Experiments with Hex in OpenSpiel and AlphaZero

by

Mohammadreza Daliri

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

Master of Science

Department of Computing Science
University of Alberta

Abstract

OpenSpiel is an open-source software system for implementing high-performance software players for many different computer games. Hex is a two-player game of perfect information used in a variety of computer games research projects. The OpenSpiel project has implemented a version of the AlphaZero algorithm used to great success in the games of Go and chess. Therefore, we use an existing implementation of Hex in OpenSpiel, along with the AlphaZero algorithm, to train and evaluate strong Hex players.

We perform experiments with OpenSpiel for both 6×6 and 8×8 Hex where, to improve the diversity and coverage of self-play games during training, a set of so-called forced-move openings are used. We show that the forced-move openings allow OpenSpiel agents to learn faster, as measured by a test suite of positions. We also developed a set of test positions, with the help of the MoHex system, which can prove the game theoretic value of a given position. Head-to-head play of game agents is often used to evaluate player strength, but a test suite based on proven winning moves (via MoHex) is computationally less expensive.

Acknowledgements

Many people should be recognized for their direct or indirect contributions to the creation of this thesis. First of all, I would like to thank my supervisors, Prof. Paul Lu and Prof. Ryan Hayward, for their support and mentorship throughout my graduate studies. The work in this thesis would not have been possible without their help and guidance.

I am incredibly grateful to Marc Lanctot and Timo Ewalds at DeepMind. Their valuable guidance helped me better understand how the AlphaZero algorithm works and what hyper-parameters matter the most. Furthermore, I thank them for inviting me to DeepMind's hackathon event, the Extravaganza. I thoroughly enjoyed those few days and learned a lot about working with AI models at scale. I would also like to acknowledge all OpenSpiel's contributors for developing a well-documented and solid software library and a well-tested version of the AlphaZero algorithm.

A special thanks go to the present and past members of the GAMES group. To Chao Gao, for fixing and maintaining the MoHex codebase and insightful advice on tuning Hex AI agents. To Prof. Jakub Pawlewicz and Prof. Martin Müller, for open discussions that have enlightened me to explore new directions in my research. And to Christian Jans, for sharing the results of his work with OpenSpiel and AlphaZero on Clobber.

I would also like to thank the Trellis group members. I learned a lot about different topics in software systems and AI out of our research meetings, and learned how to be a better researcher.

Additional acknowledgements go to the Natural Sciences and Engineering Re-

search Council (NSERC) for funding this thesis. And to Cybera and Compute Canada, for providing me with the computational resources needed to perform many of the experiments present in this thesis. I am also thankful to Antonie Bodley, who helped me with the editing of this thesis.

Finally, I am deeply grateful to my family for supporting my studies. I am indebted to my parents and, in particular, to my wife for her great companionship, unconditional support and endless love that kept me sane all along this journey.

Table of Contents

1 Introduction **1**

2 Background and Related Work **7**

2.1 Game of Hex 7

2.2 Monte Carlo Tree Search 8

2.3 AlphaZero 10

2.4 Retrograde Analysis 12

2.5 State-of-the-art Hex Players 13

2.6 Concluding Remarks 14

3 MoHex **16**

3.1 Parallel Solver 17

3.2 Hex-Go Text Protocol 18

3.3 Concluding Remarks 19

4 OpenSpiel **23**

4.1 Advantages of OpenSpiel 27

4.2 Concluding Remarks 29

5 Training AlphaZero-style Hex Players with OpenSpiel **30**

5.1 Hyper-parameters Tuning 31

5.1.1 Learning Rate 33

5.1.2 Dirichlet Noise α 33

5.1.3 Temperature and Temperature Drop 34

5.1.4 Model Type and Dimensions 36

5.1.5 Checkpoint Frequency 36

5.1.6 Actors 36

5.1.7 Evaluators 37

5.2 Forced Exploration of Search Space 37

5.2.1 Openings Database Preparation 38

5.2.2 Different Openings Combinations 40
5.3 Concluding Remarks 42

6 Evaluation Methodology **43**
6.1 Head-to-head Games 44
6.2 Test Positions 44
6.2.1 Candidate Position Extraction 45
6.2.2 Test Positions Selection 52
6.2.3 Test Positions for Hex 6×6 and Hex 8×8 54
6.3 Relationship between Head-to-head Games and Test Positions 67
6.4 Concluding Remarks 67

7 Evaluation Results **68**
7.1 Rationale for Using Test Suite and Evaluation Configuration 69
7.2 Results 71
7.2.1 Results of Evaluating Hex 6×6 Agents 72
7.2.2 Discussion of Hex 6×6 Results 77
7.2.3 Results of Evaluating Hex 8×8 Agents 81
7.2.4 Discussion of Hex 8×8 Results 85
7.3 Concluding Remarks 86

8 Summary and Conclusions **90**
8.1 Directions for Future Work 91

Bibliography **93**

Appendix A: Test Positions for Hex 6×6 and Hex 8×8 **99**
A.1 Hex 6×6 100
A.1.1 Black-to-play Test Positions 100
A.1.2 White-to-play Test Positions 111
A.2 Hex 8×8 121
A.2.1 Black-to-play Test Positions 121
A.2.2 White-to-play Test Positions 146

List of Tables

3.1 Generic Hex GTP commands used in this thesis 21
3.2 MoHex-specific Hex GTP commands for configuring and interacting with the DFPN solver [19] 22

5.1 Most important hyper-parameters that configure OpenSpiel's AlphaZero algorithm. 32
5.2 Values for Dirichlet noise α. 34
5.3 Values for temperature drop hyper-parameter. 35
5.4 Frequency of opening positions extracted for Hex 6×6 and Hex 8×8. . 40

6.1 Number of games used as source games in test suite generation. . . . 58
6.2 Detailed numbers of positions and moves used in test suite generation. 58

7.1 Runtime for two evaluation methods of head-to-head games and test positions. 69
7.2 Values of MCTS-related hyper-parameters used in evaluation step. . . 70
7.3 Average of differences (AOD) of BtoP Winning-move selection ratios (WMSR) for different 6×6 agents 79
7.4 Average of differences (AOD) of WtoP Winning-move selection ratios (WMSR) for different 6×6 agents 80
7.5 Average of differences (AOD) of BtoP Winning-move selection ratios (WMSR) for different 8×8 agents 87
7.6 Average of differences (AOD) of WtoP Winning-move selection ratios (WMSR) for different 8×8 agents 88

List of Figures

4.1 An extensive-form representation of the game of allocating two identical objects between two people. 25
4.2 Hex 2×2 shown as an extensive-form game tree 26

6.1 Two sample 3×3 games and their SGF-encoded game records 46
6.2 All unique positions extracted from Figure 6.1a 49
6.3 All unique positions extracted from Figure 6.1b 50
6.4 Two common positions between the provided sample games in Figure 6.1 that are marked as duplicate by the position extraction algorithm. 50
6.5 Solving time distributions for 10 randomly selected candidate positions for Hex 6×6. 55
6.6 Solving time distributions for 10 randomly selected candidate positions for Hex 8×8. 56
6.7 Frequency distribution of unique positions to number of stones on board in candidate extraction process. 60
6.8 Frequency distribution of unique positions to number of winning moves in candidate extraction process. 62
6.9 Randomly selected test positions from the test suite of BtoP 6×6 positions. 63
6.10 Randomly selected test positions from the test suite of WtoP 6×6 positions. 64
6.11 Randomly selected test positions from the test suite of BtoP 8×8 positions. 65
6.12 Randomly selected test positions from the test suite of WtoP 8×8 positions. 66

7.1 Winning-move selection ratio (WMSR)-Training step curves on BtoP test positions for Hex 6×6 agents trained with different forced-openings combinations 73

7.2 Winning-move selection ratio (WMSR)-Training step curves on WtoP test positions for Hex 6×6 agents trained different with forced-openings combinations 74
7.3 Smoothed version of Winning-move selection ratio (WMSR)-Training step curves on BtoP test positions for Hex 6×6 agents. 76
7.4 Smoothed version of Winning-move selection ratio (WMSR)-Training step curves on WtoP test positions for Hex 6×6 agents. 77
7.5 Winning-move selection ratio (WMSR)-Training step curves on BtoP test positions for Hex 8×8 agents trained with different forced-openings combinations 82
7.6 Winning-move selection ratio (WMSR)-Training step curves on WtoP test positions for Hex 8×8 agents trained different with forced-openings combinations 83
7.7 Smoothed version of Winning-move selection ratio (WMSR)-Training step curves on BtoP test positions for Hex 8×8 agents. 84
7.8 Smoothed version of Winning-move selection ratio (WMSR)-Training step curves on WtoP test positions for Hex 8×8 agents. 85

Abbreviations

BtoP Black-to-play.

BtoW Black-to-win.

DFPN Depth-first proof number.

GTP Go text protocol.

MCTS Monte Carlo tree search.

ML Machine learning.

NN Neural network.

PtoP Player-to-play.

RL Reinforcement learning.

WMSR Winning-move selection ratio.

WtoP White-to-play.

WtoW White-to-win.

Glossary of Terms

Game record A terminal game position together with the sequence of moves which led to that position.

Game position An arrangement of stones on a Hex board, together with the player to move next (Black or White). [Notice that the sequence of moves is not part of this definition.].

Game move A move made by a player, either placing a stone on an empty cell, or swapping identities with the opponent. [These can also be called actions. In Smart Game Format (SGF) game notation, a move is describing by giving the player who moved followed by the move's cell coordinates or the word `swap`, e.g `B[a3]` or `W[swap]`.].

Losing position For a player, a position in which they have already lost, or have no winning move.

Losing move A move which leaves the opponent in a winning position.

Move sequence An ordered list indicating the moves played by players. In Hex, this starts with Black's first move.

Terminal game position A game position in which the winner is already determined, and the game has been finished.

Winning position For a player, a position in which they have already won, or in which they have a at least one winning move.

Winning move A move which leaves the opponent in a losing position.

Chapter 1

Introduction

In this thesis, we explain how to implement an AlphaZero [1] computer Hex player based on DeepMind's OpenSpiel [2] open-source reinforcement learning framework.

Building strong game agents is long known to have been an interesting problem that led to many breakthroughs in artificial intelligence. After Deep Blue, for the first time ever, defeated the multi-year world chess champion, Garry Kasparov in 1997, the game of Go became "the new grand test of intelligence" [3]. About two decades later, in 2016, DeepMind's AlphaGo [63] won against 18-time world champion Lee Sedol and emerged as the first computer program to beat a professional human in Go [3]. AlphaGo used a complex hybrid design with supervised learning, reinforcement learning with a database of self-play games, and multiple neural networks. The very next year, DeepMind introduced AlphaGo Zero [4], an improved RL Go player trained on residual neural networks via self-play without any human expert knowledge. The same training methodology was later generalized and published as AlphaZero, which achieved superhuman strength in chess and shogi [1, 5].

AlphaZero's achievement of a general game learning architecture is significant in different aspects. Plaat [3] lists the following three aspects:

> First, it is remarkable that self-play is so powerful that it can learn from scratch and become so strong that it beats the current world champions. Second, it is remarkable that different games can be learned by the same

> network architecture. Perhaps, some kind of element of general learning must be present in the structure of the ResNets of AlphaZero, although the networks that are learned differ between games: a network trained for Chess cannot play Go well. Third, for Chess and Shogi all previous strong programs had been based on the heuristic planning approach of alpha-beta with domain-specific heuristics, but now they are beaten by a general learning approach using self-play function approximation.

Even with a powerful algorithm such as AlphaZero, there are still problems that need to be resolved [7]. We discuss two of such problems as our research questions in this thesis and propose solutions for them as our main contributions.

The first issue is that achieving perfect play performance might be computationally infeasible in games where the search space is too big to search exhaustively. For instance, Silver *et al.* [1] report that training their AlphaZero Go agent to superhuman level took 9 days on 5000 tensor processing units (TPUs). Consequently, such resources are not available to everyone, and quite a few research teams can train their models for hundreds of thousands of steps with thousands of TPUs and GPUs.

Arguably, this infeasibility is partially because AlphaZero's search algorithm, the Monte Carlo tree search (MCTS), explores better moves more frequently than others when finding the next move to play. So in games with large search space, AlphaZero requires a significant amount of time and resources to practice all possibilities, including bad moves. If such requirements were not met, the agent might not have the chance to explore all moves but only the good ones.

Exploring only good moves may not be an issue when the agent plays against another strong player who chooses the best possible move at each board position. However, the agent may fail and respond with a wrong move if the opponent does something illogical at its turn, like playing a losing move when a winning move is available. The agent has not seen such illogical moves often during training and does

not know how to respond properly. This is mainly because the agent practices only with good moves (*e.g.*, winning moves) as suggested by the MCTS.

This limited exploration could also be a result of using the information learned from previous steps to choose the next move at each training step. In the beginning, the agent may play randomly since no prior information is available yet. However, as training progresses, it learns that some moves are better than others and chooses them repeatedly. Since training is done via self-play, the experienced responses will be limited to those preferred moves. As a result, for example, the agent knows well how to start a game as the first player and how to respond as the second player when the opponent chooses one of the agent's best first moves. But it does not know the best answer to a wrong move.

Research Question (RQ) 1. What is a smarter way to explore the search space in AlphaZero?

To answer this question, we propose to enforce one or more moves as the openings of each training game to expand the explored possibilities. With such openings, the agent will be exposed to a more diverse set of moves instead of only the moves that it thinks are the best while keeping the training time and resource usage low (Section 5.2).

As a case study for our approach, we use the board game of Hex [8]. It is a two-player connection-based game with a hexagonal grid of $n \times n$ hexagons. The players, represented by the colors black and white, take turns placing stones of their respective colors in empty cells. If a chain of adjacent black stones connects one side of the board to the opposing side, the black player wins. Likewise, the white player wins if they can get a line of neighbouring white stones from the other side of the board to its opposite side. By convention, black is the first to move [9].

We choose Hex instead of Go for two main reasons. First, while being a computationally challenging game with a large search space comparable to Go [10–12], Hex has simple, easily-implemented rules that allow quick game simulations and evalu-

ation. Second, the game outcome is simple. It is either win or lose, and no tie is possible [8]. We discuss Hex and its rules in more detail in Section 2.1.

For AlphaZero and Hex implementations, we use the ones included in the OpenSpiel RL framework. OpenSpiel is an open-source library for research in RL in games. It offers environments for a variety of perfect and imperfect information games with one or more players. It also has well-tested implementations of more than ten RL and multi-agent RL algorithms (like AlphaZero) [48], and three search algorithms (MCTS, minimax [34] and alpha-beta [39]) [49].

We use OpenSpiel's AlphaZero implementation instead of writing our own for two main reasons. First, we argue that its implementation is one of the closest ones to the code used in the original AlphaZero paper. While the original code has never been published, the OpenSpiel's version is developed under the supervision of the authors of the same paper at DeepMind [1]. Second, OpenSpiel code and all of its dependencies are open-source. Thus, external researchers require minimal external information to study our methodology.More details on OpenSpiel are provided in Chapter 4.

Hyper-parameter tuning is another challenge in training agents with AlphaZero [7]. Section 5.1 provides detailed information on AlphaZero's hyper-parameters. But before the impacts of any hyper-parameter configuration on the trained agent can be observed, a large amount of computing resource commitment is required.

Therefore, we define our second research question as follows:

Research Question (RQ) 2. What hyper-parameters work well for a Hex player trained with OpenSpiel's AlphaZero algorithm?

Although the most popular board sizes for Hex are 11×11, 13×13, and 19×19 [13], we have to use smaller board sizes of 6×6 and 8×8 in this thesis. This is because we do not have the computing resources and the time required to train strong AlphaZero agents for larger boards. Hence, using smaller boards allows us to experiment with different hyper-parameters and approaches more quickly. Moreover, we have access

to MoHex and its solver (Chapter 3) which can solve Hex 6×6 perfectly and Hex 8×8 near-perfectly.

To address RQ 2, we provide two sets of hyper-parameter values for 6×6 and 8×8 Hex boards that work well for agents trained with OpenSpiel's AlphaZero implementation (Chapter 5). We also present our approach for calculating those values, which could be useful for other board sizes and similar games (Section 5.1). We should note that the reported hyper-parameter values are based on our experiments with tens of different combinations, not a result of an exhaustive systematic hyper-parameter sweep. We also discuss the results of including forced openings in training AlphaZero Hex agents (Section 7.2).

Once an agent is trained, it should be evaluated. A common way to evaluate game AIs is to play a set of head-to-head games against benchmarks. However, this method is time-consuming and may need considerable computing power. Therefore, as our third research question, we would like to find alternative evaluation methods for our Hex agents.

Research Question (RQ) 3. How can we evaluate Hex AIs and determine their strength faster?

Our proposed evaluation method to address RQ 3 is to use a set of carefully chosen test positions grouped as test suites to quickly test an agent's response in real-world situations. These test positions have been selected from actual games using a step-by-step and reproducible approach. Our approach is thoroughly described in Chapter 6. Compared to the classic evaluation methods for game AIs, such as head-to-head games, our new evaluation method using test suites is about 20 times faster (Table 7.1). While we provide test suites for Hex 6×6 and Hex 8×8 (Appendix A), our approach can easily derive test positions for other board sizes.

We can summarize the three main contributions of this work, which all directly relate to the research questions discussed previously, as below:

1. As a smarter way to explore the search space, we propose enforcing move openings for each training game to expand the explored moves over the course of training and improve the quality and speed of training (Section 5.2).

2. To address the challenge of hyper-parameters tuning, we provide two sets of hyper-parameter values for training strong AlphaZero-style Hex players for 6×6 and 8×8 boards with OpenSpiel (Chapter 5).

3. To evaluate Hex AIs and determine their strength quickly, we propose to use test suites and present a systematic method for building such test suites (Chapter 6).

Our empirical results (Chapter 7) show that enforcing move openings at the beginning of training games improves agents' performance. This improvement is, on average, 10% for Hex 6×6 and 14% for Hex 8×8. Detailed discussion of the results is provided in Section 7.2.

Chapter 2

Background and Related Work

In this chapter, we provide an overview of the concepts and techniques that we build on in the remainder of the thesis. We start with a discussion of the game of Hex, and why we choose it as a case study for our ideas (Section 2.1). Then, we describe game trees, and the Monte Carlo tree search algorithm [14] that is used by many computer Hex programs (Section 2.2). We continue by reviewing AlphaZero [1], a high-performance reinforcement learning algorithm that uses self-play to build strong game agents (Section 2.3). In Section 2.4, we briefly explain the retrograde analysis as a technique for backward searching of the game trees. Finally, in Section 2.5, we discuss the recent advancements in computer Hex programs and the current Hex Olympiad champion.

The next two chapters, Chapter 3 and Chapter 4, provide background information on MoHex [15, 16] and OpenSpiel [2]. The former is one of the strongest Hex programs that can play near-perfectly for board sizes up to 8×8. The latter is an open-source software framework for implementing high-performance players for many different games with a variety of algorithms, including AlphaZero.

2.1 Game of Hex

Hex was invented in 1942 by Danish mathematician Piet Hein and later popularized by John Nash in 1948 [8]. It is a board game that is played on a rhombus board

comprised of an $n \times n$ array of hexagons. Although the board can be any size, the popular sizes are 11×11, 13×13, and 19×19 [13]. We use smaller board sizes of 6×6 and 8×8 in this thesis because they are easier to solve and allow us to experiment with viable approaches more quickly. Moreover, we have access to MoHex and its solver (Chapter 3) which can solve Hex 6×6 perfectly and Hex 8×8 near-perfectly.

On a Hex board, two opposing sides are black, while the remaining two are white. One player's sides and stones are all black. The other player's sides and stones are white. Each player takes a turn placing a stone of their own color in an empty cell. The first player typically uses black stones, while the second player uses white stones. Whoever successfully reconciles their two sides is the winner [8]. In other words, the winning player must connect both sides of the board using a line of their color stones.

A game can never be declared a tie [8, 10]. According to Simonsen and Haddeland [10], "proving this is equivalent to proving the Brouwer fixed-point theorem for two dimensions, as done by David Gale [17]. As such, the only way to completely block the opponent is to form a connection with one's own pieces." (p. 6). The first player always has a winning strategy regardless of the board size [18]. Finding these strategies becomes more difficult as the board size increases. Current research solves up to 9×9 boards and also some 10×10 openings [19]. Indeed, it is PSPACE-complete to solve a particular state [20, 21].

A swap rule is used in one version of Hex to compensate for the first player's significant advantage. According to this rule, after the first player places the first stone, the second player may either continue playing normally with stones and sides of the opposite color, or may swap stones and sides with the first player. After that, players take turns as usual [8]. This is the variant that is most frequently used in competitions [13]. However, in this thesis, we do not consider the swap rule.

2.2 Monte Carlo Tree Search

Games like chess and Hex have enormous game trees because of their large branching factor [21, 22]. Therefore, it is almost impossible to exhaustively search trees because of time and memory constraints. As an alternative to checking all nodes, Coulom [14] proposes the Monte Carlo tree search (MCTS) which stochastically samples and visits the nodes to find a desirable root-to-leaf path. The MCTS performs several iterations known as simulations to find the best next-level node. Each iteration includes four operations: selection, expansion, simulation (rollout), and backpropagation. After completing a specific number of iterations, the data from all iterations is aggregated, and the node with the greatest visit count at the first level after the root is returned as the search result [9]. In the rest of this section, we discuss the four operations in detail.

The selection phase traverses the tree from the root node to a leaf, at which point a new child – that is not yet a member of the tree but has the best chance of winning is selected [10]. A tree policy determines which action to execute at each internal stage and which node to explore next [3].

The expansion step then expands the tree by adds one or more edges from the selected node [3]. While some variations of MCTS add only one child, other add all descendants of a leaf node [23].

