones	5	White bridge endpoints
1	White played stones	6	To play save bridge points
2	Unoccupied points	7	To play make-connection points
3	Black or White to play	8	To play form bridge points
4	Black bridge endpoints		

layers. Let d (at least one) be the number of convolutional layers. Let w be the number of convolution filters in each layer. Input to the first convolutional layer is of dimension $17 \times 17 \times 9$ (13 plus 4 padding borders, 9 feature planes in total). The first convolution layer convolves using filter size 5×5 with stride of 1, and then applies ReLU to its output. Convolution layers from 2 to $d - 1$ zero pad the input to an image of 15×15 (see Figure 2.13 on convolution operator), then convolve with filter size 3×3 and stride of 1. Likewise, each of them is followed by ReLU.

Following previous work in Go [44, 125, 200], to avoid losing information, no downsampling layers such as max-pooling are applied. The final convolutional layer convolves using one filter of kernel size 1×1 with stride 1. A bias is added before applying the final softmax function. The output of this neural net is a probability distribution over each move on the board. Figure 3.2 shows the architecture.

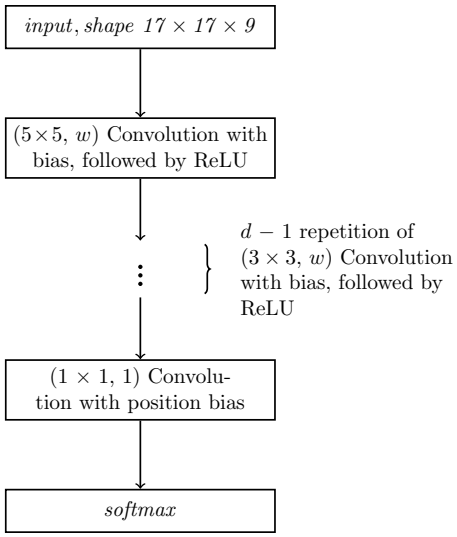


Figure 3.2: Architecture of d convolutional layers, each with w filters.

3.1.4 Data for Learning

A large amount of training data is essential for any deep learning model. According to [96], pattern weights in MoHex 2.0 were trained on a dataset consisting of 19760 13x13 games from *Little Golem* [1] and 15116 games by MoHex played against Wolve. We could not access the same dataset. Therefore, we generated a new dataset from MoHex 2.0 selfplay. To enrich the generated examples, we randomly set the time limit per move from 28 to 40 seconds, and played the tournament by iterating over all opening moves with the parallel solver turned off. In this way, we collected 15520 games on board size 13×13. These selfplay games were produced on several *Cybera*[2] cloud instances with Intel(R) Xeon(R) CPU E5-2650 v3 2.30GHz and 4GB RAM.

As a further preprocessing step, we extracted training examples from saved games, and removed duplicated state-action pairs; as a result, 1098952 distinct *state-action* pairs (s, a) are produced. We subsequently randomly selected 90% of them as the training set and used the remaining 10% as a test set. See Figure 3.3 for a visualization of the distribution of state-action pairs with move numbers. Not surprisingly, most positions are from the middle stage of the game.

No human games from Little Golem were used in our training: we found that most human players at this site are weaker than MoHex 2.0, although the top player could be significantly stronger.

[1] `http://www.littlegolem.net`
[2] `http://www.cybera.ca`

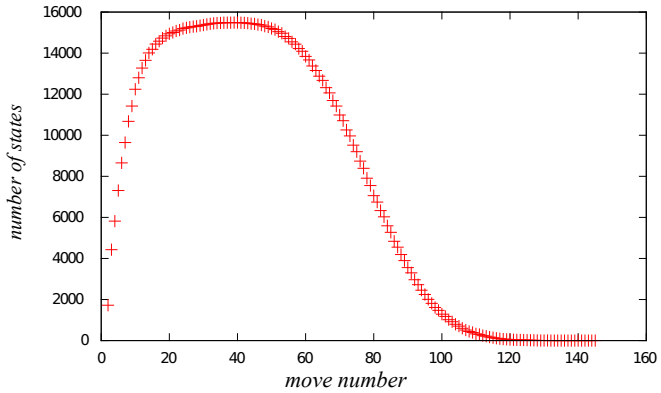


Figure 3.3: Distribution of (s, a) training data as a function of move number.

3.1.5 Configuration

The Hex board is symmetric under 180-degree rotation. Therefore, for each training example, with probability 0.5, a rotated symmetric position is randomly selected to train. The training target is to maximize the likelihood of move prediction in the training examples. Let $\mathcal{D}$ be the training set. We used this loss function:

$$\mathcal{L}(\theta; \mathcal{D}) = -\sum_{(s,a)\in\mathcal{D}} \log p_\theta(a|s) \tag{3.1}$$

For training, we used adaptive stochastic gradient descent optimizer Adam [104] with default parameters. Compared to vanilla stochastic gradient descent, Adam converged faster. We trained the neural net with a batch size of 128 per step. Every 500 steps, the accuracy is evaluated on test data. We stopped the training after 150,000 steps. The model that achieves the best test accuracy is saved. The neural net was implemented with Tensorflow [128], and trained on an Intel i7-6700 CPU machine with 32GB RAM and a NVIDIA GTX 1080 GPU.

3.1.6 Results

We present the prediction accuracy of several architectures with varying d and w. Evaluation of the playing strength of the best neural network model is presented subsequently.

Prediction Accuracy

Finding appropriate d and w requires experimentation: a too large network may require more training data while too small ones may have insufficient capacity. We

therefore experimentally vary d and w. We start by letting $w = 64, d = 5$ as in [125], and tried to double w to 128 (note that we stopped at $w = 128$ as another doubling would make training overwhelmingly slow on the GTX 1080 GPU). Table 3.2 presents the top 1 move prediction accuracy for 5 different configurations.

Table 3.2: Prediction accuracy on test set from CNN models with varying d and w

Choices of d, w	Best accuracy on test data
5 layers, 64 filters per layer	49.5%
5 layers, 128 filters per layer	53.4%
7 layers, 128 filters per layer	54.7%
8 layers, 128 filters per layer	**54.8**%
9 layers, 128 filters per layer	54.5%

Table 3.2 shows that neural nets with depths $7 - 9$ produced better results than the shallower ones. The best accuracy of 54.8% is with $d = 8, w = 128$, whose accuracy on *training* data is 57.6%. For each architecture, the result was obtained with *early stopping*, i.e., we stopped the training when the test accuracy stopped improving for 10 consecutive epochs. As the previous work in Go [125], no regularization is used.

It is of interest to know the top k accuracy for $k > 1$. If the accuracy is high for relatively small k, this characteristic could be harnessed to effectively reduce search width. Figure 3.4 shows the top k accuracy for the best $d = 8, w = 128$ neural network model with 9 input planes. For $k = 8$, the prediction accuracy is above 90%. For $k = 12$, the predication accuracy exceeds 95%. We now proceed to investigate playing strength of the best $d = 8, w = 128$ policy neural network model both without and with search. For brevity, we call this model CNN_8^{128}.

Playing Strength of CNN_8^{128} without Search

A policy network model can be used as a player whose move is sampled from the probabilistic outputs of the model. To see the strength of CNN_8^{128}, we test its performance by playing against the resistance evaluation used in Wolve [11, 13]. This is achieved by limiting Wolve to use 1-ply search. Note that 4-ply search was used by Wolve in previous tournament [11, 13].

The match consists of 6000 games from opening board position where 3000 neural net plays as Black, and 3000 as White. No swap rule was used. The result is that CNN_8^{128} won 48.9% of all games, which implies that the policy neural net model

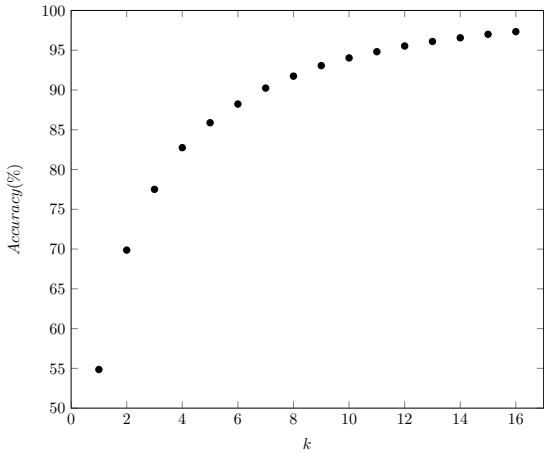


Figure 3.4: Top k prediction accuracy of the $d = 8,\ w = 128$ neural network model.

has similar playing strength as the optimized resistance evaluation in Wolve. As for speed, CNN_8^{128} is more than 10 times faster than 1-ply Wolve.

Figure 3.5 shows a typical game played by the neural net model against Wolve. The first move of Wolve is $l2$, presumably because there is a easily computable virtual connection to the top which greatly influenced the resistance evaluation. Move 11 of Wolve is problematic, as it became useless by White's response move $k8$ — there is no way that Black can connect to the bottom by $m7$ unless White misses $l9$.

The result is remarkable in the sense that, without any search, the neural network model can still answer most black moves accurately, implying that the neural net has already grasped some sophisticated aspects of the game.

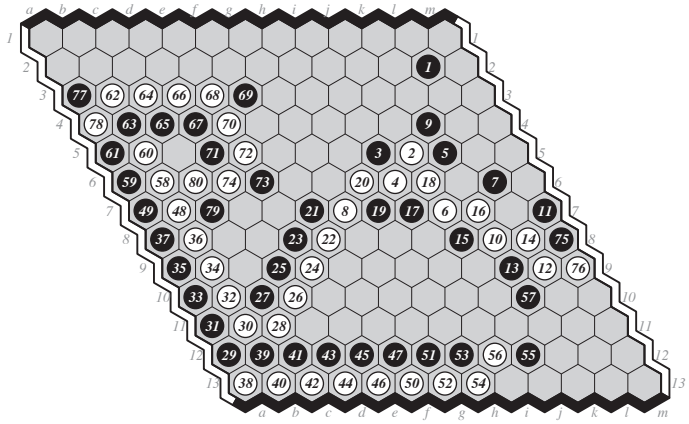


Figure 3.5: A game played by 1-ply Wolve (Black) against policy net (White) CNN_8^{1287}. The neural network model won as White. Note that we tried to use solver to solve the game state before move 11, but solver failed to yield a solution.

It is also interesting to see how well the neural net plays against 4-ply Wolve. Since this version of Wolve is slow, we only played 800 games in total. Table 3.3 shows the result. Table 3.3 also includes the result of CNN_8^{128} against 1000 simulation limited MoHex 2.0. Even against these strong Hex bots, the neural net could still achieve about 10% and 20% win as Black and White, respectively. As a comparison, using deep-Q learning [132], after two weeks of training, Young *et al*. [218] obtained a deep neural net model which achieved 20.4% win against 1 second limited MoHex 2.0 as Black and 2.1% as White. We note that in our Intel i7-6700 CPU machine — which is faster than the machine used in [218] — MoHex 2.0 with 1000 simulations almost always takes a time around 1 second, which implies that CNN_8^{128} could have achieved a better result. We suspect this is due to the following reasons: (1) The training data are different: in [218], the neural net is first trained on a set of games generated from a randomized version of Wolve, and then trained on games of neural net self-play. We believe those games are inferior to the games accumulated from MoHex 2.0 self-play. (2) The training method is different: Although Q-learning has been successful in Atari games [132], two-player strategic games are more challenging in the sense that the opponent could be regarded as a non-stationary environment. (3) The input planes in [218] contain only Black/White/empty points plus Black/White group information: our *bridge* enriched representation is better in the sense that it represents a key tactic for playing Hex.

Table 3.3: Results of $CNN_{d=8,w=128}$ against 4ply-Wolve and 1000 simulations MoHex 2.0.

Opponent	CNN_8^{128} as Black	CNN_8^{128} as White
4-ply Wolve	17%	3.25%
MoHex2.0-1000	33%	8.5%

Finally, although the comparison with Wolve shows that the playing strength of CNN_8^{128} is similar to resistance, it is advantageous over resistance in the sense that it does not have the pathological behavior (favoring fillin and dominated cells) as resistance nor does it require the computation of inferior cells and virtual connections which are typically slow [87].

Integrating CNN_8^{128} to MoHex 2.0

Our ultimate goal is to develop a stronger Hex player by combining the neural net with search. The strength of look-ahead search is that it tries to build a state-specific graph model, thus the possibility of sampling an erroneous action directly from the policy network is reduced.

The major challenge for combining a neural net with search is that move evaluation by deep networks can be slower than traditional methods [125, 180, 200]. In computer Go, previous work either employs non-blocking asynchronous batch evaluation [180] or simple synchronous evaluation [200]. In each approach, neural network evaluation is used as prior knowledge in search tree, giving preferable child moves higher prior probability.

We combine our neural network with MoHex 2.0, the reigning world champion player since 2013. MoHex 2.0 is an enhanced version of MoHex [11]. It is built upon Benzene, using the *smartgame* and *gtpengine* libraries from Fuego [58]. The major improvement of MoHex 2.0 over the 2011 version of MoHex lies on the knowledge-based probabilistic playout. Specifically, the MCTS in MoHex 2.0 work as follows:

- *In-tree phase*: In this phase, starting from the root node, a child node is selected from its parent until a leaf is reached. At tree node s, a move is selected according with the maximum score defined as:

$$(1-w)\times(Q(s,a)+c_b\times\sqrt{\frac{\ln N(s)}{N(s,a)}})+$$
$$w\times R(s,a)+c_{pb}\times\frac{\rho(s,a)}{\sqrt{N(s,a)+1}}$$

  Here, $N(s)$ is the visit count of s, $N(s,a)$ is the visit count of move a at s, $Q(s,a)$ is the Q-value of (s,a), $R(s,a)$ is the RAVE value [74], and $\rho(s,a)$ is the prior probability calculated from move pattern weights. The RAVE weight w is dynamically adjusted during the search [74]; c_b and c_{pb} are tuning parameters.

- *Expansion*: A leaf node is expanded only when its visit count exceeds an *expansion threshold*, which is set to 10 in MoHex 2.0.

- *Pattern-based playout*: Pattern weights have been trained offline. In each playout, a softmax policy selects moves according to move pattern weights.

- *Backpropagation.* After the playout, game result is recorded, then MCTS updates values in visited nodes according to the playout result.

The best performance of MoHex 2.0 is achieved after tuning its parameters by CLOP [48, 96]. Other optimizations implemented in MoHex 2.0 include: pre-search analysis (Inferior cells and connection strategies are computed from the root node before MCTS. If a winning connection is discovered, a winning move will be selected without search); knowledge computation (Whenever the visit count of a node exceeds a *knowledge threshold*, H-search [151] to compute virtual connection and inferior cell analysis are applied; this often leads to effective move pruning); and time management (A search is said to be unstable if by the end, the move with the largest visit count disagrees with the move with highest Q-value. MoHex 2.0 extends the search by half of its original time in this case [96]). After the search terminates, MoHex 2.0 finally selects a move with the largest visit count.

From MoHex 2.0 to MoHex-CNN

In MoHex 2.0, the prior knowledge $\rho(s, a)$ is computed by a rough estimate from relative move pattern weights. To see the effectiveness of our policy neural network, the straightforward modification is to replace $\rho(s, a)$ by $p_\theta(s, a)$ — the move probability computed by our policy neural network CNN_8^{128}. All other tuning parameters are left unchanged.

The new program after adding neural net CNN_8^{128} is named as MoHex-CNN. It is implemented directly from the MoHex 2.0 code in Benzene, and compiled with the Tensorflow C++ libraries. Similar to AlphaGo [180], we also prepare another program MoHex-CNN$_{puct}$ that uses a variant of PUCT [165]. It selects moves that maximize $Q(s, a) + c_{pb} \times p_\theta(s, a) \times \frac{\sqrt{N(s)}}{N(s,a)+1}$. On the same i7-6700 CPU, 32GB RAM, GTX-1080 GPU machine, we run several tournaments to compare MoHex 2.0 and MoHex-CNN.

In practice, the evaluation speed of the neural network is a concern. Whenever MoHex expands a node, *prior pruning* (using patterns to remove proved inferior cells) is applied. Since those computations are costly and are independent from neural net evaluation, we implement a evaluation method that runs in parallel with prior pruning. As a result of this implementation, the computation overhead becomes small: in our experiments on the i7-6700 CPU machine with a GTX 1080 GPU, MoHex with CNNs took about 0.19 ms per simulation, while MoHex 2.0 took

about 0.17 ms.

The first tournament uses the same number of simulations for each program. From the empty board, without the swap rule, 400 games were played by MoHex-CNN and MoHex-CNN$_{puct}$ against MoHex 2.0, in which MoHex 2.0 plays 200 games as Black and 200 as White.

Table 3.4 shows the results. Under those settings, the new programs MoHex-CNN and MoHex-CNN$_{puct}$ are stronger than MoHex 2.0, since they consistently won more than 62% over all played games. Since the difference between MoHex-CNN and MoHex 2.0 lies only in the prior probability initialization, the results suggest that the more accurate prior knowledge due to CNN can improve MoHex, given the same number of simulations.

Table 3.4: Results of MoHex-CNN and MoHex-CNN$_{puct}$ with same number of simulations against Mohex 2.0. As Black/White means MoHex 2.0 is as the Black/White.

Opponent	#Simulations	As White	As Black	Overall
MoHex-CNN	10^3	94.5%	44.5%	69.5%
	10^4	90%	56%	73%
MoHex-CNN$_{puct}$	10^3	86%	39%	62.5%
	10^4	83.5%	53%	68.3%

The next experiment compares MoHex-CNN and MoHex-CNN$_{puct}$ to MoHex 2.0 with equal time budget: 10 seconds per move. When the swap rule is used, the second player has the option to steal the opening move, so we run the tournament by iterating over all opening cells in the board. For 13×13 Hex, there are 85 distinct openings after removing symmetries. For each opening, we run 5 games for each color between MoHex-CNN$_{puct}$ or MoHex-CNN against MoHex 2.0.

Table 3.5 summarizes the result of those 850 games played by MoHex with CNNs and MoHex 2.0. Both MoHex-CNN$_{puct}$ and MoHex-CNN have better performance against MoHex 2.0 even with same computation time. Consistent with the results in Table 3.4, MoHex-CNN plays better than MoHex-CNN$_{puct}$.

Table 3.5: Results Against MoHex 2.0 of MoHex-CNN$_{puct}$ and MoHex-CNN with same time limit per move, 95% confidence.

Opponent	MoHex2.0 as White (%)	MoHex2.0 as Black (%)
MoHex-CNN$_{puct}$	71.7 ± 0.1	53.2 ± 0.1
MoHex-CNN	78.6 ± 0.4	61.2 ± 0.4

MoHex-CNN won the 2017 computer Hex Olympiad against Ezo-CNN [197], an updated version of Ezo [196] using neural networks as its move ordering function.

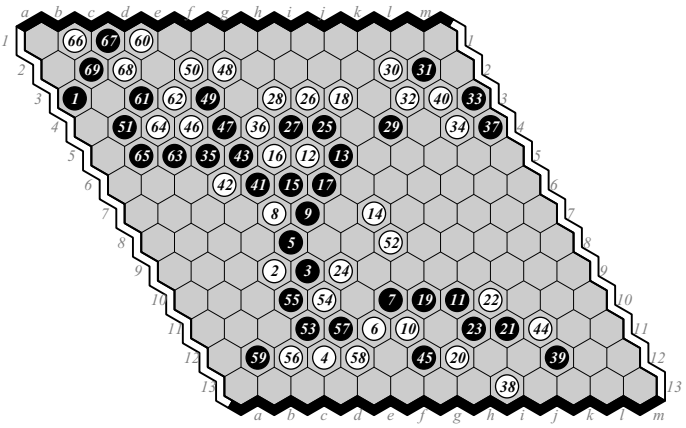


Figure 3.6: MoHex-CNN against Ezo-CNN: a sample game from 2017 computer Olympiad. MoHex-CNN won as Black.

Figure 3.6 shows a game played in the Olympiad tournament where MoHex-CNN won as Black.

3.1.7 Discussion

We achieved a prediction accuracy near 55% in predicting MoHex 2.0 moves, and used the high prediction accuracy to produce a stronger Hex player. Even without search, the neural network model plays Hex reasonably well; using a neural net model as a learned prior in MCTS, our player MoHex-CNN surpasses MoHex 2.0 — the reigning champion player until 2016. Meanwhile, a number of new general techniques have been explored by other researchers. Most of them were carried out or became popular after our work on Hex. These techniques could be used to further strengthen our Hex player:

- Instead of using a stack of convolutional layers, better neural nets have been explored in computer vision. These architectures include Residual neural nets [84, 85, 219]. A number of regularization techniques have been shown useful in image recognition, such as dropout [189, 202], batch normalization [97], stochastic depth [95], swapout [185].

- Instead of maximizing the likelihood of a single next move, studies in computer vision and natural language processing showed that predicting a more smoothed distribution can help to reduce over-fitting [154, 194]. Since the Hex data we used to train our neural net is inherently imperfect, incorporating similar techniques may also be beneficial.

- The advancement of AI accelerators such as TPU [100] have greatly improved the computational efficiency of deep neural networks. Appropriately using them may improve the playing strength of Hex as well.

3.2 Policy Gradient Reinforcement Learning

We have seen that supervised learning can produce a moderately strong policy network for playing Hex. Policy gradient reinforcement learning (PGRL) aims to *improve* a policy function by trial-and-error. In this section, we investigate the possibility of using PGRL for producing stronger policy network model in the game of Hex.

There is a large body of research in RL algorithms, so we begin with a brief review of these algorithms and then discuss their applicability to Hex. We propose a new policy gradient variant specifically for two-player games and empirically show its advantageous performance for Hex.

3.2.1 Background

Model-free reinforcement learning methods have been successfully applied to many domains, including robotics [110], Atari games [132, 133, 173], and two-player board games [180, 198]. RL algorithms can be split into two categories: value-fitting and policy gradient. The value-fitting algorithms try to learn the optimal value function for every possible states in the state-space. The primary disadvantage of value-fitting methods such as Q-learning [132, 133, 207, 209] is that they are often unstable when interacting with function approximation [27, 193]. To gain stable behavior, extra heuristic techniques [121, 173] and extensive hyper-parameter tuning are often required. By contrast, policy gradient methods explicitly represent a policy as a function of parameters: they learn through iteratively adjusting the parameterized policy by following estimated gradients of policy function, thus converging to at least a local maximum [155]. Policy gradients are applicable to continuous control [176, 177, 183] or domains with large action space [180], where action-value learning methods often become infeasible [120, 208].

Both value and policy based reinforcement learning are well-defined in MDPs, where a single agent learns by exploring a stationary environment. As discussed in Chapter 2.2.2, two-player alternate-turn perfect-information zero-sum games are alternating Markov games, which are a generalization of MDPs by allowing exactly

two players pursuing opposing goals. Many popular two-player games — including the game of Hex — are of such kind. The restriction of zero-sum in AMGs makes it possible to define a shared reward function which one agent tries to maximize while the other agent tries to minimize it. Due to such a property, using a *selfplay* policy, RL methods in MDPs have been applied to AMGs [180, 198] by simply negating the reward signal of the opponent. Tesauro [198] trained multi-layer neural networks to play backgammon. Tesauro's program learns the optimal *value function* by greedy self-play, relying on the stochastic environment in the game of backgammon for exploration; Tesauro [198] conjectured that the smoothness of the value function is one major reason for the particular success of TD-Gammon. A recent study that applies Deep Q-learning to Hex [218] failed to produce strong player.

As an alternative to value-fitting, a policy gradient method can be used to refine a neural net policy model by learning from pure neural net selfplay. In the first version of AlphaGo [180], policy gradient RL is used to improve the neural network model obtained by supervised learning. The improved model was less useful for exploration in search because of its strong bias towards a single move. However, due to its better playing strength and fast speed, it was used as a standalone player to generate training data for a value net [180], which was integrated into Monte Carlo tree search. The policy gradient employed by AlphaGo is a variant of REINFORCE [211].

In the rest of this chapter, we examine the justifications of adapting standard policy gradient RL to AMGs. Based on the difference between MDPs and AMGs, we formulate an adversarial policy gradient objective. We then develop new policy gradient methods for AMGs and apply our approach to the game of Hex. We show that by modifying REINFORCE to estimate the *minimum* rather than the *mean* return of a self-play policy, stronger pure neural net players can be obtained.

3.2.2 The Policy Gradient in MDPs

Let π_θ be a θ parameterized policy. Note that for notation convenience, in the following text we sometimes omit θ. Let $d^\pi(s)$ be the state distribution under π. In MDPs, the strength of π can be measured by

$$J(\pi) = \sum_{s \in \mathcal{S}} d^\pi(s) \sum_a \pi(a|s) q_\pi(s, a), \tag{3.2}$$

Consequently,

$$\nabla J(\pi) = \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a} \pi(a|s) \nabla \log \pi(a|s) q_{\pi}(s,a) \tag{3.3}$$

The above equation is a direct result of the Policy Gradient Theorem for MDPs [193], which implies that the gradient of the strength of a policy can be estimated by sampling according to π. The requirements are that π is differentiable and that $q_{\pi}(s,a)$ can be estimated. A PGRL that follows this scheme can be interpreted as a kind of *generalized policy iteration* [192], where the gradient ascent corresponds to policy improvement [101], and $q_{\pi}(s,a)$ is obtained by policy evaluation. Depending on how $q_{\pi}(s,a)$ is estimated, policy gradient algorithms can be categorized into two families: Monte Carlo policy gradients that use Monte carlo to estimate q_{π}, e.g., REINFORCE [211] and actor-critic methods [193, 208] that use another parameter to approximate the action-value under π. The Monte Carlo policy gradient methods have the advantage that the value estimate is unbiased, though in practice, they could have higher variance than the action-critic ones.

3.2.3 An Adversarial Policy Gradient Method for AMGs

Unlike MDPs where the *policy iteration* [27, 192] is unique, with AMGs, four different variants exist:

Algo.1 Fix π_1^t, compute the optimal counter policy π_2^t, then fix π_2^t, compute the optimal counter policy π_1^{t+1}. Continue this procedure repeatedly until convergence.

Algo.2 Policy evaluation with π_1^t, π_2^t, switch both π_1^t and π_2^t to greedy policies with respect to current state-value function. Continue this procedure repeatedly until convergence.

Algo.3 Policy evaluation with π_1^t, π_2^t, switch π_1^t to greedy policy with respect to the current state-value function and then compute the optimal counter policy for π_2^t. Continue this procedure repeatedly until convergence.

Algo.4 Policy evaluation with π_1^t, π_2^t, switch π_2^t to greedy policy with respect to the current state-value function and then compute the optimal counter policy for π_1^t. Continue this procedure repeatedly until convergence.

Denote the joint strength for a pair of parameterized policies π_1 and π_2 as:

$$J(\pi_1, \pi_2) = \sum_s d^{\pi_1,\pi_2}(s) \sum_a \pi_1(a|s) q_{\pi_1}(s, a) \tag{3.4}$$

Here, $d^{\pi_1,\pi_2}(s)$ is the state-distribution (i.e., representing the frequency that s is visited under policies π_1 and π_2) given π_1 and π_2. A natural question is what is the gradient of $J(\pi_1, \pi_2)$ with respect to π_1 and π_2 respectively? One tempting derivation is to calculate the gradient for both π_1 and π_2 simultaneously by treating the other policy as the "environment". Similar to the mutual greedy improvement in Algo.2, such a method tries to adapt each policy by referring to the value function under current π_1, π_2, ignoring the fact that the other player is an adversary who will also adapt its strategy as well. Another possible algorithm is to fix π_1 and do a fixed number of iterations to optimize π_2 by normal policy gradient as in MDP, then fix π_2 for optimizing π_1, repeating this alternatively. However, this algorithm is analogous to Algo.1, which was shown to not converge in certain cases [45].

Following Algo.3 and Algo.4, given the action-value function under π_1, π_2, a more reasonable approach for policy improvement is to switch π_2 to greedy and optimize π_1 by policy gradient. Therefore, assuming π_1 is the max player, we advocate the following objectives

$$\begin{cases} J^{\pi_1}(\pi_1, \pi_2) = \sum_s d^{\pi_1,\pi_2}(s) \sum_a \pi_1(a|s) \sum_{s'} \Pr(s'|s, a) \left[r(s, a, s') + \gamma \min_{a'} q_{\pi_2}(s', a') \right] \\ J^{\pi_2}(\pi_1, \pi_2) = \sum_s d^{\pi_1,\pi_2}(s) \sum_a \pi_2(a|s) \sum_{s'} \Pr(s'|s, a) \left[r(s, a, s') + \gamma \max_{a'} q_{\pi_1}(s', a') \right]. \end{cases} \tag{3.5}$$

Therefore, the gradients can be expressed by

$$\begin{cases} \nabla J^{\pi_1}(\pi_1, \pi_2) = \mathbb{E}_{\pi_1,\pi_2} \left[\nabla \log \pi_1(a|s) \sum_{s'} \Pr(s'|s, a)(r(s, a, s') + \gamma \min_{a'} q_{\pi_2}(s', a')) \right] \\ \nabla J^{\pi_2}(\pi_1, \pi_2) = \mathbb{E}_{\pi_1,\pi_2} \left[\nabla \log \pi_2(a|s) \sum_{s'} \Pr(s'|s, a)(r(s, a, s') + \gamma \max_{a'} q_{\pi_1}(s', a')) \right]. \end{cases} \tag{3.6}$$

The above formulation implies that, when computing the gradient for one policy, the other policy is simultaneously switched to greedy. This joint change forces the current player to adjust its action preferences according to the current worst-case response of the opponent, which is desirable due to the adversarial nature of the game.