Following that, the simulation step selects nodes down the tree using a default policy until it reaches a leaf node. At the leaf node, the reward value, *e.g.*, +1 for a win, 0 for a tie, and -1 for loss is computed. According to Anthony [9], "in the absence of domain-specific information, the default policy used is simply to choose actions uniformly from those available. Often hand-crafted default policies using domain knowledge give sizable performance improvements." (p. 59).

The backpropagation step propagates the reward value assigned to the leaf node upwards in the tree via the nodes traversed before. Then, values are updated from

the leaf node to the root node [3, 10].

According to Anthony [9], "the tree policy is based on statistics stored in the game tree (such as average values of states), and is designed to explore possible actions and gradually improve. The default policy is fixed; it is the policy taken in states outside the partial tree" (p. 59).

For each node s of the tree, we store the number of simulations that visited that node as $n(s)$. For each edge a, we keep two numbers: $n(s, a)$, the number of times it has been traversed, and $R(s, a)$, the total reward collected in simulations that travelled through it. The tree policy uses the values stored in the nodes and edges to determine what branch to visit next [3]. The most frequently used tree policy [9] is Upper Confidence bounds for Trees (UCT) [23]. The UCT formula, as listed in Equation 2.1, is sum of two terms known as exploitation and exploration, respectively. The exploitation term, $\frac{R(s,a)}{n(s,a)}$, favours actions with a high average reward [3]. The exploration term, $c_b\sqrt{\frac{\log n(s)}{n(s,a)}}$, favours rarely visited actions [9]. The c_b constant controls the level of exploration, where lower values mean less exploration [3].

$$\mathrm{UCT}(s, a) = \frac{R(s, a)}{n(s, a)} + c_b\sqrt{\frac{\log n(s)}{n(s, a)}} \tag{2.1}$$

When all iterations are completed, the most visited node after root is returned as the search result. According to Anthony [9], "because UCT will explore higher value actions more often, the most explored action [node] is usually also the highest value [reward] action. However, if an action's value changes near the end of the search procedure, then the most explored action and the highest value action differ." (p. 60). While it is possible to select the node with highest value instead, "but this has been found to be in practice slightly weaker than taking the most explored action." (p. 60) [24].

2.3 AlphaZero

AlphaZero [1] and AlphaGo Zero [4] are MCTS-based self-play reinforcement learning (RL) algorithms created by Alphabet's DeepMind. Without any human expert knowledge or supervised learning, these algorithms are designed to play board games like Go, chess, and shogi (Japanese chess). While AlphaGo Zero attained superhuman performance only in Go, AlphaZero decisively won against world champion programs: Stockfish [25] in chess, and Elmo [26] in shogi, as well as AlphaGo Zero in Go.

According to Silver *et al.* [1], "AlphaZero replaces the handcrafted knowledge and domain-specific augmentations used in traditional game-playing programs with deep neural networks, a general-purpose reinforcement learning algorithm, and a general-purpose tree search algorithm." (p. 1).

The deep neural network that is used instead of custom evaluation functions and heuristics, "takes the board position s as an input and outputs a vector of move probabilities p with components $p_a = \Pr(a|s)$ for each action a and a scalar value v estimating the expected outcome z of the game from position s, $v \approx E[z|s]$." (p. 1) [1]. The neural network parameters, as well as the move probabilities and value estimates are entirely learned by self-play. The resulting move probabilities and values are used to drive the MCTS in subsequent self-play games [1].

For all three games of chess, shogi, and Go, AlphaZero uses the same neural network architecture: "The neural network consists of a 'body' followed by both policy and value 'heads'. The body consists of a rectified batch-normalized convolutional layer followed by 19 residual blocks." (p. 16) [1]. All training hyper-parameters, except the learning rate and the Dirichlet noise, are the same as well [1]. We discuss AlphaZero hyper-parameters and their values in Section 5.1.

There are a number of differences between the AlphaZero algorithm, as described by Silver *et al.*, and AlphaGo Zero [4]. On one hand, AlphaGo Zero works by evaluating and optimizing probability of a binary outcome, given that a Go game does not end as

a tie. AlphaZero on the other hand, "evaluates and optimizes the expected outcome" (p. 2), since draws are possible in chess and shogi. The other difference is in the self-play games. Self-play games in AlphaGo Zero involved the best player of all previous iterations. Conversely, in AlphaZero, self-play games are generated by the single neural network that is updated continuously. Therefore, there is no need to wait for iteration completion [1].

To reach superior game play strength in chess, shogi, and Go, Silver *et al.* "trained separate instances of AlphaZero ... for 700,000 steps (in mini-batches of 4,096 training positions) starting from randomly initialized parameters. During training only, 5,000 first-generation tensor processing units (TPUs) were used for self-play games, and 16 second-generation TPUs were used to train the neural networks. Training lasted about 9 hours for chess, 12 hours for shogi and 13 days for Go." (p. 3) [1].

2.4 Retrograde Analysis

As discussed in Section 2.2, games like chess and Hex have game trees that are too big, and tree search algorithms may not efficiently reach the terminal positions. To overcome this limitation, heuristics are used to estimate the value of endgame positions, *i.e.*, win, loss or tie. Alternatively, a database of pre-calculated positions can be used to quickly look up a position's value. Retrograde analysis, also known as backward search, can be employed to create such a database [22].

A backward search contrasts with a forward search. According to Wu and Beal [22], in forward search, "one starts from an unknown state and tries to determine its value by recursively searching all successor values. A backward search starts from the known states and tries to prove the predecessor states' values." (p. 1).

Wu and Beal [22] describe an example usage of retrograde analysis in games as follows:

> If you programmed the game using retrograde analysis, you would start

> from an ending position. The position value is defined by the rules of the game: win, lose, or draw. Next, you look at all the moves that could have gotten you there. If you can play into a win, the position you are moving from is also a win. If you play into a loss, there is a possibility that all moves from that position lead to losses, so you examine them all. If all other moves lead to losses, the position you are examining is also a loss. Repeat this backward analysis for all ending positions. You now know the value of all positions one move away from ending positions. Repeat for positions one move away, two moves away, etc., until no more new wins or losses emerge. You now know the best end result you can achieve for the starting position, and every other position.

Retrograde analysis can be categorized as exhaustive search as well. According to Wu and Beal [22], we obtain a "full yet perfect knowledge" (p. 1) of a domain once we have completed the analysis.

So far, the retrograde analysis has been applied successfully in several games. For instance, Lake *et al.* [27] used the retrograde analysis in checkers to solve all endgame positions with up to eight stones [22]. Equipped with this database, their checkers player, Chinook, always plays the best move in positions with eight or fewer stones [28]. In 1994, Chinook became the first program to win a human world championship [27].

2.5 State-of-the-art Hex Players

Many computer Hex programs have been developed in the past couple of decades. The MoHex family, consisting of MoHex 1.0 [15], MoHex 2.0 [16], MoHex-CNN [29] and MoHex-3HNN [30], had been the champion of the computer Hex Olympiad for about a decade – from 2009 to 2018. All four versions of MoHex use the MCTS algorithm for tree search. MoHex 2.0 is equipped with a parallel depth-first number search

solver [19]. MoHex-CNN and MoHex-3HNN, unlike the other two, use convolutional and residual neural networks, respectively. These networks are used to learn prior probability distributions on the actions for the MCTS exploration phase and values for the states and actions for the MCTS evaluation phase [13]. We discuss the MoHex family in detail in Chapter 3.

There are also Hex programs that employ RL. NeuroHex [31] uses deep RL [32] to train an 11-layer CNN for Hex 13×13. The training is performed via self-play after a supervised initialization over a common Hex heuristic [13]. DeepEZO [33] uses self-play to train a convolutional neural network for creating value and policy functions. DeepEZO's player uses minimax tree search [34] with forward-pruning according to the moves determined by the policy function in self-play games. DeepEZO's policy function is trained directly from the game results and does not involve the search probabilities [33]. Expert Iteration (ExIt) [9] learns a separate expert policy in addition to the state values. It does not use any expert knowledge during its convolutional neural network training [9, 13]. For Hex 9×9, it implements two planning algorithms, one based on the MCTS, and the other based on policy gradient search, a novel model-free RL algorithm that works without a search tree [9].

Polygames [35] implements an AlphaZero-style algorithm [1, 4] with the MCTS algorithm for tree search. Since it uses board-size-invariant neural network architectures, networks trained on smaller board sizes can be scaled to larger ones. Equipped with some other innovations, Polygames won the gold medals in the 2020 Hex Olympiad for board sizes of 13×13 and 19×19 [36]. Similar to AlphaZero, Polygames also requires significant computing resources. It was trained for nine days with 500 GPUs to reach its top performance level [35]. As of 2021, the Hex Olympiad champion in all three board sizes of 11×11, 13×13 and 19×19 is Ultron [37, 38]. It is a deep RL player that uses minimax instead of MCTS and only learns state evaluations, not policy evaluations [37].

2.6 Concluding Remarks

This chapter reviewed the background information required for understanding the rest of this thesis. We provided a brief history of Hex, explained its basic rules, and discussed how having a large-enough search space while being easy to solve made Hex 6×6 and 8×8 good choices for our case study (Section 2.1). We explained the MCTS algorithm and the four operations of selection, expansion, simulation and backpropagation that it does in each iteration to eventually find the most explored move (Section 2.2). We briefly discussed how the AlphaZero algorithm trains high-performance computer agents via self-play without any prior expert knowledge (Section 2.3). We also saw that AlphaZero would need thousands of TPUs to reach the superhuman level, which is out of reach for most people. We explained the retrograde analysis, also known as backward search, which could be used to efficiently search the game tree of more complex games like Hex by solving positions endgame positions first (Section 2.4). Finally, we quickly reviewed the state-of-the-art in computer Hex programs and introduced Ultron [37], the 2021 Hex Olympiad winner.

In the following two chapters, we provide detailed information on two essential tools used in this thesis, the MoHex program (Chapter 3) and the OpenSpiel RL framework (Chapter 4).

Chapter 3

MoHex

In the previous chapter, we covered the basics of the game Hex, the Monte Carlo tree search and AlphaZero algorithm. In this chapter, we briefly introduce MoHex 2.0 [16], the last Hex player that won the Olympiad championship without using a neural engine. We also explain the Hex-Go text protocol (Hex GTP) used for communicating with Hex engines like MoHex. We use MoHex both in training AlphaZero Hex agents and in the evaluation step, discussed in detail in Chapter 5 and Chapter 6, respectively.

The computer Hex player MoHex first appeared as MoHex 1.0 [15] at the 2009 Computer Games Olympiad in Barcelona. MoHex is part of the computer Hex software collection Benzene developed at the University of Alberta, which also includes an alpha-beta [39] player Wolve and a depth-first proof-number search solver called simply Solver [16, 19, 40]. MoHex is based on the open source collection of Go programs Fuego [24] developed by Enzenberger *et al.* at the University of Alberta. From 2009 through 2018, the MoHex family won every Hex Olympiad championship on 11×11 and 13×13 board sizes. Today, the MoHex family has been able to solve all Hex openings for board sizes up to 9×9, and two non-isomorphic openings of Hex 10×10 [8].

There are four versions of MoHex, each built on the top of its ancestor. MoHex 2.0 uses pattern weights trained on human knowledge for its rollout policy instead of

a hand-crafted policy used in MoHex 1.0 [9]. The weight-based rollout policy guides both tree exploration and simulations. The next version, MoHex-CNN [29], uses a convolutional neural network to learn a prior probability distribution on the actions to play. These prior probabilities guide the exploration of the tree search. The most recent version, MoHex-3HNN[1] [30], replaces the CNN with a residual neural network and calculates not only the policy but also values for each state and action. Therefore, the simulation-based MCTS from previous versions is replaced by state values. Assigning a value to each action improves the overall performance by reducing the number of neural network calls [13].

Apart from the mentioned differences, all versions of MoHex are equipped with many Hex-specific techniques enabling them to determine a winning strategy quickly without expanding the search tree. The virtual connection and inferior cell detection engines [40] are notable unique features of MoHex 2.0. These engines allow MoHex 2.0 to prune many actions before starting the tree search [9].

In the rest of this thesis, all instances of MoHex refer to MoHex 2.0 unless otherwise stated.

3.1 Parallel Solver

As mentioned, the Benzene package includes a depth-first proof number (DFPN) solver [19]. The MoHex's MCTS itself can be used as a solver since MCTS eventually converges. However, in practise, it solves a Hex position only when the initial virtual connection computations at the root of the search tree find a win: the tree search part of MoHex almost never solves a position that the virtual connection computations did not detect.

Pawlewicz *et al.* implemented a parallel version of the Solver [19], which allowed it to solve all 9×9 and two non-isomorphic 10×10 1-stone positions. The search time for the 10×10 positions was several months on a 24-processor machine; in general, on

[1] https://github.com/cgao3/neurobenzene

a single-processor machine, 8×8 positions can take more than half an hour to solve.

When MoHex is run with its default setting of parameters, it runs on a single computation thread with the Solver off. To turn the Solver on, the user sets a flag that specifies the number of additional threads to be used for the Solver. With the Solver enabled, when MoHex is asked to play a move for a given state, it activates the parallel solving thread(s). Once the allotted MCTS computation time is up, MoHex then checks whether the Solver found a solution: if yes, the provably-winning Solver's move is returned; if no, the MCTS move (which is not necessarily a winning move) is returned.

In addition to solving a state, *i.e.*, revealing the winner, the Solver can find all or some of the winning moves for the current player, if any exists. However, it may not always find all winning moves. We propose a workaround for this limitation in Section 6.2.1.

3.2 Hex-Go Text Protocol

The Hex-Go text protocol (Hex GTP) is a textual interface for communication with Hex software [41]. By providing a standardized protocol based on Go Text Protocol (GTP), Hex GTP facilitates the separation of graphical user interfaces (GUI) from Hex strategy engines and bots [42]. Furthermore, it permits humans to interact with engines over the network directly and perform regression tests automatically. All four versions of MoHex support this protocol. And can be plugged into any compatible GUI such as HexGUI [43] and communicate directly with other GTP-compliant bots.

This protocol follows a client-server model and uses textual commands and responses represented as ASCII characters. The client and server communicate over a socket where the client writes commands in an input stream and reads the response from an output stream. Commands are composed of an optional identity number, a keyword, and zero or more arguments, all separated by a whitespace character. Each command is sent to the server as one line of text terminated by a newline character.

A successful response begins with the equal character (=), while a failure starts with a question mark (?). Immediately after that comes the optional identity number of the requested command, if any was provided. Finally, the actual response comes after one whitespace character. In case of an error, the actual response would be a short description of the failure. Responses can have one or more lines of text. Listing 3.1 shows a sample communication with MoHex over the GTP protocol to play a move on a 3×3 board using a set of commands discussed below.

Any GTP-compliant Hex bot needs to understand a subset of Hex-GTP commands related to client-server interaction, board handling, and move making. Although we are not aware of a minimum specification of such commands for GTP-compliant Hex bots, Table 3.1 shows the list of such commands that sufficed for our experiments. The command `list_commands` is used to list all supported commands.

Table 3.2 lists a set of commands provided by MoHex to configure and interact with its DFPN solver. We use such commands later in Chapters 5 and 6.

3.3 Concluding Remarks

In this chapter, we briefly reviewed the four versions of MoHex, one of the most successful computer Hex players in the last decade. We discussed the main features of MoHex: a virtual connection engine, an inferior cell detection engine used to prune the search tree, and the DFPN solver used to quickly find winning moves near the end of a game. We also discussed Hex GTP, a text-based protocol for computer Hex players to communicate with each other (Section 3.2).

In the next chapter, we will discuss OpenSpiel. This open-source software library allows fast and accurate implementation of AlphaZero-style agents for a wide range of board games, including Hex.

```
boardsize 3
=

showboard
=

  e987fcd3b3ce7cac
  a  b  c
 1\.  .  .\1
  2\.  .  .\2
   3\.  .  .\3
      a  b  c

genmove black
= b2

all_legal_moves
=  resign swap-pieces a1 b1 c1 a2 c2 a3 b3 c3

protocol_version
= 2

name
= MoHex

version
= 2.0.CMake
```

Listing 3.1: Playing a Hex 3×3 game via GTP interface of MoHex

Table 3.1: Generic Hex GTP commands used in this thesis (description provided by HexWiki [41])

Command	**Arguments**	**Meaning**	**Category**
`boardsize`	`n`	Set the board size to $n \times n$ and remove all stones from the board.	Board handling
`clear_board`	–	Remove all stones from the board.	Board handling
`showboard`	–	Display the current board in a human- and machine-readable format.	Board handling
`play`	`player move`	Play the provided `move` for the specified `player`.	Making moves
`genmove`	`player`	Generate and play a proper move for the given `player`.	Making moves
`undo`	–	Undo the most recent move.	Making moves
`all_legal_moves`	–	Display a list of all legal moves available for the current situation.	Making moves
`name`	–	Display the name of the server program.	Server interaction
`version`	–	Display the version of the server program.	Server interaction
`protocol_version`	–	Display which version of the GTP protocol the server program supports.	Server interaction
`list_commands`	–	Display the list of all commands supported by the server program.	Server interaction
`quit`	–	Shut down the connection to the server.	Server interaction

Table 3.2: MoHex-specific Hex GTP commands for configuring and interacting with the DFPN solver [19]

Command	**Arguments**	**Meaning**
`param_mohex`	`key value`	Configure MoHex by setting `value` for the requested `key`. A tuple of `key=use_parallel_solver, value=1` is used to enable the DFPN solver.
`param_dfpn`	`key value`	Configure DFPN solver by setting `value` for the requested `key`. We can set number of threads to be used by the solver by passing `key=threads`.
`dfpn-solve-state`	–	Solve the current state using DFPN solver and return the winner.
`dfpn-solver-find-winning`	`player`	Find zero or more winning moves for the given `player`.

Chapter 4

OpenSpiel

In this chapter, we briefly introduce OpenSpiel [2], a reinforcement learning (RL) framework for research in games developed by Alphabet's DeepMind. We also discuss the reasons for using this framework to train AlphaZero-style Hex players. The next chapter describes our training process in detail and discusses our methodology for tuning hyper-parameters.

OpenSpiel is an open-source library for research in RL and search/planning in games. It offers environments for a variety of perfect and imperfect information games with one or more players and well-tested implementations of more than ten RL and multi-agent RL algorithms [48]. For zero-sum turn-taking games with perfect information like Hex, OpenSpiel has implementations of three classical search algorithms [49]: minimax [34], alpha-beta [39], and Monte Carlo tree search (MCTS) [2, 14, 23, 44]. Furthermore, it includes tools to analyze learning dynamics and other standard evaluation metrics.

In this thesis, we use the Hex game environment and Python implementations of the AlphaZero training algorithm with MCTS from OpenSpiel. OpenSpiel provides a general API implemented in C++ and exposed to Python via `pybind11` [47]. Having this API almost the same in the two languages allows users to translate code if needed quickly. It contains over 30 different games [48] and more than 20 algorithms [49]. While games are written in C++ to increase memory efficiency and faster runtime,

most algorithms that require machine learning (ML) are implemented in Python and use Tensorflow [50] and PyTorch [51].

Perfect-information games like Hex are represented as procedural extensive-form games [52, 53] in OpenSpiel. A tree symbolises a perfect-information game in its extensive form, with each node indicating one of the players choice, each edge indicating a possible action, and the leaves representing final outcomes for all players [53]. If a game has stochastic events, a special player called *chance* (also known as *nature*) is added to the tree. This player takes actions according to a given probability distribution [54]. In artificial intelligence, these extensive-form games are known simply as game trees. Because the choice nodes are arranged in a tree structure, we can precisely identify a node by its history, that is, the series of choices that led to it from the root node [53]. Therefore, each conceivable sequence of players' actions is represented explicitly as a root-to-leaf path in the tree.

Figure 4.1 represents a simple game in extensive form in which two identical indivisible objects are shared between two people. Osborne and Rubinstein [52]) describe this game as follows:

> One of them [players] proposes an allocation, which the other then either accepts or rejects. In the event of rejection, neither person receives either of the objects. Each person cares only about the number of objects he obtains. The small circle at the top of the diagram represents the initial history $\emptyset$ (the starting point of the game).
>
> ... The small circle at the top of the diagram represents the initial history $\emptyset$ (the starting point of the game). The 1 above this circle indicates that $P(\emptyset) = 1$ (player 1 makes the first move). The three line segments that emanate from the circle correspond to the three members of $A(\emptyset)$ (the possible actions of player 1 at the initial history); the labels beside these line segments are the names of the actions, $(k, 2-k)$ being the proposal to

give k of the objects to player 1 and the remaining $2-k$ to player 2. Each line segment leads to a small disk beside which is the label 2, indicating that player 2 takes an action after any history of length one. The labels beside the line segments that emanate from these disks are the names of player 2's actions, y [yes] meaning "accept" and n [no] meaning "reject". The numbers below the terminal histories are payoffs that represent the players' preferences. (The first number in each pair is the payoff of player 1 and the second is the payoff of player 2.)

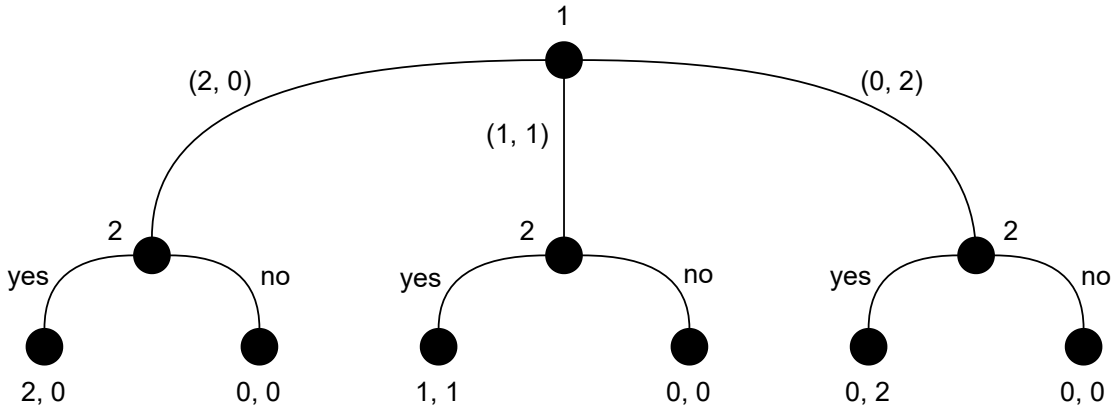


Figure 4.1: An extensive-form representation of the game of allocating two identical objects between two people (recreation of Figure 91.1 of Osborne and Rubinstein [52]).

Figure 4.2 depicts the extensive-form game tree for Hex 2×2 generated by OpenSpiel's visualizer tool and slightly modified for the sake of clarity. The blue and red edges correspond to black and white players. The diamonds correspond to the terminal positions and are labelled by the outcome for black. Since Hex does not have stochastic events, there is no edge for the chance player.

For more information on extensive-form games and how they are implemented in OpenSpiel, see Section 5.1 of Shoham *et al.* [53] and Section 3.1 of Lanctot *et al.* [2], respectively.

OpenSpiel describes games using two concepts [2]:

- **Game class:** It contains the high-level description of a game. Properties like

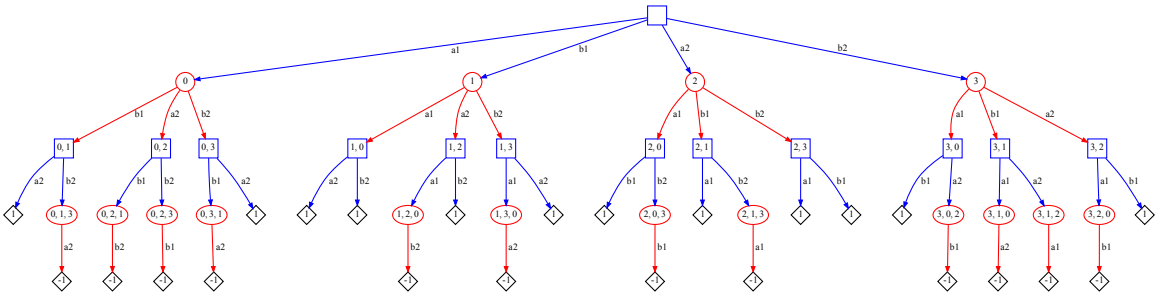


Figure 4.2: Hex 2×2 shown as an extensive-form game tree. Blue and red edges correspond to black and white players. The leaves (shown as diamonds) are terminal game positions labelled with the game result for black.

the number of players, minimum and maximum scores, and having perfect or imperfect information are stored in instances of this class. This class also knows how to initialize and set up the board for a new game.

- **State class:** In computer games, the current state of the board is referred to as a game position. In OpenSpiel, game positions are represented by instances of state class. State objects provide APIs for generating and applying the next move, and a reward function for scoring a terminal game position. In this thesis, we use game position and states interchangeably.

Listing 4.1 provides an example of using the game class and the states APIs to play a complete Hex game between two random players. At lines $6-10$, we set up two uniform random bots as players. OpenSpiel includes MCTS and Go Text Protocol (GTP) compatible bots in addition to multiple kinds of stochastic ones (not shown on the listing.) At line 13, we initialize an object of a game class that implements Hex. Each game could be configured by one or more variables. In the case of Hex, the only configurable variable is board size (line 13.) At line 16, the starting game state, an empty board, is initialized and returned by the game object. In each iteration of the loop defined by lines $18-38$, the following actions happen. First, the current player is identified by the state object (lines $19-20$). Second, the current player is

asked to generate its next move (line 29). Third, the opponent gets informed of the selected move (line 35) to ensure all players have perfect information about the game. Finally, the selected move is applied to the state object (line 38). When a terminal state is reached (line 18), the loop ends, and the print statement of line 41 announces the winner.

4.1 Advantages of OpenSpiel

We opt to use the OpenSpiel framework instead of writing our game agent for Hex from scratch for two main reasons:

1. **Includes one of the strongest available algorithms:** As described in Chapter 1, one of the main goals in this thesis is to train a strong Hex player. From our discussion in Section 2.3, we know that the AlphaZero is established as a strong high-performance training algorithm [1]. Therefore, using this algorithm in our Hex agents allows us to study the effectiveness of our ideas when applied on an existing high-performance technique.

   However, the original AlphaZero implementation has not been released to the public to date. So we decided to use the one included in OpenSpiel instead of writing our version based on the original paper. Since it is maintained by DeepMind and is developed under the supervision of the original AlphaZero paper's authors [1].

2. **Open-source and flexible:** OpenSpiel code and all of its dependencies are open-source and available online. Thus, external researchers require minimal information to study our methodology.

Among the other interesting features of OpenSpiel is having a diverse list of supported games with a similar programming interface. Since its release in September 2019, OpenSpiel has gained much attention and has become popular in the RL community. Researchers and engineers from across the world have participated in the

```python
import numpy as np
import pyspiel
from open_spiel.python.bots import uniform_random

# Set up two random rollout players.
rng = np.random.RandomState()
players = [
  uniform_random.UniformRandomBot(player_id, rng)
  for player_id in range(0, 2)
]

# Initialize an object of Hex game class.
game = pyspiel.load_game('hex(board_size=2)')

# Initialize the starting game state (an empty board for Hex).
state = game.new_initial_state()

while not state.is_terminal():
  current_player = state.current_player()
  player = players[current_player]

  # The algorithm can pick an action based on an observation
  # (fully observable games) or an information state (information
  # available for that player).
  # However, since both players are uniform random players,
  # they randomly select one of the legal actions in each step.
  # The body of `UniformRandomBot.step` method looks like the following:
  #   return self.rng.choice(state.legal_actions())
  action = player.step(state)

  for i, player in enumerate(players):
    # Hex is a perfect information game,
    # so all players will be informed of the selected action.
    if i != current_player:
      player.inform_action(state, current_player, action)

  # Apply the selected action on the current state.
  state.apply_action(action)

# Print the winner when a terminal position is reached.
print('Game results: {}'.format(state.returns()))
```

Listing 4.1: Python code to play a Hex game with OpenSpiel

project and are actively working on adding new games and algorithms under the direct supervision of DeepMind's scientists. Currently, more than 30 games other than Hex are included in OpenSpiel.

4.2 Concluding Remarks

In this chapter, we provided a high-level introduction to the OpenSpiel RL framework and presented how its use of game/state classes can be used to play the game of Hex. We also discussed how the strength of OpenSpiel's implementation of the AlphaZero algorithm and its open-source nature make it a proper tool for our research. In the next chapter, our focus will be on using OpenSpiel's AlphaZero algorithm to train strong Hex players. We will review AlphaZero's hyper-parameters and propose our solution to improve search space exploration.

Chapter 5

Training AlphaZero-style Hex Players with OpenSpiel

As a continuation of our discussion on the AlphaZero algorithm [1] and the OpenSpiel reinforcement learning (RL) framework [2] in Section 2.3 and Chapter 4, this chapter describes techniques, such as the forced exploration of search space (Section 5.2) to train strong AlphaZero Hex players with OpenSpiel. Since we need to compare different trained models as part of our discussion, we also briefly introduce a test method later in the current chapter and leave the complete description of our evaluation methodology for the next chapter.

For this thesis, we use the Python implementation of OpenSpiel with Tensorflow [50] and train and evaluate all Hex models without GPU for the reasons that follow. However, OpenSpiel is bundled with three implementations of AlphaZero: One in Python with Tensorflow, and two in C++ using Tensorflow and PyTorch [51] C++ APIs.

The Python implementation is claimed to be considerably slower than the Tensorflow C++ version since it does not support batching for inference and cannot utilize graphical processing units (GPU) for inference or training [56]. Hence, the C++ version should be used to take advantage of additional hardware and train significantly faster.

However, the current C++ implementation depends on some APIs that are not yet

fully supported on the open-source version of Tensorflow. Although we attempted to address this limitation in collaboration with OpenSpiel's primary engineers [57], we were unable to completely fix the problem and make the Tensorflow C++ version work on our systems. The PyTorch C++ implementation was added recently by the community [58] and was not available at the time this thesis was started.

We do not impose a clock-based or any other time-related restriction in the training and evaluation of our Hex agents. Therefore, the performance difference between C++ and Python implementations of the AlphaZero algorithm is irrelevant to our results.

5.1 Hyper-parameters Tuning

As with any machine learning model, models trained with AlphaZero-style algorithms have two types of parameters: model parameters and hyper-parameters. The former are the parameters that the model uses internally and are determined during training, such as the model weights. The latter are adjustable parameters that must be tuned manually before starting the training, like the number of layers in a neural network [59].

The most important hyper-parameters for configuring OpenSpiel's AlphaZero algorithm are listed in Table 5.1. On the same table, we also include the values that we think work best for training agents for Hex 6×6 and Hex 8×8. While finding optimal values for hyper-parameters is non-trivial and requires many years of experience [60], we derive our proposed values based on the literature [1, 61] and through consultation with other experts.

The rest of this section discusses hyper-parameters that require additional details: Learning rate, Dirichlet noise α, temperature and temperature drop, model type and dimensions, checkpoint frequency, actors and evaluators. Note that some hyper-parameters, *e.g.*, `uct_c`, `max_simulations` and `train_batch_size` are not discussed in this chapter, even though they are listed on Table 5.1.

Table 5.1: Most important hyper-parameters that configure OpenSpiel's AlphaZero algorithm.

Hyper-parameter	Description	Value for chess, shogi, Go [1]	Value for Tic-tac-toe [2]	Our value for Hex 6×6	Our value for Hex 8×8
`uct_c`	UCT's exploration constant. Used in calculating PUCT value.	Not listed	1	1.5	1.5
`max_simulations`	How many simulations to run.	1600	20	150	150
`train_batch_size`	Batch size for learning.	2048	128	512	512
`replay_buffer_size`	How many states to store in the replay buffer.	Not listed	2^{14}	2^{17}	2^{17}
`replay_buffer_reuse`	How many times to learn from each state.	Not listed	4	2	2
`learning_rate` (Section 5.1.1)	Learning rate.	0.2, 0.02, 0.002, 0.0002	10^{-2}	10^{-3}	10^{-3}
`weight_decay`	L2 regularization strength.	10^{-4}	10^{-4}	10^{-4}	10^{-4}
`policy_epsilon`	What noise epsilon to use.	0.25	1	0.25	0.25
`policy_alpha` (Section 5.1.2)	What Dirichlet noise alpha to use.	0.3 (chess), 0.15 (shogi), 0.03 (Go)	0.25	0.310	0.303
`temperature` (Section 5.1.3)	Temperature for final move selection.	1	1	1	1
`temperature_drop` (Section 5.1.3)	Drop the temperature to 0 after this many moves.	30	4	8	17
`nn_model` (Section 5.1.4)	What type of model should be used? (resnet, conv2d, mlp)	resnet	resnet	resnet	resnet
`nn_width` (Section 5.1.4)	How wide should the network be.	256	128	128	128
`nn_depth` (Section 5.1.4)	How deep should the network be.	20	2	7	9
`checkpoint_freq` (Section 5.1.5)	Save a checkpoint every N steps.	1000	25	1	1
`actors` (Section 5.1.6)	How many actors to run.	Not listed	4	15	15
`evaluators` (Section 5.1.7)	How many evaluators to run.	Not listed	4	1	1
`eval_levels` (Section 5.1.7)	The difficulty level of the built-in MCTS+Solver evaluator.	Not listed	7	5	5
`evaluation_window`	How many games to average results over.	400	50	100	100
`max_steps`	How many learn steps before exiting.	25000	25	100	200

5.1.1 Learning Rate

The learning rate corresponds to the same hyper-parameter on Momentum optimizer [62]. In Silver *et al.* [1], it is initially set to 0.2 for chess and shogi and later drops to 0.02, 0.002, and 0.0002 during training after 100K, 300K, and 500K steps, respectively. For Go, the initial rate is 0.02, which later drops to 0.002 and 0.0002 after 300K and 500K steps. In our training of Hex 6×6 and Hex 8×8, we use a value of 0.001 for the learning rate, which is the default value of OpenSpiel's AlphaZero implementation.

5.1.2 Dirichlet Noise α

Adding a Dirichlet noise ($Dir(\alpha)$) to the prior probabilities of the root node of MCTS was first proposed by Silver *et al.* in the original AlphaGo paper [63]. For each game, the noise is scaled in inverse proportion to the average number of legal moves in a typical position [61]. Since the exact formula is not mentioned in Silver *et al.* [63] or its succeeding papers [1, 4], we use polynomial regression to fit a polynomial to the existing values we have for chess, shogi and Go (listed on Table 5.2). Equation 5.1 shows the fitted fourth-degree polynomial in which x is average number of legal moves and y is the α value. Now we use Equation 5.2 to calculate branching factor $B(n)$ of board size of n as an approximation for the average number of legal moves in a typical position. Then we put the result as x in Equation 5.1 to calculate the α value. The last two rows of Table 5.2 show the α values for Hex 6×6 and Hex 8×8.

$$\begin{aligned} y = &-5.777\,144\,43 \times 10^{-20}x^4 - 3.709\,983\,09 \times 10^{-11}x^3 \\ &-4.285\,651\,27 \times 10^{-9}x^2 - 3.183\,094\,43 \times 10^{-7}x \\ &1.201\,152\,67 \times 10^{-9} \end{aligned} \tag{5.1}$$

$$B(n) = \frac{n^2}{2} \tag{5.2}$$

Table 5.2: Values for Dirichlet noise α.

Game	**Average number of legal moves**	α **value**
chess	35 [61]	0.3 [1]
shogi	92 [61]	0.15 [1]
Go	250 [61]	0.03 [1]
Hex 6×6	18	0.310
Hex 8×8	32	0.303

5.1.3 Temperature and Temperature Drop

The MCTS keeps track of the number of times each potential next move is visited during a round of simulations as *visit count*. The search work such that the good moves are eventually visited more often than bad moves and have a greater visit count.

When a standard MCTS agent wants to make a decision about the next move, visit counts are normalized and used as a distribution to choose the best move [61]. However, AlphaZero's MCTS agent uses a temperature hyper-parameter (τ) to exponentiate the distribution proportional to $N^{-\tau}$, where N is the visit count of each move.

The temperature hyper-parameter ensures diversity in the set of encountered positions [1]. When $\tau = 1$, a random move is selected according to the visit counts distribution. And when an infinitesimal temperature is used ($\tau \to 0$), the best move with the greatest visit count is chosen. For our Hex agents, we use the same value of $\tau = 1$ that was used for chess, shogi and Go in Silver *et al.* [1].

Another hyper-parameter called *temperature drop* (τ_{drop}) controls the temperature value based on the total number of moves played so far in the game. For the first τ_{drop} moves of each game, the initial temperature value (*e.g.*, $\tau = 1$) is used, and after that, the temperature will drop to 0.

Table 5.3: Values for temperature drop hyper-parameter.

Game	**Average game length**	**Temperature drop**
Go	150 [66]	30 [1]
Connect 4	36 [66]	10 [65]
Tic-tac-toe	9 [66]	4 [2]
Hex 6×6	31	8
Hex 8×8	56	17

The current literature does not recommend any particular way to calculate the temperature drop hyper-parameter to the best of our knowledge. However, considering the effect of temperature hyper-parameter on move selection, we argue that it is reasonable to say that τ_{drop} should not be greater than the game length. Otherwise, the agent continues picking random moves towards the end of the game when the best moves with maximum visit count are preferred.

Inspired by Jans [64], we calculate temperature drop based on the average game length. Again, since the exact relationship between this hyper-parameter and the average game length is not studied in the literature, we use polynomial regression to devise a value for Hex 6×6 and Hex 8×8 according to AlphaZero implementations of Go [1], Connect 4 [65] and Tic-tac-toe [2]. Table 5.3 depicts average game lengths and temperature drop values for the mentioned games. Equation 5.3 shows the fitted third-degree polynomial in which x is average game length and y is the τ_{drop} value. We use batches of 10000 games played between two random-play agents to calculate the average game length for Hex, giving us 31 for Hex 6×6 and 56 for Hex 8×8. The last two rows of Table 5.3 show the τ_{drop} values calculated using Equation 5.3 for Hex 6×6 and Hex 8×8.

$$\begin{aligned} y = &-5.789\,893\,71 \times 10^{-16}x^3 + 1.709\,340\,71 \times 10^{-4}x^2 \\ &+ 6.200\,790\,34 \times 10^{-3}x - 3.350\,045\,70 \times 10^{-5} \end{aligned} \tag{5.3}$$

5.1.4 Model Type and Dimensions

OpenSpiel AlphaZero provides three dual-headed residual neural network architectures: a multilayer perceptron (MLP) network, a residual network, and a convolutional network without residual connections [64]. The residual network architecture works well for the games of Go, chess, and shogi [1], mainly due to its immunity against the Vanishing Gradient Problem in deep neural networks [64, 67, 68]. Therefore, we use the same architecture for Hex.

It is possible to configure the width and depth of the network architecture. In Silver *et al.* [1], a network with 256 filters and 20 layers is used for Go 19×19. Since Hex 6×6 and Hex 8×8 are much smaller than Go 19×19 we use only 128 filters. Having 20 layers for Go 19×19 inspires us to approximate layer count based on the board size. So we use $D = N + 1$ in which D is the number of layers (depth) of the neural network and N is board size to 7 layers for Hex 6×6 and 9 layers for Hex 8×8.

5.1.5 Checkpoint Frequency

OpenSpiel AlphaZero can be set to save a snapshot of the trained model as a checkpoint in specific intervals. Each checkpoint is a complete model that can be used to play the game independently. Since we want to analyze the training progress by looking at the performance gain of each training step (Section 7.1), we set this hyper-parameter to 1 to have one playable model per step.

5.1.6 Actors

The training script creates a set of actor processes to perform self-play games and puts game states in the replay buffer. Having more actors results in faster training since it increases the speed of self-play simulations and training data generation.

However, considering that each actor is maintained by a process in OpenSpiel AlphaZero Python implementation, we should not have more actors than the number of available processing cores. Otherwise, the performance will decrease due to the

continued context switches in CPU cores.

We limit the number of actors to 15 because our training machines have 16 CPU cores. The remaining core is dedicated to an evaluator process which will be discussed in Section 5.1.7. For detailed information on hardware configuration of our training machines, see Section 7.1.

5.1.7 Evaluators

It is possible to have a set of evaluator processes continually evaluate the model as it is being trained to see how training is progressing. These evaluators play games against vanilla MCTS+Solver players with scalable strength. The strength of the MCTS opponent is correlated with the number of simulations it is allowed to perform per move. For each MCTS player of a level n, the simulations count, $S(n)$, is calculated using Equation 5.4 [64]. By setting OpenSpiel AlphaZero `eval_levels` hyper-parameter, we can specify the strength of MCTS opponents.

$$S(n) = \text{max\_simulations} \times 10^{\frac{n}{2}} \tag{5.4}$$

Increasing the number of evaluators means more test games to be played, which can result in better evaluation accuracy. Similar to the number of actors, the number of evaluators should not be greater than number of available CPU cores. In fact, the sum of actors and evaluators should be less than or equal to the number of processing cores.

In this thesis, we use test positions to evaluate our Hex agents (Chapter 6) and do not rely on OpenSpiel AlphaZero's built-in evaluator. Therefore, we use one evaluator process during training only to get a general idea about the training progress.

5.2 Forced Exploration of Search Space

Our performance analysis shows that although our Hex agents can perform reasonably well against standard MCTS players like what is shipped with OpenSpiel framework,

they sometimes make unwise decisions in less-common game states and lose frequently to stronger players like MoHex. One possible reason could be over-fitting the proof tree and lack of proper search space exploration due to insufficient training.

AlphaZero, by design, learns to play very well on its own data, *i.e.*, the game positions that it has seen during training. But it does not quite know what to do when it faces a completely unfamiliar position. So it tries to generalize its learned knowledge in order to find the right move.

In cases where AlphaZero has been trained for a long time with enough computing power on enormous positions, like in Silver *et al.* [1], it will eventually choose the correct move. However, with limited training computing power and time, it may not be able to generalize its knowledge in unknown positions properly.

Since most research teams do not have access to powerful computers and are on tight timing budgets, we propose the following technique as an alternative to the long-run trainings.

First, we create a database of opening positions containing up to first three game moves. Then, for each training game, we select one position from the list randomly and enforce it as the opening game state. This technique expands AlphaZero search space exploration, and makes it visit and evaluate positions that otherwise might have been visited late.

In the rest of this section, first we discuss the openings database (Section 5.2.1). Then, we describe the two different lists of openings: all winning moves, and all legal moves, and discuss the way to choose positions from a list to have the best result (Section 5.2.2). A comparison between models trained on opening databases with various move and position selections is provided in Chapter 7.

5.2.1 Openings Database Preparation

The opening database has one table and stores one row per position. As shown on Listing 5.1, the table comprises one column for each move and one for storing current

```sql
CREATE TABLE positions (
        id INTEGER
                CONSTRAINT first_moves_pk
                        PRIMARY KEY,
        move TEXT NOT NULL,
        player INTEGER NOT NULL,
        first_move INTEGER
                REFERENCES positions,
        second_move INTEGER
                REFERENCES positions
);
```

Listing 5.1: The opening database schema consisting of one table with one column per move.