A straightforward implementation of Equation (3.5) is to obtain separate Monte Carlo estimates for each next action, and then apply the min or max operator. However, this may not be practically feasible when the action space is large. We

introduce a parameter k, approximating the true min/max by only considering a subset of actions.

As a minimal modification on self-play REINFORCE, this algorithm works as follows: (i) generate a batch of n games by a self-play policy; (ii) for each game, sample a single state-action pair (s, a) uniformly as a training example for this game; (ii) instead of directly using the observed return z in the game as the estimated action-value for (s, a), perform extra Monte Carlo simulations to estimate the *minimum* return for (s, a). We consider two Adversarial Monte Carlo Policy Gradient (AMCPG) methods:

AMCPG-A: Run k self-play games from (s, a) using self-play policy π, then take the minimum of these k returns and z.

AMCGP-B: Sample a state s' from (s, a) according to the state-action transition $p(\cdot|s, a)$, and select the top k actions suggested by π for s'. For each selected action, obtain an Monte Carlo estimate using self-play policy π, then take the minimum of these k returns and z.

Algorithm 2: Adversarial Monte-Carlo Policy Gradient (AMCPG-A and AMCPG-B)

Input: A policy network π_θ
Result: Improved policy $\hat{\pi}_\theta$
$ite \leftarrow 0$;
while $ite < maxIterations$ **do**
  Self-play a mini-batch of n games E using π ;
  $N \leftarrow \emptyset$;
  for $e_i \in E$ **do**
    Select a state-action pair (s_j, a_j) uniform randomly;
    Let $z(s_j, a_j)$ be the outcome with respect to action a_j at s_j in e_i;
    Let s'_j be the next state after taking a_j at s_j; at s'_j, let

$$z'(s_j, a_j) = \begin{cases} A: & \text{Self-play } k \text{ games using } \pi, \text{ record the minimum outcome with respect to } s_j; \\ B: & \text{Select top } k \text{ moves of } \pi, \text{ from each move self-play a game using } \pi, \\ & \text{record the minimum outcome w.r.t } s_j; \end{cases}$$

    $R^i \leftarrow \min\{z(s_j, a_j), z'(s_j, a_j)\}$;
    Append (s_j, a_j, R^i) to the mini-batch N ;
  end
  $\theta \leftarrow \theta + \frac{\alpha}{|N|} \sum_{i=1}^{|N|} \nabla \log \pi(s^i_j, a^i_j; \theta) R^i$;
  $ite \leftarrow ite + 1$;
end
return π_θ;

Here, only *minimum* is used, because we assume the state-value are always given with respect to the player to play at that state. It is easy to see that in **AMCPG-A**, if *mean* operator is used rather than *minimum*, a variant of REINFORCE is

Table 3.6: Input feature planes.

Plane	Description	Plane	Description
0	Black played stones	6	To play save bridge points
1	White played stones	7	To play make-connection points
2	Unoccupied points	8	To play form bridge points
3	Black or White to play	9	Opponent's save bridge points
4	Black bridge endpoints	10	Opponent's form bridge points
5	White bridge endpoints	11	Opponent's make-connection points

recovered. When $k = |\mathcal{A}(s')|$, **AMCPG-B** becomes a genuine implementation of Equation (3.5), but in this situation, even though each action-value's estimation is unbiased, bias could still be incurred when applying the min or max; this is known as the *winner's curse* problem in economics [41, 187].

3.2.4 Experiment Results in Hex

We apply policy gradient reinforcement learning to improve a policy net which was trained by supervised learning. As a neural net model obtained by deep Q-learning [218], although the policy gradient refined neural net model can be stronger at playing, it is not helpful in guiding best-first search [180]. Therefore, in this experiment we focus on studying the behavior of policy gradient reinforcement learning.

3.2.5 Setup

Since this experiment is intended to verify a new method rather than produce a competitive player, we choose 9×9 Hex for the study due to its fast speed of training, while using fully-convolutional neural network architecture that can output policies for multiple board sizes. Figure 3.7 shows the detailed architecture design. For $W \times W$ input, after applying $F \times F$ convolution with stride S and padding P, the output is $\frac{W+2P-F}{S} + 1$, which means that given fixed F, S and P, the output size is solely decided by input width W; therefore, for a given set of convolution filters, fixed stride and padding, by feeding $W \times W$ input feature planes, a vector of move probabilities of size W^2 can be obtained. See Figure 2.13 for an example of 2D convolution. We implement the network graph using Tensorflow [128].

3.2.6 Data and Supervised Learning for Initialization

The input has 12 binary planes, shown in Table 3.6. To initialize the neural net weights, we first generate a dataset containing 10^6 state-action pairs by MoHex

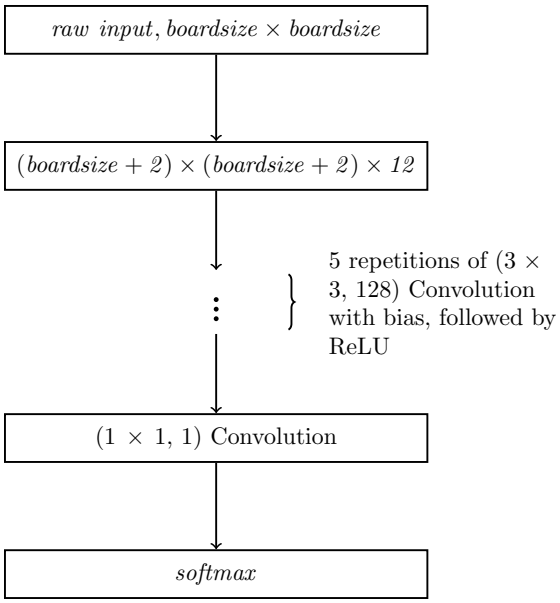


Figure 3.7: Neural network architecture: It accepts different board size inputs, padded with an extra border using black or white stones; the reason for this transferability is that the network is fully convolutional.

2.0 selfplay on 9×9 board. We train the policy neural net to maximize the log-likelihood on this dataset. The training took 100,000 steps, where each step is with a mini-batch size of 64 and optimized using Adam [104] with learning rate 0.001.

After supervised learning, the obtained neural net model can be used for playing in multiple board sizes. Its win-percentages against 1-play Wolve on 9×9 and 11×11 are respectively 13.2% and 4.6%; each test is done by iterating all opening moves and with 10 games for each opening with Wolve as Black and White.

3.2.7 Results of Various Policy Gradient Algorithms

For comparison purposes, we implement three REINFORCE variants:

- REINFORCE-V: Vanilla REINFORCE using a parameterized self-play policy. After a batch of n self-played games, each game is replayed to determine the batch policy gradient update $\frac{\alpha}{n}\sum_i^n\sum_t^{T_i}\nabla\log\pi(s_t^i,a_t^i;\theta)z_t^i$, where z_t^i is either $+1$ or -1.

- REINFORCE-A: An "AlphaGo-like" REINFORCE. It differs from REINFORCE-V by randomly selecting an opponent from former iterations for self-play.

- REINFORCE-B: For each self-played game, only one state-action pair is uniformly selected for policy gradient update. This algorithm differs from AMCPG-

A by using the mean of all $k+1$ observed returns.

All methods are implemented using Tensorflow, sharing the same code base. They only differ in a few lines of code. A self-play policy is employed for all algorithms, which is equivalent to forcing π_1 and π_2 to share the same set of parameters. The game batch size n is set to 128. For each self-play game, the opening move is selected uniformly random. The learning rate is set to 0.001; vanilla stochastic gradient ascent is used as the optimizer. The reward signal $z \in \{+1, -1\}$ is observed only after a complete self-play game. For all algorithms, the same neural net architecture is used. As mentioned, the initial parameter weights were obtained from supervised learning. Because the architecture is fully convolutional, the parameter weights can be used on multiple board sizes.

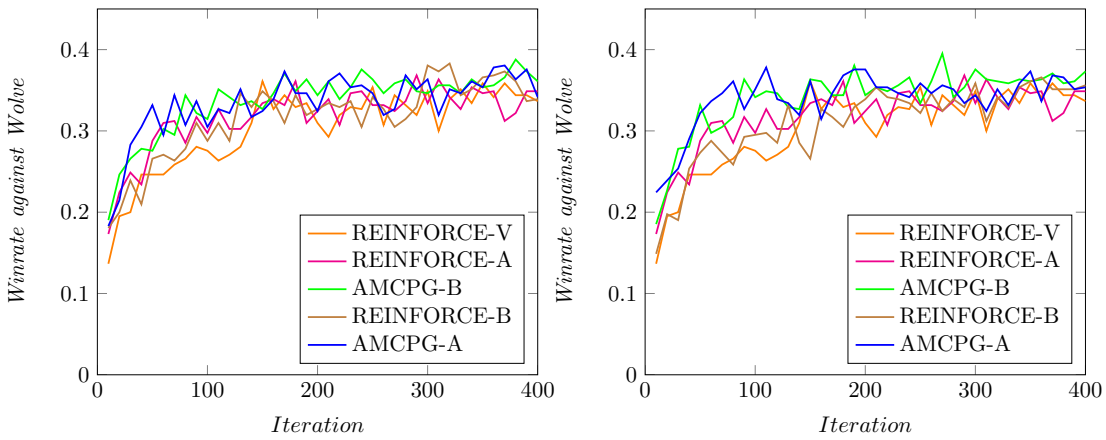
We run policy gradient reinforcement learning for 400 iterations for each method, on two different board sizes, 9×9 and 11×11, varying $k \in \{1, 3, 6, 9\}$. We use Wolve [87, 151] as a benchmark to measure the relative performance of our learned models. After every 10 iterations, model weights are saved and then evaluated by playing against 1-ply Wolve. The tournaments with Wolve are played by iterating all opening moves, each is repeated 5 times with Wolve as Black or White.

Figure 3.8 compares the strengths of these five algorithms. REINFORCE-B is able to achieve similar learning speed with REINFORCE-V as a function of iteration number, though in one game, REINFORCE-B only extracted one example to train the neural network. This is perhaps because the reward signal in the same game is too much correlated. Consistent with the finding in Go [180], REINFORCE-A performed better on 9×9 and 11×11 Hex. However, the newly developed algorithms AMCPG-A and AMCPG-B achieved better results even when $k = 1$.

While error bands in Figure 3.8 were not plotted for clear visualization of the average performance, to further see the relative strength, we then show the comparison of REINFORCE-V and AMCPG-B with error bands plotted in Figure 3.9. It clearly shows that AMCPG-B performed significantly better than REINFORCE-V.

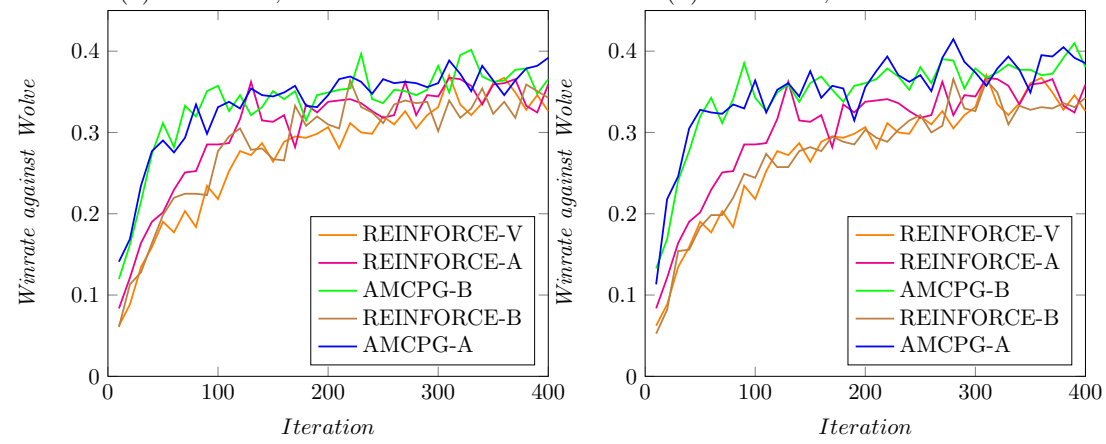
3.2.8 Discussion

We have presented a new policy gradient objective for AMGs. Experiment results on the game of Hex show that optimizing towards the new objective led to a better neural net player. However, as a model-free method, the merit of a policy gradient



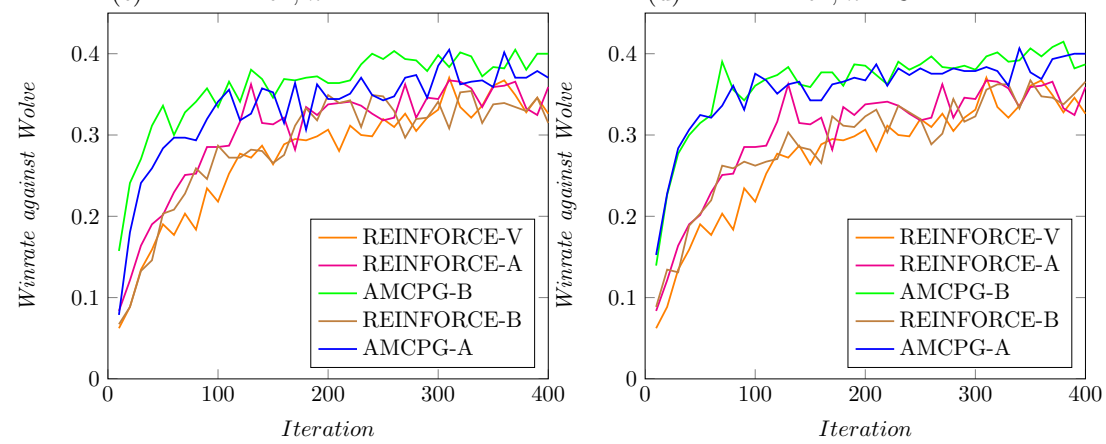
(a) 9×9 Hex, $k = 1$

(b) 9×9 Hex, $k = 3$



(c) 11×11 Hex, $k = 1$

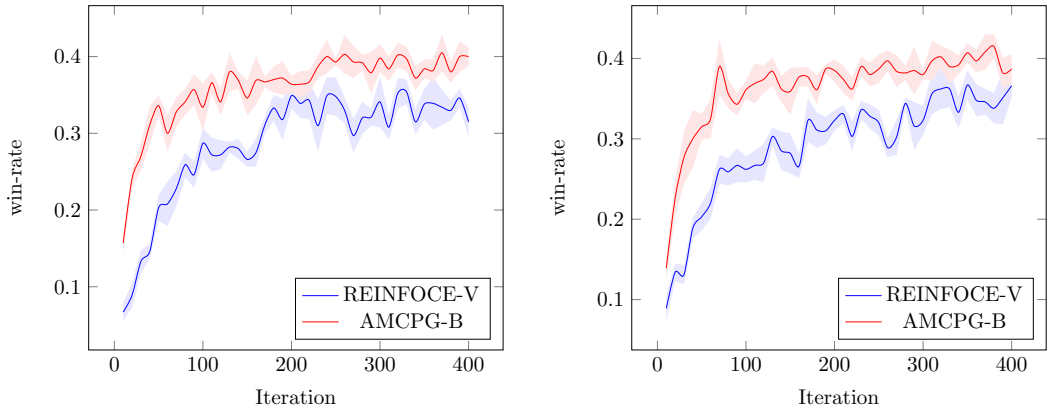
(d) 11×11 Hex, $k = 3$



(e) 11×11 Hex, $k = 6$

(f) 11×11 Hex, $k = 9$

Figure 3.8: Comparison of playing strength against Wolve on 9×9 and 11×11 Hex with different k. The curves represent the average win percentage among 10 trials with Wolve as black and white.



(a) k=6 (b) k=9

Figure 3.9: On 11×11 Hex, comparisons between AMCPG-B and REINFORCE-V with error bands, 68% confidence. Each match iterates all opening moves; each opening was tried 10 times with each player as Black or White.

refined neural net is rather limited. In the first version of AlphaGo [180], a policy gradient refined network was used to produce data for training a value net. However, later studies [184] show that training value net directly on *human professional data* or *search generated data* with a novel architecture produce significantly better results for search-based player. We report the use of value estimation network for stronger Hex player in the next chapter.

Chapter 4

Three-Head Neural Network Architecture for MCTS and Its Application to Hex

This chapter contains content from the papers "Three-Head Neural Network Architecture for Monte Carlo Tree Search" [66], "A transferable neural network for Hex" [69], and a discussion with content from "Hex 2018: MoHex3HNN over DeepEzo" [67]. We review the grand success of AlphaGo Zero and AlphaZero, discuss some of their shortcomings, describe our three-head neural network and show empirical results in Hex.

4.1 Background: AlphaGo and Its Successors

By defeating European champion Fan Hui, AlphaGo [180] became the first computer Go program to reach professional level. AlphaGo was created as follows [180]:

1. Training a policy network p_σ on human expert games from the KGS Go server. These 160,000 games contain roughly 30 million state-action pairs.

2. Refining the policy network by policy gradient reinforcement learning, resulting in a stronger neural net player p_ρ.

3. Training a value network v_θ on a dataset consisting of roughly 30 million state-value pairs generated by p_ρ selfplay.

4. Training a fast rollout policy p_π on 8 million Go positions from the Tygem server.

5. Integrating p_σ, p_π and v_θ into a policy value MCTS (PV-MCTS) algorithm.

Table 4.1: p_σ, p_π, p_ρ, p_σ and v_θ architectures and their computation consumption.

Function	Architecture	Computation Hardware and Cost
p_σ	13 layers, 192 filters per layer	50 GPUs for three weeks
p_ρ	13 layers, 192 filters per layer	50 GPUs for one day
v_θ	13 layers, 192 filters per layer	50 GPUs for one week
p_π	linear combination of features	not provided

The architecture and computation for each neural network are listed in Table 4.1. These trained linear or non-linear functions do not give superhuman playing strength. They were used along with MCTS, where each search node s represents a game state. The stored statistics for each action a of s are

$$\{P(s,a), N_v(s,a), N_r(s,a), W_v(s,a), W_r(s,a), Q(s,a)\}.$$

$P(s,a)$ is the prior probability obtained by calling $p_\sigma(s)$, $W_v(s,a)$ and $W_r(s,a)$ are accumulated total action value over $N_v(s,a)$ and $N_r(s,a)$ leaf evaluations by v_θ and rollout results by p_π, respectively, and $Q(s,a)$ is the mean action value. PV-MCTS repeatedly performs the following steps until a predefined search time is reached:

- Select: Starting from the root, at each node, select an action leading to the child node that maximizes a PUCT [165] score:

  $$Q(s,a) + c_{puct} P(s,a) \frac{\sqrt{\sum_b N_r(s,b)}}{1 + N_r(s,a)}.$$

  This process is repeated until a leaf node s is reached.

- Evaluate: The leaf node s is evaluated by v_θ and p_π.

- Expand: When a leaf's visit count exceeds a threshold, i.e., $N(s) > n_{th}$, the node is expanded, and statistics over its actions are initialized to $\{N(s,a) = 0, W(s,a) = 0, Q(s,a) = 0, P(s,a) = p\}$, where p is the move probability of action a from the output of policy network p_σ.

- Backup: The evaluation results for each leaf node are backed up until the root is reached.

After search terminates, in the root node, if the move with largest visit count does not also have the best action-value, an extended search is conducted. Finally, the most visited move is selected. The program played in two configurations, both with parallel MCTS [43, 57]:

1. Single machine with 40 search threads, 48 CPUs and 8 GPUs.

2. Distributed with 40 search threads. 1,202 CPUs, and 176 GPUs.

The number of CPUs used is larger than the number of search threads because the algorithm was implemented in asynchronous style, where many CPU workers are used to perform the rollout after a leaf node is selected. The distributed version is significantly stronger than the single-machine one, winning 77% games in head-to-head match against the single machine version; it defeated Fan Hui, becoming the first computer program achieving professional level strength [180]. A later version called AlphaGo Lee defeated human world champion player Lee Sedol by 4–1 in a five-game public match; the key improvements over AlphaGo Fan are briefly highlighted in [184]:

- Policy and value networks were enlarged to contain 256 filters per layer.
- Value network was trained on games played by AlphaGo selfplay. This process is iteratively repeated for a few times.
- More advanced hardware TPUs were used when playing against Lee Sedol.

Continual improvements were made, resulting in AlphaGo Master which defeats Ke Jie by 3–0. AlphaGo Master improves over AlphaGo Lee as follows [184]:

1. Plain CNNs were replaced with 20-block ResNet; each block (except the first one) consists of 2 convolution layers, where each layer is with 256 3×3 filters, and instead of using two separate policy and value networks, a single two-head network with policy and value outputs was used.

2. The two-head network was first trained on human professional game records, then on data generated from AlphaGo selfplay. Again, the later phase was repeated for a few times.

Later, the idea behind step 2 was further explored [184]: without using any human data to initialize the neural network, the program called AlphaGo Zero [184] (with 40-block ResNet) achieved a strength stronger than AlphaGo Master. We recapitulate the AlphaGo Zero algorithm in Algorithm 3.

Algorithm 3 is an iterative procedure alternating among three subroutines: (1) using player PV-MCTS(f_θ) for data generation; (2) using played games to retrain

Algorithm 3: AlphaGo Zero Algorithm

Input: Neural network hyperparameters, e.g., number of residual blocks; allocated training resource, e.g., total number of training games

Result: f_θ and PV-MCTS(f_θ)

Function `Main()`**:**
- Let f_θ be a θ parameterized two-head ResNet
- Let PV-MCTS(f_θ) a PV-MCTS with network f_θ
- $\mathcal{D} \leftarrow \emptyset$
- **while** *computation resource not exhausted* **do**
  - 1. Run m parallel workers, each $g_i \leftarrow$ `Search-Selfplay`$(f_\theta, 25000)$, put g_i to $\mathcal{D}$
  - 2. Sample the most recent 500,000 game data from $\mathcal{D}$ and reoptimize the neural net parameter θ, obtaining θ'
  - 3. Run a match between PV-MCTS($f_\theta, 1600$) and PV-MCTS($f'_\theta, 1600$), if θ' won more than 55%, $\theta \leftarrow \theta'$
- **end**

End Function

Function `Search-Selfplay(`f_θ, n_games`)`**:**
- Let s be the initial state of Go
- $g \leftarrow \{\}$
- **while** *s is not the end of a game* **do**
  - $a, \boldsymbol{n}(s) \leftarrow$ `PV-MCTS`$(f_\theta, s, 1600)$
  - $s \leftarrow Play(s, a)$
  - Resign the game if estimated root value is below a threshold v_{resign}
  - Append a and $\boldsymbol{n}(s)$ to g
- **end**
- Add the game result to g
- **return** g

End Function

Function `PV-MCTS(`f_θ, s, n`)`**:**
- $i \leftarrow 0$
- **while** $i < n$ **do**
  - Starting from s, select using PUCT until a leaf l is obtained
  - Evaluation by $v, \boldsymbol{p} \leftarrow f_\theta(l)$ and Expand
  - Backup v along the selection path back to s
- **end**
- Let $\boldsymbol{n}(s)$ be a vector of visit counts for each move in s
- Select an action a of s proportionally to its visit count
- **return** $a, \boldsymbol{n}(s)$

End Function

the neural net; and (3) using a gating subroutine to examine the strength of newly produce neural net model, ensuring that the best network parameter will always be used for data generation. The step (1) is embarrassingly parallel. The search procedure `PV-MCTS` is crucial to the whole algorithm because it controls how data are produced for improving the neural network weights; thus, some important details omitted in Algorithm 3 deserve to be noted:

- In the selection formula, a noise is added to $P(s,a)$: $P(s,a) = (1-\epsilon)P(s,a) + \epsilon\eta(\alpha)$ where $\eta(\alpha) \sim \text{Dir}(0.03)$ (Dir represents a Dirichlet distribution), $\epsilon = 0.25$.

- After PV-MCTS iteration finishes, select a move to play by sampling

$$a \propto \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}.$$

  Here, τ is a temperature; $\tau = 1$ for the first 30 moves and $\tau \leftarrow 0$ in the remaining moves. For gating, $\tau \leftarrow 0$ for all the moves.

By removing the Go specific designs in Algorithm 3, an AlphaZero algorithm was first proposed in a preprint article [181], whose formal version was published in [182]. AlphaZero is described as a general algorithm and its effectiveness was verified in three games: chess, Shogi and Go. AlphaZero differs from AlphaGo Zero by the following modifications:

- For input feature planes to the neural network, the augmentation using symmetry is removed because this is only applicable to Go, not Shogi and chess.

- It uses a new PUCT selection formula:

$$Q(s,a) + c(s)P'(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)},$$

  where $P'(s,a) = (1-\epsilon)P(s,a) + \epsilon\eta(\alpha)$ where $\eta(\alpha) \sim \text{Dir}(\alpha)$, $\epsilon = 0.25$, α is a Dirichlet noise parameter. That is, the coefficient is now game-state dependent $c(s) = \log((1+N(s)+c_{base})/c_{base}) + c_{init}$, where $c_{init} = 1.25$ and $c_{base} = 19625$; however, this dynamic $c(s)$ was not described in [181].

- Gating is removed.

- Number of MCTS simulations (i.e., iterations) per move is reduced to 800.

AlphaZero mastered Go, Shogi and chess after separate training on each.

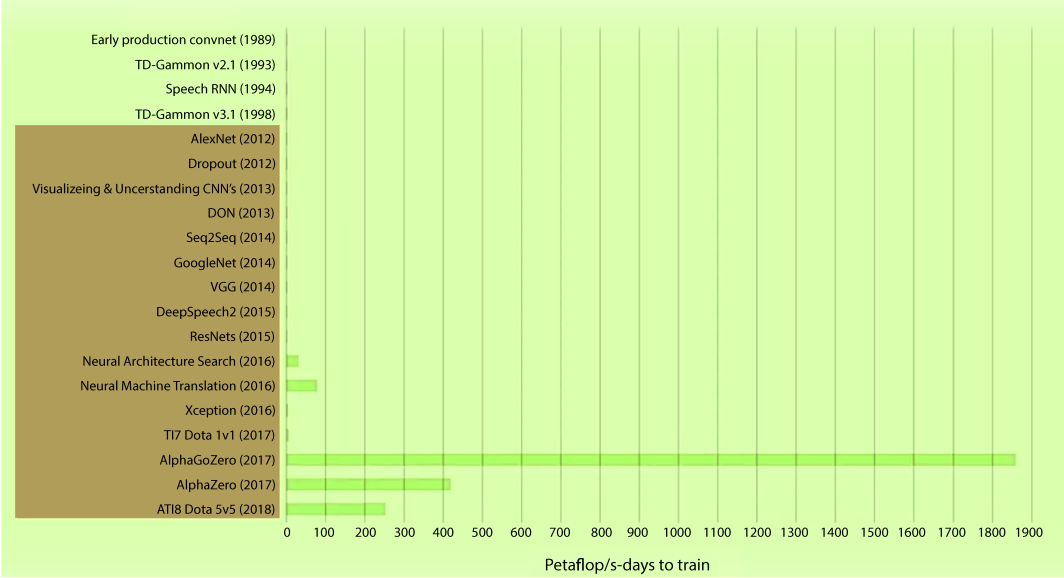


Figure 4.1: Computation costs for AlphaGo Zero and AlphaZero are far ahead of other major achievements in AI [157].

4.2 Sample Efficiency of AlphaGo Zero and AlphaZero

Although AlphaGo Zero and AlphaZero are seemingly simple, an important drawback is that they are sample inefficient. As in Figure 4.1, The computational cost of training 40-block AlphaGo Zero is far larger than for other prominent AI achievements [157]. To see the strong correlation between computation and playing strength in Go, in Table 4.2 we summarize the Elo strength and the computation hardware for each AlphaGo variant. Except for AlphaGo Fan, all other programs used an iterative scheme illustrated in Figure 4.2. Training AlphaGo Zero 20-block, AlphaGo Zero 40-block and AlphaZero used 4.9 and 29 and 140 million games, respectively. The number of parallel workers used in training AlphaGo Zero was not given, but can be estimated from provided data — on current commodity hardware the training process could 1700 machine years [147].