PtoP. The `move` column always contains the last move of a position in textual format, *e.g.*, `a1`. The `first_move` and `second_move` contain foreign keys to a previous position, *i.e.*, the last and the second-to-last positions, respectively. Therefore, for positions with just one stone on board, only column `move` is filled with the actual move. For positions with two stones, both `move` and `first_move` columns are filled. And for positions with three stones, all columns are filled.

With the mentioned database design, selecting a random position is as easy as randomly picking one row from the table. This can be done quickly and has no negative impact on the training time.

When choosing moves for the opening database, we can consider either all legal moves from a position (*e.g.*, the blank board) or only a subset of winning moves available to the current player. A solver like MoHex [19] can be used to find all winning moves early in the game. For Hex 6×6, we have a total of 541 positions derived from winning moves and a total of 44136 ($= 36 \times 35 \times 34$) positions from all legal moves. Solving and finding all wining moves in 8×8 positions with few stones on board could take considerable time even for MoHex (Section 3.1). Therefore, for Hex 8×8, we cannot extract positions based on winning moves, and we use a database of 254080 ($= 64 \times 63 \times 62$) positions generated by following all legal moves. Table 5.4

Table 5.4: Frequency of opening positions extracted for Hex 6×6 and Hex 8×8.

Game	Move selection criteria	1-stone positions	2-stone positions	3-stone positions	Total
Hex 6×6	Winning moves only	9	58	474	**541**
Hex 6×6	All legal moves	36	1260	42840	**44136**
Hex 8×8	All legal moves	64	4032	249984	**254080**

explains the frequency of opening positions in more detail.

By limiting the openings to winning moves only, we cap the exploration to a subset of all positions resulting from constantly playing winning moves. An agent trained over this kind of openings may learn the winning opening moves better since it repeatedly sees them. However, it may not play well against imperfect players who may play a losing move in the beginning of game. This is because the agent has not seen enough positions that resulted from losing moves. We include all legal moves in the opening database to combat this limitation.

5.2.2 Different Openings Combinations

Considering that the opening database includes positions with one, two, and three stones on board, the question is whether we should use all of them in training or not. We consider four possible ways to use the forced openings:

- **A1**: one-move openings only
- **A2**: two-move openings only
- **A3**: three-move openings only
- **A123**: all one-, two- and three-move openings

One-move openings guide AlphaZero to try a variety of first moves throughout the training instead of sticking with what it eventually learns as the best opening move.

This way, it learns the consequences of different opening moves and the reasonable second move in response to a possibly losing first move made by the opponent. Considering that the number of all first moves is relatively small (< 1000), it is possible to cover all of them multiple times during training.

Two-move openings include the first two moves of a game and can diversify the training games up to the second move. Compared to one-move openings, they are greater in count (1260 for Hex 6×6 and 4032 for Hex 8×8), so it is harder to cover all of them more than once during training. The risk of not encountering some two-move openings multiple times is that the agent may not learn the right first, second or third moves since it has not had the chance to see the results of different choices.

Similar to two-move openings, three-move openings include the first three moves and improve exploration up to the third move. We have by far a higher number of them (42840 for Hex 6×6 and 249984 for Hex 8×8), and they are even harder to cover multiple times than two-move openings. A similar risk of not properly learning consequences comes with using exclusively three-move openings.

So far, we have discussed the risks of using only two- or three-move openings, and described the limited advantage of one-move openings on its own. Therefore, we argue that it is reasonable to include the three categories at the same time to see if it maximizes the benefits while minimizing the risks.

We train our agents with all four possible combinations to see which one leads to a greater improvement in terms of training speed and accuracy. We also train one agent with exact same hyper-parameters but without any forced openings as the baseline to see if including forced openings always improves training. Finally, we should note that we use random uniform distribution to select an opening position per game to ensure that all positions are equally selected.

Our evaluation results, as discussed in Chapter 7, show that including forced openings generally improves the winning-move selection ratios compared to the baseline. But we found no significant difference among the four combinations.

5.3 Concluding Remarks

In this chapter, we briefly explained the important hyper-parameters that can be used to configure OpenSpiel's implementation of the AlphaZero algorithm. We also provided the values we used for training AlphaZero-style Hex agents for 6×6 and 8×8 board sizes and described how we calculated them. In Section 5.2, we discussed how insufficient search space exploration and the possible over-fitting the proof tree could result in unwise moves by our Hex agents. We proposed using forced-move openings in training to expand the explored search space to address this issue. We described our method for creating and using a database of one-, two- and three-move openings. We also discussed the risks and limitations of using each set of opening moves separately and suggested using a combination of them instead.

The next chapter will discuss our evaluation methodology based on test positions. In Chapter 7, we report the results of using forced openings in training Hex agents as measured by a test suite of positions.

Chapter 6

Evaluation Methodology

In the previous chapter, we described how we trained AlphaZero agents for the game of Hex. In this chapter, we describe our methodology for evaluating the performance of these agents. In Chapter 7, we present and discuss the results.

In order to understand how well the trained agents perform and to provide a way to compare them systematically, we propose the following two techniques while being primarily focused on the second one:

1. **Head-to-head Games**: In this method, an agent plays a fixed number of complete games, *e.g.*, 1000 games, against a strong or perfect player such as MoHex with its DFPN solver (Section 3.1).

2. **Test Positions**: In this method, each agent makes one move against each of a set of test positions, and is assigned an overall score in $[0, 1]$ that measures the fraction of time it selected a winning move.

We do not rely solely on head-to-head games to evaluate a Hex agent. This is because they are expensive in terms of computing power and runtime, mainly due to high chance of seeing duplicate games during evaluation. Therefore, to cover as many unique game scenarios as possible, hundreds or thousands of head-to-head games are needed. Test positions can be used as a shortcut for a more practical evaluation. Evaluation using test positions takes a fraction of time required for playing complete games, as the agent plays only one move per each test position.

In the rest of this chapter, we describe the two techniques in detail. In the next chapter, we present the evaluation results for our Hex 6×6 and Hex 8×8 agents.

6.1 Head-to-head Games

One traditional way to evaluate the strength of an agent is by looking at head-to-head games it played versus a strong or perfect player. For example, for Hex boards up to 8×8, it is shown that MoHex can play near-perfectly with its DFPN solver [16, 19]. For this reason, we use MoHex with DFPN solver as the opponent in this method. To remove the inherent first player advantage, players should switch turns in half of all played games.

Since one side of the game is a strong or perfect player who tends to play the same winning moves repeatedly, a random opening move should be enforced in all games. Similar to enforcing move openings during training to expand the search space (Section 5.2), a random opening diversifies game records during evaluation and pushes both players to explore new moves. Considering the number of possible game positions of a Hex board, we arguably need hundreds or thousands of games for a proper evaluation. For Hex 6×6 and 8×8, we use batches of 1000 games where the agent in one half of them (500 games) plays black and in the other half plays white.

Even with a random opening, it is still possible to have duplicate game records. All duplicates are removed before further analysis. Once we have a set of unique games, we count the number of wins for each side of the game.

6.2 Test Positions

A test is like a multiple-choice question consists of the following:

- **Question**: a game position
- **Choices**: all legal moves at that game position for the current player-to-play (PtoP)

- **Correct answer(s)**: all winning move for the current PtoP as proven by Mo-Hex's solver (Section 6.2.2)

To evaluate an agent with a group of test positions, we conduct the following procedure for each test:

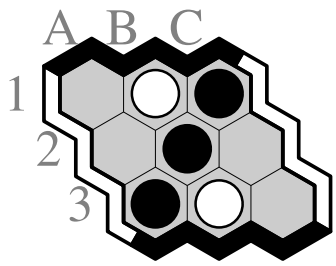
1. A new game is created.
2. All moves required to reach that position are applied on the board.
3. The agent is asked to play a move.
4. If the chosen move exists in the set of winning moves, the agent gets a reward of +1. Otherwise, it gets nothing.

To calculate the final score of an agent, called *winning-move selection ratio* (WMSR), we divide the total reward by the number of tested positions.

A set of test positions could serve as an accurate way for performance evaluation and a proper alternative for head-to-head games only if they have been picked precisely. The rest of this section describes our approach for finding proper test positions. First, we explain our multi-step process for finding candidate positions (Section 6.2.1), and second, we describe the criteria for choosing the right positions from the pool of candidates (Section 6.2.2). Finally, in Section 6.2.3, we discuss our test suites for Hex 6×6 and Hex 8×8.

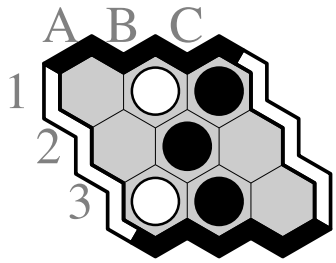
6.2.1 Candidate Position Extraction

The candidate position extraction process begins with listing all unique positions that occurred in a set of played games, referred to as *source games*. Source games are encoded as game records in the Smart Game Format (SGF) notation. Figure 6.1 visualizes two sample 3×3 games and lists their SGF-encoded game records. Even though we are working on 6×6 and 8×8 boards in this thesis, we use 3×3 games




```
(;AP[HexGui:0.9.GIT]FF[4]GM[11]SZ[3];B[b2];W[b1];B[c1];W[b3];B[a3])
```

(a)




```
(;AP[HexGui:0.9.GIT]FF[4]GM[11]SZ[3];B[c1];W[b1];B[b2];W[a3];B[b3])
```

(b)

Figure 6.1: Two sample 3×3 games and their SGF-encoded game records

in this chapter's examples to simplify the discussion. For examples of 6×6 and 8×8 games, see Figures 6.9 to 6.12.

We extract source games from real Hex games involving MoHex, is a strong Hex player that plays near-perfectly for boards up to 8×8 [16, 19], as opposed to the made-up games between random players. Detailed information on our Hex 6×6 and 8×8 source games is provided in Section 6.2.3. We use real games since only a subset of all legal moves, and hence, all possible positions are likely to be seen in competitive games. This is because Hex is a win-for-black game, and black can win by constantly playing the winning move. In contrast, all the opponent needs for winning is to block black's moves [8].

Listing 6.1 provides our algorithm for processing game records and extracting all unique positions. In other words, it shows how we find all the unique ways of reaching

source games' final position from an empty board. For each game record in the set of the source games, we do the following (lines $4-10$). First, at line 5, the entire game record is copied as a move sequence. Then, in the loop defined by lines $6-10$, we take out moves from the end of the move sequence one by one (line 10) and store the result as a new move sequence until the initial game position, an empty board, is reached (line 6). At line 7, we use a helper function to convert the move sequence to an SGF-encoded game position by removing moves ordering. If the returned game position was not found in the final set of game positions, it gets added there (lines $8-9$). Conversion of a move sequence to a game position is simply done by marking all odd moves as black stones, and all even moves as white stones (lines $19-20$). To encode a position in SGF notation, we also need to include the current PtoP, which is easily determined based on the parity of the number of stones on board (line 17).

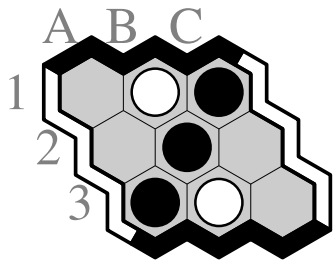
Figures 6.2 and 6.3 show all unique positions extracted from the two sample games presented in Figures 6.1a and 6.1b, respectively. Each sample game has five stones on board. So by removing each of the five moves, we extract a total of six game positions (including the initial empty board) for each game.

By definition, we only consider the final arrangement of stones on the board for each game position. So two positions that have the same stone arrangements but resulted from different move sequences are marked as duplicates. Figure 6.4 shows two duplicate positions found while processing the previously mentioned sample games. Note that Figure 6.4b shows an empty board with no stones which is a valid game position.

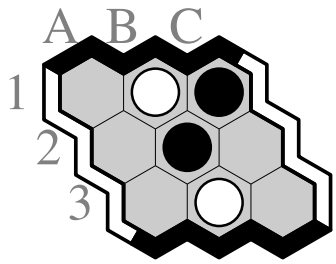
Now that we have a set of unique positions, we need to find all winning moves for them. While the position extraction process starts is about going backwards from an end position, the winning move finding is about going forwards from a given position. We use MoHex and its DFPN solver [16, 19, 40] for finding all winning moves for a position. The DFPN solver is hosted on Compute Canada's Cedar cluster [69] and uses 24 processors working at 2.1 GHz frequency. It is allowed to spend up to 5

```
def extract_unique_positions(R):
  # R is the set of source game records
  P = set() # P is the set of all unique positions
  for r in R:
    move_seq = r
    while move_seq is not empty:  # an empty value means the blank board
     p = make_position(move_seq)
     if p not in P:
      P.append(p)
     move_seq = move_seq[:-1]  # remove the last move from sequence
  return P

def make_position(move_seq):
  # move_seq is a move sequence
  # move sequence is BtoP if it has even number of stones on board,
  #  and WtoP otherwise.
  player_to_play = len(move_seq) % 2

  black_moves = position[0::2]  # all odd moves have been made by Black
  white_moves = position[1::2]  # all even moves have been made by White

  # return the position in SGF notation
  return encode_position_in_sgf(black_moves, white_moves, player_to_play)
```

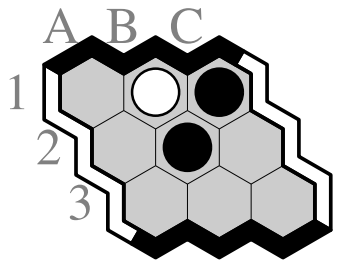
Listing 6.1: Pseudocode for unique positions extraction algorithm.



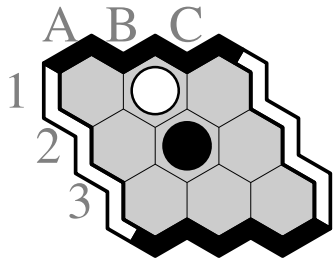
(a) Original board state at the end of the game



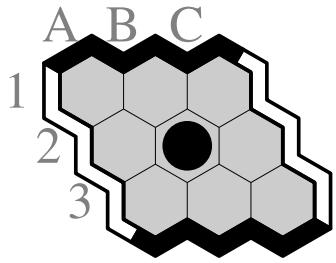
(b) Board state after removing the last move



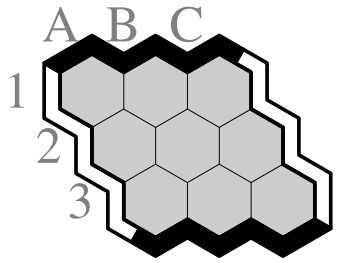
(c) Board state after removing the last 2 moves



(d) Board state after removing the last 3 move

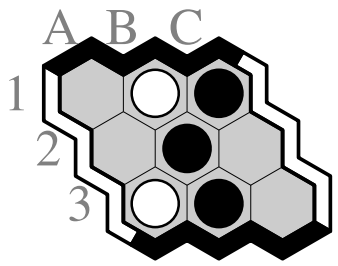


(e) Board state after removing the last 4 moves

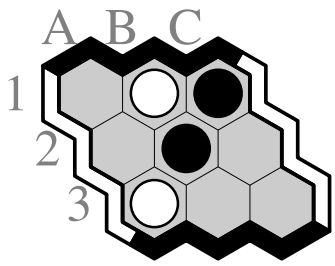


(f) Board state after removing all moves

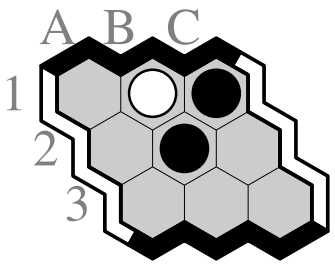
Figure 6.2: All unique positions extracted from Figure 6.1a



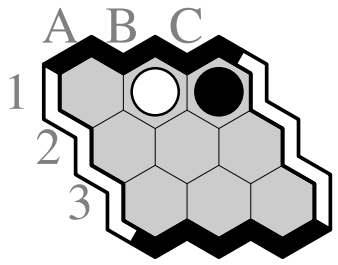
(a) Original board state at the end of the game



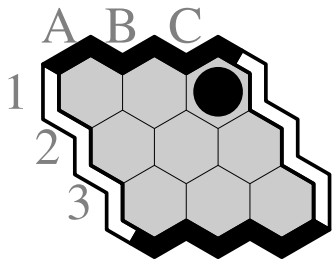
(b) Board state after removing the last move



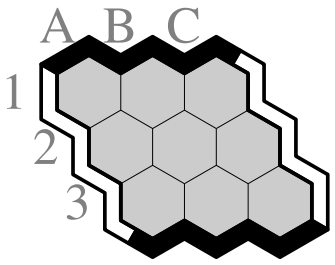
(c) Board state after removing the last 2 moves



(d) Board state after removing the last 3 move

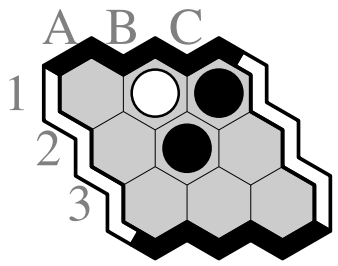


(e) Board state after removing the last 4 moves

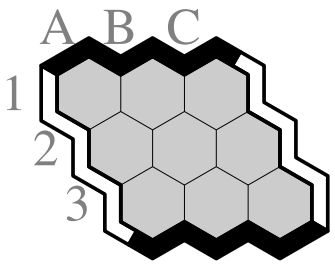


(f) Board state after removing all moves

Figure 6.3: All unique positions extracted from Figure 6.1b



(a)



(b)

Figure 6.4: Two common positions between the provided sample games in Figure 6.1 that are marked as duplicate by the position extraction algorithm.

minutes on any given position.

The MoHex's command-line interface has a dedicated command for finding winning moves for a given position (`dfpn-solver-find-winning`). While every move returned by this command is a proven winning move, it does not necessarily return all winning moves. In other words, it only returns a subset of all winning moves, and if one of available legal moves is not present in the return set, we cannot say whether it was a losing move or not. We cannot find a way to tell MoHex to return all wining moves without substantial code changes. Therefore, we propose an alternative way by solving the resulting position of playing every available legal move.

Our alternative method uses the retrograde analysis (Section 2.4) to solve positions backwards. Thus, we look at positions that are closer to the end of the game first. Listing 6.2 describes our algorithm in pseudocode.

First, we sort the set of all unique positions by the number of stones on board in descending order. This way, the positions with more stones come at the beginning of the list. Second, we iterate over the sorted list and find all winning moves for each position. Since we cannot use MoHex's winning moves finder command, we check every legal move and ask the solver to solve the position resulting from playing that move. If the new position was winning for the current player, we would add the originating move to *the winning set*. Otherwise, we conclude that the originating move was a loser and place it in *the losing set*.

Since we start from the most-crowded positions at the end of the game and head towards the beginning of the game, it is likely to revisit some positions while checking their parent positions. In that case, we do not need to ask the solver as we already know the answer. The total processing time is reduced considerably by following this technique.

The output of this algorithm is a set of solved positions which is referred to as the set of *candidate positions*. Based on the number of source games, we could have thousands of candidate positions. We cannot use all of them in a test suite

since checking a great number of positions takes a long time. So we should pick a fraction of them that can identify the agent's strengths and weaknesses as accurately as possible. In the next subsection, we explain some criteria to choose the best test positions from a pool of candidates.

6.2.2 Test Positions Selection

As described earlier, the extraction process can generate many candidates as it goes through all unique positions that occurred in source games. However, not every candidate position is of distinguishing quality that can properly evaluate an agent. Thus, we define the following criteria for sorting candidates and selecting the most challenging ones.

1. **Having at least one winning move.** When there is no winning move for the current player, all legal moves are considered as losing. In this case, playing any of them results in losing the game, assuming the opponent plays a winning move in response. So there is no definitive right move and the position cannot be used in the test suite.

2. **Having only a few winning moves.** When many of the legal moves are winning, the player can quickly find one winning move and go on. Therefore, challenging positions are usually the ones that have just one or very few winning moves.

3. **Taking more time to be fully solved.** We consider the computational effort required to completely solve a position, *i.e.*, solve all of its legal moves, as an indicator for its difficulty. As a proxy for computational effort, we examine the time it takes to solve every legal move. We argue that having a higher median solving time per move means a position is harder to solve and hence, a better candidate for the test suite.

```
def find_winning_moves(P):
  # Input is the list of unique positions.
  # Output is a dictionary of solved positions
  #  in which keys are positions and values are winning sets
  solved_positions = dict()

  # Sort list of positions by number of stones in descending order.
  P.sort(key=lambda p: len(p.stones), reverse=True)
  for p in P:
    board = Game() # Create a new game.

    # Put Black and White stones on a blank board as described by position.
    board.put_stones(p.stones)

    # Find all legal moves for current PtoP.
    legal_moves = board.get_legal_moves(p.player)

    winning_set = set()  # Set of winning moves
    losing_set = set()  # Set of losing moves

    for move in legal_moves:
      board.apply(move, p.player) # Play the one legal move for current PtoP.

      if board in solved_positions:
        # When we have already solved the new board state,
        #  derive who wins by looking at the set of winning moves.
        # A non-empty set means the player wins
        winner = who_wins(solved_positions.get(board))
      else:
        # When the new board state has not yet been visited,
        #  ask MoHex to see who wins.
        winner = MoHex().solve(board)

      if winner == p.player:
        winning_set.add(move)
      else:
        losing_set.add(move)

      # Undo the last move, so we can reuse the board for the next legal move.
      board.undo()

    # Save the winning and losing sets in the output dictionary.
    solved_positions[board] = winning_set
  return solved_positions
```

Listing 6.2: Pseudocode for winning moves finder algorithm.

We use median solving time instead of the total solving time of all moves. This is because the total time correlates with the number of legal moves and cannot fairly compare positions with different moves. We also decide to use the median instead of the mean since we find that the distribution of solving times generally is not normal. Figures 6.5 and 6.6 display the solving time histograms for 10 randomly selected candidate positions for Hex 6×6 and Hex 8×8.