To see the data inefficiency of these iterative algorithms, consider AlphaZero 20-block: 140 million games were produced while the total number of neural net parameter updates is $700,000$, each with mini-batch 4096; multiplying these numbers shows a total of 286 million forward-backward passes. Considering that 140 million games were generated, this implies that on average two game positions were selected from each game for training the neural net only once. In Go, a typical game usually consists of several hundred game states, where each single game state was

Table 4.2: Computation used for producing each player. For all Zero variants, computation used to optimize the neural network was ignored.

Version	Computation	Elo
AlphaGo Fan	50 GPUs for a few weeks	3144
AlphaGo Lee	not given	3739
AlphaGo Master 20-block	not given	4858
AlphaGo Zero 20-block	a few thousand GPUs for 3 days	≈ 4500
AlphaGo Zero 40-block	a few thousand GPUs for 40 days	5185
AlphaZero (Science)	5000 TPUs for 13 days	≈ 4500

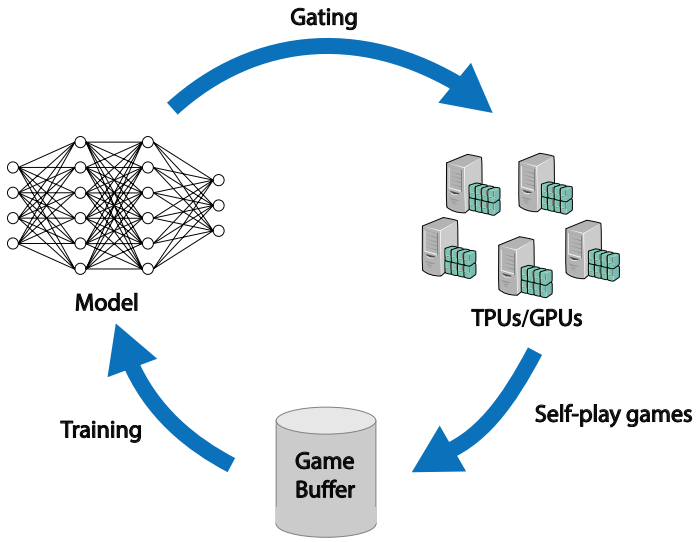


Figure 4.2: A closed scheme for iterative learning. Gating was removed in AlphaZero, but some implementation found gating is important for stable progress [148].

processed by 800 iteration MCTS — calling the neural net 800 times while building the search tree. This is in sharp contrast to other deep learning applications [75] where no examples are discarded, and moreover, every example will be learned by the neural network many times. This learning is also inefficient when compared to how human professional players master the game of Go — apparently, they do not play many millions of games to master the game.

The enormous demand on computation makes reproducing AlphaGo Zero and AlphaZero, or applying them to other games difficult. But reproducing these results is relevant, because how and why AlphaGo/Alpha Zero converge and what is their most crucial component is largely unknown. Research projects studying such questions include Leela Zero [148] and ELF OpenGo [201]. These approaches either use crowd-sourcing for computation or have access to thousands of GPUs for parallel selfplay game generation. Here are some discoveries reported in [201]:

- Performance is almost always bounded by the neural network capacity, that is, improvement can always be achieved by increasing neural network size as more selfplay games are produced.
- The PUCT parameter in PV-MCTS is important.
- Learning (as measured by strength of the program) has high variance even in the later stage.
- Elo measurement by selfplay has inflation; the monotonic increase of Elo score did not capture the fluctuation in strength against a variety of opponents.
- The ladder tactic in Go was never fully mastered even with increased MCTS simulations.

4.3 Three-Head Neural Network Architecture for More Efficient MCTS

In terms of reinforcement learning, the algorithmic framework of AlphaGo Zero and AlphaZero has been summarized as *approximate policy iteration* [26], where the *policy improvement* is due to running PV-MCTS at each state, and *policy evaluation* is achieved by regression with respect to the search-based selfplay game result [184]. However, such a high-level interpretation does not provide a detailed explanation on convergence properties of the algorithm, nor does it give useful practical guidance on how to select the hyperparameters for the search, the neural network or the training. Another interpretation views the whole framework as an *expert iteration* algorithm [8] by drawing connections to the Systems I and II psychological theory of the human mind (*fast* intuition is represented by deep neural network and the *slow* thinking is implemented by tree search) [59]. They applied the idea to 9×9 Hex and showed that they can produce a player stronger than MoHex 2011 but did not surpass MoHex 2.0; see [64].

Due to our limited computational resources, instead of showing results of applying AlphaGo Zero, AlphaZero algorithms to the game of Hex, we developed a three-head neural network architecture to improve the efficiency of PV-MCTS. In rest of this chapter, we show the merits of this architecture in supervised learning, transfer learning, and search-based Zero-style iterative learning.

4.3.1 PV-MCTS with Delayed Node Expansion

In AlphaGo Zero and AlphaZero, the policy value Monte Carlo tree search (PV-MCTS) merges the *expansion* and *evaluation* phases. In the implementation, whenever expanding and evaluating a non-terminal leaf node, after new child nodes are created, the move probabilities are saved to these child nodes; the state-value estimate is backed up.

However, a common practice in MCTS is expanding a leaf node only when it has been visited for a certain number of times. This scheme is used in the first version of AlphaGo [180]. The rationale is that, since the neural net has two heads, when evaluating the leaf node s, two outputs are given: the value estimate can be backed up immediately, but if the node is not expanded, the policy output would be wasted. See Figure 4.3.

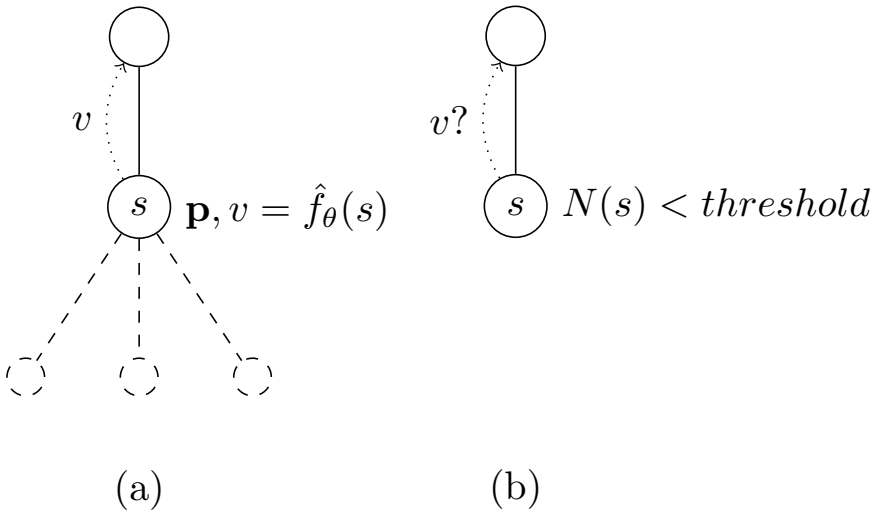


Figure 4.3: A problem with two-head architecture in PV-MCTS. The leaf node is expanded (a) with threshold 0, otherwise (b) if $N(s)$ is below the threshold, no expansion and evaluation indicates that no value to back up. $\hat{f}_\theta$ is the two-head neural net that each evaluation of state s yields a vector of move probabilities $\mathbf{p}$ and state-value v. $N(s)$ is the visit count of s.

To allow PV-MCTS to have delayed node expansion, we propose a three-head neural net that outputs also a vector of action-values, illustrated in Figure 4.4. Each evaluation of a leaf node s yields three outputs: a policy $\mathbf{p}$, a vector of next action-values $\mathbf{q}$, and a state-value v. The policy and action-value information can be stored by newly created child nodes. Therefore, in future MCTS iterations, when the visit count of a leaf is below the expansion threshold, the stored action-value can be

backed up immediately. See Figure 4.4. On the other hand, if we tailor 2HNN to delayed node expansion by forcing a leaf to back up its parent's evaluation, a 3HNN would be advantageous by enabling a deeper MCTS.

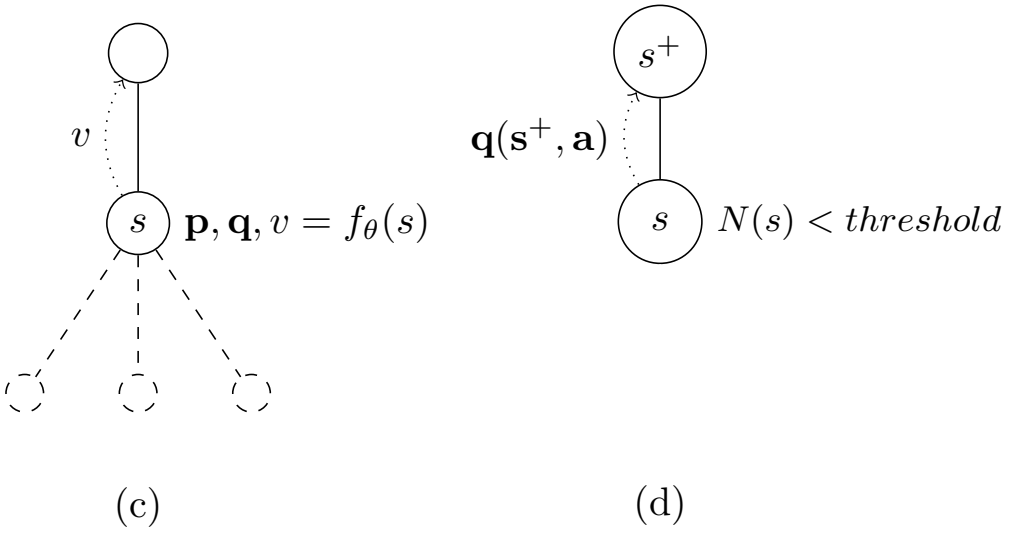


Figure 4.4: PV-MCTS with a three-head neural net f_θ: The leaf node s can be expanded with any threshold. If the visit count of s reaches the expansion threshold, s is expanded, the value estimate is backed up, action values and move probabilities are saved to new child nodes. If the visit count of s is below the threshold, the previously saved action-value estimate can be backed up.

Let PV-MCTS-2HNN be the PV-MCTS with two-head network, and PV-MCTS-3HNN be the one with three-head net. If an expansion threshold parameter must be used, PV-MCTS-2HNN can either 1) force the expansion threshold to be 0, or 2) use fast rollout as leaf evaluation when the leaf node is below the expansion threshold $\zeta \geq 1$.

For case 1), suppose PV-MCTS-2HNN and PV-MCTS-3HNN are allocated with the same amount of computation time $\tilde{T}$ on the same hardware. Let t and t' respectively be time cost per simulation for PV-MCTS-2HNN and PV-MCTS-3HNN. Assuming the time overhead for the extra action-head in 3HNN is negligible, then $t' \leq t$ since PV-MCTS-2HNN calls the neural net every simulation while PV-MCTS-3HNN does not. Thus, the number of neural value leaf estimates received by PV-MCTS-3HNN is $\frac{\tilde{T}}{t'} - \frac{\tilde{T}}{t}$ more than that of PV-MCTS-2HNN.

For case 2), a reasonable assumption is that PV-MCTS-3HNN and PV-MCTS-2HNN will use equal computation time for the same number of simulations, we then have the following observation:

Observation 1. *Suppose that the total number of simulations for MCTS is T, and the expansion threshold is $\zeta \geq 1$. Then, after the search terminates, PV-MCTS-3HNN has received at least $T - \frac{T}{\zeta}$ more neural leaf value estimates than PV-MCTS-2HNN.*

Proof. The number of neural leaf estimates received by PV-MCTS-3HNN is T, since every simulation will back up a leaf estimate of either state-value or action value. For PV-MCTS-2HNN, let $\hat{n}$ be the number of internal nodes after T simulations, and L be the set of leaf nodes. The number of neural leaf estimates used by PV-MCTS-2HNN is thus $\hat{n}$, since the neural net is called whenever a leaf is expanded. Another observation is that $\hat{n}\zeta \leq T$, because $T = \hat{n}\zeta + \sum_{\check{s} \in L} N(\check{s})$, so $\hat{n} \leq \frac{T}{\zeta}$. Therefore, PV-MCTS-3HNN uses at least $T - \frac{T}{\zeta}$ more neural leaf estimates than PV-MCTS-2HNN. □

4.3.2 Training 3HNN

Given a fixed set of games $\mathcal{D}$ generated by strong players, the loss function described in [184] is this:

$$\hat{L}(\hat{f}_\theta; \mathcal{D}) = \sum_{(s,a,z_s) \in \mathcal{D}} \left(w(z_s - v(s))^2 - \log p(a|s) + c||\theta||^2 \right) \tag{4.1}$$

Here, $0 < w \leq 1$ is a weighting factor that is used to control the relative weight of the value loss. c is a constant for the level of L_2 regularization. (s, a) is a state-action pair from dataset $\mathcal{D}$, and z_s is the game result with respect to the player to play at s. $v(s)$ is the state-value head output, and $p(a|s)$ is the policy head output. The loss function simply combines policy, value losses and L_2 regularization. Assuming infinite expressiveness of the neural network, training by such a loss function forces the neural network to predict the move probabilities and state-value of the players that have been used for producing the dataset.

At first glance, it seems more data intensive to train a three-head neural net, since all action-values must be available. Instead of just using the actions appeared in $\mathcal{D}$, we show that by using the optimal consistency between parent and child node values, we can augment extra action-values to train 3HNN on the same raw training data as for 2HNN. Assume a game state has binary outcomes, either win or lose with respect to the player at s. Then, for any given game state s, the following relations hold:

- OR constraint: if s is winning, then at least one action leads to a losing state for the opponent.
- AND constraint: if s is losing, then all actions lead to a winning state for the opponent.

If the games in $\mathcal{D}$ were played by perfect players, for each game $g \in \mathcal{D}$, g implies a tree rather than a single trajectory. See Figure 4.5.

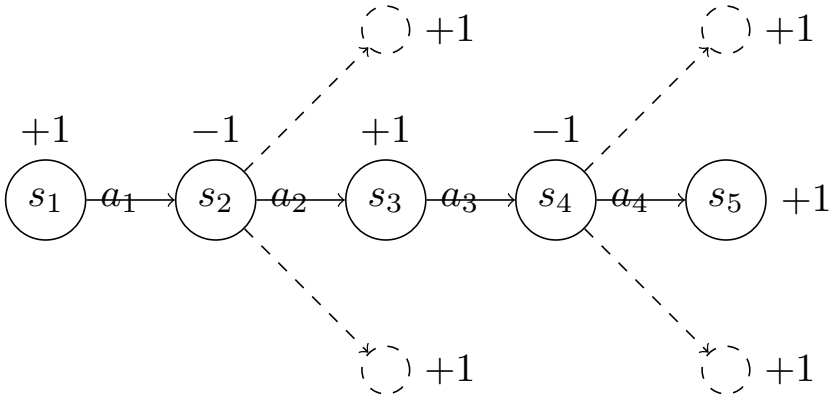


Figure 4.5: A game implies a tree, rather than a single path. For each state, the value (+1 or -1) is given with respect to the player to play there.

Therefore, we introduce the following loss term:

$$L_Q(f_\theta; \mathcal{D}) = \sum_{(s,a,-1)\in\mathcal{D}} \frac{1}{|\mathcal{A}(s)|} \sum_{a'\in\mathcal{A}(s)} \Big(q(s,a') - 1\Big)^2 \tag{4.2}$$

where $\mathcal{A}(s)$ is the action set at s (i.e., to average up the augmented error).

The other observation is that for an input state s, we now have both action- and state-value predictions. In two-player alternate-turn zero-sum perfect-information games, suppose the value function $v(s)$, $q(s,a)$ are with respect to the player to play at s or (s,a). The optimal Bellman equation is as follows:

$$v^*(s) = -\min_a q^*(s,a), s \in \mathcal{S}, \tag{4.3}$$

Here, v^* and q^* are the optimal state-value and action-value functions. To force the state and action values to satisfy the optimal consistency, we further augment the loss function by adding the optimal Bellman error as a penalty.

$$L_P(f_\theta; \mathcal{D}) = \sum_{(s,a,z_s)\in\mathcal{D}} (\min_{a'} q(s,a') + v(s))^2 \tag{4.4}$$

Combining the usual state-, action-value and policy losses, we propose the following loss function:

$$L(f_\theta; \mathcal{D}) = \sum_{(s,a,z_s)\in\mathcal{D}} \Bigg(w\bigg(\frac{1}{2}(z_s - v(s))^2 + \frac{1}{2}(z_s + q(s,a))^2\bigg) - \log p(a|s) + c||\theta||^2 \Bigg) + wL_Q + wL_P \tag{4.5}$$

Here, $q(s,a)$ is equivalent to $v(s')$ if taking a at s leads to s'. w and c are weighting parameters as in (4.1), $v(s)$, $q(s,a)$ and $p(a|s)$ are predictions from a 3HNN f_θ. The coefficient $\frac{1}{2}$ is used to equally combine the weight of action-value and state-value prediction errors. To ease parameter tuning, we also set the weight of L_Q and L_P the same.

4.4 Results on 13×13 Hex with a Fixed Dataset

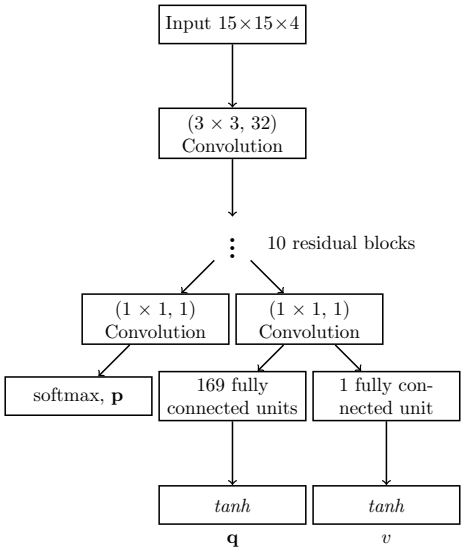
We demonstrate the effectiveness of 3HNN on 13×13 Hex. We use the same search-generated dataset as for obtaining MoHex-CNN; see Chapter 3.1.

4.4.1 ResNet for Hex

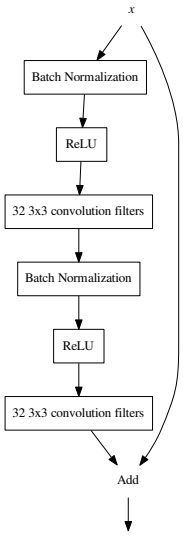
We adopt a residual neural net [84] with 10 blocks; each block has two convolutional layers; each layer has 32 3×3 filters. We use pre-activation [85] in each residual block, which applies batch normalization [97] and ReLU before convolution. The input features consists of 4 binary planes that contain only basic board state information: black stones, white stones, empty points, and to-play. In Hex, each player owns two of the board's four sides: to indicate this, we pad each board's side with a row of stones of the appropriate color. See Figure 4.6 for the detailed neural network design, where a batch normalization layer normalizes the signals for each batch signal (see [97] for details). Refer to Chapter 3 Section 3.1 for the usefulness of enriched input feature planes. Here, we follow the AlphaGo Zero and AlphaZero trend, and rely on the advancement of residual network to automatically learn more meaningful features.

4.4.2 Setup

The neural nets are implemented with Tensorflow 1.0, and trained with the Adam optimizer [104] using the default learning rate with a mini-batch size of 128 for 100



(a) ResNet for Hex



(b) A residual block

Figure 4.6: A ResNet architecture for Hex with three heads. Each residual block repeats twice batch normalization, ReLU, convolution using 32 3×3 filters, then adds up original input before leaving the block.

epochs, where one epoch is defined as a full sweep of the training data. As the previous in Go [184], we set the L_2 regularization constant c to 10^{-5}, and the value loss weight w to 0.01.

4.4.3 Prediction Accuracy of 3HNN

As in Chapter 3.1, the dataset is partitioned into disjoint training and testing sets. To show the learning progress of the neural nets, at the end of each epoch, the model parameters are saved and evaluated on the test data. We then plot the move prediction accuracy and mean square errors (MSEs) in Figures 4.7 and 4.8. The possible range of MSE is $[0, 4.0]$.

We also train a 2HNN with identical residual architecture as our 3HNN except that it has only policy and state-value heads. This can be emulated by ignoring the action-value head in Figure 4.6, and optimizing the neural network with the loss function (4.1).

Figures 4.7 (left) shows that the MSE from the newly introduced action-value head in 3HNN achieves similar accuracy with the state-value head of 2HNN. Although 3HNN did not produce significantly better predictions than 2HNN, it is

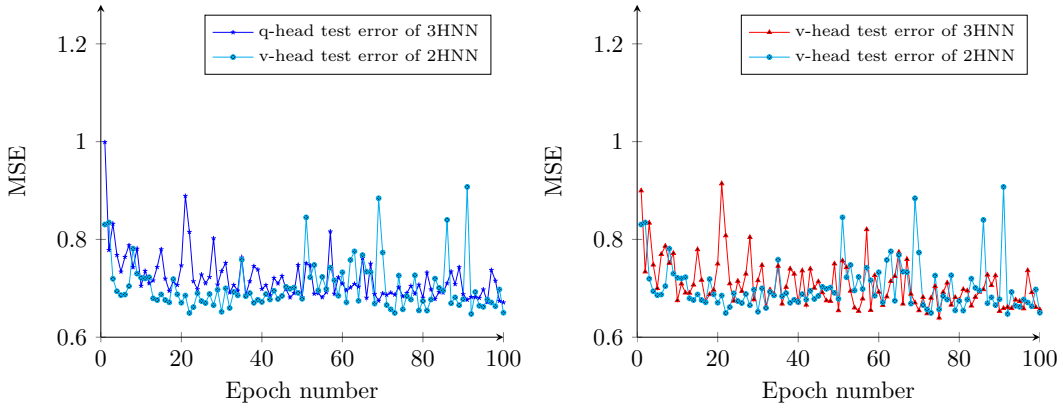


Figure 4.7: Mean Square Errors of two- and three-head residual nets.

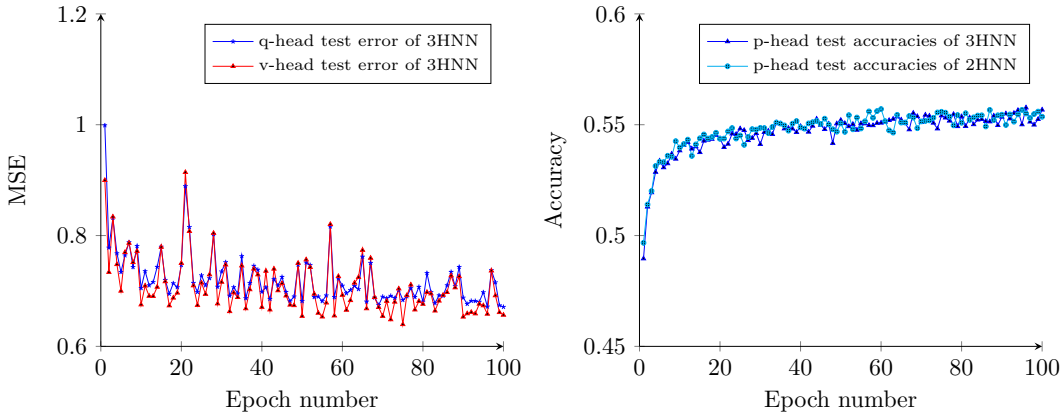


Figure 4.8: MSE (left) and top one move prediction accuracies (right) of two- and three-head residual nets.

advantageous in the sense that it gives an additional vector of all action-values with a single neural net forward pass.

Figure 4.8 compares the move prediction accuracy of 2HNN and 3HNN. The networks have almost indistinguishable learning curves, because they are trained on the same dataset with identical policy loss.

4.4.4 Evaluation in the Integration of PV-MCTS

We now present experimental comparisons between two and three heads nets in the same search framework. As before, we build our programs upon the codebase of MoHex 2.0 [58, 96, 151], and use MoHex-CNN from previous chapter as a benchmark to measure the relative strength of the new programs. Denote MoHex-3HNN as the implementation of PV-MCTS-3HNN, and MoHex2-2HNN for PV-MCTS-2HNN. The key differences between MoHex-3HNN, MoHex-2HNN and MoHex-CNN are reflected in their leaf evaluation:

- MoHex-3HNN uses the default expansion threshold 10 of MoHex 2.0.
- To simply experiment, we use either default expansion threshold or an expansion threshold of 0. In the former case, when the leaf node's visit count is below the expansion threshold, MoHex-2HNN uses the pattern-based rollout as MoHex 2.0.
- MoHex-CNN uses the default expansion threshold; it always use pattern-based rollout from MoHex 2.0 as it does not use neural network for value estimation.

The same RAVE in-tree selection formula (as in Eq. (3.1.6)) is used for all programs.

We evaluate neural net models at epochs 20, 30, ..., 100 with an interval of 10. Each neural net model is combined the same MCTS implementation and played against MoHex-CNN. To facilitate evaluation, we set the same 1000 simulations per move for these players, which generally consumes about 1 second per move for each player. Following a practice in the literature [96], the matches are played by iterating all openings with symmetric moves removed. Each round of match consists of 170 games, where each player plays an opening twice (as Black and White). Due to the fast speed of evaluation, we repeat 5 rounds for each match. No swap rule was applied.

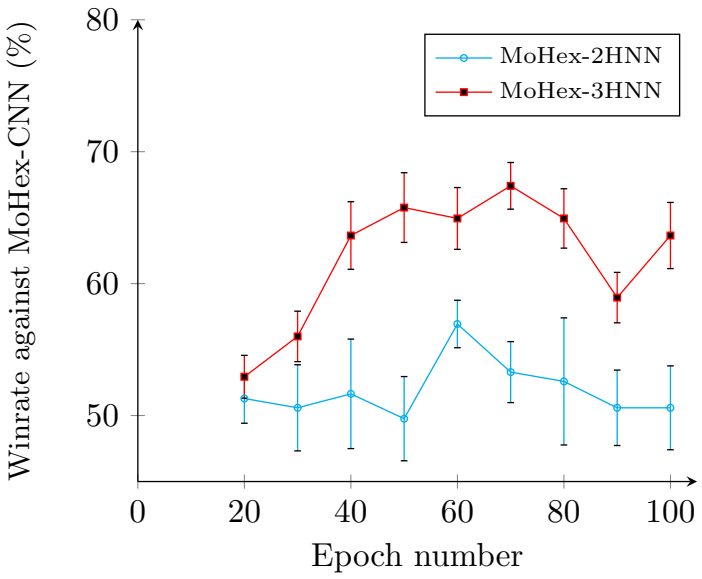


Figure 4.9: Results of MoHex-2HNN and MoHex-3HNN against MoHex-CNN. All programs use the same 1000 simulations per move. MoHex-2HNN uses playout result when there is no node expansion. After epochs 70 and 60, MoHex-3HNN and MoHex-2HH's performance decreased, possibly due to over-fitting of the neural nets: Figures 4.7 and 4.8 show that around epoch 70, the value heads of 3HNN generally achieve smaller value errors than epochs around 80 and 90. The error bar represents the standard deviation of each evaluation.

Figure 4.9 shows that PV-MCTS-3HNN is better than PV-MCTS-2HNN in this setting. Both PV-MCTS-3H and PV-MCTS-2H achieve winrates larger than 50% against MoHex-CNN, presumably because MoHex-CNN only uses neural net as prior knowledge during the in-tree phase. As in observation (1), given that more neural net estimates are more accurate than random rollouts, PV-MCTS-3HNN is better than PV-MCTS-2HNN because it always uses neural net value estimation while PV-MCTS-2HNN has to use playout result for each leaf below the expansion threshold.

We then compare PV-MCTS-2HNN to PV-MCTS-3HNN by letting the former have an expansion threshold of 0. To have a fair comparison, we give all programs the same search time of $10s$ per move. The results are summarized in Table 4.3. As a reference, we also include the result from PV-MCTS-2HNN with the default expansion threshold.

Consistent to Figure 4.9, the results in Table 4.3 show that MoHex-3HNN still achieves the best performance, significantly outplaying MoHex-CNN. With the default expansion threshold, MoHex-3HNN takes $0.3ms$ to execute one simulation on

Table 4.3: Winrates of MoHex-2HNN and MoHex-3HNN against Mohex-CNN with the same time per move. For best performance, MoHex-3HNN and MoHex-2HNN respectively use the neural net models at epochs 70 and 60.

Player	As Black	As White	Overall Win
MoHex-3HNN	76.5%	70.6%	73.5%
MoHex-2HNN threshold 0	65.9%	57.6%	61.8%
MoHex-2HNN default threshold	69.4%	56.5%	62.9%

average. MoHex-2HNN with expansion threshold 0 takes $1.8ms$ per simulation, about 6 times slower, which explains MoHex-3HNN's better performance given the same time per move.

In head-to-head match, under the same setting using $10s$ per move, MoHex-3HNN's won 64.1% against MoHex-2HNN-0 and 70.6% against MoHex-2HNN-10, where MoHex-2HNN-x means MoHex-2HNN with an expansion threshold of x. MoHex-3HNN won 82.4% against MoHex 2.0 when both programs are giving the same 10s per move.

4.5 Transferring Knowledge Using 3HNN

As we can see in Figure 1.1, the similarities between Hex on different board sizes raise the following questions:

- To what extent can learned neural net knowledge be reused on a smaller board?
- To what extent can such knowledge be generalized to a larger board?

It is known that convolution filters are translation invariant. We have seen in Chapter 3.2 that this property can be harnessed by designing a policy network that contains only such filters, thereby creating a policy network that is transferable. Similar to action probabilities, the $\boldsymbol{q}$ head in 3HNN could also be fully transferable if the final fully connected layer is removed. This architecture is shown in Figure 4.10. The major difference between Figures 4.6 and 4.10 is that the fully-connected layers are removed from the **q**-head: by doing so, we expect that the action-value vector can generalize to board sizes not used in training. This neural net can be trained as before with the loss function of Eq. (4.5).

To study the how transferable the architecture is, we use three datasets from different board size:

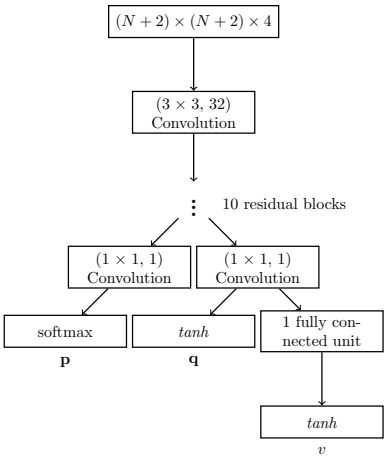


Figure 4.10: A given Hex state of board size $8 \leq N \leq 19$ is padded with black or white stones along each border, then fed into a feedforward neural net with convolution filters. A fully-connected bottom layer compresses results to a single scalar v.

- On 8×8 Hex, 62853 games among MoHex 2011, MoHex 2.0 and Wolve.
- On 9×9 Hex, as in Chapter 3.2.
- On 13×13 Hex, as in Chapter 3.1.

4.5.1 Setup

We train the network with learning rate 0.001 and a mini-batch size of 128 for 100 epochs. Model parameters are saved after each epoch. L_2 regularization constant c is set to 10^{-5}; value loss weight $w = 0.01$. We conduct two sets of experiments:

- Train the neural net on 13×13 games and investigate transferability to smaller boards.
- Train the neural net on 9×9 games and investigate transferability to larger boards.

In either board size, the dataset is split into 90% for training and the rest for test.

4.5.2 Prediction Accuracy In Different Board Sizes

We measure the prediction accuracy of the saved neural net models. Figure 4.11 shows the test prediction accuracy of $\mathbf{p}$ and $\mathbf{q}$ heads of the neural net models obtained from learning on 13×13 games; for $\mathbf{p}$ head, the label is the move played in the game dataset; for $\mathbf{q}$ head, the label for $\mathbf{q}_a$ is the game result after playing the

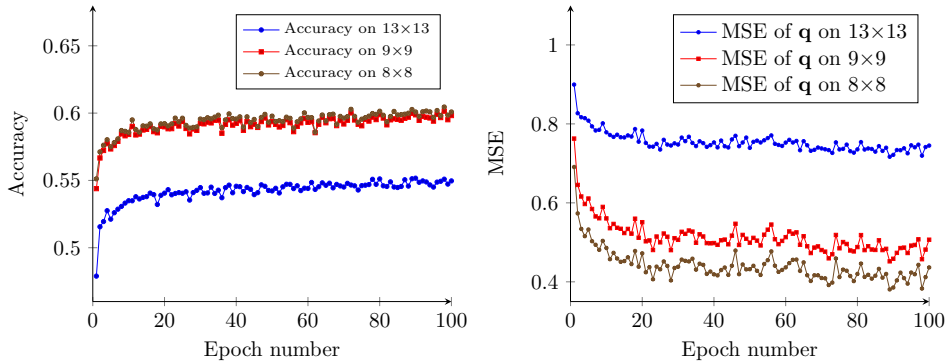


Figure 4.11: 13×13 training: prediction accuracy across boardsizes.

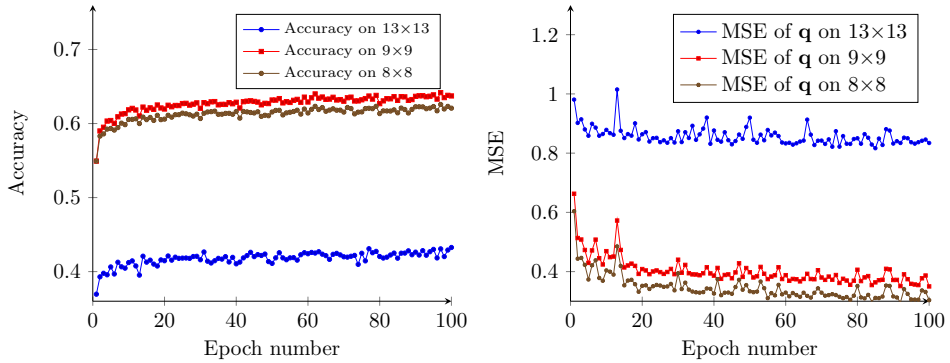


Figure 4.12: 9×9 training: prediction accuracy across board sizes.

corresponding action a. As expected, neural net models optimized on 13×13 games achieved high accuracy on smaller boards. Prediction accuracy on 8×8 and 9×9 are better than on 13×13, reflecting that smaller board sizes are easier to master. Compared to Figures 4.8 and 4.7, this network's **q**- and **p**-heads achieved slightly larger errors. The benefit is that the new architecture to generalize its action-value prediction to other board sizes.

It is more interesting to measure transferability from smaller boards to larger, as on smaller boards it is easier produce high quality games. Figure 4.12 shows prediction accuracy after training the net on 9×9 games. Surprisingly, both **q** and **p** yield reasonable prediction accuracy even on 13×13 Hex. However, on 13×13 Hex, by comparison, prediction accuracy of Figure 4.12 is about 10% lower than in Figure 4.11.

4.5.3 Usefulness When Combined with Search

How useful are our neural nets when used in search? We answer this question by combining our nets with the MCTS of MoHex 2.0 as before. The v-head is board size

dependent, so we use it only when the running board size is the same as that used in training; otherwise, only the **q**- and **p**-heads are used. In our implementation, move prior and action-value are both stored at each node creation. We call this new transferable neural net program MoHex3H.

We evaluate 10 neural net models at an epoch interval of 10. Each model is combined with MoHex3H and played against MoHex 2.0. Table 4.4 shows the results on boardsizes 9×9 to 13×13. We did not include 8×8 data, because positions on this board are easily solved [89, 151]. For all programs, we allow 10^4 simulations per move. Each tournament is played by iterating over all $N \times N$ opening moves. As before, each opening is used twice, with each program playing as first-player and second-player; and no swap rule.

Table 4.4: MoHex3H using 13×13-trained nets: win rate (%) versus MoHex 2.0 and MoHex-CNN. Columns 2-11 show strength by epoch.

Board \ Epoch	10	20	30	40	50	60	70	80	90	100
9×9	71.0	64.8	69.1	75.9	73.5	77.2	69.1	74.7	77.8	67.9
10×10	71.0	78.0	66.5	71.0	82.5	83.5	80.5	78.0	78.5	72.5
11×11	67.4	67.8	71.5	74.0	71.9	76.4	74.8	78.5	78.1	76.4
12×12	67.0	71.9	70.5	73.6	76.4	78.5	78.1	77.4	79.9	75.7
13×13	63.0	63.3	68.3	69.8	73.7	76.3	74.9	75.4	74.3	74.9
13×13	44.1	42.9	46.5	55.3	58.8	61.2	55.3	52.4	61.2	55.3

Table 4.4 shows results against MoHex 2.0 using net models learned from the 13×13 dataset. The high **p**- and **q**-head prediction accuracy shown in Figure 4.11 yields strong play on smaller board sizes. MoHex-CNN only plays on 13×13 Hex, so we include the result against MoHex-CNN in the last row: MoHex3H defeats MoHex-CNN with this transferable neural net, although the win rate is lower than that of a three-head net with fully connected bottom layer (MoHex-3HNN achieved win-rate over 70%; see Table 4.3).

Table 4.5 shows the results against MoHex 2.0 using the models learned from the 9×9 dataset. The resulting program defeats MoHex 2.0 even on boardsize 13×13,

Table 4.5: MoHex3H using 9×9-trained nets: win rate (%) versus MoHex 2.0.

Board \ Epoch	10	20	30	40	50	60	70	80	90	100
9×9	63.6	69.1	63.0	65.4	65.4	72.2	74.7	72.8	71.6	72.8
10×10	63.5	70.0	71.5	68.0	67.0	75.5	76.0	76.0	76.5	77.5
11×11	62.8	58.3	64.0	66.1	69.0	66.1	67.8	64.0	66.1	65.7
12×12	51.0	45.8	56.2	53.5	45.8	60.1	54.9	49.3	58.7	58.0
13×13	35.3	39.4	47.1	47.6	46.5	42.9	42.4	42.9	52.9	54.1

Table 4.6: MoHex3H using 9×9-trained nets, with **p**-head only: win rate (%) versus MoHex 2.0.

Epoch \ Board	9×9	10×10	11×11	12×12	13×13
90	68.5	75.5	68.2	65.3	60.6
100	68.5	70.5	70.7	67.4	63.5

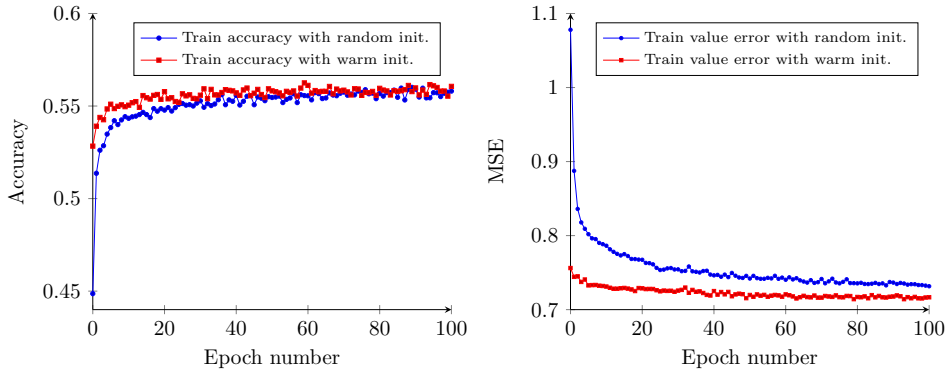


Figure 4.13: 13×13 training errors, with and without warm initialization.

although its margin of victory there is less than on other boardsizes. MoHex 2.0's playout policy was trained mostly on 13×13 games [96]. A possible explanation is that the low **q**-head prediction accuracy is insufficient to match the strength of MoHex 2.0 playouts. To verify this, we ran extra tournaments with the **q**-head in MoHex3H's search disabled, forcing MoHex3H to use the same pattern-based playouts as MoHex 2.0. By comparing the results in Tables 4.5 and 4.6, we can see that, at epochs 90 and 100, the winrate against MoHex 2.0 dropped on boardsizes 9×9 and 10×10 but rose on boardsizes 11×11 and larger, suggesting that the **p**-head consistently aided the search, while the **q**-head was insufficiently accurate to replace pattern-based playouts on larger boardsizes.

In summary, we conclude that (1) without fine-tuning, transferring neural knowledge to larger board sizes is harder than to smaller and (2) when transferring to larger boar sizes, the learned **p**-head seems more robust than the **q**-head.

4.5.4 Effect of Fine-tuning

We further investigate fine-tuning 9×9-trained neural models on 13×13 data. Specifically, we use the neural net model at epoch 100 in Figure 4.12 to initialize training on the 13×13 dataset.

Figures 4.13 and 4.14 compare the learning curves on the same 13×13 dataset

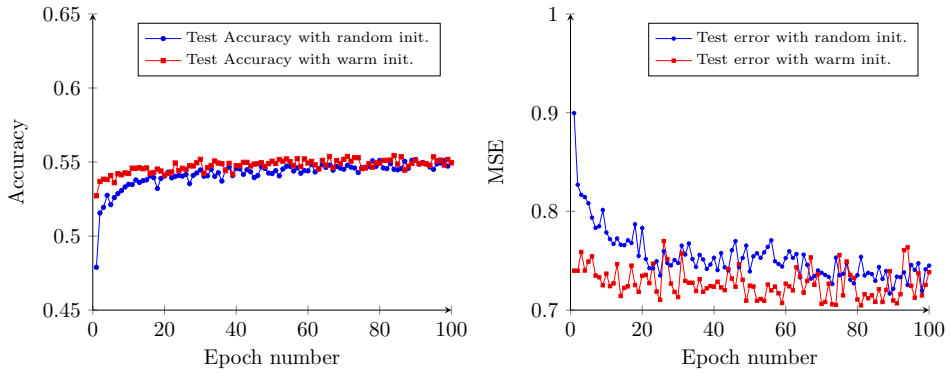
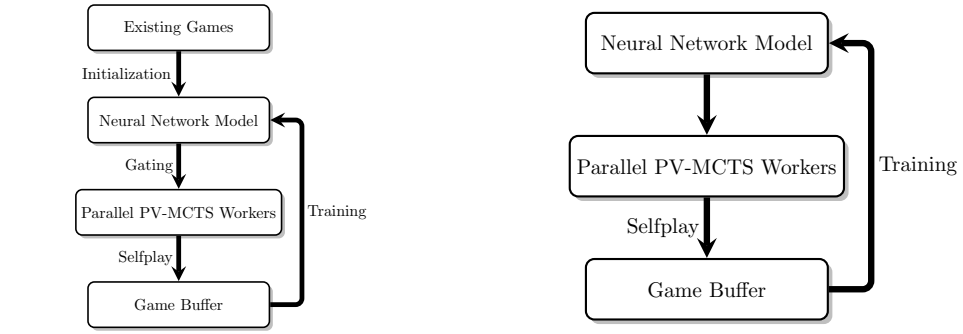


Figure 4.14: 13×13 test errors, with and without warm initialization.



(a) Closed-loop training for Olympiad

(b) Zero-style closed-loop learning

Figure 4.15: Closed-loop learning schemes

with warm initialization and starting from random weights. Notice that learning benefits from warm initialization: the neural net learns faster and achieves better accuracy. This strength gain is amplified with search: when used with the MCTS of the previous section, the resulting player quickly passes 70% wins against MoHex 2.0: at epoch 10, the win-rate is already 72.4%. By comparison, Table 4.4 shows that with random weights initialization this took more than 40 epochs.

4.6 Closed Loop Training with 3HNN

We now present AlphaGo Master, AlphaGo Zero and AlphaZero style closed loop training with 3HNN. These can be summarized as the flowcharts in Figure 4.15.

4.6.1 Training For 2018 Computer Olympiad

To prepare for the 2018 Hex tournament, we conducted a closed loop training on 13×13 and 11×11 Hex, following the scheme in Figure 4.15a.

Figure 4.16 shows the testing results of the saved neural net models tested on games of Maciej Celuch, the best human player on the website `little golem` [124]. Due to limited computation power and our uncertainty about key parameter choices, these results are noisy as the experiment was not conducted in a fully automatic manner. Some intentional or accidental interruptions happened during the process around one months of training. For example, we thought a 10 block ResNet with 32 3×3 filters per layer would be large enough for training on 13×13, but we did not see obvious playing strength increases from iteration 2 to 3; we then increased the network to 64 filters and observed some improvement. Then, two iterations later, because of the slower speed, we changed back to 32 filters for another 2 iterations. These unsatisfying results eventually pushed us to use 128 filters per layer, and then we saw significant improvement in MoHex-3HNN. Finally, we used a 128 filters per layer 10-block three-head ResNet for participating in the 2018 Computer Hex tournament. The other parameters we were uncertain about and had switched between a few choices include:

- Whether to use RAVE.
- PUCT constant c_{pb}.
- How many MCTS iterations to use for selfplay.
- How many epochs to train the neural network given recently generated selfplay games.

AlphaGo Fan [180] used a PUCT constant of 5.0 and no RAVE. We initially considered such a choice but did not observe better performance. We tried to tune the PUCT constant, and eventually found that the default 2.47 with RAVE worked best. This is consistent with later empirical analysis reported for ELF OpenGo [201]. We tried different numbers of MCTS iterations for selfplay game generation varying from 1000 to 10000, and different epochs for training the neural networks on each set of newly generated sample games (one epoch means a full sweep of the set of training dataset). We had to trade-off between game quality and speed of learning because the goal was to obtain strong player in time for the competition. On the 11×11 board, a similar approach was used on another contributed [1] GPU computer; the starting neural network was the same as for 13×13. Therefore, in the first iteration,

[1] Thanks Mengliao Wang

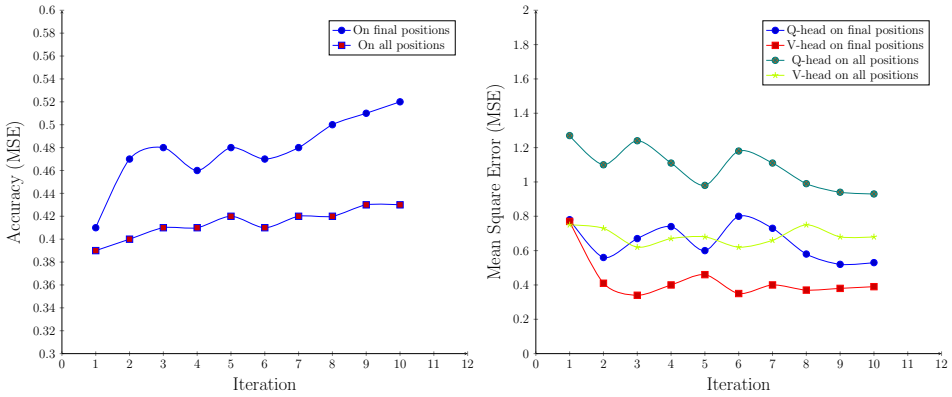


Figure 4.16: On 13×13 Hex, around 10 training iterations was finished before participating 2018 Computer Olympiad Hex tournament, using 2 4-core GPU computers with GTX 1080 and GTX 1080Ti. 3HNN was initialized using MoHex generated data. The first three iteration used 32 filters per layer, 4–5 used 64 filters per layer, 6–7 used 32 filters per layer, 8–10 used 128 filters per layer. The curve shows MSE or accuracy on all games played by Maciej Celuch from little golem. Celuch is presumably the strongest human Hex player [124]; we dumped 620 games played by 2018 December. Noticeably, he did not lose any of these games.

random playout rather than a value net was used for evaluating leaf nodes. In total, for 13×13 Hex, a total number of 0.48 million selfplay games were produced before participating the Computer Olympiad.

Our competitor DeepEzo [195] used a variant of minimax that adopts a policy and value network, trained iteratively for 2.6 million of games [102, 195]. DeepEzo achieved over 80% win-rate against MoHex 2.0 on 13×13 Hex. In the competition, MoHex-3HNN defeated DeepEzo 7–0 on 13×13 and 5–0 on 11×11. See [67].

Finally, we present statistics in our selfplay games for the winning probability for each opening on 13×13 and 11×11 Hex.

4.6.2 Zero-style Learning

To further verify the performance of 3HNN in zero-style training, considering that automatically running the iterative procedure could be computationally intensive, we conducted the Hex experiment on relatively small board size 9×9. Following the flowchart of Figure 4.15b, we used these configurations:

- For MCTS, we turn off playout and *knowledge computation*. We leave the in-tree selection formula with RAVE unchanged. To be consistent with AlphaGo Zero and AlphaZero, we add Dirichlet noise to prior probability provided by neural net: $p(s,a) = (1-\epsilon)p(s,a) + \epsilon\text{Dirichlet}(\alpha)$ where $\epsilon = 0.25, \alpha = 0.03$.

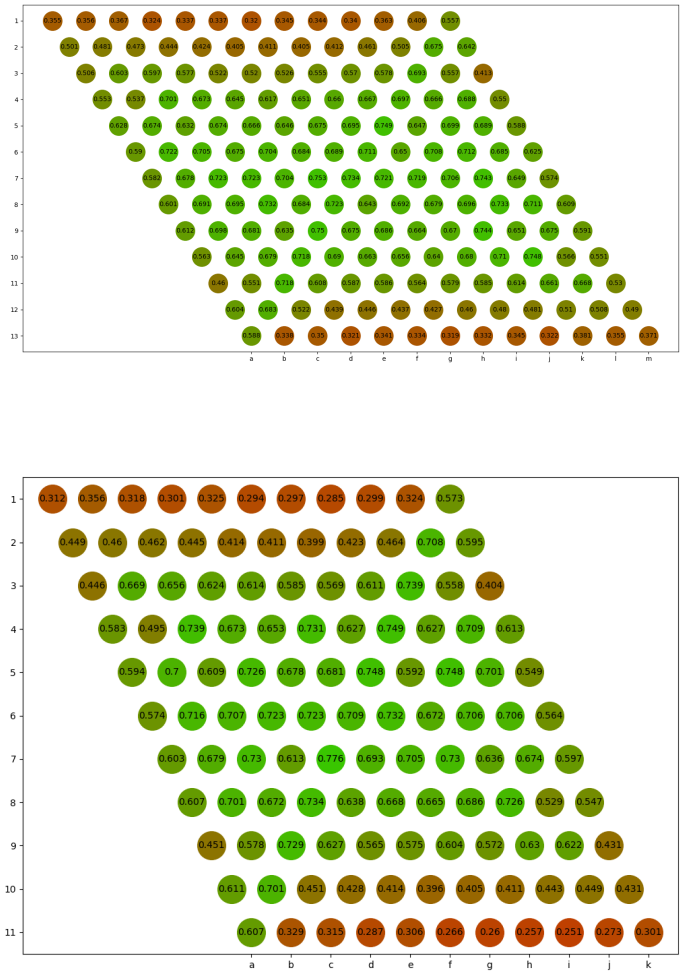


Figure 4.17: Averaged win value for each opening cell on 13×13 and 11×11 Hex. The numbers are consistent to some common belief in Hex; for example, every cell more than two-row away from the border is likely to be a Black win.

The default expansion threshold of 10 is used.

- For each selfplay game, at each state, we conduct a $n_{mcts} = 800$ iteration MCTS. For the first n_{dither} moves in a game, after search terminates, randomly sample move $a \propto \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$. We set n_{dither} to 10 due to short move sequence in 9×9 Hex.

- For neural net, we use the 10-block three-head ResNet as in Figure 4.6.

- We use 60 selfplay workers; each produces 200 games per iteration.

- We use synchronous training. At each iteration, the training worker collects all games from all selfplay workers, and then re-optimizes the neural net with stochastic gradient descent with momentum 0.9. The initial learning rate is 0.005, decayed by factor 0.95 each iteration. Training is with a mini-batch size of 128, using the following loss function:

$$L(f_\theta; \mathcal{D}) = \sum_{(s,a,z_s)\in\mathcal{D}} \Bigg(\Big((z_s - v(s))^2 + \underbrace{(z_s + q(s,a))^2}_{R1} + \underbrace{\frac{\max(-z_s, 0)}{|\mathcal{A}(s)|} \sum_{a'\in\mathcal{A}(s)} (z_s + q(s,a'))^2}_{R2} + \underbrace{(\min_{a'} q(s,a') + v(s))^2}_{R3} \Big) - \boldsymbol{\pi}^T(s) \log \boldsymbol{p}(s) + c||\theta||^2 \Bigg)$$

  Here, $\mathcal{D}$ contains the set of 200 games generated at each iteration; for each $\mathcal{D}$, we optimize the neural net for 5 epochs, i.e., sweeping $\mathcal{D}$ for 5 times. $R1$, $R2$ and $R3$ are the three newly added loss terms for the loss used in AlphaZero.

On 9×9 Hex for MoHex 2.0, around 10 seconds per move generates strong playing, but there is no guarantee that these data used for testing is error-free. To further see the result, we randomly produce a set of game states and using solver to perfectly label these positions; we obtained 8485 examples. See Figure 4.18 for the comparison in detail. From Figure 4.18 we see that AlphaZero-3HNN learned faster than AlphaZero-2HNN. The high fluctuations of the curve from AlphaZero-2HNN is due to its slow learning. We do not use selfplay Elo to measure learning progress primarily because of the inflation phenomenon [201].

The major reason for the slow learning of AlphaZero-2HNN in Figure 4.18 is that setting $n_{mcts} = 800$ makes the generation of a training game time-consuming.

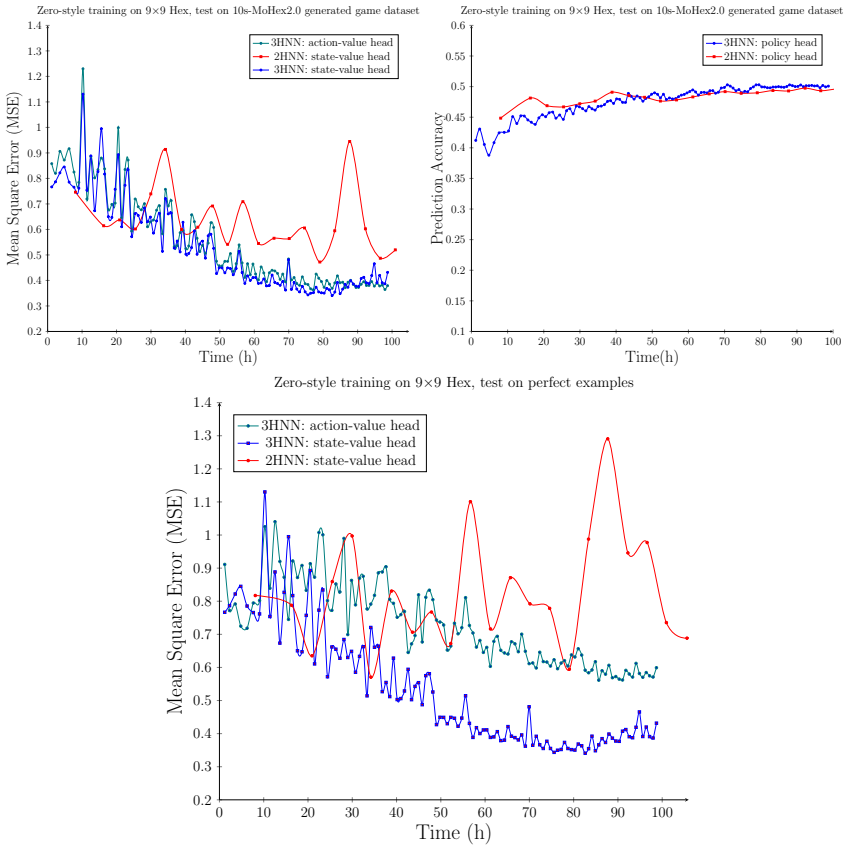


Figure 4.18: On 9×9 Hex zero-style training with MoHex3HNN and MoHex2HNN. Test the neural network model on MoHex2.0 selfplay games or randomly generated perfectly labeled game states. Red curve is result obtained under the same configuration except that a 2HNN is used. The above two plots were tested on a test set of 149362 examples, while the last one was from a smaller set of 8485 examples.

We therefore conducted another experiment by reducing $n_{mcts} = 160$ and run AlphaZero-2HNN and AlphaZero-3HNN. To further encourage exploration, we changed the Dirichlet noise parameter α to 0.15, $n_{dither} = 30$, and in-tree formula to PUCT with $c_{puct} = 1.5$. Figure 4.19 contains the comparison: AlphaZero3HNN did not achieve better prediction accuracy on the test data. To see if the learned 3HNN is really weaker than 2HNN, we conduct a head-to-head match between the networks generated at each iteration. Figure 4.20 contains the match results, which indicate that MCTS-3HNN is mostly stronger than MCTS-2HNN. Same as for training, in our test, we turn off the knowledge computation for MCTS-2HNN and MCTS-3HNN.