A high-quality test suite should evaluate agents' performance in different stages of the game. Therefore, in addition to the three criteria, we also want the test positions to be as diverse as possible with regard to the number of stones. So we put the candidates that match the criteria in separate bins by the stones count.

Listing 6.3 provides the pseudocode for applying the selection criteria on candidate positions and picking test positions. In lines $10-11$, we apply the limitation on the number of winning moves on candidate positions. The default minimum is 1, but it can change as needed. In line 14, we put remaining candidates in bins based on the number of stones on board.

In lines $16-19$, we sort positions in each bin by median solving time of their winning moves (Figures 6.5 and 6.6). We only consider the solving time of the winning moves and ignore losing moves. This is because losing moves can be found quickly as MoHex can identify them without searching and just by using its virtual connection/inferior detection engine. So it makes sense to remove them from calculation to prevent a great number of near-zero values from affecting the median.

Finally, in lines $21-28$, we iterate over bins and pick positions with the highest median solving time as test positions. Not all bins have the same number of candidates, so we may encounter an empty bin during iteration. Line 24 checks this situation and skips the empty bins.

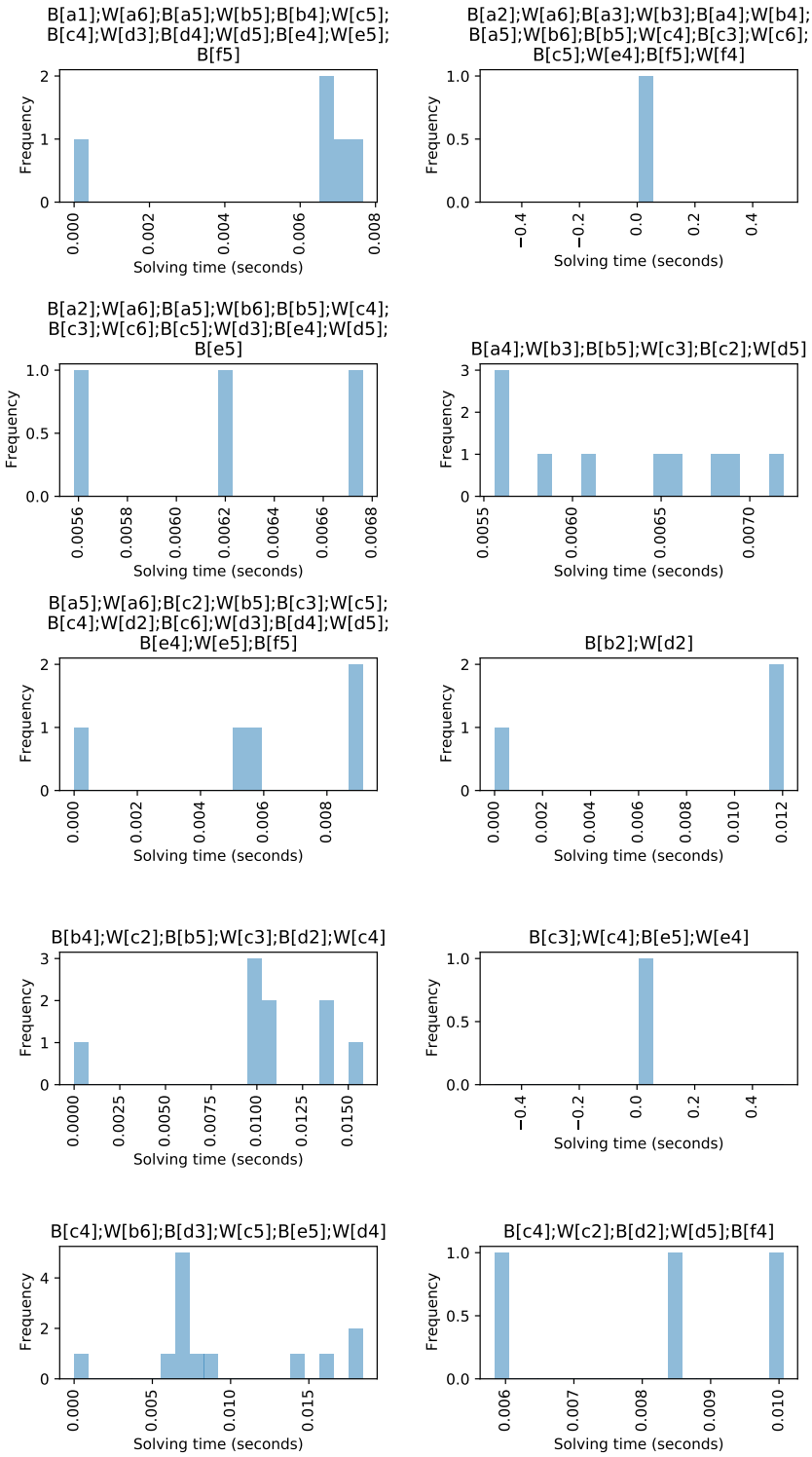


Figure 6.5: Solving time distributions for 10 randomly selected candidate positions for Hex 6×6.

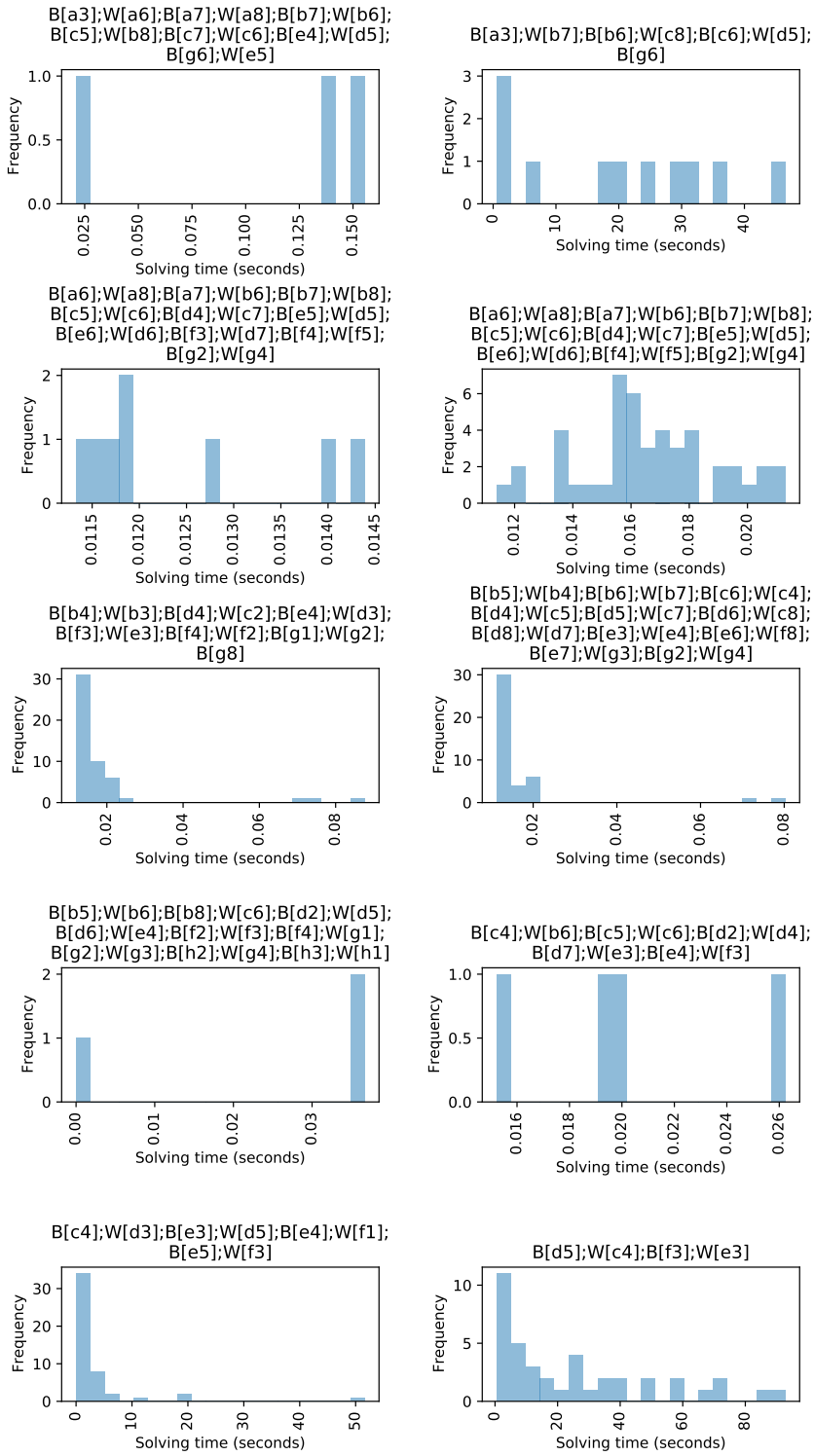


Figure 6.6: Solving time distributions for 10 randomly selected candidate positions for Hex 8×8.

```
def pick_positions(C, selection_size, min_wins=1, max_wins=4):
  # C is the list of all solved candidate positions
  # selection_size indicates how many positions
  #  should be selected from the list.
  # Output is a list of test positions selected according to the criteria
  test_positions = []

  # Filter out any position that does not have minimum number of wins
  #  or has more wins than the allowed maximum.
  winning_candidates = C.filter(lambda c:
                                  min_wins < len(c.winning_moves) < max_wins)

  # Put candidate positions in separate bins by number of stones on board
  bins, binned_candidates = bin_candidates(winning_candidates)

  for bin_ in binned_candidates:
    # Sort candidates by median solving time of winning moves
    #  in ascending order
    bin_.sort(key=lambda c: c.median_time_winning_moves)

  bin_index = 0  # Start picking from the first bin
  while len(test_positions) < selection_size:
    # Bin may have been already emptied and should be skipped.
    if len(binned_candidates[bin_index]) > 0:
      # Last item in the sorted bin has the greatest solving time.
      position = binned_candidates[bin_index].pop()  # Remove the last item.
      test_positions.append(position)  # Add position to the output list
    bin_index = (bin_index + 1) % len(bins)  # Move to the next bin

  return test_positions
```

Listing 6.3: Pseudocode for test positions selection algorithm.

6.2.3 Test Positions for Hex 6×6 and Hex 8×8

In this subsection, we discuss how we prepare test suites for Hex 6×6 and Hex 8×8 agents according to the process explained in the previous subsections.

The candidate position extraction process starts with collecting source games. For each board size, our best agent plays 500 games against MoHex as black and another 500 games as white. Although we use random opening for all games to avoid duplicate games as much as possible, only 36 and 39 games are found unique for Hex 6×6 playing as Black and White, respectively. The number of unique games for Hex 8×8 is 249 for Black and 219 for White. Table 6.1 summarizes the source games we used in the extraction process.

Table 6.1: Number of games used as source games in test suite generation.

Game	**Games played against MoHex**		**Unique games**	
	Black	White	Black	White
Hex 6×6	500	500	36	39
Hex 8×8	500	500	249	219

From the collected source games, we extract a total of 1344 and 10547 unique positions for Hex 6×6 and Hex 8×8, respectively. As listed in Table 6.2, about 34000 and 496000 legal moves are checked out and solved to find all winning and losing moves for Hex 6×6 and Hex 8×8. Based on these checks, we removed about half of all positions from consideration as they were found to be losing moves. The ratio of winning to losing moves is 0.18 ($= \frac{5246}{28506}$) for Hex 6×6 and 0.25 ($= \frac{100505}{395304}$) for Hex 8×8. Although we use backward search for both board sizes, we found it primarily helpful in 8×8, where solving game states is generally hard and takes considerable time.

Figure 6.7 depicts the frequency distribution of unique positions to number of stones on board for all positions solved in the extraction process. As we see, for both

Table 6.2: Detailed numbers of positions and moves used in test suite generation.

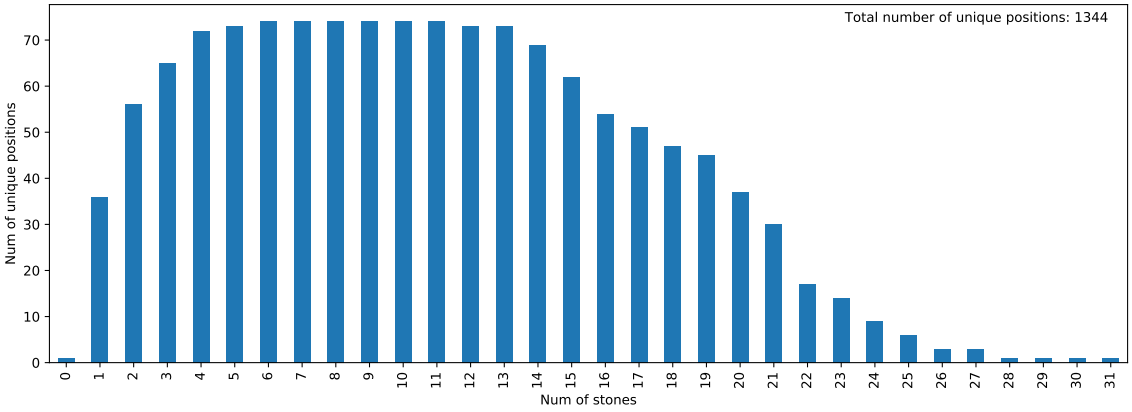
Game	**Unique Positions**			**Moves** (solved with retrograde analysis)		
	Total	Winning	Losing	Total	Winning	Losing
Hex 6×6	1344	653	691	33752	5246	28506
Hex 8×8	10547	5285	5262	495854 (1971)	100550 (972)	395304 (999)

board sizes, the positions are almost evenly distributed. While positions with the lowest and the highest number of stones are less frequent as they are close to the beginning and end of games, the majority of positions for 6×6 and 8×8 have between $4-14$ and $7-20$ stones on board, respectively.

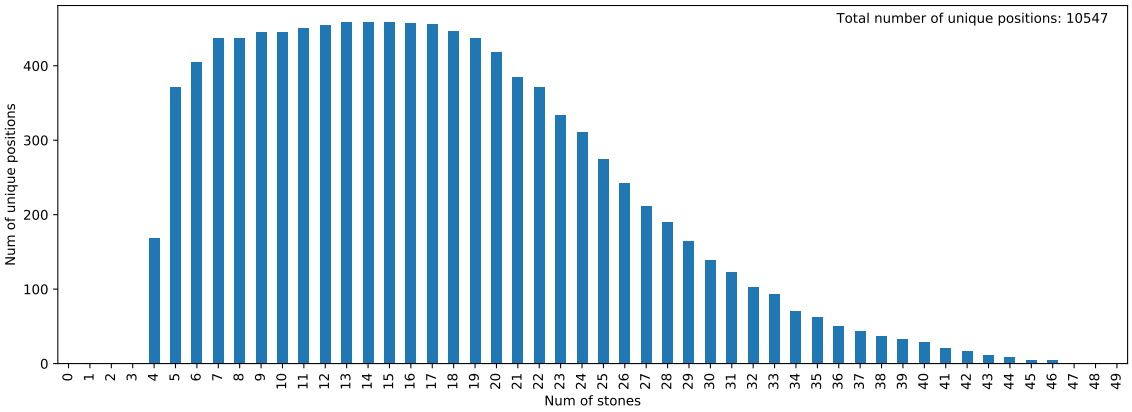
Now that we have a set of winning positions, we apply the limitation on the number of winning moves. We determine the upper limit for winning moves count based on test suite size. In general, we decided that we want a maximum of 250 test positions for each player. Also, in order to maintain the quality of positions, we should select no more than half of all candidate positions. So the formula for devising the number of test positions for each player can be written as $min\left(250, \frac{N}{2}\right)$ where N is the number of winning positions.

For Hex 8×8, we have 2650 Black-to-Win (BtoW) 2635 White-to-Win (WtoW) positions. So the test suite size for Hex 8×8 is 250. For Hex 6×6, although we have 443 BtoW positions, the number of WtoW positions is only 210. So Hex 6×6's test suite size would be 105. To simplify further evaluation results analysis, we decide to have 100 test positions instead of 105.

The frequency distribution of positions to the number of winning moves is displayed in Figure 6.8. Having set the test suite size, we can use this distribution to determine the limit for number of winning moves. In addition to setting a maximum, we would exclude positions with only one winning move if we have enough positions. This is



(a) Among 1344 unique positions found for Hex 6×6.



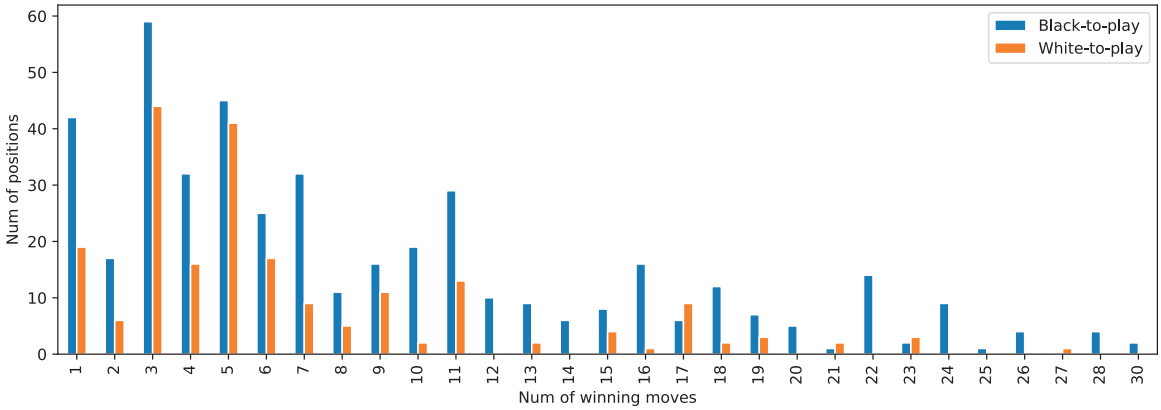
(b) Among 10547 unique positions found for Hex 8×8.

Figure 6.7: Frequency distribution of unique positions to number of stones on board in candidate extraction process.

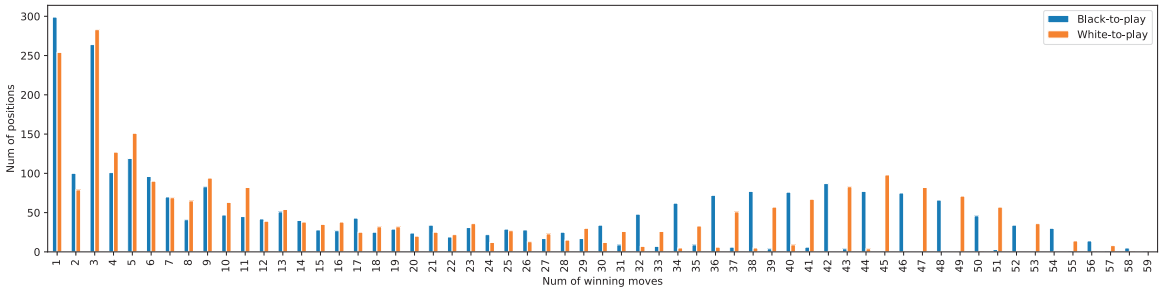
because, in those positions it is often relatively obvious what the winning move is.

For Hex 8×8, we have many winning positions with more than one winning move. So we can select 250 required test positions from candidates with 2, 3, or 4 winning moves. However, for Hex 6×6, winning positions are already quite limited. So we include all candidates with $1-5$ winning moves. It is better to include one-winning-move positions instead of positions with six or more winning moves, since the more winning moves we have in a position, the easier it would be to solve (Section 6.2.2). To apply the third criteria in test positions selection process, we sort filtered candidates by median solving time in descending order.

The complete list of 200 and 500 test positions for Hex 6×6 and Hex 8×8 can be found be in Appendix A. Figures 6.9 to 6.12 visualize ten sample test positions for each player and board size. The winning moves are marked with small dots in all figures.

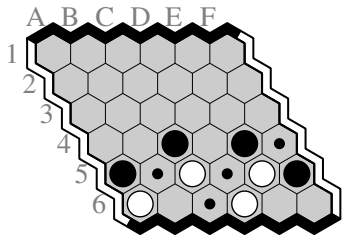


(a) Of the selected 6×6 positions, 653 are winning: 443 BtoW, 210 WtoW.

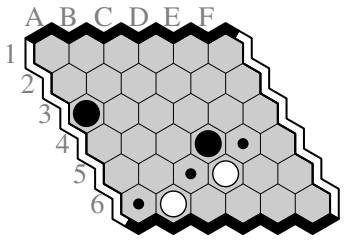


(b) Of the selected 8×8 positions, 5285 are winning: 2650 BtoW, 2635 WtoW.

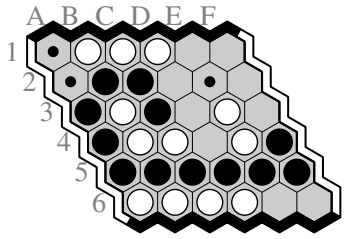
Figure 6.8: Frequency distribution of unique positions to number of winning moves in candidate extraction process.