We suspect that one reason that 3HNN failed to achieve smaller state-value prediction error is due to the $R2$ term in the loss function. By injecting $R2$, we

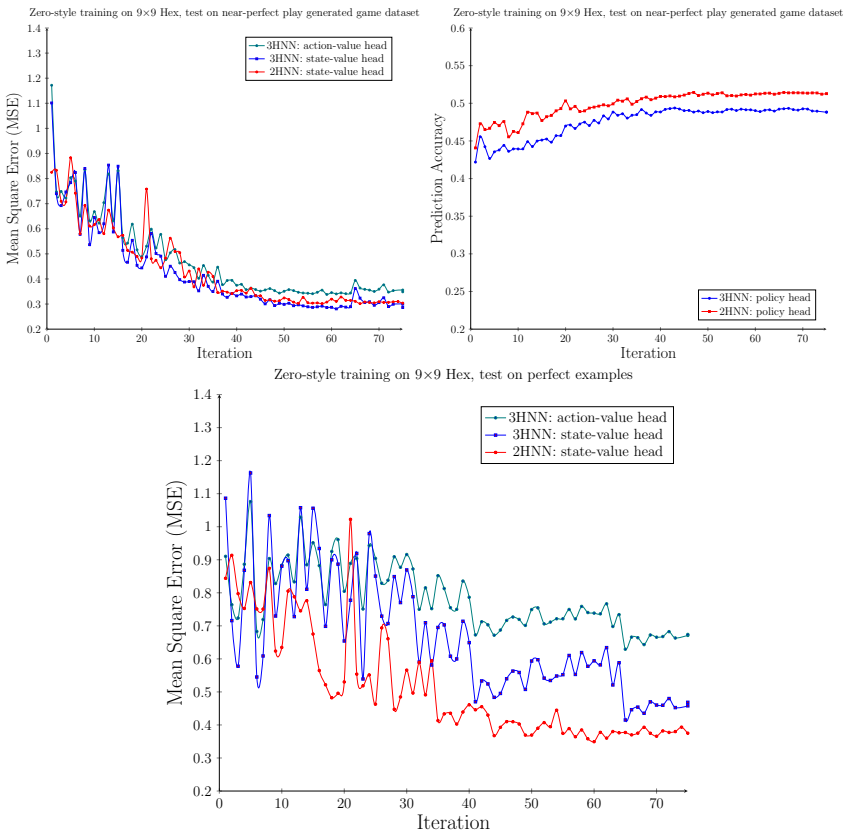


Figure 4.19: AlphaZero-2HNN versus AlphaZero-3HNN. 2HNN uses expansion threshold 0, $n_{mcts} = 160$; 3HNN uses expansion threshold 10, $n_{mcts} = 800$; all other parameters are the same.

assume the action made by MCTS player is the best action. However, this is not true because of the dithered action selection mechanism (i.e., $n_{dither} = 30$). To see the effect of $R2$, we run another AlphaZero-3HNN by removing $R2$. Figure 4.21 shows the result along with comparison to AlphaZero-2HNN: without $R2$, AlphaZero-3HNN obtained neural network models with smaller state-value prediction errors on both test sets.

Figure 4.22 shows the head-to-head match results between MCTS-3HNN and MCTS-2HNN, where the 3HNN models were obtained without using $R2$ in training. MCTS-3HNN defeated MCTS-2HNN as learning goes on. This is consistent to the learning curves in Figure 4.21.

In addition, we test our final iteration models of 3HNN and 2HNN by playing against MoHex 2.0. The same as in [9], we let MCTS-2HNN and MCTS-3NN use $n_{mcts} = 800$, and MoHex 2.0 use $n_{mcts} = 10000$. MCTS-3HNN and MCTS-2HNN respectively achieved mean win-rates 91.57 ± 1.8 and 89.7 ± 3.317 (both with 95%

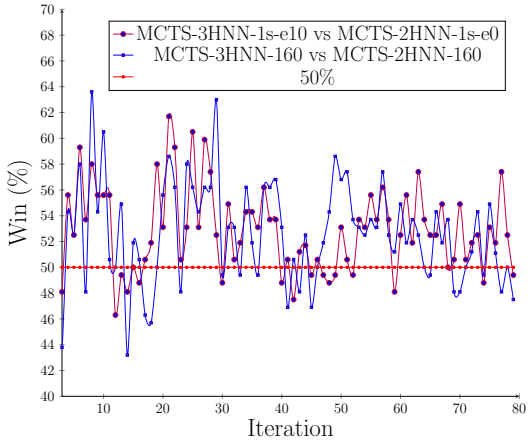


Figure 4.20: MCTS-3HNN against MCTS-2HNN. Each match consists of 162 games by letting each player starting from each opening cell once as Black and White. MCTS-3HNN-160 means it uses 3HNN for 160 iteration MCTS with expansion threshold of 0. MCTS-2HNN-160 means it uses 2HNN for 160 iteration MCTS and expansion threshold of 0. MCTS-3HNN-1s-e10 means the player uses 1s per move with expansion threshold of 0. MCTS-2HNN-1s-e0 means it uses 1s per move with expansion threshold of 0.

confidence); see Table 4.7. In comparison, the Zero implementation PGS-EXIT in [9] achieved a win-rate of 58% against MoHex2.0-10000 for 800-simulation search. This indicates the high quality of our AlphaZero implementation for both 3HNN and 2HNN.

Table 4.7: Detailed match results against MoHex 2.0. Each set of games were played by iterating all opening moves; each opening is tried twice with the competitor starts first and second, therefore each match consists of 162 games. We use the final iteration-80 models for 2HNN and 3HNN from Figure 4.21. The overall results are calculated with 95% confidence. MCTS-2HNN and MCTS-3HNN used 800 simulations per move with expand threshold of 0. MoHex 2.0 used default setting with 10,000 simulations per move.

Player \ Set	1	2	3	Overall
MCTS-2HNN-800	86.4%	92.0%	90.7%	$89.7\% \pm 3.32$
MCTS-3HNN-800	93.2%	91.4%	90.1%	91.57 ± 1.77

The results in this section were obtained on a server with 56 Intel(R) Xeon(R) CPUs (E5-2690 v4 2.60GHz), 500GB RAM, with 6 Tesla P100 GPU each with 16 GB RAM [2].

[2]Thanks Huawei Canada for providing the computation resource

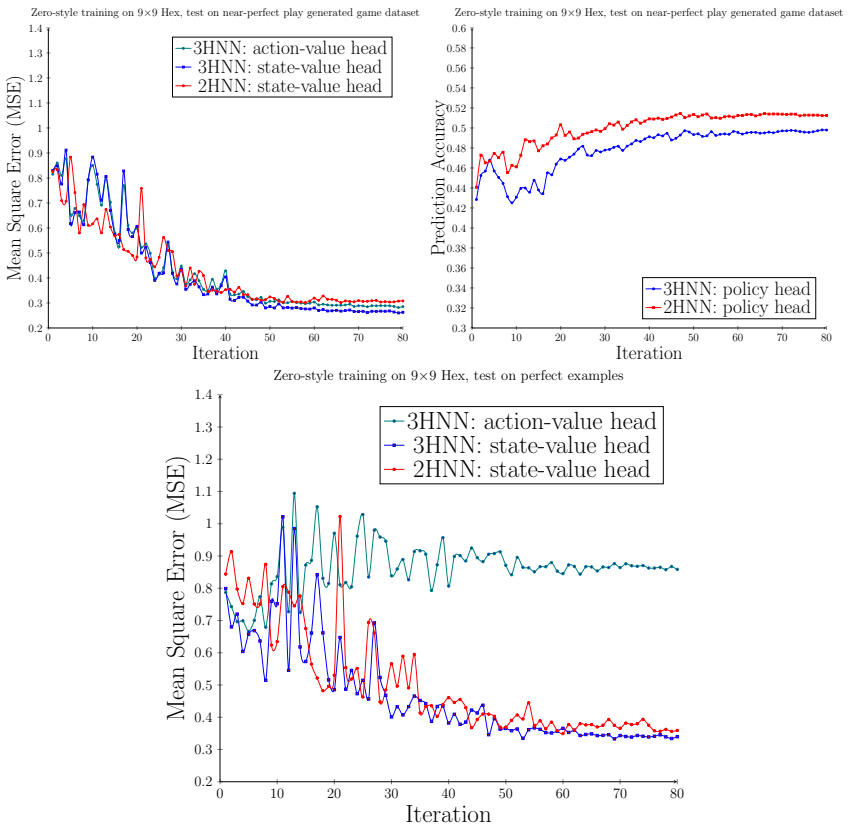


Figure 4.21: AlphaZero-2HNN versus AlphaZero-3HNN. 2HNN used expansion threshold 0, $n_{mcts} = 160$; 3HNN used expansion threshold 10, $n_{mcts} = 800$; all other parameters same.

4.7 Discussion

In this chapter, we provided a thorough review of recent advances in the game of Go. We developed a new three-head network and applied it to the game of Hex, and obtained significant playing strength improvement over the previous state-of-the-art. In the 2018 Computer Hex tournament, our player convincingly defeated a strong competitor which uses a two-head deep neural networks for move selection and game state evaluation. We further investigated the merit of the three-head network in zero-style learning, and showed that it can lead to stronger 9×9 Hex player. There have been other research explorations on improving the efficiency of AlphaZero learning, e.g., using multiple small and large networks in the tree search [115]. Combining these improvements with three-head network might lead to further improvement.

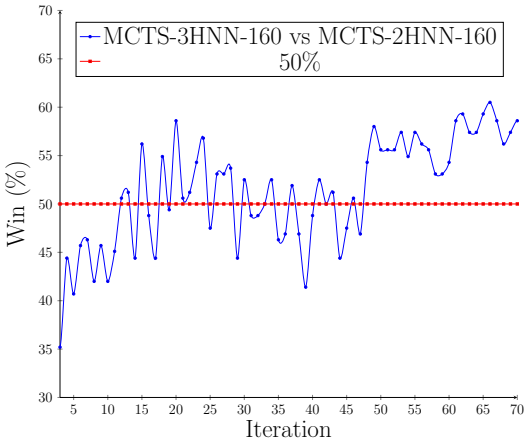


Figure 4.22: MCTS-3HNN against MCTS-2HNN. Each match consists of 162 games by letting each player starting from each opening cell once as Black and White.

Chapter 5

Solving Hex with Deep Neural Networks

In this chapter we focus on solving Hex with the help of deep neural networks.

5.1 Focused Proof Number Search for Solving Hex

As discussed in Chapter 2.4.5, all 8×8 Hex openings were first solved by Henderson *et al.* [89] after extending the inferior cell analysis of Hex positions [82], adding a better implementation of H-search, and using DFS for exhaustive search. Focused Depth First Proof Number Search (FDFPN) [87] performed better than DFS with backtracking and straightforward DFPN. We can view FDFPN as a combination of *best-first* (BF) and *backtracking* (BT) search strategies (see *BF-BT* combinations in [153]). At each node, by setting a search window that is smaller than the branching factor, FDFPN behaves locally like normal BF, while globally it looks like DFS with BT. DFS with BT works well when there is an accurate move ordering, but fails disastrously if the move ranking is poor, while BF ranks the potential of each frontier node based all nodes explored so far, so the combination of BF and BT enables the search to harness the strength of a good move ordering function while preserving its basic best-first principle.

With proof number search (PNS), reducing the width to a small fixed value may not be helpful, because PNS exploits the non-uniform branching factor for selecting a most promising node to expand. When expanding a node, FDFPN uses the following formula to select a subset of child nodes to the search:

$$child\ limit = base + \lceil factor \times |live\ children| \rceil \tag{5.1}$$

Here, $base \geq 1$, and the widening factor $0 < factor < 1$; *live children* is a set that

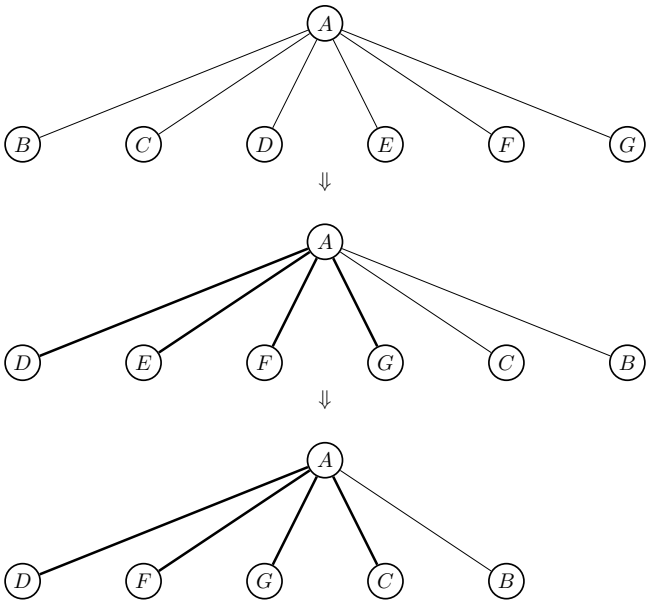


Figure 5.1: Focused Best-first Search uses two external parameters: a function f_R for ranking moves, and a way for deciding window size. Here, assume $f_R(D) > f_R(E) > f_R(F) > f_R(G) > f_R(C) > f_R(B)$, and the window size is simply set to a fixed number of 4. E is found to be winning, then E is removed, and the next best node is added to the search window.

contains all child nodes that have not been solved. Once a node x in *live children* is found to be winning (corresponding to a losing move from its parent), it is removed. Then, FDFPN either maintains the same *child limit* or introduces a new child to the search. If x is losing, it is clear that other child nodes can be ignored since its parent node becomes *solved* as a win. Figure 5.1 shows the scheme behind FDFPN.

The strength of Equation (5.1) comes from the observation that assuming node n has N child nodes, and k nodes are proved winning before a losing child is found, then at most $\max(0, N - k - base - \lceil factor \times (N - k) \rceil)$ child nodes are ignored by the BF search. However, the potential drawback is that when the move ordering function is poor, i.e., when the losing child node x is given a rank k_x and $k_x > l_{max}$, where l_{max} is the maximum index of the selected child nodes when expanding their parent, then at least $\lfloor \frac{k_x - b - factor \times N}{1 - factor} \rfloor + 1$ nodes must be solved as winning before including x to the search. To see why, assume that at least n nodes are solved before including x to search: then $n + b + factor \times (N - n) \geq k_x$, thus n is at least $\lfloor \frac{k_x - b - factor \times N}{1 - factor} \rfloor + 1$.

So, the effectiveness of FDFPN depends on two features: (1) the computation of widening size and (2) the use of an external move ordering function for accurately

selecting promising moves. In [87], (1) was manually set to a fixed factor, and (2) was achieved with the resistance-based heuristic evaluation [87] function in Hex.

For (1), it is uninformative to adopt a static widening factor for all expanding nodes regardless their likelihood of being a win or a loss; for (2), the resistance-based evaluation function, even with ICA and H-search [151], is often inaccurate, as noted in [87].

Deep convolutional neural networks (CNNs) [114, 119] have been successfully used in a variety of domains for providing reliable heuristic knowledge. Motivated by their success in game-playing [180, 184], as we have seen in Chapter 4, we show how to improve FDFPN using policy and value neural networks.

5.2 Focused Proof Number Search with Deep Neural Networks

The move ordering function in FDFPN is only used to select a set of promising moves in the search tree: The strict order of the selected moves does not matter much since proof number search uses proof or disproof numbers to choose a most proving node (MPN). Trained policy networks can provide reliable move selection by mimicking expert play [125]; therefore, our first modification of FDFPN uses such a policy network to replace resistance as the move ordering function.

The next question is how many promising moves should be selected. A widening factor that is too large leads to decreased performance, while a factor that is too small increases the likelihood of initially missing an existing winning move. Ideally, the widening size should also be a function of value of the expanding state, which can be estimated by a value network. Hence our second modification is this: whenever expanding a state s, pick the widening size of s by the information provided by a value net $v_\theta(s)$. Specifically, for state s, suppose $v_\theta(s) = -1$ indicates a loss and $+1$ a win, the revised formula is as follows:

$$l(s) = \mathit{base} + \lceil f(s) \times |\mathit{live\ \ children}| \rceil \tag{5.2}$$

where $f(s)$ is defined as

$$f(s) = \begin{cases} \min\{\mathit{factor}, 1 + v_\theta(s)\} & \text{if } v_\theta(s) < 0 \\ \mathit{factor} & \text{otherwise} \end{cases}. \tag{5.3}$$

As in Eq. (5.1), *factor* is a parameter. This minor revision causes only one difference: when $v_\theta(s)$ is close to -1, the smaller noisy estimation value will be

used as the widening factor. So the smaller the estimation, the fewer child nodes are selected. This may seem counter-intuitive, since if a state s has an optimal value of -1, then eventually all its children must be solved to prove that s is losing. However, a losing node corresponds to a winning move for its parent state. Proof number search prefers nodes with small branching factor, therefore giving a losing node a small widening size makes its parent favor this node, which is desirable. On the other side, if a state is losing, all child nodes must be solved, and it is better to solve them consecutively rather than letting the search jump around.

The algorithm for FDFPN with CNNs is sketched in Algorithm 4. In practice, the `TTStore` function may have collision (resulting in failure for future `TTLookup`) , a collision resolution as in [149] is adopted. The revision lies on line 12 where policy and value networks are called. Results are then used to set search window size and select promising moves.

5.3 Results on 8×8 Hex

5.3.1 Preparation of Data

This 8×8 board size dataset was used in Chapter 4.5. Here, we describe how it is produced. We generated the expert games by playing tournaments between Olympiad computer players in the Benzene project: Mohex 2.0 [96], MoHex 2011 [11] and Wolve [13]. To enrich this data, (1) we conducted tournaments with randomly selected time settings each move (5s to 10s); (2) we iterated all one or two stone openings; and (3) neither player used solver during play. We collected 62853 games in total. Each game includes win/loss game result. No swap rule and no resign are used. Extracting these games gives 6.5×10^5 distinct state-action (s, a) and state-value (s, z) pairs. A position s can appear in two or more games but be associated with different win/loss results: we thus use the average to label these states. Let A be the set of games that s have been played, and $r(s, g)$ is the playing result (either $+1$ or -1) for game g: we label s by $z = \frac{\sum_{g \in A} r(s,g)}{|A|}$. The dataset is split into 90%, 10% training and test sets.

5.3.2 Policy and Value Neural Networks

We use separate policy and value networks in our experimentation. We use a policy network with 5 hidden layers. After padding borders, the input has a size of $10 \times 10 \times 5$. The first layer uses 48 3×3 filters with stride of 1, and then applies ReLU.

Algorithm 4: Focused depth-first proof number search with external policy function p_σ and value function v_θ.

```
   Input: A board position
   Result: +1 or -1
 1 Procedure FDFPN-CNN(root)
 2     root.φ, root.δ ← ∞, ∞
 3     MID(root)
 4     if root.φ = 0 then return +1
 5     else return −1
 6 Function MID(s)
 7     TTLookup(s)
 8     if node is terminal then
 9         Evaluate(s)
10         TTStore(s)
11         return
12     Compute child limit l(s) by v_θ, p_σ
13     while s.φ > δMin(s, l) and s.δ > φSum(s, l) do
14         child, δ2 ← SelectChild(s, l)
15         child.φ ← s.δ − φSum(s, l) + child.φ
16         child.δ ← min{s.φ, δ2 + 1}
17         MID(child)
18         if child.φ = 0 then
19             Prune child
20             Select a new child if necessary
21         end
22     end
23     s.φ ← δMin(s, l)
24     s.δ ← φSum(s, l)
25     TTStore(s)
26 Function SelectChild(s, l)
27     δ_best ← δ2 ← ∞
28     c_best ← nil
29     foreach c ∈ l(s) do
30         TTLookup(c)
31         if c.δ < δ_best then
32             c_best ← c
33             δ2 ← δ_best
34             δ_best ← c.δ
35         end
36         else if c.δ < δ2 then
37             δ2 ← c.δ
38         end
39     end
40     return c_best, δ2
41 Function δMin(s, l)
42     minδ ← ∞
43     foreach c ∈ l(s) do
44         TTLookup(c)
45         minδ = min{minδ, c.δ}
46     end
47     return minδ
48 Function φSum(s, l)
49     sumφ ← 0
50     foreach c ∈ l(s) do
51         TTLookup(c)
52         sumφ = sumφ + c.φ
53     end
54     return sumφ
```

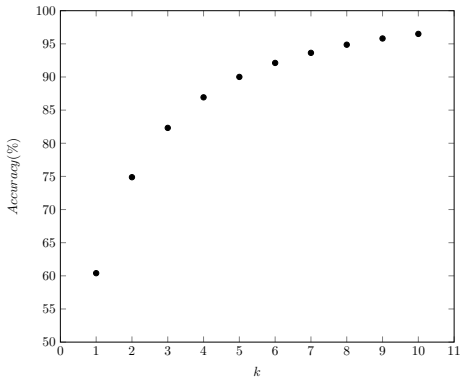


Figure 5.2: Top-k prediction accuracy of the policy network on 8×8 Hex.

The second layer zero pads an image into 10×10 and convolves using 3×3 filters with stride of 1, and then applies ReLU. Hidden layers 3 and 4 repeat the process of layer 2 . Hidden layer 5 convolves with 1×1 kernel size filters with 64 biases for each cell. The final layer is a softmax function.

We train the policy network to maximize the log-likelihood of move a at state s with $\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma}$. Using Adam optimizer [104] with a mini-batch of size 64, we train the network for 2×10^5 steps. The top one prediction accuracy on the test and training data are respectively 60.86% and 67%. Figure 5.2 contains the top-k prediction results. It shows that the top-10 accuracy exceeds 95%.

The first 6 layers of our value network are exactly the same as the policy network. After layer 6, we use one kernel size 1×1 filters with stride 1. We then use a layer with 48 fully-connected units. The final layer uses a *tanh* function to compress the output to $[-1, +1]$.

We train the value network to minimize the mean square error (MSE) between the predicted value $v_\theta(s)$ and label value z, *i.e.*, $\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial \theta}(z - v_\theta(s))$. Using Adam optimizer [104], with a mini-batch of 64, we train the value network for 1.6×10^5 steps. The obtained model achieves a MSE of 0.067 on the training set, and 0.083 on the test set.

5.3.3 Empirical Comparison of DFPN, FDFPN and FDFPN-CNN

As before, we built FDFPN-CNN upon the FDFPN [151] solver inside of Benzene. We use the serial version, accessible from `https://github.com/jakubpawlewicz/benzene/tree/jvc2`. After training, the network models are exported as static graph models for solely forward-inference in `C++`. Two parallel calls were used for

neural network inference while doing knowledge computation.

As suggested in [87], *base* is set to 1 and widening factor is set to 0.2 for FDFPN. We also prepared a DFPN that includes all child nodes to search at each node expansion. The DFPN is implemented on the same codebase by removing the focused behavior of FDFPN in Benzene. To see the effect of adding the neural network models in search, we use the same $factor = 0.2, base = 1$ for FDFPN-CNN.

We now compare the performance of DFPN, FDFPN and FDFPN-CNN on solving 8×8 Hex openings.

Table 5.1: DFPN, FDFPN and FDFPN-CNN results for 8×8 Hex. The best results are marked by boldface. Results were obtained on the same machine. Computation time was rounded to seconds.

Opening	DFPN		FDFPN		FDFPN-CNN	
	#node	time	#node	time	#node	time
a1	47383	240	30462	71	**12063**	**46**
a2	264718	489	104581	161	**61900**	**116**
a3	370973	1350	212140	486	**103940**	**275**
a4	1418942	4482	570207	1167	**217130**	**477**
a5	3929824	11128	1797393	3377	**1226009**	**2856**
a6	525308	1177	**272614**	**473**	295163	474
a7	3230008	10067	2874465	5496	**1058361**	**2615**
a8	1408664	4024	844403	1799	**572892**	**1300**
b1	49607	265	30317	73	**12490**	**43**
b2	204342	421	123728	140	**48074**	**82**
b3	89920	405	39683	115	**19077**	**82**
b4	541376	2003	270571	565	**186693**	**455**
b5	463360	1974	193799	526	**100489**	**368**
b6	563084	1808	497961	931	**223486**	**576**
b7	19182	53	12146	25	**6601**	**16**
b8	54590	201	15106	47	**12150**	**43**
c1	46926	226	28775	58	**10645**	**38**
c2	128112	287	**43899**	**74**	118384	133
c3	125365	521	56597	154	**25819**	**82**
c4	56951	281	18687	54	**11618**	**54**
c5	54388	271	28816	85	**17600**	**63**
c6	41018	205	30637	88	**10536**	**45**
c7	93673	278	60134	114	**42389**	**90**
c8	40562	183	**14157**	**40**	20327	63
d1	44295	192	14858	35	**10649**	**34**
d2	51451	147	89900	140	**16309**	**37**
d3	62776	269	24926	75	**12177**	**48**
d4	20368	94	13265	43	**8469**	**35**
d5	13821	85	6914	22	**4068**	**20**
d6	63014	345	93313	256	**13876**	**59**
d7	63574	192	79214	137	**40337**	**87**
d8	44634	198	15472	**37**	**14363**	40
SUM	14132209	43861	8509140	16864	**4534084**	**10752**

Table 5.1 shows the results of DFPN, FDFPN and FDFPN-CNN for solving all 32 8×8 Hex openings. Both FDFPN and FDFPN-CNN solved every opening faster than DFPN — the cumulative time of DFPN is respectively 2.6 and 4.1 times larger than those of FDFPN and FDFPN-CNN. This indicates the focused search has been beneficial.

FDFPN-CNN solved most opening positions faster than FDFPN. The largest improvement was on opening $a7$ where FDFPN-CNN used computation time 2614s, instead of 5496s for FDFPN. FDFPN-CNN solved all 32 openings with cumulative time 64% that of FDFPN. The total number of node expansions of DFPN, FDFPN and FDFPN-CNN are respectively 14132209, 8509140 and 4534084. Therefore, FDFPN-CNN expanded 53% nodes of FDFPN for all 32 openings. FDFPN-CNN did not perform better than FDFPN on every case: on openings $a6, c2, c8$, FDFPN-CNN's results are worse than those of FDFPN.

Effectiveness of the policy neural network

To further show the advantage of the policy net over the resistance evaluation, we conducted an ablation comparison between FDFPN and FDFPN-p_σ, a program that replaces the move ordering function with the policy net model p_σ. Using $base = 1$, we varied the widening factors 0.1 to 1.0 at an interval of 0.1.

Figure 5.3 shows the results with varying factor values, where FDFPN-*resistance* is the original FDFPN, FDFPN-p_σ uses the policy net. Both programs achieved best performance with widening factor of 0.2; however, FDFPN-p_σ used less time than FDFPN. This is true for all widening factors below 0.5. FDFPN-p_σ becomes slight worse than FDFPN when the widening factor is larger than 0.7. Perhaps that the policy net was not trained to rank all moves but to predict the *strong* moves played by expert players. Overall, FDFPN-p_σ is 17.3% faster than FDFPN with the best factor of 0.2. In terms of node expansion, it is 14.3% less than FDFPN.

The results in Figure 5.3 confirm that with small widening factors, which are desirable for focused search, the policy net is more reliable at selecting promising moves than the resistance evaluation function. Although FDFPN-CNN enabled faster solving, an exponential reduction in search space was not observed. This is because the neural network knowledge, unlike these from inferior cells, are essentially heuristic. That is, they can provide more accurate guidance to the search, but do not lead any exponential pruning in the state-space.

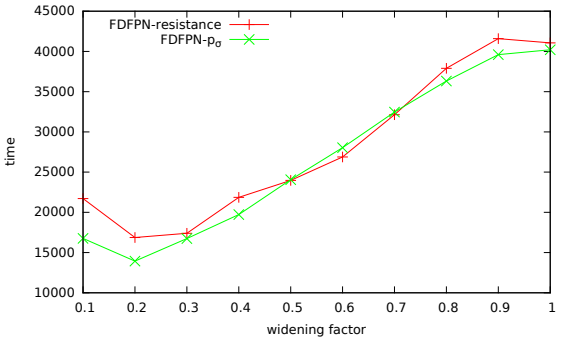


Figure 5.3: A comparison between the performance of FDPFN using *resistance* and policy network p_σ with varying *factor*.

5.3.4 Using Three-Head ResNet

Although the value network helped solving 8×8 Hex openings faster, research progress in Go [180, 184] showed that a single-head value net can exhibit overfitting while by forcing policy and value share one ResNet achieved better results. We conjecture that similar overfitting could happen in Hex when using a single-head value network, and using a three-head architecture may alleviate the problem. To verify this, we prepared a set of randomly generated 8×8 positions, and then use our solver to label the *true* value of these examples. We generated 58590 such positions in total. See Figure 5.4 for the training results. With the single-head value network, the training value error was quickly reduced to almost 0, so did the test error on examples split from the same game dataset for training, but test result on the randomly generated perfect examples implies that such value net has poor generalization. With three-head network of the same size, the generalization error was reduced. In the latter case, to improve generalization, a simple and effective approach is enlarging the network size — as shown in Figure 5.4 (right) after doubling the layers in three-head net, test error on randomly generated examples decreased.

So far there are no satisfying theoretical results on how deep neural networks generalize. It has been shown that deep network can fit randomly generated data just as well as natural images [220]. These deep models also have great memorization capacity [14]. Empirical evidence [98] suggests that using one deep network with auxiliary learning tasks often achieve good results. The three-head ResNet adds two auxiliary tasks (policy and action-value) for the state-value learning. Figures 5.4 shows that the auxiliary tasks indeed helped the neural net to achieve better generalization.

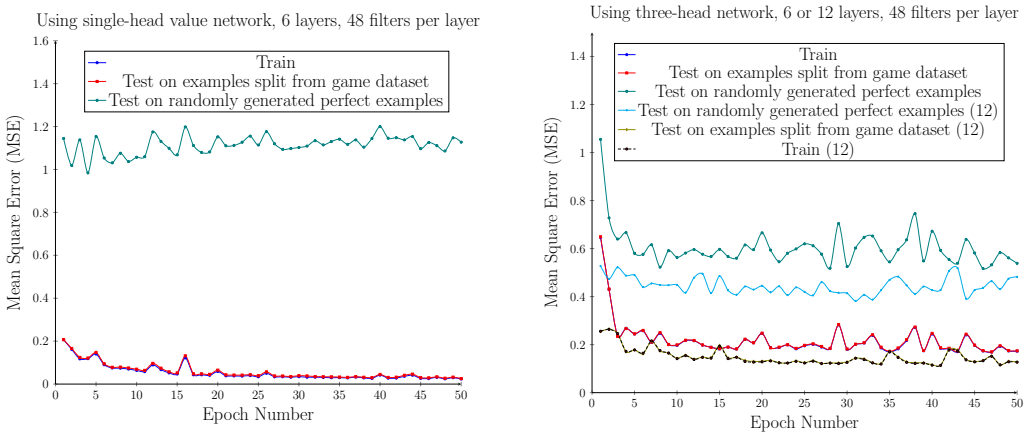


Figure 5.4: Results of value training from single versus three-head architectures. At each epoch, neural net model is saved and evaluated on each dataset. In the right figure, the cyan line is produced by doubling the neural network size to 12 layers. One epoch is a full sweep of the training data.

5.4 Solving by Strategy Decomposition

We saw in Chapter 2.1 that a fundamental difficulty in solving Hex is that strategy representation in a state-space graph can be overwhelmingly large, while decomposition-based representation can reduce the solution size. To further see this in Hex, consider the following example in Figure 5.5. Assume that we do not use H-search, and our goal is to find the solution for this Hex position. Further assume that we can access an oracle, who always provides a winning move for Black player if there is one. To solve this position by state-space search, we can expand the tree as follows:

- 1-ply: Enumerate all White moves.
- 2-ply: For each White move, use the oracle to obtain a Black winning move.
- 3-ply: For each frontier, enumerate all White moves.
- 4-ply: For each frontier node, use the oracle to obtain a Black winning move.
- 5-ply: For each frontier node, enumerate all White moves.
- 6-ply: For each frontier node, use the oracle to obtain a Black winning move.
- 7-ply: For each frontier node, enumerate all White moves.
- 8-ply: For each frontier node, use the oracle to find a Black winning chain.

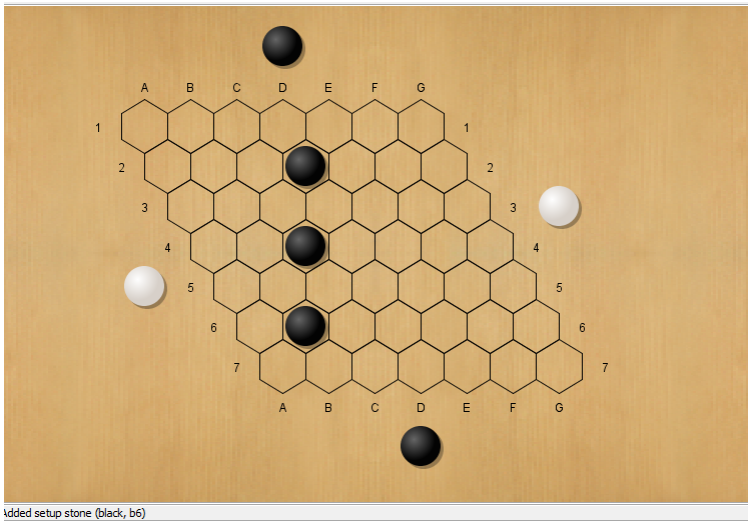


Figure 5.5: A Hex position; White is to play. The optimal value of this position is a Black win. Given an oracle machine, which can always predict the winning move for Black, searching for the solution represented by a solution-tree in the state-space graph requires to examine $44 \cdot 1 \cdot 42 \cdot 1 \cdot 40 \cdot 1 \cdot 38 \cdot 1 \approxeq 2.8$ million nodes

Since Black can surely make a winning-chain using 4 Black stones, the above process takes about 2.8 million steps ($44 \cdot 1 \cdot 42 \cdot 1 \cdot 40 \cdot 1 \cdot 38 \cdot 1$). So, in this way, even we have an error-free guessing, solving Hex positions as in Figure 5.5 quickly becomes intractable as the number of empty cells increases.

We saw in Chapter 4 that with algorithms similar to AlphaZero, formidable playing strength can be obtained. However, these players remain heuristic: there is no theoretical guarantee that their move prediction will converge to provably optimal play. By contrast, if we believe that an AlphaZero player has converged to optimal play, verifying in the state-space that it can indeed lead to a solution-graph could still take unbearably long time. This observation calls for state compression techniques.

Instead of representing the solution-object in state-space graph, Figure 5.6 shows how to decompose the solution to Figure 5.5 into a conjunction of 9 subproblems. The idea is to aggregate White moves and Black responses into 9 different categories. This means now what we are interested in is not a move predictor but a family of set function predictors. The argument for each function f_i is a set of White moves S_i. The output of f_i is a single Black move. The requirement is that all the arguments of these mappings in union shall be the set of all legal White moves. Denote any family of mappings satisfying this property a *decomposition*. A *decomposition prediction*

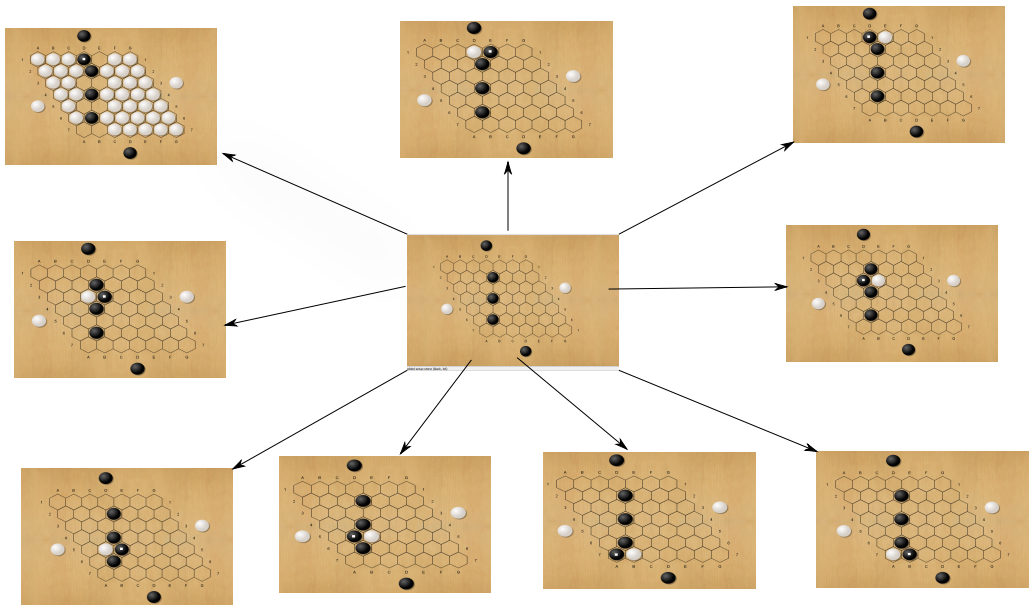


Figure 5.6: Decomposition-based solution to the Hex position in Figure 5.5. Solution to the original problem can be represented by a conjunction of 9 subproblems; each of them can be decomposed further using a similar scheme. Since it is known that Black can win using 4 black stones, the solution-graph found by such decomposition contains at most $9^4 = 6561$ nodes.

can be emulated by a move predictor. This idea is described in Algorithm 5, where α is a given move predictor — for example, suppose it has been obtained by running AlphaZero on the game of Hex. Its utility in Algorithm 5 is that, for any given position s and player to-play P, it returns a best move for P (or it gives a vector of scores for each move thus the best move can be selected). The idea is that for each P move, we use α to obtain the best $\bar{P}$ response. Then, we aggregate those P moves that share the same $\bar{P}$ response into a single group whose key is the corresponding $\bar{P}$ move. Now the problem boils down to finding a set of $\bar{P}$ keys whose groups in union shall cover all P moves in s.

We now describe a solving algorithm for searching a feasible decomposition-based solution-graph. The procedure is depicted in Algorithm 6. After predicting a decomposition, the `do` function is used to carry out the decomposition. That is, it tries to play a set of P moves T and response $\bar{P}$ move k; the requirement is that playing a set T of P moves must not overturn the game to a P win — again, this can be checked by referring to move predictor α. In the worst case, each T would contain a single P move, such that the solution-graph found by decomposition would

Algorithm 5: Decomposition prediction by move predictor α

```
1  Function Predict-Decomposition(s, P, α):
       // s: game position; P: player to play; α: move predictor
       /* For each move, record the best response.                    */
2      Let A be the set of moves available at s for P
3      d ← {}                                          /* a dictionary */
4      for a ∈ A do
5          s' ← play(s, a, P)                           /* play a at s */
6          x ← α(s', P̄)                    /* P̄ means opponent of P */
7          d[x] ← d[x] ∪ {a}
8      end
       /* Each value in d is a subset of P moves.                    */
9      U ← A
10     F ← {}                                          /* a dictionary */
11     A' ← sorted(d.keys())
12     for a ∈ A' do
13         U ← U\d[a]
14         F[a] ← d[a]
15         if U = ∅ then break
16     end
17     return F
18 End of Function
```

be exactly the same size as conventional state-space search. However, as Figure 5.6 indicates, in Hex, the size of space reduction can be huge as in many cases letting the losing player to play multiple moves simultaneously does not alter the game result. Formally, suppose in state-space, there is a solution-graph with size $\Omega(b^{d/2})$, using decomposition-based representation, it is guaranteed that a solution-graph of size $\Omega(c^{d/2})$ ($c \leq b$) exits. In Hex, c can be much smaller than b, as in Figure 5.6. In the literature, there have been similar work trying to establish proof for different games using strategy space search, e.g., [199, 213, 216]. The distinctive feature of our algorithm outlined here is that it does not rely on human expert knowledge, but uses a move predictor as a guide.

The unspecified utility functions in Algorithm 6 are explained as follows: `winner(s, player)` checks if s is a terminal position and returns its winner given *player* to play; `play(s, x, player)` plays cell x (or set of cells if x is a set) with color of *player* at position s; `value`(s'', P, α) uses α to query the estimated position value s'' with P to play. We do not empirically evaluate the merit of Algorithm 6 but give a proof on its correctness. We assume the Hex position to solve is noted as s; the

Algorithm 6: Automatic search for a decomposition-based solution

Input : s: Hex position to solve; P: player to play for s; α a move predictor
Result: If *true*, s is a $\bar{P}$ win, otherwise unknown

```
1  Function SolveByDecompose(s, P, α):
2      if winner(s, P) != Unknown then
           // For successful decomposition, winner must be P̄
3          return winner(s, P) = P̄
4      end
5      F ← Predict-Decomposition(s, P, α)
6      r ← true
7      for f ∈ F do
8          k, V ← key(f), values(f)
9          T ← do(s, k, P̄, V, P)    /* carryout one-step decomposition */
10         for s' ∈ T do
11             r ← r ∧ SolveByDecompose(s', P, α)
12         end
13         if r = false then return false
14     end
15     return true
16 End of Function
17 Function do(s, k, P̄, V, P, α):
18     T ← ∅
19     s' ← play(s, k, P̄)
20     X ← ∅
21     for v ∈ V do
22         s'' ← play(s', v, P)
23         if value(s'', P, α) = P̄ then
24             X ← X ∪ {v}
25         end
26         else
27             t ← play(s', X, P)
28             T ← T ∪ {t}
29             X ← ∅
30         end
31     end
32     t ← play(s', X, P)
33     T ← T ∪ {t}
34     return T
35 End of Function
```

player to play is P, and there is a winning strategy for $\bar{P}$: the opponent of P.

Definition 1. *Given a Hex position s, suppose the set of empty cells is noted as $M(s)$, a* decomposition branch *f is defined as a set function: $2^{M(s)} \to M(s)$.*

Definition 2. *For s, a* decomposition *is thus a collection mappings $f_1, f_2, \ldots, f_n$, where $\cup_i^n argument(f_i) = M(s)$. That is, a* decomposition *is a feasible solution to a set-covering problem where the union set is $M(s)$, and the argument of each f_i represents a subset. A* decomposition *splits the second-player winning strategy into a set of Hex positions where each position is reached by referring to f: fill the cells of $argument(f_i)$ with P stones and $output(f_i)$ using $\bar{P}$ stones.*

Observation 2. *If* `SolveByDecompose` *in Algorithm 6 returns true, the solution-graph that it identifies sufficiently represents a second playing winning strategy for Hex position s.*

Proof. The only condition that `SolveByDecompose` in Algorithm 6 can return true is that every node by a decomposition branch is true, *i.e.*, the solution-graph is a pure AND graph. Since we assume the terminal checking function `winner(s, player)` is correct, it follows that all nodes in the solution graph should be correctly true, and thus the root node is correctly true. Since we have guaranteed in Algorithm 5 that a *decomposition* has covered all P moves in the given Hex position, the identified solution-graph must be a feasible solution to the root node. □

Algorithm 5 is critical to the success for Algorithm 6. While for Algorithm 5, α is critical. If α is an oracle, Algorithm 5 always identifies a correct decomposition. In practice, α can be approximated by training a player before solving by decomposition, for example, by AlphaZero. It is worth noting that the `for` loop in line 4 of Algorithm 5 is parallel. In practice, this property can be used for faster computation of a decomposition. As the developments of super strong players with deep neural networks scale well with board sizes (given sufficient computation resource), searching for a decomposition represented solution using strong player as guidance might be a promising direction for solving large board size Hex positions.

Chapter 6

Conclusions and Future Work

This thesis gives a number of contributions to the computational and algorithmic approaches to Hex. New search and learning algorithms which uses the advancement in deep neural networks have been developed and applied to the game. State-of-the-art solving and playing programs have been created due to these algorithmic innovations.

Many challenges remain. This research sheds light on some important future working directions on the game of Hex. The transformation brought by AlphaGo and its successors has led some AI researchers to opine that future research in two-player alternate-turn zero-sum perfect-information games is inconsequential [39]. However, AlphaGo [180], AlphaGo Zero [184] and AlphaZero [182] are essentially heuristic algorithms without theoretical guarantee on success. They contain a number of important hyperparameters and how they affect the overall performance is only meagerly understood. Public re-implementations such as Leela Zero [148] and OpenELF Go [201] are towards the investigation of this phenomenon.

One fundamental difference that distinguishes Hex from other classic board games is its conspicuous mathematical structure [80], which has enabled much research on Hex to be presented in an exact rather than heuristic manner. Some examples are the proof that there is no draw [137], the identification of dead, dominated and inferior cells [81, 87], and the computation of connection strategies [7]. The accumulation of these mathematical knowledge, combined with sophisticated search, has allowed computer programs to solve Hex openings on board sizes up to 10×10, whose state spaces are already far beyond the limit of any simple brutal-force search. For example, the number of states for 9×9 and 10×10 Hex are respectively 10^{37} and 10^{46}. See Table 2.1.

Table 6.1: Status of solved Hex board sizes. For 10×10, only 2 openings are solved. For other smaller board sizes, *all* openings have been solved.

Board size	status	year	method	computation time
6×6	solved, all	2000	DFS [204]	seconds
7×7	solved, all	2003	DFS [82]	minutes
8×8	solved, all	2008	DFS [89]	hours
9×9	solved, all	2013	parallel FDFPN [149]	months
10×10	solved, 2	2013	parallel FDFPN [149]	months

However, *mathematical knowledge* accumulation becomes more difficult as it requires to invoke more complicated reasoning. On the other side, algorithms that learn with deep neural networks have shown great capacity of acquiring *heuristic knowledge* from data. These two kinds of knowledge are different and arguably complementary: given the seemingly intractable problem, the mathematical knowledge states what we can *at least* identify, while the later represents what we can be guessed *at most* after seeing a number of noised observations. Both have their merit and limitation. For example, the continual identification of inferior cells [28, 82, 87, 89] and the development of H-search [7, 88, 151] since the 2000s have led to feasible computer solutions for board sizes from 6×6 to 10×10. See Table 6.1 for a summarization. However, it is unlikely that 11×11 Hex can be solved if no overwhelmingly larger amount of pruning due to inferior cells analysis or H-search is introduced, i.e., although the pruning they brought is often exponential, the state-space complexity grows at a faster rate [1]. Although can be quite accurate in practice, the predictions from well-trained deep neural networks do not enable any instant pruning except for leading to a preferential search. Yet, guessing guided look-ahead search in a state-space graph faces another challenge: the solution graph itself could be intractably large which implies that even an error-free guessing technique is employed, verification could still be infeasible. This observation highlights the use of state-abstraction method in informed guess-based forward search. In Hex, strategy decomposition is such a method. Given that advancement in machine learning has enabled more accurate heuristic guidance, together with existing exact knowledge computation techniques, we conjecture that a promising future direction for solving Hex is to search decomposition-based solutions. This is arguably how humans solve

[1] For a state-space of b^d, the effect of these pruning is a constant reduction of b and d, but as board size increases, both b and d increase

Hex positions [214].

In summary, Hex is a game that has interested mathematicians and computer scientists since its invention; its graph-theoretical, combinatorial, game theoretic, and artificial intelligence aspects are perpetual incentives to attract future research. Despite grand successes, deep learning techniques have been questioned by the lack of reasoning [50]; as a domain where reasoning is ubiquitous and of utmost importance, Hex could be a valuable domain for pushing machine learning research to incorporate reasoning techniques.

References

[1] Selim G Akl and Monroe M Newborn. "The principal continuation and the killer heuristic." In: *Proceedings of the 1977 annual conference*. ACM. 1977, pp. 466–473. 25

[2] L Victor Allis, Maarten van der Meulen, and H Jaap Van Den Herik. "Proof-number search." In: *Artificial Intelligence* 66.1 (1994), pp. 91–124. 24, 138, 141

[3] L. Victor Allis, H. Jaap van den Herik, and M. P. H. Huntjens. "GoMoku Solved by New Search Techniques." In: *Computational Intelligence* 12 (1996), pp. 7–23. 5

[4] LV Allis. "Searching for solutions in games and artificial intelligence." PhD thesis. Universiteit Maastricht, 1994. 3, 26, 139

[5] Vadim V Anshelevich. "Hexy wins Hex tournament." In: *ICGA Journal* 23.3 (2000), pp. 181–184. 37, 41

[6] Vadim V Anshelevich. "The game of Hex: An automatic theorem proving approach to game programming." In: *AAAI/IAAI*. 2000, pp. 189–194. 37, 41

[7] Vadim V Anshelevich. "A hierarchical approach to computer Hex." In: *Artificial Intelligence* 134.1 (2002), pp. 101–120. 37, 38, 41, 113, 114

[8] Thomas Anthony, Zheng Tian, and David Barber. "Thinking Fast and Slow with Deep Learning and Tree Search." In: *arXiv preprint arXiv:1705.08439* (2017). 73

[9] Thomas Anthony et al. "Policy gradient search: Online planning and expert iteration without search trees." In: *arXiv preprint arXiv:1904.03646* (2019). 94, 95

[10] David L Applegate et al. *The traveling salesman problem: a computational study*. Princeton university press, 2006. 5

[11] Broderick Arneson, Ryan B Hayward, and Philip Henderson. "Monte Carlo tree search in Hex." In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 251–258. 32, 41, 49, 52, 101

[12] Broderick Arneson, Ryan B Hayward, and Philip Henderson. "Solving Hex: beyond humans." In: *International Conference on Computers and Games*. Springer. 2010, pp. 1–10. 36

[13] Broderick Arneson, Ryan Hayward, and Philip Henderson. "Wolve wins Hex tournament." In: *ICGA Journal* 32.1 (2008), pp. 49–53. 41, 49, 101

[14] Devansh Arpit et al. "A closer look at memorization in deep networks." In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 233–242. 106

[15] Argimiro A Arratia-Quesada and Iain A Stewart. "Generalized Hex and logical characterizations of polynomial space." In: *Information processing letters* 63.3 (1997), pp. 147–152. 35

[16] A Bagchi and A Mahanti. "Admissible heuristic search in AND/OR graphs." In: *Theoretical Computer Science* 24.2 (1983), pp. 207–219. 136

[17] Don F Beal. "A generalised quiescence search algorithm." In: *Artificial Intelligence* 43.1 (1990), pp. 85–98. 25

[18] Anatole Beck, Michael N Bleicher, and Donald Warren Crowe. *Excursions Into Mathematics: Millennium Edn.* Universities Press, 2000. 36

[19] Michael Z Bell. "Why expert systems fail." In: *Journal of the Operational Research Society* 36.7 (1985), pp. 613–619. 6

[20] Richard Bellman. "A Markovian decision process." In: *Journal of Mathematics and Mechanics* (1957), pp. 679–684. 21, 28

[21] Yoshua Bengio. "Learning deep architectures for AI." In: *Foundations and trends in Machine Learning* 2.1 (2009), pp. 1–127. 30

[22] Yoshua Bengio, Aaron Courville, and Pascal Vincent. "Representation learning: A review and new perspectives." In: *IEEE transactions on pattern analysis and machine intelligence* 35.8 (2013), pp. 1798–1828. 30

[23] Claude Berge. "Some remarks about a Hex problem." In: *The Mathematical Gardner*. Springer, 1981, pp. 25–27. 35

[24] Elwyn R Berlekamp, John Horton Conway, and Richard K Guy. *Winning ways for your mathematical plays*. Vol. 3. AK Peters Natick, 2003. 2, 36

[25] Dimitri P Bertsekas. *Dynamic programming and optimal control*. Vol. 1. 2. Athena scientific Belmont, MA, 1995. 22

[26] Dimitri P Bertsekas. "Approximate policy iteration: A survey and some new methods." In: *Journal of Control Theory and Applications* 9.3 (2011), pp. 310–335. 73

[27] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. 1st. Athena Scientific, 1996. ISBN: 1886529108. 28, 31, 56, 58

[28] Yngvi Björnsson et al. "Dead cell analysis in Hex and the Shannon game." In: *Graph Theory in Paris*. Springer, 2006, pp. 45–59. 35, 114

[29] Blai Bonet and Héctor Geffner. "Planning as heuristic search." In: *Artificial Intelligence* 129.1-2 (2001), pp. 5–33. 5

[30] Édouard Bonnet, Florian Jamain, and Abdallah Saffidine. "On the complexity of connection games." In: *Theoretical Computer Science* 644 (2016). Recent Advances in Computer Games, pp. 2–28. 3

[31] Léon Bottou. "Online Algorithms and Stochastic Approximations." In: *Online Learning and Neural Networks*. Ed. by David Saad. revised, oct 2012. Cambridge, UK: Cambridge University Press, 1998. URL: `http://leon.bottou.org/papers/bottou-98x`. 31

[32] Bruno Bouzy and Tristan Cazenave. "Computer Go: an AI oriented survey." In: *Artificial Intelligence* 132.1 (2001), pp. 39–103. 26

[33] Dennis M Breuker. “Memory versus search in games.” PhD thesis. 1998. 26

[34] Dennis Michel Breuker, Joseph Willem Hubertus Marie Uiterwijk, and Hendrik Jacob Herik. *”The PN2-search algorithm”*. Universiteit Maastricht, Department of Computer Science, 1999. 26

[35] Cameron Browne. *Hex strategy - making the right connections.* Jan. 2000. ISBN: 978-1-56881-117-8. 23, 33

[36] Cameron B Browne et al. “A survey of Monte Carlo tree search methods.” In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43. 32

[37] Arthur E Bryson. “A gradient method for optimizing multi-stage allocation processes.” In: *Proc. Harvard Univ. Symposium on digital computers and their applications*. Vol. 72. 1961. 30

[38] Garikai Campbell. “On optimal play in the game of Hex.” In: *INTEGERS: Electronic Journal of Combinatorial Number Theory* 4.2 (2004), pp. 1–23. 34

[39] Murray Campbell. “Mastering board games.” In: *Science* 362.6419 (2018), pp. 1118–1118. 113

[40] Murray Campbell, A Joseph Hoane, and Feng-hsiung Hsu. “Deep Blue.” In: *Artificial intelligence* 134.1-2 (2002), pp. 57–83. 26, 40

[41] Edward C Capen, Robert V Clapp, William M Campbell, et al. “Competitive bidding in high-risk situations.” In: *Journal of petroleum technology* 23.06 (1971), pp. 641–653. 61

[42] Chin-Liang Chang and James R. Slagle. “An admissible and optimal algorithm for searching AND/OR graphs.” In: *Artificial Intelligence* 2.2 (1971), pp. 117–128. 135, 138

[43] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. “Parallel Monte-Carlo tree search.” In: *International Conference on Computers and Games*. Springer. 2008, pp. 60–71. 67

[44] Christopher Clark and Amos Storkey. “Training deep convolutional neural networks to play Go.” In: *International Conference on Machine Learning*. 2015, pp. 1766–1774. 33, 44–46

[45] Anne Condon. “On Algorithms for Simple Stochastic Games.” In: *Advances in computational complexity theory*. 1990, pp. 51–72. 28, 59

[46] Anne Condon. “The complexity of stochastic games.” In: *Information and Computation* 96.2 (1992), pp. 203–224. 28, 29

[47] Rémi Coulom. “Efficient selectivity and backup operators in Monte-Carlo tree search.” In: *International Conference on Computers and Games*. Springer. 2006, pp. 72–83. 32

[48] Rémi Coulom. “Computing Elo ratings of move patterns in the game of Go.” In: *Computer Games Workshop*. 2007. 53

[49] Joseph C Culberson and Jonathan Schaeffer. “Searching with pattern databases.” In: *Conference of the Canadian Society for Computational Studies of Intelligence*. Springer. 1996, pp. 402–416. 5

[50] Adnan Darwiche. "Human-level Intelligence or Animal-like Abilities?" In: *Commun. ACM* 61.10 (2018), pp. 56–67. ISSN: 0001-0782. 115

[51] Jia Deng et al. "Imagenet: A large-scale hierarchical image database." In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255. 30

[52] Christian Donninger. "Null move and deep search." In: *ICGA Journal* 16.3 (1993), pp. 137–143. 25

[53] James E Doran and Donald Michie. "Experiments with the graph traverser program." In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 294.1437 (1966), pp. 235–259. 4

[54] Stuart Dreyfus. "The numerical solution of variational problems." In: *Journal of Mathematical Analysis and Applications* 5.1 (1962), pp. 30–45. 30

[55] Stuart Dreyfus. "The computational solution of optimal control problems with time lag." In: *IEEE Transactions on Automatic Control* 18.4 (1973), pp. 383–385. 30

[56] Arpad E Elo. *The rating of chessplayers, past and present*. Arco Pub., 1978. 42

[57] Markus Enzenberger and Martin Müller. "A lock-free multithreaded Monte-Carlo tree search algorithm." In: *Advances in Computer Games*. Springer. 2009, pp. 14–20. 67

[58] Markus Enzenberger et al. "Fuego—an open-source framework for board games and Go engine based on Monte Carlo tree search." In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 259–270. 32, 33, 52, 81

[59] Jonathan St BT Evans and Keith E Stanovich. "Dual-process theories of higher cognition: Advancing the debate." In: *Perspectives on psychological science* 8.3 (2013), pp. 223–241. 73

[60] Shimon Even and Robert Endre Tarjan. "A combinatorial problem which is complete in polynomial space." In: *Journal of the ACM (JACM)* 23.4 (1976), pp. 710–719. 33, 35

[61] Kunihiko Fukushima. "Neural network model for a mechanism of pattern recognition unaffected by shift in position-Neocognitron." In: *IEICE Technical Report, A* 62.10 (1979), pp. 658–665. 30

[62] David Gale. "The game of Hex and the Brouwer fixed-point theorem." In: *The American Mathematical Monthly* 86.10 (1979), pp. 818–827. 2

[63] C. Gao, R. Hayward, and M. Müller. "Move Prediction using Deep Convolutional Neural Networks in Hex." In: *IEEE Transactions on Games* PP.99 (2017), pp. 1–1. 11, 44