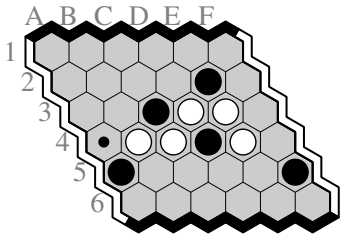
(a)



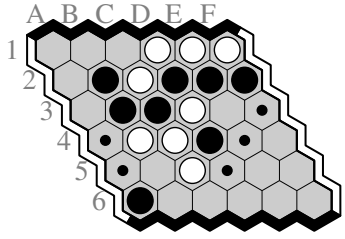
(b)



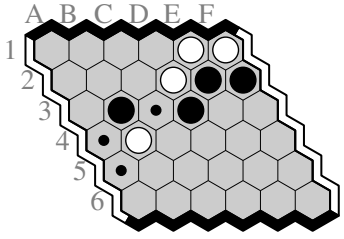
(c)



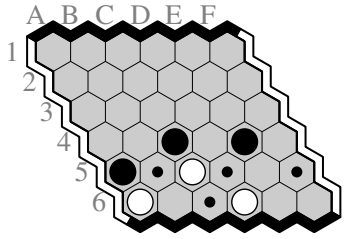
(d)



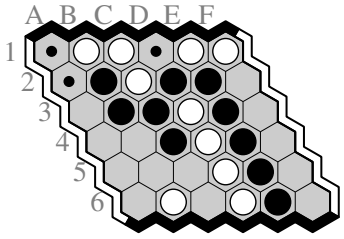
(e)



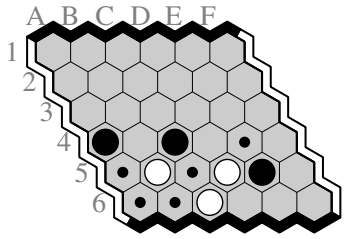
(f)



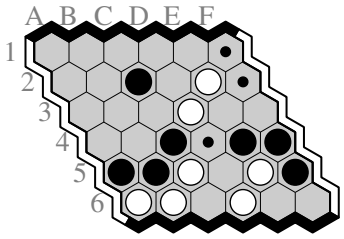
(g)



(h)

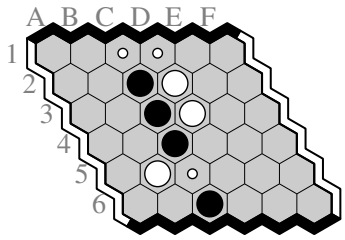


(i)

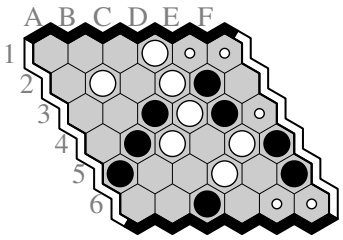


(j)

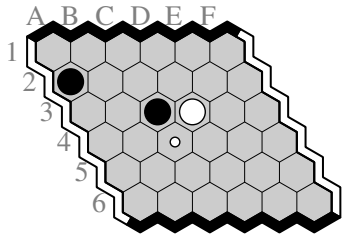
Figure 6.9: Randomly selected test positions from the test suite of BtoP 6×6 positions.



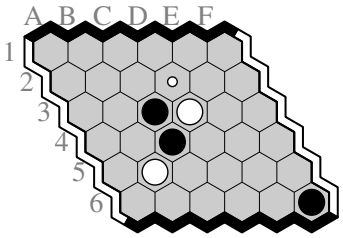
(a)



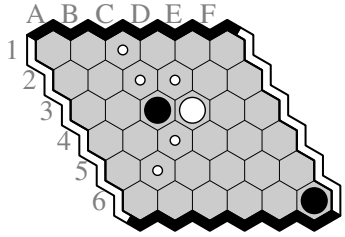
(b)



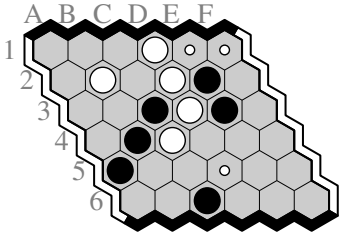
(c)



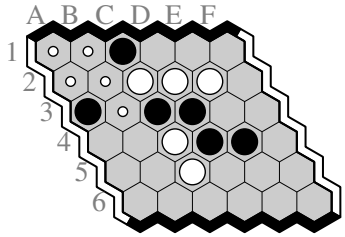
(d)



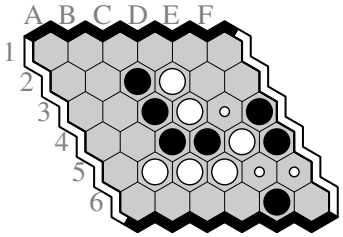
(e)



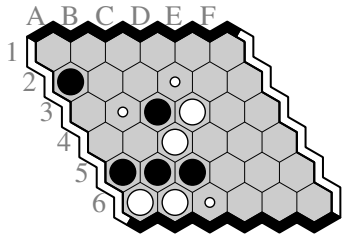
(f)



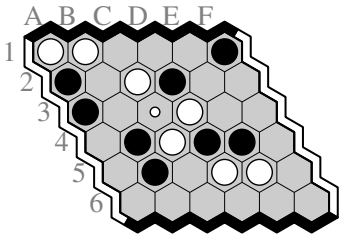
(g)



(h)

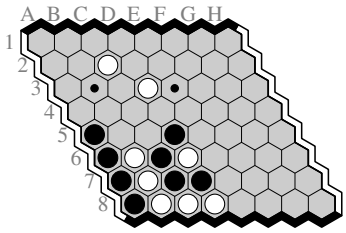


(i)

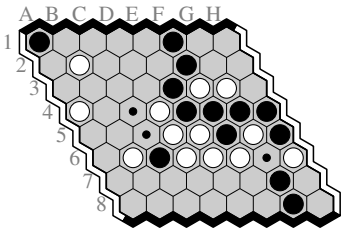


(j)

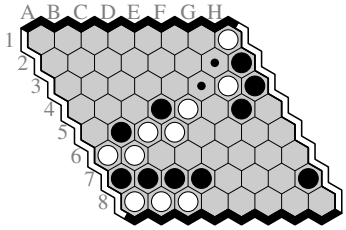
Figure 6.10: Randomly selected test positions from the test suite of WtoP 6×6 positions.



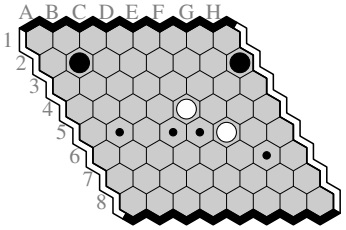
(a)



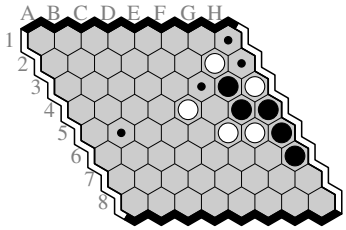
(b)



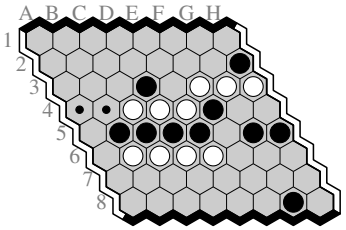
(c)



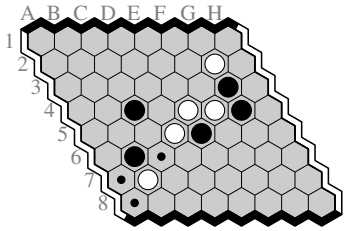
(d)



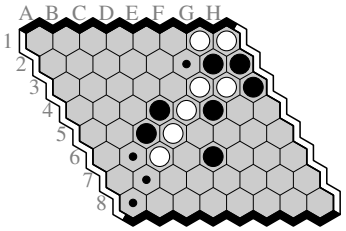
(e)



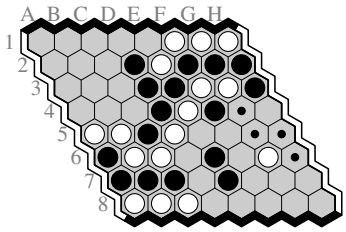
(f)



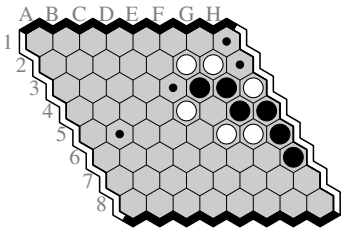
(g)



(h)

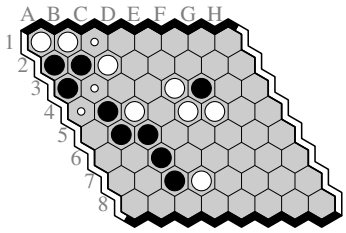


(i)

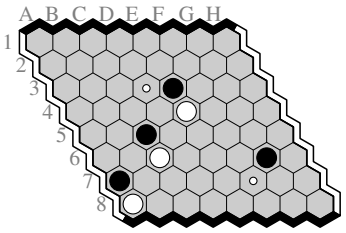


(j)

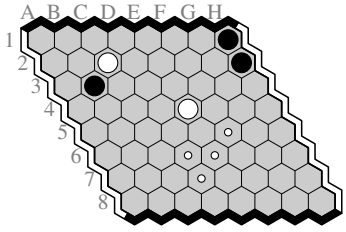
Figure 6.11: Randomly selected test positions from the test suite of BtoP 8×8 positions.



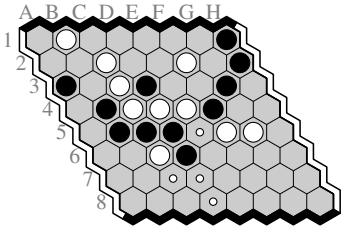
(a)



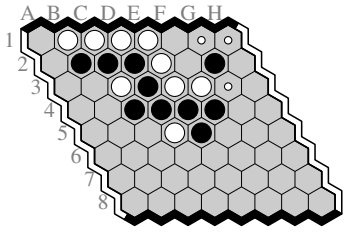
(b)



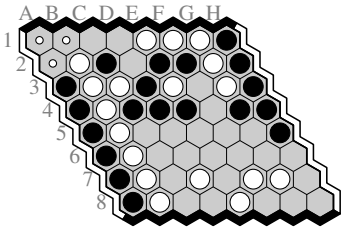
(c)



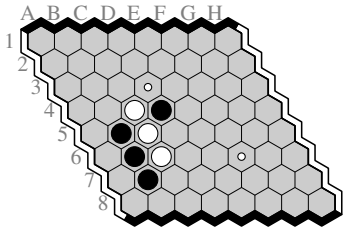
(d)



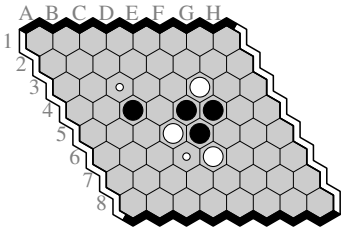
(e)



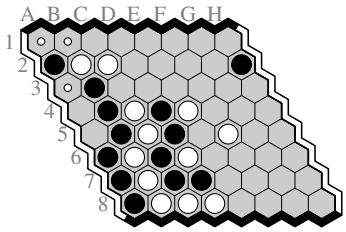
(f)



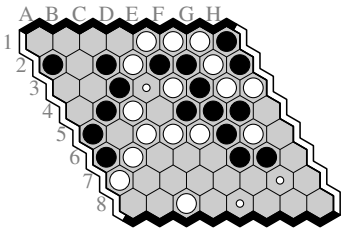
(g)



(h)



(i)



(j)

Figure 6.12: Randomly selected test positions from the test suite of WtoP 8×8 positions.

6.3 Relationship between Head-to-head Games and Test Positions

We discussed the advantages of using test positions over head-to-head games for evaluation, namely being less resource-intensive, in Section 6.2. However, we have done no extensive experimentation to compare the results of these two methods. Therefore, while we may not claim test positions can replace head-to-head games altogether, we use test positions for evaluating our Hex agents to overcome time and computing power limitations. For more information on the runtime of each method, see Section 7.1.

6.4 Concluding Remarks

In this chapter, we explained two methods for evaluating the performance of Hex agents: Head-to-head games, and test positions. We briefly discussed how requiring less resources makes test positions a more practical evaluation method for our Hex agents. In Section 6.2, we explained the four-step procedure required to test an agent against a given test position. We also introduced the winning-move selection ratio (WMSR) to assign one value to each agent based on all tested positions. We proposed the candidate positions extraction process to extract prospect game positions from game histories (Section 6.2.1). We also provided three criteria of having at least one winning move, not having too many winning moves, and taking considerable time to be fully solved for picking proper test positions from the pool of candidates in Section 6.2.2. Finally, we applied both test positions extraction and selection processes for Hex 6×6 and Hex 8×8 to generate test suites of 200 and 500 positions for each board size, respectively (Section 6.2.3).

In the next chapter, we will use these test suites to evaluate our Hex agents and study the efficacy of enforcing move openings during training.

Chapter 7

Evaluation Results

The previous chapter explains how we evaluate Hex agents by playing a series of head-to-head games and testing against a set of test positions. This chapter quantifies the performance of our Hex agents, trained with different forced-openings combinations (Chapter 5), against the two test suites prepared according to the test position selection procedure described in Section 6.2.2. There is one test suite per board size: Our test suite has 200 test positions (100 per player) for Hex 6×6; and 500 test positions (250 per player) for Hex 8×8. For detailed information on the test positions, see Section 6.2.3.

Although we provide two ways for evaluation — head-to-head games and test positions — we use only the test positions in our analysis and discussions. The reason is that head-to-head game evaluation is too slow, allowing insufficient time to consider all trained agents. As shown in Table 7.1, a complete set of head-to-head games could take up to 33.33 hours, which is at least 20 times higher than checking all test positions. More details are in Section 7.2.

When evaluating Hex agents on test suites, results are expressed as winning-move selection ratio (WMSR). It is the ratio of the number of correctly-answered test positions to the number of tested positions (Section 6.2).

Overall, our findings (Sections 7.2.1 and 7.2.3) show that including forced-move openings generally improves the winning-move selection ratios. However, we found

no significant difference among the various forced openings we tested.

7.1 Rationale for Using Test Suite and Evaluation Configuration

As explained in Chapter 6, our two evaluation methods check an agent's performance in two complementary ways. However, playing an adequate number of head-to-head games between each agent and MoHex takes more time than checking agents against test positions. For that reason, we compared different agents only on test positions.

Table 7.1 lists the runtime of evaluating Hex 6×6 and Hex 8×8 agents with the two evaluation methods. While the average runtime of one full game are 30 s and 118 s for Hex 6×6 and Hex 8×8, the average check-time for one test position are just about 4.5 s and 12 s, respectively.

Comparing agents with a full set of head-to-head games requires 1000 games, totalling 30000 s and 120000 s (8.33 hours and 33.33 hours) for the 6×6 and 8×8 boards. On these board sizes we have 200 and 500 test positions respectively, so comparing agents requires only 1000 s and 6000 s (0.28 hours and 1.67 hours), at least 20 times faster than playing head-to-head games.

For this reason we use only our test suites for evaluating agents. Each agent's evaluation score is measured as WMSR, a fraction in the interval of $[0, 1]$.

As mentioned in Section 5.1, we train our AlphaZero Hex agents for 100 and 200 steps beyond the initial step (zero) on the Hex 6×6 and Hex 8×8 board sizes. We also set the checkpoint frequency hyper-parameter to 1 to capture one snapshot per training step. This allows us to examine an agent's evolution during training. Rather than saving data only from the final snapshot, we save data from all 101 and 201 training snapshots for the two board sizes respectively, as maximum agent accuracy might occur before the last snapshot.

Each test position can be considered an arbitrary position in a game of Hex, requiring the agent to select the next move. AlphaZero uses MCTS for move selection, so

Table 7.1: Runtime for two evaluation methods of head-to-head games and test positions.

Measurement	**Hex 6×6**	**Hex 8×8**
Avg. time of one complete game against MoHex	30 s	120 s
Avg. time of checking one test position	5 s	12 s
Total time required for one full set of 1000 head-to-head games	30000 s (8.33 hours)	120000 s (33.33 hours)
Total time required for checking all test positions	1000 s (0.28 hours)	6000 s (1.67 hours)
Runtime factor of head-to-head games to test positions	30	20

we configure MCTS-related hyper-parameters in the evaluation step as in the training step, described in Section 5.1. Table 7.2 lists MCTS hyper-parameter values during evaluation of Hex 6×6 and Hex 8×8 agents. These values are the default ones defined by the OpenSpiel framework. We do not know whether these hyper-parameter settings are optimal. We keep the default values fixed to avoid introducing bias during the evaluation process.

All evaluations have been conducted using OpenSpiel AlphaZero Python implementation on Compute Canada's Cedar cluster [69], with 16 CPU cores working at 2.1 GHz frequency with 16 GB of memory.

Table 7.2: Values of MCTS-related hyper-parameters used in evaluation step.

Hyper-parameter	Description	Value
`uct_c`	UCT exploration constant used in PUCT	2
`max_simulations`	number of simulations to run	1000
`solve`	whether to back up solved states	True

7.2 Results

For each board size, we have five agents trained with the exact same hyper-parameters but different combinations of forced openings (Section 5.2.2), as follows:

- **A1**: all one-move openings only
- **A2**: all two-move openings only
- **A3**: all three-move openings only
- **A123**: all one-, two- and three-move openings
- **Baseline**: no forced openings

To study the training progress, we evaluate each training snapshot using our test suite. Considering that we have hundreds of snapshots per each of the five agents, we need a way to compare them numerically to see which forced-openings combination works best. We use the average of differences (AOD) in WMSRs for two agents throughout training as the metric of goodness. The formula for calculating the AOD for two agents $A = \{a_i, 0 \leq i \leq N\}$ and $B = \{b_i, 0 \leq i \leq N\}$ over n training steps is provided in Equation 7.1. A positive AOD means B has a higher WMSR and a negative one means A has a higher WMSR on average. For a detailed explanation of WMSR calculation, see Section 6.2.

We bisect these checkpoints because the WMSRs are indistinguishable in the early steps of training, but they diverge as evaluations progress. We will see this pattern later in Figures 7.1 and 7.2. Therefore, including all steps increases the risk of having a skewed and less accurate average value.

$$AOD(A, B) = \frac{\sum_{i=0}^{N} WMSR(b_i) - WMSR(a_i)}{N} \tag{7.1}$$

Given the properties of Hex, one agent may not play at the same level of strength for Black and White: the strongest BtoP and WtoP agents could be different.

In the rest of this section, we discuss the results of evaluating Hex 6×6 and Hex 8×8 models using the described framework in detail (Sections 7.2.1 and 7.2.3.) Here are the main conclusions:

1. Including forced-move openings in the training of AlphaZero models for Hex improves the quality of the agents trained to play board sizes of 6×6 and 8×8. Compared to the baseline, such agents reach higher WMSR values sooner.

2. While all Hex 6×6 and 8×8 agents with forced-move openings have better WMSR values than the baseline, there is no noticeable difference between the various forced-move opening combinations.

3. Some agents trained with forced-move openings solve all test positions for Hex 6×6 at some point in a 100-step training period. No Hex 8×8 agent can solve more than 76% of its related test suite in a 200-step training period.

7.2.1 Results of Evaluating Hex 6×6 Agents

Figures 7.1 and 7.2 show the performance of the five agents for Hex 6×6 in terms of WMSR based on 100 BtoP and 100 WtoP test positions, respectively. All agents trained with forced openings start below the baseline (the dotted line on Figure 7.1) at step 0 but rapidly increase toward the baseline and surpass it at step 4. Four steps later, at step 8, the baseline regains its position as the best, followed by A1 (the blue line) as the runner-up. Then it diminishes, and for the rest of the training, it cannot score higher than the majority of its competitors.

At step 16, A123 (the red line) and A2 (the orange line) experience a sudden drop followed by a fast recovery. For A123, the drop is almost three times sharper than A2. A123 drops by about 8% (8 test positions), while A2 drops by only 3% (3 test positions). At the same time, both A1 and A3 show an increase of 8% (8 test positions) and 2% (2 test positions.) Overall, agents trained with A1, A3, and A123 follow a steadily improving trend, but A2 has a wildly fluctuating upward trend.

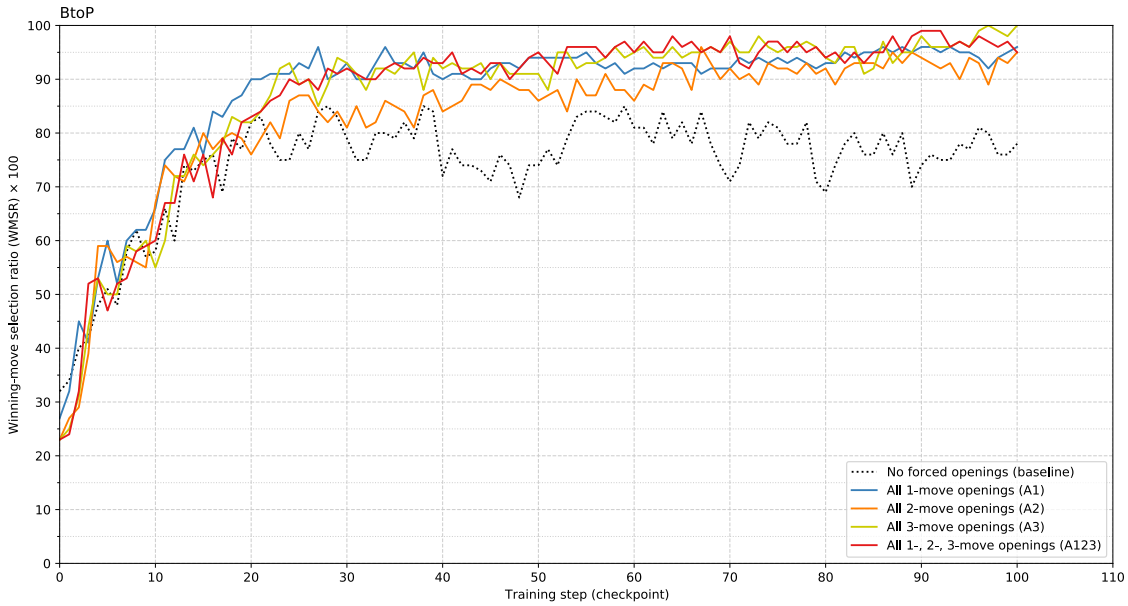


Figure 7.1: Winning-move selection ratio (WMSR)-Training step curves on BtoP test positions for Hex 6×6 agents trained with different forced-openings combinations.
First global maximum value for baseline is 85.0% (85 correct out of 100 test positions) at step 28.
First global maximum value for A1 is 96.0% (96 correct out of 100 test positions) at step 27.
First global maximum value for A2 is 96.0% (96 correct out of 100 test positions) at step 67.
First global maximum value for A3 is 100.0% (correct answers for all test positions) at step 97.
First global maximum value for A123 is 99.0% (99 correct out of 100 test positions) at step 90.

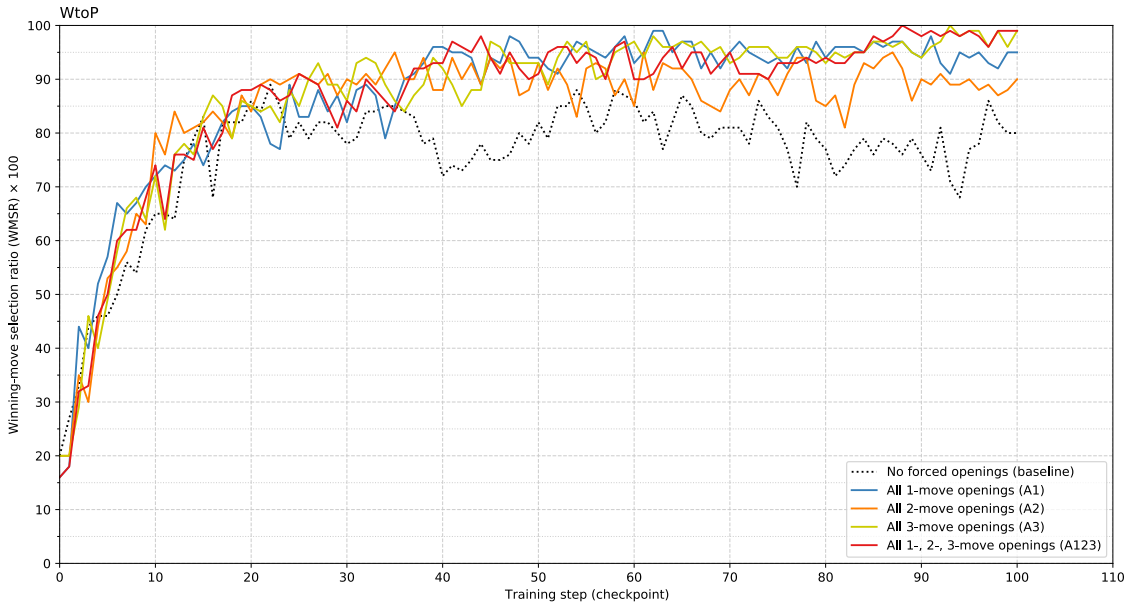


Figure 7.2: Winning-move selection ratio (WMSR)-Training step curves on WtoP test positions for Hex 6×6 agents trained different with forced-openings combinations.
First global maximum value for baseline is 89.0% (89 correct out of 100 test positions) at step 22.
First global maximum value for A1 is 99.0% (99 correct out of 100 test positions) at step 62.
First global maximum value for A2 is 95.0% (95 correct out of 100 test positions) at step 35.
First global maximum value for A3 is 100.0% (correct answers for all test positions) at step 93.
First global maximum value for A123 is 100.0% (correct answers for all test positions) at step 88.

Regarding maximum values, the baseline reaches the maximum value of 85% relatively soon at step 28, but it cannot maintain that value and loses it immediately. Later it reaches the exact maximum two more times, at steps 38 and 58, but it ends at 77% after 100 training steps.

A1 also attains its maximum of 96% very early at step 27. Nevertheless, this peak is followed by a sharp decrease of 7%. After that, it reaches the same maximum a few more times, at steps 34, 86, $89-92$, and 94. Finally, it finishes at the same maximum of 96% at step 100.

The maximum for A2 only happens once. At step 67, it peaks at 96%. However, it cannot maintain that value for the rest of the training and closes 90% at step 100.

A3 reaches its maximum of 100% quite late at step 97. Although it drops by 2% in the next two steps, it finishes at the same maximum value of 100% at the end of training. The behaviour of A123 is a little different. Like A3, it reaches its peak of 99% at step 90, but it maintains that value for four steps before starting a downward trend. This trend continues until the end of the training, where the final WMSR is 90%.

We see a new set of trends for WtoP. Like BtoP, all agents with forced openings start below the baseline but surpass at step 5. Then, at step 12, all agents, including the baseline, experience a peak followed by a deep valley.

The baseline peaks only once, reaching 89% at step 22. It then follows a fluctuating trend and concludes training with a WMSR value of 80%. A1 reaches the maximum value of 99% for the first time at step 62. It maintains that peak value in the next step, then a downward trend begins, which leads to 95% at the end of training. A2 peaks relatively early at step 35 with a WMSR value of 95%. Although it reaches the same maximum value 52 steps later at step 87, its training ends with a WMSR of 90%.

Both A3 and A123 peak late at steps 93 and 88 by solving all test positions and scoring 100% WMSR. However, none of them can keep this maximum to the end of

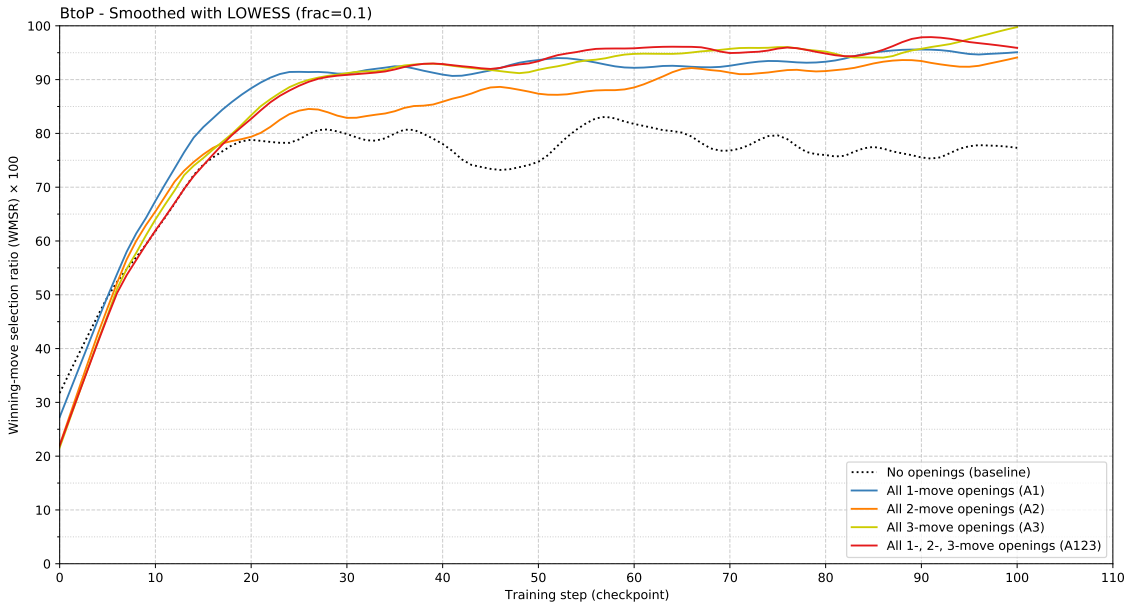


Figure 7.3: Smoothed version of Winning-move selection ratio (WMSR)-Training step curves on BtoP test positions for Hex 6×6 agents.

the training, and they both finish at 99% after 100 steps.

Due to the high fluctuation rate in winning-move selection ratios, it is not easy to compare agents or discuss trends using the WMSR-Training step curves (Figures 7.1 and 7.2). To make things easier, we smooth the curves using the Locally Weighted Scatterplot Smoothing (LOWESS) technique [70] with a fraction value equal to 0.1. The smoothed versions of Figures 7.1 and 7.2 are provided in Figures 7.3 and 7.4.

From Figure 7.3, it is clear that all agents, including the baseline, rise sharply until step 18. After that, the baseline starts a fluctuating pattern which continues until the end of training (step 100) and makes the agent finish at an even lower value than step 18. Conversely, the three agents of A1, A3, and A123 continue their upward trend at a reduced pace to a WMSR of 90% at step 30. From this point to the end of the training, the increasing trend for the group of three gets slower. By contrast, the rate for A2 behaves differently, where it maintains a steady rising trend from step 18 to the end (step 100.) However, since it gets to higher values (above 90%) 30 steps later than the other three at around step 60, even with a steady increasing trend, it

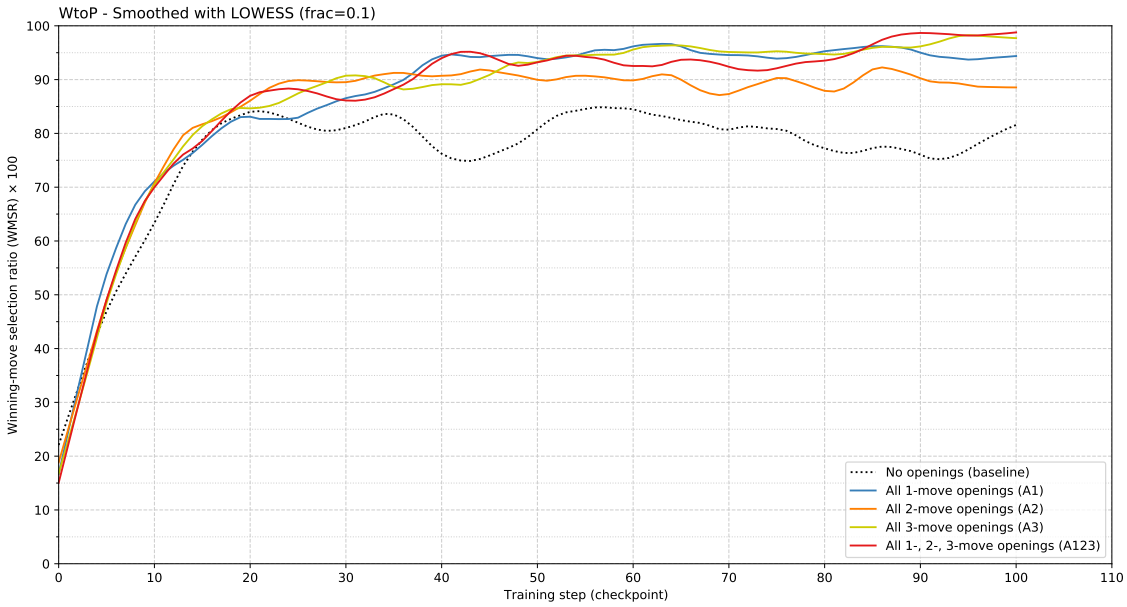


Figure 7.4: Smoothed version of Winning-move selection ratio (WMSR)-Training step curves on WtoP test positions for Hex 6×6 agents.

cannot surpass them until the end of training.

Figure 7.4 shows the smoothed version of WMSR curves for WtoP. The trends visible on that figure are comparable with what we see on the smoothed version of BtoP curves (Figure 7.3). From the beginning until step 20, the WMSR values for all five agents increase sharply. Then, the baseline starts fluctuating for the rest of the training and finishes at a lower value than step 20. A1 and A123 continue their upward trend until step 42, when they pass 90% in WMSR. From that point to the end, A1 remains almost stable, while A123 experiences a gradually-rising trend. Similarly, A3 continues its slowly-increasing trend until the end. In contrast, A2 fluctuates gradually to the end of the training after reaching a peak at step 25 and finishes below its maximum value.

7.2.2 Discussion of Hex 6×6 Results

Overall, evaluation of all four 6×6 agents trained with forced openings over BtoP and WtoP test positions shows a similar increasing trend throughout training. The

WMSR curves tend to stay close to each other. Based on this observation, there is no significant difference between the different combinations of opening moves. To support this observation with quantitative data, we calculate the average of differences (AOD) for winning-move selection ratios of each agents' pair using Equation 7.1. The calculated AODs for BtoP and WtoP test positions are provided in Tables 7.3 and 7.4, respectively. We should remember that the sign of an AOD has significance. For example, in $AOD(A, B)$, a positive number indicates the superiority of B (on average) while a negative one indicates otherwise.

As shown in Table 7.3.a, in the first half of training, the pairwise AOD of WMSR over BtoP test positions between A1, A3, and A123 is negligible and ranges from 0.06% to 2.9%. However, A1 performs best by 2.87% WMSR on average among these three. At the same time, A2 has the most negative pairwise AOD among all four agents trained with forced openings. Its performance is worse than A3 and A123 by an average WMSR of 2.87%. The difference between A2 and A1 is almost double, where A2 falls behind by 5.74%. All forced-openings agents outperform the baseline by an AOD ranging from 4.88% (for A2) to 10.62% (for A1).

We see a similar trend in the second half, as listed in Table 7.3.b. There is still a minor pairwise AOD of WMSR between A1, A3, and A123. Nevertheless, this time, A123 is the best among the group of three. A1 and A123 underperform A2 by about 4.44%, and A1 by 2.65%. Again, all agents with forced openings surpass the baseline by at least 12.90% WMSR.

Table 7.4 shows the pairwise AOD of WMSR for WtoP test positions. In the first half of training (Table 7.4.a), all four forced-opening agents perform better than the baseline, by about 7%. At the same time, the difference between the four agents is insignificant (under 1%.), to the point that A3 and A123 have AOD of zero.

In the second half of training (Table 7.4.b), the pairwise AOD between A1, A3, and A123 is still below 1%. However, A2 performs relatively poorly against the other three, with a pairwise AOD of at least -5.43%. Similar to the first half, all four agents

surpass the baseline by an AOD ranging from 9.41% (for A2) to 15.59% (for A3).

The quantitative analysis discussed above confirms our observation from WMSR-Training step curves (Figures 7.1 to 7.4). Therefore, we can conclude that using forced-move openings in the training step improves the quality of a Hex 6×6 agent. This allows the agent to achieve better results in test positions evaluation than the baseline with no forced-move openings, even when all training and testing hyper-parameters were unchanged. However, there is no evidence of a significant difference between the various combinations of forced-move openings for Hex 6×6 agents.

Table 7.3: Average of differences (AOD) of BtoP Winning-move selection ratios (WMSR) for different 6×6 agents

(a) First half: Training steps $0 - 50$

Table is symmetric.	A123	A1	A2	A3	Baseline
A123	0	-2.9	2.84	-0.06	7.72
A1	2.9	0	5.74	2.84	10.62
A2	-2.84	-5.74	0	-2.9	4.88
A3	0.06	-2.84	2.9	0	7.78
Baseline (no openings)	-7.72	-10.62	-4.88	-7.78	0

(b) Second half: Training steps $51 - 100$

Table is symmetric.	A123	A1	A2	A3	Baseline
A123	0	2.039	4.686	0.49	17.588
A1	-2.039	0	2.647	-1.549	15.549
A2	-4.686	-2.647	0	-4.196	12.902
A3	-0.49	1.549	4.196	0	17.098
Baseline (no openings)	-17.588	-15.549	-12.902	-17.098	0

Table 7.4: Average of differences (AOD) of WtoP Winning-move selection ratios (WMSR) for different 6×6 agents

(a) First half: Training steps $0 - 50$

Table is symmetric.	A123	A1	A2	A3	Baseline
A123	0	-0.2	-0.84	0	6.74
A1	0.2	0	-0.64	0.2	6.94
A2	0.84	0.64	0	0.84	7.58
A3	0	-0.2	-0.84	0	6.74
Baseline (no openings)	-6.74	-6.94	-7.58	-6.74	0

(b) Second half: Training steps $51 - 100$

Table is symmetric.	A123	A1	A2	A3	Baseline
A123	0	-0.059	5.431	-0.745	14.843
A1	0.059	0	5.49	-0.686	14.902
A2	-5.431	-5.49	0	-6.176	9.412
A3	0.745	0.686	6.176	0	15.588
Baseline (no openings)	-14.843	-14.902	-9.412	-15.588	0

7.2.3 Results of Evaluating Hex 8×8 Agents

The WMSR curves over BtoP test positions for Hex 8×8 agents, as shown in Figure 7.5, are considerably fluctuating and close to each other. Therefore, we cannot extract many meaningful trends from this graph. However, it can be seen that all agents experience a sharp increase in the first 30 steps of training. From that point, the rising pace of the baseline (the dotted line) and A2 (the orange line) slows down until it disappears at around step 62. After that, both curves start a fluctuating pattern with sharp ups and downs until the end of training (step 200.) At the same time, A1, A3, and A123 show an overall rising trend through the training. Although they also experience sharp peaks and valleys, their overall trend is increasing. We see that they move together and stay close to each other in almost all training steps.

Regarding the peak WMSR values, the baseline reaches its maximum of 48.4% relatively soon at step 62. However, it loses this maximum immediately at the next step. Later it reaches the exact maximum again at step 103, but it stands at 42% at the end of the 200-step training period. All four forced-opening agents attain their peaks in the last twenty steps: 72.8% for A1 at step 188, 54.8% for A2 at step 199, 72.0% for A3 at step 186, and 70.8% for A123 at step 196.

The WMSR curves for WtoP test positions (Figure 7.6) are not much different from BtoP ones. Again, all agents have an increasing trend until step 64. At that point, the baseline starts a strong downward trend that continues for ten steps and finishes at a local minimum of 38% at step 74. From this point to the end of the training, the baseline shows continuous fluctuations between 35% and 52% and eventually stands at 43%. A2 also experiences strong fluctuations between 48% and 60.8% from step 64 to step 200 (the last training step). It finishes close to the baseline at 45.5%.

The other three agents with forced openings follow an overall rising trend throughout the training. Like in BtoP evaluation, they have sharp fluctuations, but overall, they increase slowly while staying close to each other. Their only peak happens rel-

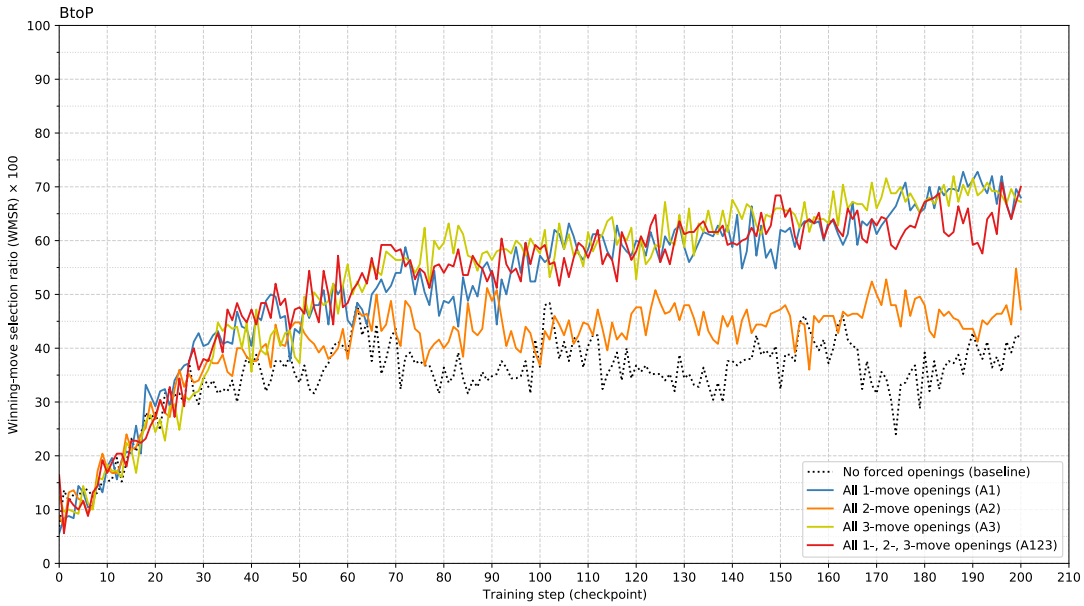


Figure 7.5: Winning-move selection ratio (WMSR)-Training step curves on BtoP test positions for Hex 8×8 agents trained with different forced-openings combinations.
First global maximum value for baseline is 48.4% (121 correct out of 250 test positions) at step 62.
First global maximum value for A1 is 72.8% (182 correct out of 250 test positions) at step 188.
First global maximum value for A2 is 54.8% (137 correct out of 250 test positions) at step 199.
First global maximum value for A3 is 72.0% (180 correct out of 250 test positions) at step 186.
First global maximum value for A123 is 70.8% (177 correct out of 250 test positions) at step 196.