[64] Chao Gao, Martin Mueller, and Ryan Hayward. "Adversarial policy gradient for alternating Markov games." In: *International Conference on Learning Representations*. 2018. 11, 44, 73

[65] Chao Gao, Martin Müller, and Ryan Hayward. "Focused Depth-first Proof Number Search using Convolutional Neural Networks for the Game of Hex." In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 2017, pp. 3668–3674. 11

[66] Chao Gao, Martin Müller, and Ryan Hayward. "Three-Head Neural Network Architecture for Monte Carlo Tree Search." In: *IJCAI*. 2018, pp. 3762–3768. 11, 66

[67] Chao Gao, Kei Takada, and Ryan Hayward. "Hex 2018: MoHex3HNN over DeepEzo." In: *ICGA JOURNAL* 41.1 (2019), pp. 39–42. 11, 66, 90

[68] Chao Gao et al. "An iterative pseudo-gap enumeration approach for the Multidimensional Multiple-choice Knapsack Problem." In: *European Journal of Operational Research* 260.1 (2017), pp. 1–11. 11

[69] Chao Gao et al. "A transferable neural network for Hex." In: *ICGA Journal* 40.3 (2018), pp. 224–233. 11, 66

[70] Chao Gao et al. "On Hard Exploration for Reinforcement Learning: a Case Study in Pommerman." In: *The 15th AAAI Conference on Artificial Intelligence and Interactive Digital EntertainmentT*. 2019. 11

[71] Martin Gardner. "Mathematical Games: Concerning the game of Hex, which may be played on the tiles of the bathroom floor." In: *Scientific American* 197.1 (1957), pp. 145–150. 2

[72] Martin Gardner. "The Scientific American book of mathematical puzzles and diversions." In: (1959). 36

[73] Sylvain Gelly and David Silver. "Combining online and offline knowledge in UCT." In: *Proceedings of the 24th international conference on Machine learning*. ACM. 2007, pp. 273–280. 32, 42

[74] Sylvain Gelly and David Silver. "Monte-Carlo tree search and rapid action value estimation in computer Go." In: *Artificial Intelligence* 175.11 (2011), pp. 1856–1875. 32, 52

[75] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. 9, 72

[76] Eric A Hansen and Shlomo Zilberstein. "LAO*: A heuristic search algorithm that finds solutions with loops." In: *Artificial Intelligence* 129.1-2 (2001), pp. 35–62. 26

[77] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths." In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. 4, 136

[78] Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat. *Building Expert Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-10686-8. 6

[79] Ryan Hayward. "Six wins Hex tournament." In: *ICGA Journal* 29.3 (2006), pp. 163–165. 41

[80] Ryan B Hayward and Bjarne Toft. *Hex: The Full Story*. CRC Press, 2019. 2, 113

[81] Ryan B Hayward and Jack Van Rijswijck. "Hex and combinatorics." In: *Discrete Mathematics* 306.19-20 (2006), pp. 2515–2528. 35, 113

[82] Ryan Hayward et al. "Solving 7x7 Hex: Virtual connections and game-state reduction." In: *Advances in Computer Games*. Springer, 2004, pp. 261–278. 5, 42, 98, 114

[83] Ryan Hayward et al. "Mohex wins 2015 Hex 11× 11 and Hex 13× 13 tournaments." In: *ICGA Journal* 39.1 (2017), pp. 60–64. 42

[84] Kaiming He et al. "Deep residual learning for image recognition." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778. 31, 44, 55, 78

[85] Kaiming He et al. "Identity mappings in deep residual networks." In: *European Conference on Computer Vision*. Springer. 2016, pp. 630–645. 55, 78

[86] Piet Hein. "Vil de laere Polygon." In: *Article in Politiken newspaper* 26 (1942). 2

[87] Philip Thomas Henderson. "Playing and solving the game of Hex." PhD thesis. University of Alberta, 2010. 35, 36, 38–42, 51, 63, 98

[88] Philip Henderson, Broderick Arneson, and Ryan B Hayward. "Hex, braids, the crossing rule, and XH-search." In: *Advances in Computer Games*. Springer. 2009, pp. 88–98. 114

[89] Philip Henderson, Broderick Arneson, and Ryan B Hayward. "Solving 8x8 Hex." In: *Proc. IJCAI*. Vol. 9. Citeseer. 2009, pp. 505–510. 42, 86, 98, 114

[90] Philip Henderson and Ryan B Hayward. "Captured-reversible moves and star decomposition domination in Hex." In: *Integers: Annual* 2013 (2014), p. 75. 36

[91] Philip Henderson and Ryan B Hayward. "A handicap strategy for Hex." In: *Games of No Chance 4* 63 (2015), p. 129. 36

[92] H Jaap van den Herik and Mark HM Winands. "Proof-number search and its variants." In: *Oppositional Concepts in Computational Intelligence*. Springer, 2008, pp. 91–118. 139

[93] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks." In: *Neural networks* 4.2 (1991), pp. 251–257. 31

[94] Ronald A Howard. "DYNAMIC PROGRAMMING AND MARKOV PROCESSES." In: (1960). 28, 29

[95] Gao Huang et al. "Deep networks with stochastic depth." In: *European Conference on Computer Vision*. Springer. 2016, pp. 646–661. 55

[96] Shih-Chieh Huang et al. "MoHex 2.0: a pattern-based MCTS Hex player." In: *International Conference on Computers and Games*. Springer. 2013, pp. 60–71. 32, 42, 47, 53, 81, 87, 10

[97] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In: *International Conference on Machine Learning*. 2015, pp. 448–456. 55, 78

[98] Max Jaderberg et al. "Reinforcement learning with unsupervised auxiliary tasks." In: *arXiv preprint* (2016). 106

[99] Tommy R Jensen and Bjarne Toft. *Graph coloring problems*. Vol. 39. John Wiley & Sons, 2011. 35

[100] Norman P. Jouppi and Yang et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. 33, 56

[101] Sham M Kakade. "A natural policy gradient." In: *Advances in neural information processing systems*. 2002, pp. 1531–1538. 58

[102] Takada Kei. "A Study on Learning Algorithms of Value and Policy Functions in Hex." PhD thesis. Hokkaido University, 2019. 90

[103] Henry J Kelley. "Gradient theory of optimal flight paths." In: *Ars Journal* 30.10 (1960), pp. 947–954. 30

[104] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." In: *International Conference on Learning Representations*. 2014. 48, 62, 78, 103

[105] Akihiro Kishimoto. "Dealing with Infinite Loops, Underestimation, and Overestimation of Depth-first Proof-number Search." In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI Press, 2010, pp. 108–113. 26, 141

[106] Akihiro Kishimoto and Radu Marinescu. "Recursive Best-First AND/OR Search for Optimization in Graphical Models." In: *UAI*. 2014, pp. 400–409. 141

[107] Akihiro Kishimoto and Martin Müller. "About the completeness of depth-first proof-number search." In: *International Conference on Computers and Games*. Springer. 2008, pp. 146–156. 26

[108] Akihiro Kishimoto et al. "Game-tree search using proof numbers: The first twenty years." In: *ICGA Journal* 35.3 (2012), pp. 131–156. 139, 141

[109] Donald E Knuth and Ronald W Moore. "An analysis of alpha-beta pruning." In: *Artificial intelligence* 6.4 (1975), pp. 293–326. 8, 25

[110] Jens Kober, J Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey." In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274. 56

[111] Levente Kocsis and Csaba Szepesvári. "Bandit based Monte-Carlo planning." In: *European conference on machine learning*. Springer. 2006, pp. 282–293. 32

[112] Richard E Korf. "Finding optimal solutions to Rubik's cube using pattern databases." In: *AAAI/IAAI*. 1997, pp. 700–705. 5, 9

[113] Richard E Korf and David Maxwell Chickering. "Best-first minimax search." In: *Artificial intelligence* 84.1-2 (1996), pp. 299–337. 25

[114] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems*. 2012, pp. 1097–1105. 30, 44, 100

[115] Li-Cheng Lan et al. "Multiple Policy Value Monte Carlo Tree Search." In: *IJCAI*. 2019. 96

[116] Ailsa H. Land and Alison G. Doig. "An Automatic Method for Solving Discrete Programming Problems." In: *50 Years of Integer Programming*. 2010. 5

[117] Tor Lattimore and Csaba Szepesvári. "Bandit algorithms." In: *preprint* (2018). 23

[118] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444. 30

[119] Yann LeCun et al. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. 30, 100

[120] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning." In: *ICLR*. 2016. 32, 56

[121] Long-H Lin. "Self-improving reactive agents based on reinforcement learning, planning and teaching." In: *Machine learning* 8.3/4 (1992), pp. 69–97. 56

[122] Seppo Linnainmaa. "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors." In: *Master's Thesis (in Finnish), Univ. Helsinki* (1970), pp. 6–7. 30

[123] Michael Lederman Littman. "Algorithms for sequential decision making." PhD thesis. Brown University Providence, RI, 1996. 19, 22, 23, 28, 29

[124] *Maciej Celuch*. `https://www.hexwiki.net/index.php/Maciej_Celuch`. Accessed: 2019-08-24. 89, 90

[125] Chris J Maddison et al. "Move evaluation in Go using deep convolutional neural networks." In: *International Conference on Learning Representations*. 2015. 33, 44–46, 49, 52, 100

[126] Alberto Martelli and Ugo Montanari. "Additive AND/OR Graphs." In: *IJCAI*. Vol. 73. 1973, pp. 1–11. 136

[127] Alberto Martelli and Ugo Montanari. "Optimizing decision trees through heuristically guided search." In: *Communications of the ACM* 21.12 (1978), pp. 1025–1039. 136

[128] Abadi et al. Martin. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `http://tensorflow.org/`. 48, 61

[129] David Allen McAllester. "Conspiracy numbers for min-max search." In: *Artificial Intelligence* 35.3 (1988), pp. 287–310. 26

[130] Drew McDermott et al. "PDDL-the planning domain definition language." In: (1998). 5

[131] Thomas M. Mitchell. *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997. 6

[132] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (2015), pp. 529–533. 32, 51, 56

[133] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning." In: *International Conference on Machine Learning*. 2016, pp. 1928–1937. 32, 56

[134] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018. 6

[135] Martin Müller. "Computer Go." In: *Artificial Intelligence* 134.1-2 (2002), pp. 145–179. 25

[136] Ayumu Nagai. "Df-pn algorithm for searching AND/OR trees and its applications." PhD thesis. PhD thesis, Department of Information Science, University of Tokyo, 2002. 26, 141

[137] John F Nash. *Some games and machines for playing them*. Tech. rep. Rand Corporation, 1952. 2, 113

[138] Dana S Nau. "Pathology on Game Trees: A Summary of Results." In: *AAAI*. 1980, pp. 102–104. 26

[139] Dana S Nau. "An investigation of the causes of pathology in games." In: *Artificial Intelligence* 19.3 (1982), pp. 257–278. 26

[140] Dana S Nau. "Pathology on game trees revisited, and an alternative to minimaxing." In: *Artificial intelligence* 21.1-2 (1983), pp. 221–244. 26

[141] Allen Newell, John C Shaw, and Herbert A Simon. "Report on a general problem solving program." In: *IFIP congress*. Vol. 256. Pittsburgh, PA. 1959, p. 64. 4, 20, 21

[142] Nils J Nilsson. *Problem-Solving in Artificial Intelligence*. McGraw-Hill, 1971. 135

[143] Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 1980. 5, 6, 136

[144] Kyoung-Su Oh and Keechul Jung. "GPU implementation of neural networks." In: *Pattern Recognition* 37.6 (2004), pp. 1311–1314. 30

[145] Manfred Padberg and Giovanni Rinaldi. "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems." In: *SIAM review* 33.1 (1991), pp. 60–100. 5

[146] Sinno Jialin Pan and Qiang Yang. "A survey on transfer learning." In: *IEEE Transactions on knowledge and data engineering* 22.10 (2010), pp. 1345–1359. 32

[147] Gian-Carlo Pascutto. *[Computer-go] Zero performance*. `http://computer-go.org/pipermail/computer-go/2017-October/010307.html`. Accessed: 2019-08-18. 71

[148] Gian-Carlo Pascutto. *Leela Zero*. `https://github.com/leela-zero/leela-zero`. Accessed: 2019-08-18. 72, 113

[149] Jakub Pawlewicz and Ryan B Hayward. "Scalable parallel DFPN search." In: *International Conference on Computers and Games*. Springer. 2013, pp. 138–150. 18, 42, 101, 114

[150] Jakub Pawlewicz and Ryan B Hayward. "Sibling conspiracy number search." In: *Eighth Annual Symposium on Combinatorial Search*. 2015. 42

[151] Jakub Pawlewicz et al. "Stronger Virtual Connections in Hex." In: *IEEE Transactions on Computational Intelligence and AI in Games* 7.2 (2015), pp. 156–166. 38, 39, 42, 53, 63, 81, 86

[152] Judea Pearl. "On the nature of pathology in game searching." In: *Artificial Intelligence* 20.4 (1983), pp. 427–453. 26

[153] Judea Pearl. "Heuristics: intelligent search strategies for computer problem solving." In: (1984). 4, 5, 9, 18–20, 24, 98, 13

[154] Gabriel Pereyra et al. "Regularizing neural networks by penalizing confident output distributions." In: *ICLR 2017 workshop* (2017). 55

[155] Jan Peters and J Andrew Bagnell. "Policy gradient methods." In: *Encyclopedia of Machine Learning*. Springer, 2011, pp. 774–776. 29, 56

[156] Aske Plaat et al. "Best-first fixed-depth minimax algorithms." In: *Artificial Intelligence* 87.1-2 (1996), pp. 255–293. 25

[157] *Progress towards the OpenAI mission*. https://www.slideshare.net/AIFrontiers/ilya-sutskever-at-ai-frontiers-progress-towards-the-openai-mission. Accessed: 2019-08-19. 71

[158] Martin L Puterman. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014. 21, 26, 27

[159] *Rating Inflation - Its Causes and Possible Cures*. https://en.wikipedia.org/wiki/ChessBase. Accessed: 2019-08-11. 42

[160] A Ravindran, Don T Phillips, and James J Solberg. "Operations research: principles and practice." In: (1987). 5

[161] Ali Sharif Razavian et al. "CNN features off-the-shelf: an astounding baseline for recognition." In: *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*. IEEE. 2014, pp. 512–519. 32

[162] Stefan Reisch. "Hex ist PSPACE-vollständig." In: *Acta Informatica* 15.2 (1981), pp. 167–191. 3, 33

[163] JA Robinson. "An overview of mechanical theorem proving." In: *Theoretical Approaches to Non-Numerical Problem Solving*. Springer, 1970, pp. 2–20. 38

[164] Igor Roizen and Judea Pearl. "A minimax algorithm better than alpha-beta? Yes and no." In: *Artificial Intelligence* 21.1-2 (1983), pp. 199–220. 25

[165] Christopher D Rosin. "Multi-armed bandits with episode context." In: *Annals of Mathematics and Artificial Intelligence* 61.3 (2011), pp. 203–230. 53, 67

[166] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. "Learning representations by back-propagating errors." In: *Cognitive modeling* 5.3 (1988), p. 1. 30

[167] Sartaj Sahni. "Computationally related problems." In: *SIAM Journal on Computing* 3.4 (1974), pp. 262–279. 133, 136

[168] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers." In: *IBM J. Res. Dev.* 3.3 (July 1959), pp. 210–229. ISSN: 0018-8646. DOI: 10.1147/rd.33.0210. URL: http://dx.doi.org/10.1147/rd.33.0210. 8

[169] Arthur L Samuel. "Some studies in machine learning using the game of checkers. II—Recent progress." In: *IBM Journal of research and development* 11.6 (1967), pp. 601–617. 8

[170] Jonathan Schaeffer. "The history heuristic." In: *ICGA Journal* 6.3 (1983), pp. 16–19. 25

[171] Jonathan Schaeffer et al. "A world championship caliber checkers program." In: *Artificial Intelligence* 53.2-3 (1992), pp. 273–289. 26

[172] Jonathan Schaeffer et al. "Checkers is solved." In: *Science* 317.5844 (2007), pp. 1518–1522. 5

[173] Tom Schaul et al. "Prioritized experience replay." In: *ICLR*. 2016. 32, 56

[174] Martin Schijf, L Victor Allis, and Jos WHM Uiterwijk. "Proof-number search and transpositions." In: *ICGA Journal* 17.2 (1994), pp. 63–74. 141

[175] Jürgen Schmidhuber. "Deep learning in neural networks: An overview." In: *Neural networks* 61 (2015), pp. 85–117. 9, 30

[176] John Schulman et al. "Trust region policy optimization." In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015, pp. 1889–1897. 32, 56

[177] John Schulman et al. "Proximal policy optimization algorithms." In: *arXiv preprint arXiv:1707.06347* (2017). 32, 56

[178] Claude E Shannon. "Computers and automata." In: *Proceedings of the IRE* 41.10 (1953), pp. 1234–1241. 3, 40, 41

[179] Lloyd S Shapley. "Stochastic games." In: *Proceedings of the national academy of sciences* 39.10 (1953), pp. 1095–1100. 23

[180] David Silver et al. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587 (2016), pp. 484–489. 29, 32, 33, 44, 52, 53, 56

[181] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm." In: *arXiv preprint arXiv:1712.01815* (2017). 33, 70

[182] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play." In: *Science* 362.6419 (2018), pp. 1140–1144. 70, 113

[183] David Silver et al. "Deterministic policy gradient algorithms." In: *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*. 2014, pp. 387–395. 32, 56

[184] David Silver et al. "Mastering the game of Go without human knowledge." In: *Nature* 550.7676 (2017), pp. 354–359. 33, 65, 68, 73, 76, 79, 10

[185] Saurabh Singh, Derek Hoiem, and David Forsyth. "Swapout: Learning an ensemble of deep architectures." In: *Advances in Neural Information Processing Systems*. 2016, pp. 28–36. 55

[186] James R Slagle. "A heuristic program that solves symbolic integration problems in freshman calculus." In: *Journal of the ACM (JACM)* 10.4 (1963), pp. 507–520. 4

[187] James E Smith and Robert L Winkler. "The optimizer's curse: Skepticism and postdecision surprise in decision analysis." In: *Management Science* 52.3 (2006), pp. 311–322. 61

[188] Sylvain Sorin. *A first course on zero-sum repeated games*. Vol. 37. Springer Science & Business Media, 2002. 23

[189] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958. 55

[190] George C. Stockman. "A minimax algorithm better than alpha-beta?" In: *Artificial Intelligence* 12.2 (1979), pp. 179–196. 25

[191] Richard S Sutton. "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming." In: *Machine Learning Proceedings 1990*. Elsevier, 1990, pp. 216–224. 29

[192] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Second. Preliminary Draft, MIT press Cambridge, 2017. 9, 19, 29, 58

[193] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation." In: *Advances in neural information processing systems*. 2000, pp. 1057–1063. 29, 56, 58

[194] Christian Szegedy et al. "Rethinking the inception architecture for computer vision." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2818–2826. 55

[195] Kei Takada, Hiroyuki Iizuka, and Masahito Yamamoto. "Reinforcement Learning to Create Value and Policy Functions using Minimax Tree Search in Hex." In: *IEEE Transactions on Games* (2019). 90

[196] Kei Takada et al. "Developing computer hex using global and local evaluation based on board network characteristics." In: *Advances in Computer Games*. Springer. 2015, pp. 235–246. 42, 54

[197] Kei Takata, Hiroyuki Iizuka, and Masahito Yamaoto. "Computer Hex using Move Evaluation Method based on Convolutional Neural Network." In: *IJCAI 2017 Computer Games Workshop*. 2017. 54

[198] Gerald Tesauro. "Temporal difference learning and TD-Gammon." In: *Communications of the ACM* 38.3 (1995), pp. 58–68. 29, 31, 56, 57

[199] Thomas Thomsen. "Lambda-search in game trees—With application to Go." In: *International Conference on Computers and Games*. Springer. 2000, pp. 19–38. 110

[200] Yuandong Tian and Yan Zhu. "Better computer Go player with neural network and long-term prediction." In: *International Conference on Learning Representations*. 2015. 33, 44–46, 52

[201] Yuandong Tian et al. "ELF OpenGo: an analysis and open reimplementation of AlphaZero." In: *ICML*. 2019, pp. 6244–6253. 72, 89, 92, 113

[202] Jonathan Tompson et al. "Efficient object localization using convolutional networks." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 648–656. 55

[203] Toru Ueda et al. "Weak proof-number search." In: *International Conference on Computers and Games*. Springer. 2008, pp. 157–168. 141

[204] Jack Van Rijswijck. "Computer Hex: Are Bees Better Than Fruitflies?" MA thesis. University of Alberta, 2002. 3, 41, 114

[205] Jack Van Rijswijck. "Search and evaluation in Hex." In: *Master of science, University of Alberta* (2002). 3, 40

[206] Jack Van Rijswijck. "Set colouring games." PhD thesis. University of Alberta, 2006. 3, 34, 35

[207] Ziyu Wang et al. "Dueling network architectures for deep reinforcement learning." In: *ICML*. 2016. 56

[208] Ziyu Wang et al. "Sample efficient actor-critic with experience replay." In: *ICLR*. 2017. 32, 56, 58

[209] Christopher JCH Watkins and Peter Dayan. "Q-learning." In: *Machine learning* 8.3-4 (1992), pp. 279–292. 56

[210] Paul J Werbos. "Applications of advances in nonlinear sensitivity analysis." In: *System modeling and optimization*. Springer, 1982, pp. 762–770. 30

[211] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning." In: *Machine learning* 8.3-4 (1992), pp. 229–256. 29, 57, 58

[212] Mark HM Winands, Jos WHM Uiterwijk, and H Jaap van den Herik. "An effective two-level proof-number search algorithm." In: *Theoretical Computer Science* 313.3 (2004), pp. 511–525. 26

[213] I-Chen Wu and Ping-Hung Lin. "Relevance-zone-oriented proof search for connect6." In: *IEEE Transactions on computational intelligence and AI in games* 2.3 (2010), pp. 191–207. 110

[214] Jing Yang, Simon Liao, and Mirek Pawlak. "On a decomposition method for finding winning strategy in Hex game." In: *Proceedings ADCOG: Internat. Conf. Application and Development of Computer Games (ALW Sing, WH Man and W. Wai, eds.), City University of Honkong*. 2001, pp. 96–111. 4, 18, 115

[215] Jing Yang, Simon Liao, and Mirek Pawlak. "A New Solution for 7x7 Hex Game." In: *to appear* 106 (2002). 4, 18

[216] Kazuki Yoshizoe, Akihiro Kishimoto, and Martin Müller. "Lambda Depth-First Proof Number Search and Its Application to Go." In: *IJCAI*. 2007, pp. 2404–2409. 26, 110

[217] Jason Yosinski et al. "How transferable are features in deep neural networks?" In: *Advances in neural information processing systems*. 2014, pp. 3320–3328. 32

[218] Kenny Young, Gautham Vasan, and Ryan Hayward. "NeuroHex: A Deep Q-learning Hex Agent." In: *Computer Games*. Springer, 2016, pp. 3–18. 51, 57, 61

[219] Sergey Zagoruyko and Nikos Komodakis. "Wide residual networks." In: *arXiv preprint arXiv:1605.07146* (2016). 55

[220] Chiyuan Zhang et al. "Understanding deep learning requires rethinking generalization." In: *International Conference on Learning Representations*. 2017. 106

Appendix A

Additional Documentation

Extending `benzene` to contain neural networks results `neurobenze`, we document the key changes and how to use them in below. We have also developed software tool for visualizing the decomposition-based solution for solving $e5$ opening on 9×9 Hex. The pattern files are provided by Jing Yang.

A.1 Neurobenzene

Newly added commands:

- `nn_ls`: it lists all available neural net models in share/nn/.
- `nn_load`: it follows by a nn name to load.
- `param_nn`: it displays parameter setting for neural net.
- `param_mohex use_playout_constant`, default 0, indicating no playout is used (otherwise, setting it to 1.0 will use sole playout result for leaf evaluation).
- `param_mohex moveselect_ditherthreshold`, default 0, indicating the threshold for soft-selection if the number of stones on the board is less than the threshold.
- `mohex-self-play`, it follows by two arguments n (number of games) and $filetosave$ (location to save played games).
- `param_mohex root_dirichlet_prior`, it follows by a float number α, indicating the parameter for Dirichlet(α).
- `param_dfpn use_nn`, it follows by 0 or 1 indicating whether to use neural net in DFPN solver.

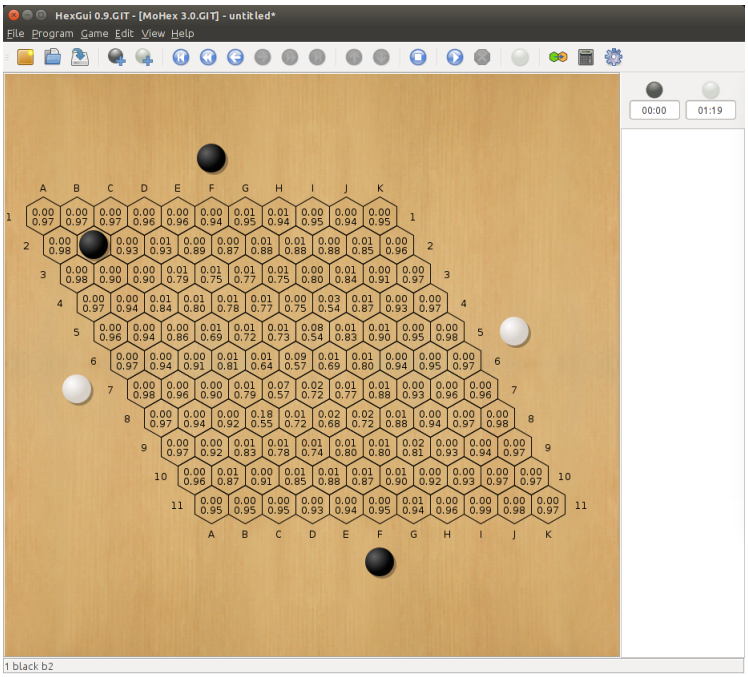


Figure A.1: An example shows the evaluation of actions provided by neural net model. For each cell, the upper number is the prior probability provided by policy head, the lower number is the value provided by action-value head. Note that here the value is with respect to the player to play after taking that action, thus a smaller value indicating higher preference for the current board state. Here, both policy and action-value are in favor of $f6$.

Newly added module:

- `simplefeature3/`: this directory contains a sub-project of using Python to train neural networks.
- `src/neuralnet/`: this directory contains Tensorflow inference written in Tensorflow C API.
- `closedloop/`: this directory contains all scripts for doing closed loop training inside of neurobenzene.

A.2 Visualization for a Human Identified 9×9 Solution

Starting from initial state, Jing Yang's handcrafted solution expresses how a board state should be decomposed after every opponent move. For 9×9 Hex, the solution provided by him contains around 700 patterns. We implemented an visualization tool for expressing the perfect solution, allowing user to play any moves and see the response move, how a larger pattern is decomposed into smaller ones.

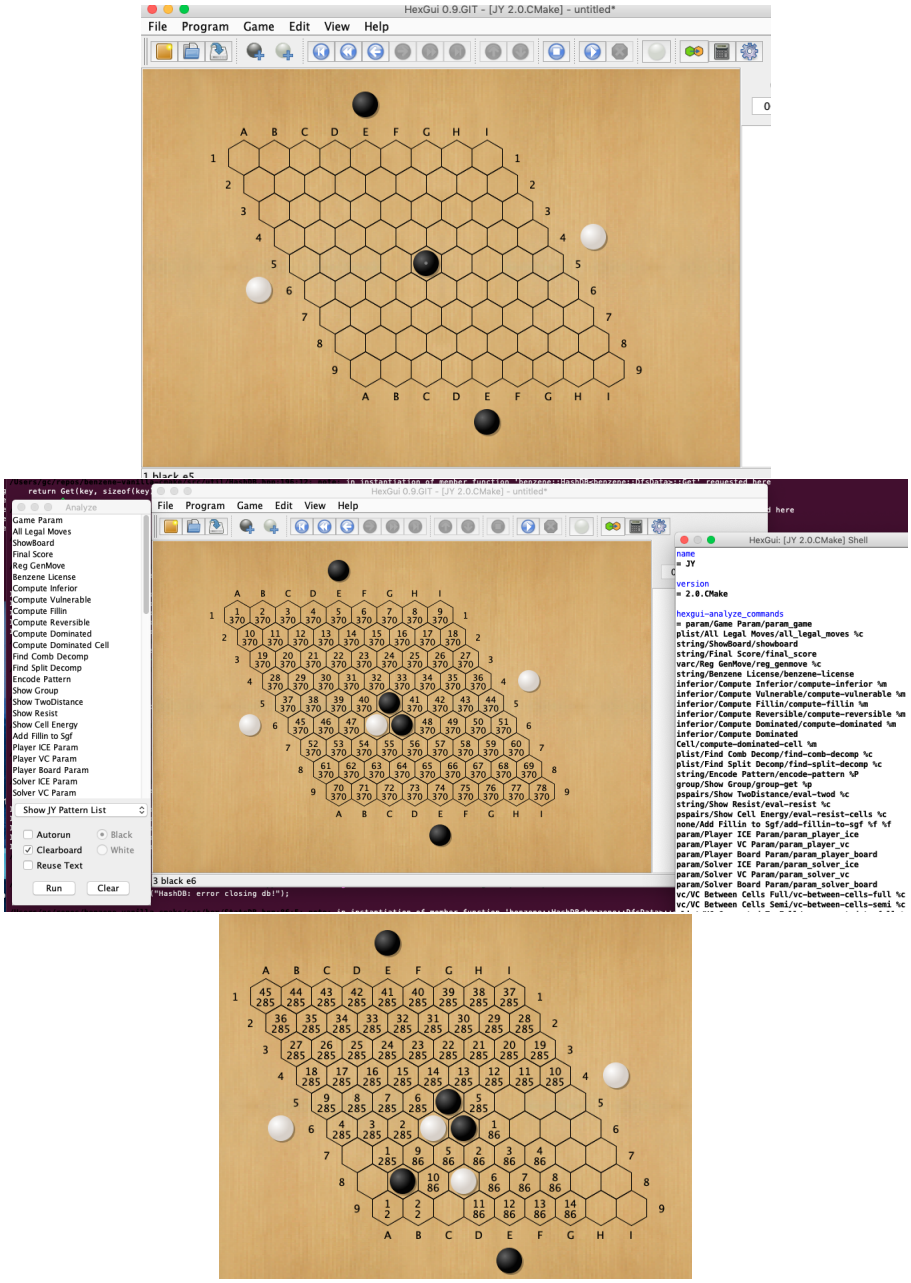


Figure A.2: An example play by White. If it plays at $D6$, Black will respond at $E6$ resulting to pattern 370. If White then plays at $D8$, then Black will respond at $B8$, splitting pattern 370 into patterns 285 and 86. In each cell, the lower number is pattern id, upper number is local move name inside of that pattern.

A.3 Published Dataset

Datasets for 8×8, 9×9 and 13×13 are available in `https://drive.google.com/drive/u/0/folders/18MdnvMItU7O2sEJDlbmk_ZzUhZG7yDK9`. These data were produced by players MoHex 2.0 and Wolve in moderately strong setting.

A.4 From GBFS to AO* and PNS

At each iteration, a clever search selects the *seemingly best* node for expansion. Such a feature is called *best-first*. Assuming that the problem to be solved is represented by a single start node s and the task is to find a solution graph with *minimum cost* defined by scheme Ψ, a General Best-first Search (GBFS) procedure for loop-free AND/OR graphs is depicted in Algorithm 7. The expansion is driven by the *small-is-quick* and *face-value* principles, i.e., it always selects the best (smallest) solution-base to explore based on estimated values of candidate nodes (so called face-value). Given that the heuristic estimation h is always optimistic, upon termination the returned solution graph must be optimal [153] even though in some cases we are merely interested in finding any solution rather than the optimum.

Algorithm 7: General Best-First Search for AND/OR graphs

Input: Start node s, selection function f_1, f_2, rules for generating successor nodes, evaluation function h, cost scheme Ψ
Output: Solution graph G_0^*

```
1  Function GBFS(s, f1, f2):
2      Let the explicit graph be G' = {s}
3      while true do
4          G0 ← f1(G') // G0 is solution-base
5          if G0 is solution graph then
6              G0* ← G0, then exit;
7          end
8          t ← f2(G0) // t is selected frontier node
9          Generate all successors of t, append to G'
10         for t' ∈ successor(t) do
11             Evaluate t' by h
12             UpdateAncestors(t', Ψ, G')
13         end
14     end
15     return G0*
16 Function UpdateAncestors(t', Ψ, G'):
17     Update ancestor nodes of t' in G' according to the definition of Ψ
```

Algorithm 7 contains abstract functions f_1, f_2 and h, which are respectively used for selecting the *solution-base* G_0 in G', a frontier node n and evaluating n. A solution-base is a subgraph of G' that may eventually be developed to a solution for the implicit graph G. f_1 selects the best solution-base from G'. The price made for generality is that Algorithm 7 leaves three important functions, f_1, f_2 and h , unspecified. While the content of h relies on domain-knowledge, implementations

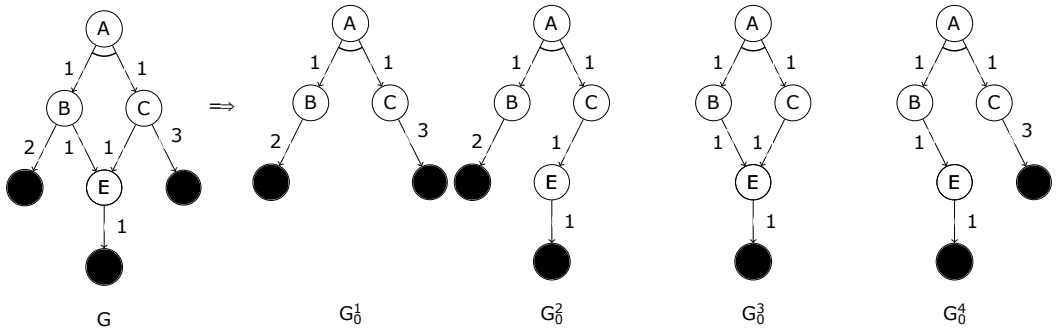


Figure A.3: Leftmost is the full graph G; all the tip nodes are solvable terminal with value 0; each edge has a positive cost. Four different solution graphs exist for G, among which G_0^3 is with the *minimum total cost* of 5. However, if the cost scheme Ψ is defined as a *recursive sum-cost*, both G_0^2 and G_0^3 are with *minimum cost of* 6.

of f_1 and f_2 depend on definition of the *cost scheme* Ψ. See Appendix A.4 for a full account on the impact of *cost scheme* for various instances of GBFS.

To see the impact of *cost scheme* in GBFS (as in Algorithm 7), consider a simple toy AND/OR graph shown in Figure A.3 where all tip nodes are solvable terminal assigned with value 0. In this problem, there are four solution graphs, noted as G_0^1, G_0^2, G_0^3 and G_0^4. Each edge has a cost either 1, 2 or 3. If the objective is to find a solution graph with the minimum summed edge cost, G_0^3 will be is the optimal. We then consider how the cost for each candidate solution is computed. In this example, the cost for each solution graph is calculated by summing all edges within the graph. A naive algorithm for finding the best solution subgraph is enumerating all possible candidates and select the smallest one. If the AND/OR graph has d OR nodes, and on average the branching degree for these OR nodes is b, then the total number of candidate subgraphs will be $O(b^d)$, suggesting that such a f_1 could be computationally prohibitive. It has been shown that computing the graph with minimum summed edge cost is NP-hard [167].

In essence, the computational difficulty comes from the fact that each candidate subgraph needs to be evaluated separately, which is unavoidable in general since each tip node could have multiple parents and whether they are going to or how they will meet somewhere upward in the graph is unpredictable. In other words, for arbitrary node, the cost rooted at this node depends not only upon its immediate successors but also on the structure of all its descendants — neglecting which could cause one edge cost being counted multiple times. However, the computation could be largely simplified if we define the solution cost for arbitrary node n from only on

its immediate successors. A cost scheme Ψ is *recursive* if it satisfies the following property:

$$h^*(n) = \begin{cases} \Psi_{n'}(c(n,n') + h^*(n')) & \text{if } n \text{ is non-terminal AND node} \\ \min_{n'}(c(n,n') + h^*(n')) & \text{if } n \text{ is non-terminal OR node} \\ 0 \quad \text{if } n \text{ is solvable terminal} \\ \infty \quad \text{if } n \text{ is unsolvable terminal} \end{cases} \tag{A.1}$$

where $c(n,n') \geq 0$ is the edge cost between n and its successor n', $cost(n)$ represents the cost rooted at n. By such, the implementation of f_1 becomes easy: starting from s, at each OR node, it only needs to select the minimum child node; at each AND node, all successors will be selected. There are, of course, many potential choices for which specific function to use in this formula. One natural choice is the recursive *sum-cost* shown in below:

$$h^*(n) = \begin{cases} \sum_{n'}(c(n,n') + h^*(n')) & \text{if } n \text{ is non-terminal AND node} \\ \min_{n'}(c(n,n') + h^*(n')) & \text{if } n \text{ is non-terminal OR node} \\ 0 \quad \text{if } n \text{ is solvable terminal} \\ \infty \quad \text{if } n \text{ is unsolvable terminal} \end{cases} \tag{A.2}$$

However, even the optimal cost at each node is defined recursively, these values are unknown to the problem-solver simply because the explicit graph G is hidden and fully expanding it is unachievable. Following the *face-value* principle, GBFS* always selects upon G' for a best solution-graph defined by the same recursive cost scheme as depicted above, except that for each non-terminal tip node, an estimation for the cost rooted is given by heuristic function h. Formally, denote $f(n)$ as the cost rooted at n for arbitrary n in G':

$$f(n) = \begin{cases} \Psi_{n'}(c(n,n') + f(n')) & n \text{ is non-tip AND node} \\ \min_{n'}(c(n,n') + f(n')) & n \text{ is non-tip OR node} \\ 0 \quad \text{if } n \text{ is solvable terminal} \\ \infty \quad \text{if } n \text{ is unsolvable terminal} \\ h(n) \quad n \text{ non-terminal tip node} \end{cases} \tag{A.3}$$

To see how f_1 selects the most promising subgraph G_0 when the recursive sum-cost is adopted, consider the example shown in Figure A.4. Each tip node was given an estimated cost; every non-leaf node was computed from its successor values. Starting from s, a solution-base graph is identified by simply choosing a minimum successor at each OR nodes. We recognize that the recursive sum-cost is not the only cost scheme that will facilitate the computation of f_1. Other recursive cost

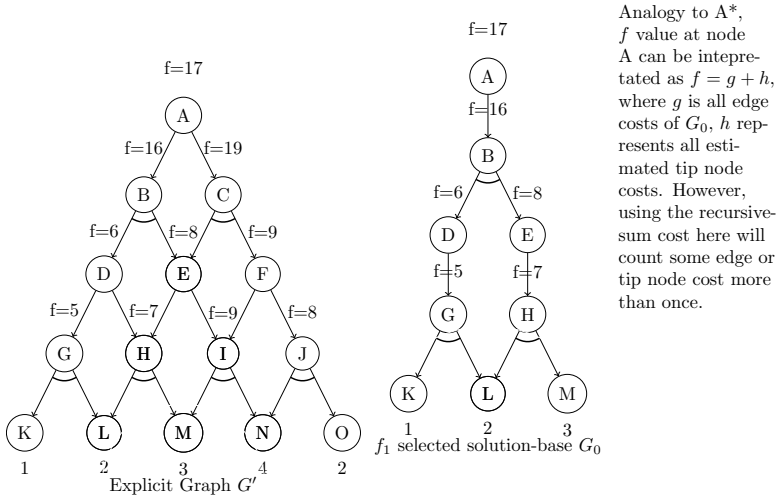


Figure A.4: f_1 selects solution-base from explicit graph G'. Every edge is with cost 0; each tip node has an estimation cost provided by function h. By definition of the *recursive sum-cost* scheme, G_0 is the minimum solution-base with cost 17 even though we see the true summed edge cost is 15.

schemes include *max*, *expected-sum* and so on. In particular, the *max-cost* does not suffer from the *over counting* problem because it does not try to accumulate the cost from its descendants. The sum-cost, however, is free from over-counting only when the underlying graph is a tree. The recursively defined cost scheme also makes the `UpdateAncestors` procedure in Algorithm 7 straightforward — all influenced ancestors just need to be updated in bottom-up manner by Equation (A.1).

The next question to answer is how f_2 selects frontier nodes for expansion (except in some special cases only one frontier exists in the solution base, e.g., in pure OR graphs). As the pursuit is essentially a graph-represented solution, it seems that the selection of frontier nodes is inconsequential, i.e., arbitrary non-terminal tip node can be selected; or alternatively, all expandable tip nodes shall be expanded simultaneously.

Indeed, the specific choice of f_1, f_2 and how to update ancestor nodes signify the key differences between various realizations of GBFS*. Nilsson [142] studied the use of *sum-* and *max-cost* when the underlying graph is an AND/OR tree. Chang and Slagle [42] discussed the use of AND/OR graphs to represent the process of problem-solving by reduction; the graph is directed and acyclic as cyclic reasoning is apparently meaningless; they drew connections between Boolean functions and

AND/OR graphs and define the minimum cost solution as the summed edge cost in the solution graph, but the function (i.e., f_1 in GBFS*) to compute such a measurement was not specified. However, following a similar fashion to OR graph search A* [77], they showed that if the heuristic estimation is admissible and consistent, the algorithm will be optimal. It was proved that computing a solution graph with the minimum summed edge cost is NP-hard in AND/OR graphs [167]. Martelli and Montanari [126, 127] defined an additive AND/OR graph which is essentially AND/OR graphs equipped with the recursive sum-cost; they described detailed edge marking and revising procedures for selecting the solution-base and at each expansion, how to modify this solution-base when updating ancestors. The algorithm was named as HS. Nilsson [143] defined AO* in a similar fashion and discussed the differences between recursive sum- and max-cost schemes. HS can be viewed as AO* with a minor optimization — it updates a node's value only when the new value is lower while AO* updates whenever there is a change. Given h is both admissible and consistent, or the graph is a tree, HS behaves exactly the same as AO*, though they may differ in other scenarios. Bagchi and Mahanti [16] proved that for AO* to obtain optimal solutions, the requirement of heuristic function to be admissible and consistent can be relaxed to only being admissible. Our exposition of GBFS* in Algorithm 7 is adapted from Pearl [153], which includes AO*, A* all as its special cases. As we are majorly interested in AND/OR not pure OR graphs, we reiterate the details of AO* in Algorithm 8.

In Algorithm 8, the selection of a solution-base is achieved simply by following all marked edges from start node s. Then, a non-terminal node is arbitrarily chosen for expansion. After that, all influenced ancestors of the just expanded node will be updated. The cost at arbitrary node t in G' is noted by $f(t)$. Each newly created node is assessed by heuristic function h — it provides an estimated cost if the node is non-terminal, otherwise it yields 0 or ∞ if the node is either solvable or unsolvable terminal. The function f is used to store the accumulated estimation as G' grows. For tip nodes, values by f are always identical to those from h. Using h^* to denote the optimal value at each node for the implicit graph G, we see that the information provided by h can be thought as a rough first estimation on h^*, as G' grows larger, f becomes a more and more finer estimation for h^* until the optimal value is achieved. AO* terminates with an optimal solution graph G_0^* if the heuristic function h is *admissible*, i.e., $h(t) \leq h^*(t)$ for any node t in the search graph.

Algorithm 8: AO* algorithm

Input: Start node s, rules for generating successor nodes, evaluation function h, cost scheme Ψ

Output: Solution graph G_0^*

Function `AO*(`s, f_1, f_2`):`
  Let the explicit graph be $G' = \{s\}$
  while *true* **do**
    Start from s, compute a solution base graph G_0 by tracing down *marked* edges.
    if G_0 *is solution graph* **then** $G_0^* \leftarrow G_0$, exit
    Select any non-terminal tip node t of G'
    Expand t, generate all successors of t
    for $t_j \in successor(t)$ **do**
      if $t_j \in G'$ **then** continue
      evaluate t_j by h, let $f(t_j) \leftarrow h(t_j)$ **if** $h(t_j) = 0$ **then** label t_j as solvable terminal
      if $h(t_j) = \infty$ **then** label t_j as unsolvable terminal
    end
    `UpdateAncestors`(t, Ψ, G')
  end
  return G_0^*
Function `UpdateAncestors(`t', Ψ, G'`):`
  Initialize $S \leftarrow \{t\}$
  while S *not empty* **do**
    Remove any node u from S such that no descendents of u in S **if** u *is OR node* **then**
      $e \leftarrow \min_{u_j \in successor(u)} (c(u, u_j) + f(u_j))$ Mark the edge for which the minimum occurs
    end
    if u *is AND node* **then**
      $e \leftarrow \Psi u_j \in successor(u)(c(u, u_j) + f(u_j))$ Mark all edges between u and its successor
    end
    if $e = 0$ **then** label u as solvable terminal
    if $e = \infty$ **then** label u as unsolvable terminal
    if $f(u) \neq e$ **then**
      $f(u) \leftarrow e$
      Add all predecessors of u to S
    end
  end

However, one potential drawback of AO* in Algorithm 8 is that the selection of tip node from G_0 for expansion is ad hoc. Intuitively, at each step, AO* tries to identify the most promising solution base by following information provided by h and its accumulation f to compute G_0; at each OR node, it selects the successor with smallest f value, while at each AND node, all successors are selected. The later selection mechanism is the culprit for the resultant outcome that more than one frontier nodes exist in G_0. Although earliest research suggests that all non-terminal nodes in G_0 shall be expanded simultaneously [42], this does not seem to be practically feasible, as this expansion approach could quickly exhaust the computer memory due to the exponential growth rate of the size of G'. On the other hand, the ad-hoc strategy used in Algorithm 8 is naive as it does not consider the relative merits of those tip nodes in G_0 at all.

Take the example in Figure A.5 where all edges are with uniform cost 4. Following AO*, given the current implicit graph G', the left branch B shall be selected, and either successors of B could be chosen for expansion. If D is expanded first, it instantly reveals that the selection of B from start node is mistaken. However, if E is expanded first, in the next iteration, AO* will continue to expand E before switching to the true solution branch C. This example shows that a better decision at AND nodes can indeed save our computation for finding the solution object, given the heuristic nature of the decisions made at OR nodes.

The next question to answer is from what measurement shall we make the decision at AND nodes? Th example in Figure A.5 clearly shows that choosing the tip node with the minimum f value is incorrect too. Intuitively, the true metric we wish to have is a measurement on how likely a selected successor could refute our decision at the parent OR nodes. In Figure A.5, we would want such a measurement lead us to select D rather than E, as proving D is an infeasible successor immediately denies our decision of selecting B at A — consequently it enables the search to switch for C instantly without wasting further efforts.

Such a mechanism for making decisions at AND nodes is essentially symmetric to how AO* implements function f_1 at OR nodes. So, if we now equip a pair of functions to AO* based on two heuristics — one for selecting in OR nodes while the other for AND nodes — a top down selection scheme, which eventually selects a single tip node for expansion, exists. There resulting algorithm was never elaborated in the literature. However, a special variant called Proof Number Search (PNS) [2,

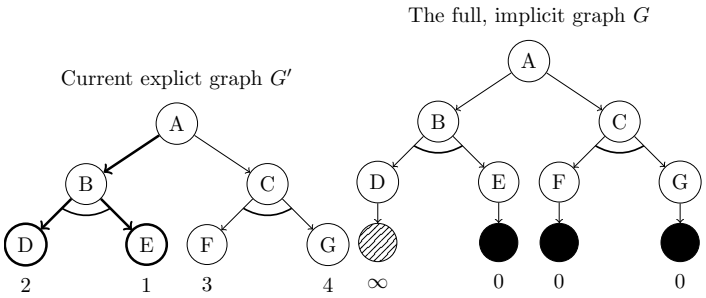


Figure A.5: An example shows the drawback of AO* in selecting frontier node for expansion. If D is expanded first, E will never be expanded; however, if E is chosen first, both D and E will be expanded before the search switches to correct branch C. We assume all edges in the graph have a cost of 4, therefore the estimation provided by h in G' is admissible.

4, 92, 108] has been extensively used in solving games, though its connection to AO* was seldom discussed. From this regard, we shall call the general version of PNS as PNS*, i.e., PNS only counts the number of leaf nodes by proof and disproof numbers, while PNS* works exactly under the same assumption as AO* except that it employs a dual of heuristic functions, h and $\bar{h}$, respectively for making decisions on OR and AND nodes. More specifically, let $< p(n), d(n) >$ denote the estimated cost rooted at node n in G', we have the following recursive relations:

$$p(n) = \begin{cases} h(n) & n \text{ is non-terminal tip node} \\ \min_{n_j \in successor(n)} (c(n, n_j) + p(n_j)) & n \text{ is OR node} \\ \Psi_{n_j \in successor(n)} (c(n, n_j) + p(n_j)) & n \text{ is AND node} \end{cases}$$
$$d(n) = \begin{cases} \bar{h}(n) & n \text{ is non-terminal tip node} \\ \min_{n_j \in successor(n)} (c(n, n_j) + d(n_j)) & n \text{ is AND node} \\ \Psi_{n_j \in successor(n)} (c(n, n_j) + d(n_j)) & n \text{ is OR node} \end{cases} \tag{A.4}$$

When n is terminal:

$$p(n) = \begin{cases} 0 & n \text{ is solvable} \\ \infty & n \text{ is unsolvable} \end{cases} \quad d(n) = \begin{cases} 0 & n \text{ is unsolvable} \\ \infty & n \text{ is solvable} \end{cases} \tag{A.5}$$

Naturally, the optimal $h^*(n)$ and $\bar{h}^*(n)$ are defined on the implicit graph in the same fashion where all tip nodes are terminal. By these definitions, PNS* selects a frontier node to expand by a top down manner: at each OR nodes, it selects a successor n_j with minimum $p(n_j)$ value; at each AND node, it selects a successor n_i with the minimum $d(n_i)$ value. The celebrated PNS algorithm is a special variant of PNS* where $c(n, n_j) = 0$ everywhere in Equation A.4. Figure A.6 is an example

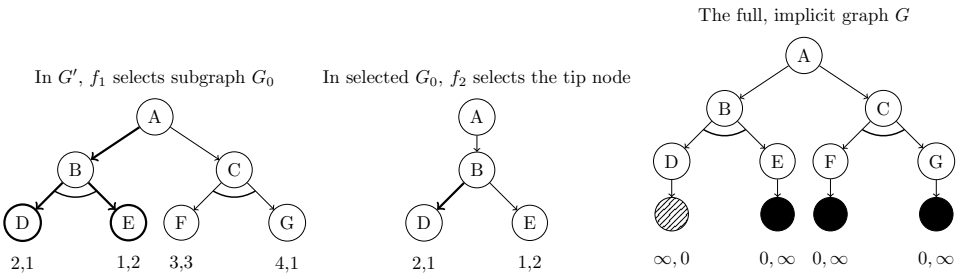


Figure A.6: The same as Figure A.5, edge costs are all 4. The difference is that now each tip node has a pair of heuristic estimations, respectively representing the estimated cost for being *solvable* and *unsolvable*. All tip nodes are with admissible estimations from both h_1 and h_2. Here h_2 successfully discriminates that D is superior to E because $h_2(D) < h_2(E)$. Indeed, as long as $h_2(D) \in [0, 4] \wedge h_2(E) \in [0, \infty]$, h_2 will be admissible, hinting that the chance that an arbitrary admissible h_2 can successfully choose D is high. In respect to PNS, we call algorithm AO* employing a pair of admissible heuristics PNS*.

showing the merit of PNS* in comparison to AO*. To further show the relation between these two algorithms, we define the concept of dual graph in below.

Definition 3. *Suppose arbitrary AND/OR graph is noted as* $G =< V_a, V_o, E >$*, where* V_a *and* V_o *are respectively the set of AND and OR nodes,* E *is the set of edges. The dual of* G*, denoted as* $\bar{G}$ *is defined as* $< \bar{V_a}, \bar{V_o}, E >$ *where* $\bar{V_a} = V_o, \bar{V_o} = V_a$*. That is,* $\bar{G}$ *is obtained by reversing all AND nodes from* G *into OR nodes in* $\bar{G}$*, all OR nodes from* G *into AND nodes in* $\bar{G}$*, all edges remain unchanged.*

We then have the following observations:

Observation 3. *For arbitrary node* n *in AND/OR graph* G*,* $p(n)$ *and* $d(n)$ *are* recursive cost schemes *respectively defined on* G *and* $\bar{G}$*. Let* $G_0 \leftarrow f_1(G)$*,* $\bar{G_0} \leftarrow f_1(\bar{G})$*, then, at each iteration, the tip node selected by PNS* for expansion is the unique intersecting tip node between* G_0 *and* $\bar{G_0}$*.*

Observation 4. *Given the same tie-breaking and* h *from AO* is identical to the one in PNS*, then for the same explicit graph* G'*, the frontier node selected by PNS* must also be a tip node in the solution base* G_0 *selected by AO*.*

Observation 5. *PNS* can be viewed a version of AO*, where* $f_2(G_0)$ *is implemented by selecting the unique tip node of the intersection between* $f_1(G')$ *and* $f_1(\bar{G}')$*. Conversely, AO* can also be viewed as a less informed variant of PNS* by treating all edge cost as* 0 *in* $\bar{G}'$ *and* $\bar{h} = 0$ *when applying* $f_1(\bar{G}')$*.*

After seeing that PNS* be a more informed version of AO*, following the results in A* [153], we conjecture that PNS* is more efficient than AO*:

Conjecture 1. *PNS* dominates AO*, given that AO* uses heuristic function h, and PNS* uses heuristics h and $\hat{h}$; they use the same recursive cost scheme Ψ; both h and $\bar{h}$ are admissible and consistent.*

A.5 Computing True Proof/Disproof Number is NP-hard

PNS is well-defined on AND/OR trees [2], but as described earlier, the space graph of a two-player game is an AND/OR graph. Therefore, using the recursive sum-cost may over-count some leaf node. This could be a problem in some domains, *e.g.*, in Tsume-Shogi some easy to solve transposition node can be assigned huge proof numbers, which prevents PNS from selecting that node without using long search time [105, 136]. Algorithms with exponential complexity are known to find the true proof and disproof numbers at each node in directed acyclic AND/OR graphs [174], but they are impractical even for toy problems. Heuristic techniques address this by replacing sum-cost with a variant of max-cost at each node [203], identifying some specific cases and curing them individually [105, 136]. Such over-counting was called *overestimation* in PNS [106, 108], but the theoretical complexity for dealing with this problem has never been formally discussed. We now show that this task is NP-hard.

Theorem 1. *Deciding whether a SAT instance is satisfiable can be reduced to finding the true proof (or disproof) number of an AND/OR graph.*

Proof. Consider a SAT instance in *conjunction normal form*

$$P = \wedge_i^k C_i,$$

where each clause is a disjunction of literals, $C_i = \vee_j l_j$. Let $x_1, x_2, \ldots, x_n$ be all the variables, each literal l_j is either x_j or $\neg x_j$. Construct an AND/OR graph as follows.

1. Let the start node be an AND node, denoted by P.
2. P contains $n+k$ successors C_i and X_j, $\forall i = 1, \ldots, k,\ \forall j = 1, \ldots, n$. They are all OR nodes.

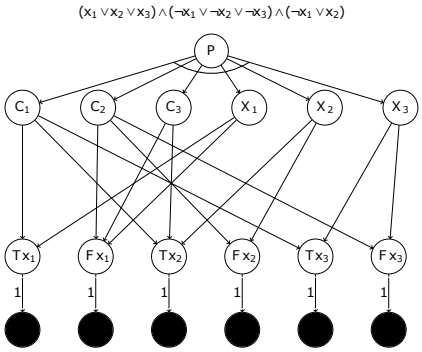


Figure A.7: Deciding whether an SAT instance is satisfiable can be reduced to finding the true proof number in an AND/OR graph. All tip nodes are terminal with value 0. All edges are with cost 0, except those linking to terminal (which have cost 1).

3. Each X_j contains two successors Tx_j and Fx_j representing x_j and $\neg x_j$ respectively. The successors of each C_i are those literals that appear in that clause.

4. Each Tx_j or Fx_j is connected to a terminal node whose value is true or false.

5. All edges have cost 0 except those between Tx_j or Fx_j and terminal nodes: these edges have cost 1.

6. All leaf nodes are *solvable* terminal.

For such a graph, to satisfy the start node P, every clause must be satisfied and each variable node X_j must be assigned to a value. The exact proof number for P is n, because in the best case, to satisfy each X_j, only one of Tx_j and Fx_j is needed. Therefore, finding if an SAT instance is satisfiable can be transformed into finding the true proof number of P in this AND/OR graph. For disproof number, just replace all leaf nodes as *unsolvable* in the construction, then the true and minimum disproof number of P must be n, which is equivalent to finding a solution for the SAT. □

The graph construction from SAT is with polynomial time; it follows that computing proof or disproof number exactly in general AND/OR graph is NP-hard. An example formula $P = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2)$ and its constructed AND/OR graph is shown in Figure A.7.