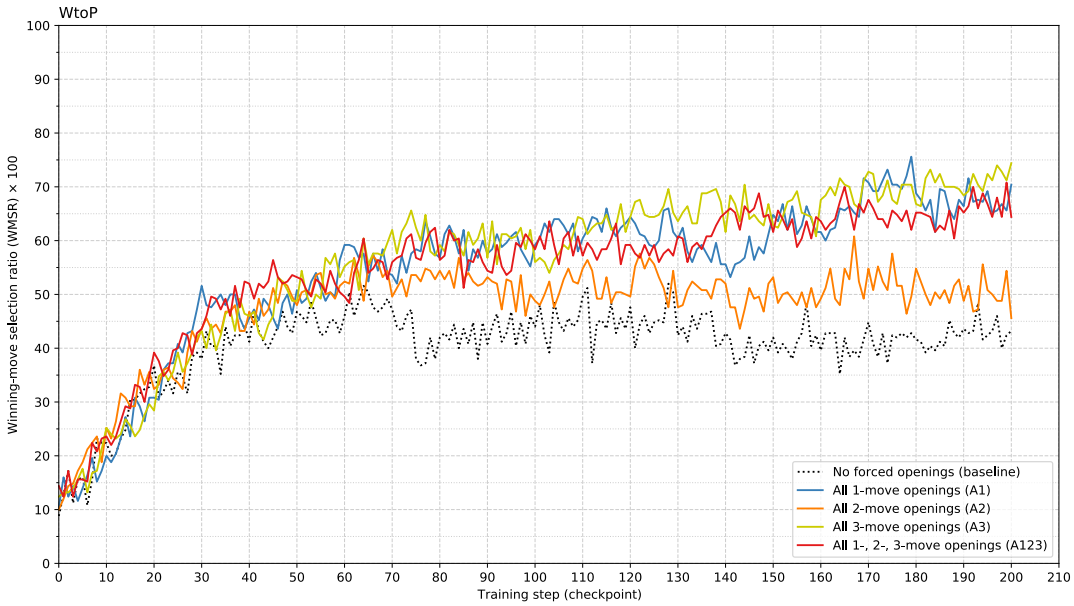


Figure 7.6: Winning-move selection ratio (WMSR)-Training step curves on WtoP test positions for Hex 8×8 agents trained different with forced-openings combinations.
First global maximum value for baseline is 52.0% (130 correct out of 250 test positions) at step 128.
First global maximum value for A1 is 75.6% (189 correct out of 250 test positions) at step 179.
First global maximum value for A2 is 60.8% (152 correct out of 250 test positions) at step 167.
First global maximum value for A3 is 74.4% (186 correct out of 250 test positions) at step 200.
First global maximum value for A123 is 70.8% (177 correct out of 250 test positions) at step 199.

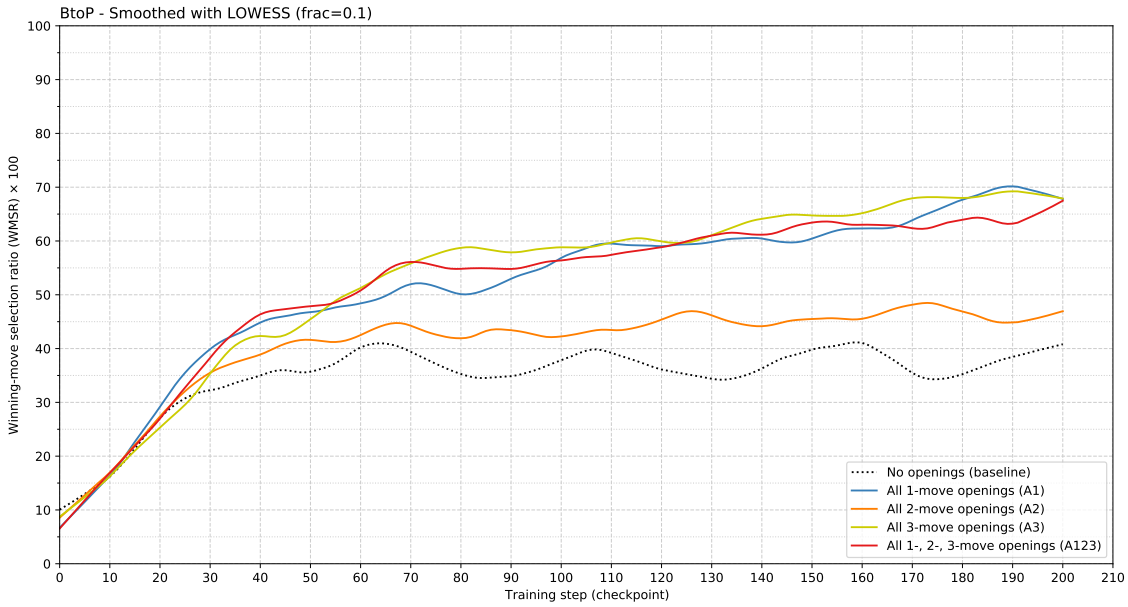


Figure 7.7: Smoothed version of Winning-move selection ratio (WMSR)-Training step curves on BtoP test positions for Hex 8×8 agents.

atively late: A1 reaches its maximum WMSR of 75.6% at step 179, A123's peak of 70.8% occurs at step 199 (the end of training), and A3 develops its maximum at the end of the training, at step 200.

As mentioned earlier, the Hex 8×8 BtoP and WtoP evaluation trends are not easily visible by looking at the WMSR-Training step curves (Figures 7.5 and 7.6) due to the high fluctuation rates. Therefore, we smooth them using the LOWESS method with $fraction = 0.1$ to see the generalized trends.

Figure 7.7 displays the smoothed version of BtoP WMSR curves. An aggressive rising trend can be seen until step 30 for all agents. After that, the baseline begins to fluctuate, but other curves continue going up. At around step 70, A2 starts a fluctuation while the other three rise steadily. A1, A3, and A123 keep increasing until the end of the training period.

Figure 7.8 shows the smoothed version of WtoP WMSR curves. Again, they look very similar to the smoothed BtoP ones. The main difference is that A2 keeps increasing steadily with the rest of the forced-opening agents until step 55 before it

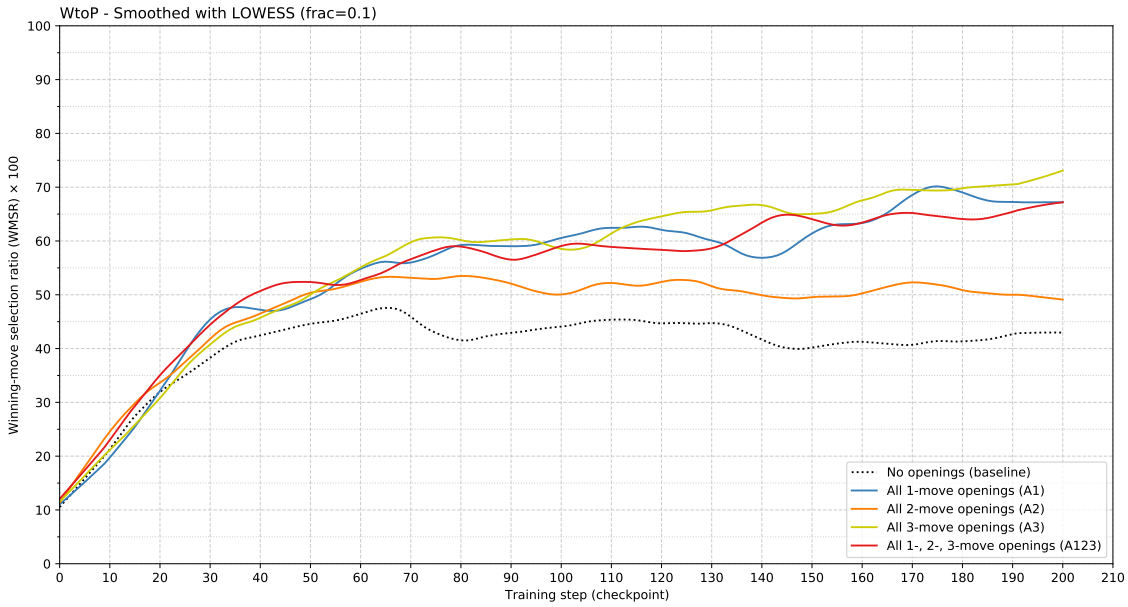


Figure 7.8: Smoothed version of Winning-move selection ratio (WMSR)-Training step curves on WtoP test positions for Hex 8×8 agents.

begins to fluctuate for the remaining training period.

7.2.4 Discussion of Hex 8×8 Results

Overall, based on Figures 7.5 to 7.8, we can see that agents trained with forced-move openings perform better than the baseline with no openings on both BtoP and WtoP test positions. Besides, while A1, A3, and A123 score closely and follow the same increasing trend to the end of the training, A2 falls behind them and stops developing relatively early.

To examine our observations with quantitative data, we calculate the average of differences (AOD) for both BtoP and WtoP evaluations using Equation 7.1. Tables 7.5 and 7.6 list pairwise AODs of the five evaluated agents over BtoP and WtoP test positions.

In the first half of training, A123 has the highest AOD against all other agents over BtoP test positions (Table 7.5.a.) However, the pairwise AOD between A1, A3 and A123 is relatively tiny. Conversely, A2 has an AOD of -5.86% (on average) against the

other three forced-opening agents. All forced-opening agents outperform the baseline by a minimum of 4.03% (for A2) and a maximum of 10.68% (for A123.) In the second half (Table 7.5.b), A3 takes the lead while A1, A3, and A123 show a slightly higher pairwise AOD than the first half. At the same time, the gap between A2 and the other three almost triples at a minimum of -17.34%. While the AOD of the baseline and A2 doubles at 7.8%, the AOD of the baseline and the other three agents increases 2.5 times to 25%.

The AODs are mostly similar on WtoP test positions (Table 7.6) In the first half, the AOD between A1, A3, and A123 is negligible (under 1%.) A2 falls behind them by a minor absolute AOD of about 2%. The baseline ranks last with an AOD ranging from 5.6% (against A2) to 8.27% (against A123.) In the second half, A3 scores slightly higher than A1 and A123 by a minimum AOD of 2.82%. The gap between A1 and A123 remains relatively low at 1.14%. The difference between A2 and the other three becomes six times greater than the first half and ranges from -15.46% to -11.5%. The gap between the baseline and others also increases to a minimum of 8.27% (with A2) and 23.72% (with A3.) It is almost 2.5 times the first half.

Our quantitative analysis confirms our observations from the WMSR-Training step curves (Figures 7.5 to 7.8) regarding Hex 8×8 agents. Therefore, we conclude that using forced-move openings in the training of a Hex 8×8 agent improves its results in test positions evaluation. Such agent scores better than the baseline when all training and testing hyper-parameters were unchanged. The one with all 2-move openings scores the worst between various combinations of forced openings. No combination seems better than the others.

7.3 Concluding Remarks

In this chapter, we compared the runtime of the two proposed evaluation methods – head-to-head games and test positions. We showed that evaluation with test positions takes $0.28 - 1.67$ hours (for Hex 6×6 and 8×8) which is at least 20 times faster than

Table 7.5: Average of differences (AOD) of BtoP Winning-move selection ratios (WMSR) for different 8×8 agents

(a) First half: Training steps $0 - 100$

Table is symmetric.	A123	A1	A2	A3	Baseline
A123	0	1.69	6.64	0.66	10.68
A1	-1.69	0	4.95	-1.03	8.98
A2	-6.64	-4.95	0	-5.98	4.03
A3	-0.66	1.03	5.98	0	10.01
Baseline (no openings)	-10.68	-8.98	-4.03	-10.01	0

(b) Second half: Training steps $101 - 200$

Table is symmetric.	A123	A1	A2	A3	Baseline
A123	0	-1.11	16.23	-2.77	24.03
A1	1.11	0	17.34	-1.66	25.14
A2	-16.23	-17.34	0	-19	7.8
A3	2.77	1.66	19	0	26.8
Baseline (no openings)	-24.03	-25.14	-7.8	-26.8	0

Table 7.6: Average of differences (AOD) of WtoP Winning-move selection ratios (WMSR) for different 8×8 agents

(a) First half: Training steps $0 - 100$

Table is symmetric.	A123	A1	A2	A3	Baseline
A123	0	0.82	2.67	0.58	8.27
A1	-0.82	0	1.85	-0.24	7.45
A2	-2.67	-1.85	0	-2.09	5.6
A3	-0.58	0.24	2.09	0	7.69
Baseline (no openings)	-8.27	-7.45	-5.6	-7.69	0

(b) Second half: Training steps $101 - 200$

Table is symmetric.	A123	A1	A2	A3	Baseline
A123	0	-1.14	11.5	-3.96	19.76
A1	1.14	0	12.64	-2.82	20.9
A2	-11.5	-12.64	0	-15.46	8.27
A3	3.96	2.82	15.46	0	23.72
Baseline (no openings)	-19.76	-20.9	-8.27	-23.72	0

head-to-head games with $8.33 - 33.33$ hours (Table 7.1). Therefore, we used only test positions for evaluating our models.

To analyze the efficacy of using forced-move openings (Section 5.2) in AlphaZero's search space exploration, we evaluated five agents with the same hyper-parameters but different combinations of forced openings: all one-move openings only, all two-move openings only, all three-move openings only, all one-, two- and three-move openings, and the baseline with no forced openings. As described in Section 6.2, the results of testing agents against test positions were expressed in WMSR.

Considering that the maximum agent accuracy might have occurred before the last snapshot, we studied agents' performance during the whole training period (101 steps for Hex 6×6 and 201 steps for Hex 8×8.) rather than evaluating only from the final snapshot of each agent. To compare the agents' performance over multiple snapshots, we defined a new pairwise metric called the average of differences (AOD) for WMSR.

Our experiments showed notable improvements in WMSR values and positive AODs compared to the baseline for all four agents trained with forced-move openings for both board sizes of 6×6 and 8×8 regardless of the player's color. While the baseline was never succeed in solving all test positions, one agent (A3) was able to solve all test positions for 6×6 BtoP (Figure 7.1) and two agents (A3 and A123) did the same for 6×6 WtoP test positions. No tested agent could solve all test positions for either colors of 8×8 and the maximum WMSR values were 72.8 for BtoP (Figure 7.5) and 75.6 for WtoP (Figure 7.6) both scored by A1.

The pairwise AOD of WMSR between the four agents and the baseline had a range of $4.88 - 17.588$ for Hex 6×6 BtoP (Table 7.3) and $6.74 - 15.588$ for WtoP (Table 7.4). The same value for Hex 8×8 ranged from 4.03 to 26.8 for BtoP (Table 7.5) and from 5.6 to 23.72 for WtoP (Table 7.6).

Despite the improvement over the baseline, we found no significant difference between the four combinations of forced-move openings. Therefore, we could not draw a conclusion about the superiority of one combination over the others.

Chapter 8

Summary and Conclusions

In this thesis we have tried to improve search space exploration during the training phase of AlphaZero agents by enforcing short opening-move sequences for each game, called a forced-move opening. During training, agents trained with this technique are exposed to a diverse range of moves and can learn the proper response to a broader set of opponent's actions in less time. As we showed in our experiments, this led to faster learning in the first 100 and 200 training steps for Hex 6×6 and 8×8, respectively.

In this thesis, we use the game of Hex as a case study for our proposed approach. Hex is a computationally challenging game with a large search space, easily-implemented rules, and a simple win-or-lose outcome. Such properties made it a good choice for a case study allowing quick game simulations and evaluation.

This thesis presents a set of hyper-parameters that work well for training AlphaZero agents for Hex 6×6 and 8×8, although an extensive hyper-parameter sweep is not part of this work. It also provides brief documentation on DeepMind's OpenSpiel RL framework and how it should be tuned for training AlphaZero agents.

In order to speed up the evaluation of trained agents, we propose to use test suites consisting of carefully chosen test positions instead of classical methods such as head-to-head games. By following four simple steps for each test position, we calculate the winning-move selection ratio (WMSR) for an agent evaluated with a test suite. Our experiments showed that such evaluation is significantly faster than playing complete

games and can be used as a comparison tool.

We present a systematic and repeatable method to extract a test suite from actual game histories. We also provide three criteria for selecting worthy positions from a pool of thousands of candidates for Hex boards of 6×6 and 8×8.

We trained Hex 6×6 and Hex 8×8 agents with four different combinations of forced move openings: all 1-move openings, all 2-move openings, all 3-move openings, and all 1-, 2-, 3-move openings. Our evaluation showed that all four combinations reached higher WMSR values sooner than the baseline with no forced openings. However, the difference between the four combinations was not significant enough to firmly conclude that one combination worked better than the others. We also observed that while the baseline never reached a WMSR of 1 (*i.e.*, solving all test positions), some forced openings combinations were able to solve all test positions for Hex 6×6 in a 100-step training period, *e.g.*, Figures 7.1 and 7.2. Conversely, no agent could solve more than 76% of the 8×8 test suite in a 200-step training period, *e.g.*, Figures 7.5 and 7.6.

Overall, we can conclude that this thesis's goals – presented in Chapter 1 as research questions are addressed as follows. For RQ 1, we have explored and presented some empirical evidence that a forced-move opening technique improves search space exploration and training speed. With respect to RQ 2, we have presented hyperparameters that work well for training Hex AlphaZero agents. For RQ 3, we have proposed using test positions as a faster way to evaluate Hex agents and provided a systematic approach to create test suites for different Hex board sizes.

We discuss some possible avenues for future work in the rest of this chapter.

8.1 Directions for Future Work

- Due to time and resource constraints, we are limited to small Hex board sizes. With more time and computing power, we can study our approach on competitive board sizes like 11×11 and 13×13.

- We exclusively used OpenSpiel's implementation of AlphaZero algorithm which is based on the original AlphaZero paper [1]. In the past couple of years, new variations of AlphaZero have been introduced [35]. It would be interesting to see whether including forced-move openings would be beneficial for them.

Bibliography

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018, ISSN: 0036-8075. DOI: 10.1126/science.aar6404. eprint: https://science.sciencemag.org/content/362/6419/1140.full.pdf. [Online]. Available: https://science.sciencemag.org/content/362/6419/1140.

[2] M. Lanctot, E. Lockhart, J.-B. Lespiau, V. Zambaldi, S. Upadhyay, J. Pérolat, S. Srinivasan, F. Timbers, K. Tuyls, S. Omidshafiei, D. Hennes, D. Morrill, P. Muller, T. Ewalds, R. Faulkner, J. Kramár, B. D. Vylder, B. Saeta, J. Bradbury, D. Ding, S. Borgeaud, M. Lai, J. Schrittwieser, T. Anthony, E. Hughes, I. Danihelka, and J. Ryan-Davis, "Openspiel: A framework for reinforcement learning in games," Aug. 26, 2019. arXiv: 1908.09453 [cs.LG].

[3] A. Plaat, *Learning to Play*. Springer International Publishing, Nov. 21, 2020, 330 pp. [Online]. Available: https://www.ebook.de/de/product/41242278/aske_plaat_learning_to_play.html.

[4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017, ISSN: 1476-4687. DOI: 10.1038/nature24270. [Online]. Available: https://doi.org/10.1038/nature24270.

[5] D. J. Wu, "Accelerating self-play learning in go," Feb. 27, 2019. arXiv: 1902.10565 [cs.LG].

[6] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 11, pp. 1203–1212, 1989. DOI: 10.1109/34.42858.

[7] T.-R. Wu, T.-H. Wei, and I.-C. Wu, "Accelerating and improving alphazero using population based training," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, pp. 1046–1053, Apr. 2020. DOI: 10.1609/aaai.v34i01.5454. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5454.

[8] R. B. Hayward and B. Toft, *Hex. The full story*, English. Boca Raton, FL: CRC Press, 2019, pp. xxii + 297, ISBN: 978-0-367-14425-8/hbk; 978-0-367-14422-7/pbk.

[9] T. W. Anthony, "Expert iteration," Ph.D. dissertation, University College London., Mar. 5, 2021.

[10] A. L. Simonsen and O. A. Haddeland, "Playing the game of hex with thetsetlin machine and tree search," M.S. thesis, University of Agder, Grimstad, Oct. 15, 2020. [Online]. Available: https://hdl.handle.net/11250/2683053.

[11] J. Tromp and G. Farnebäck, "Combinatorics of go," in *Computers and Games*, Springer Berlin Heidelberg, 2007, pp. 84–99. DOI: 10.1007/978-3-540-75538-8_8.

[12] C. Browne, *Hex strategy : making the right connections*. Natick, Mass: A.K. Peters, 2000, ISBN: 1568811179.

[13] Q. Cohen-Solal, "Learning to play two-player perfect-information games without knowledge," Aug. 3, 2020. arXiv: 2008.01188 [cs.AI].

[14] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Computers and Games*, Springer Berlin Heidelberg, 2007, pp. 72–83. DOI: 10.1007/978-3-540-75538-8_7.

[15] B. Arneson, R. Hayward, and P. Henderson, "Mohex wins hex tournament," *ICGA Journal*, vol. 32, pp. 114–116, 2009, ISSN: 13896911, 24682438. DOI: 10.3233/ICG-2009-32218.

[16] S.-C. Huang, B. Arneson, R. B. Hayward, M. Müller, and J. Pawlewicz, "Mohex 2.0: A pattern-based mcts hex player," in *Computers and Games*, H. J. van den Herik, H. Iida, and A. Plaat, Eds., Cham: Springer International Publishing, 2014, pp. 60–71, ISBN: 978-3-319-09165-5.

[17] D. Gale, "The game of hex and the brouwer fixed-point theorem," *The American Mathematical Monthly*, vol. 86, no. 10, pp. 818–827, Dec. 1979. DOI: 10.1080/00029890.1979.11994922.

[18] E. Berlekamp, *Winning ways for your mathematical plays*. Boca Raton: CRC Press, Taylor & Francis Group, 2018, ISBN: 9781568811307.

[19] J. Pawlewicz and R. B. Hayward, "Scalable parallel dfpn search," in *Computers and Games*, H. J. van den Herik, H. Iida, and A. Plaat, Eds., Cham: Springer International Publishing, 2014, pp. 138–150, ISBN: 978-3-319-09165-5.

[20] S. Reisch, "Hex is pspace-complete," 1981.

[21] É. Bonnet, F. Jamain, and A. Saffidine, "On the complexity of connection games," *Theoretical Computer Science*, vol. 644, pp. 2–28, Sep. 2016. DOI: 10.1016/j.tcs.2016.06.033.

[22] R. Wu and D. Beal, "Parallel retrograde analysis on different architecture," in *Proceedings 10th IEEE International Symposium on High Performance Distributed Computing*, IEEE Comput. Soc. DOI: 10.1109/hpdc.2001.945203.

[23] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2006, pp. 282–293. DOI: 10.1007/11871842_29.

[24] M. Enzenberger, M. Muller, B. Arneson, and R. Segal, "Fuego—an open-source framework for board games and go engine based on monte carlo tree search," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 259–270, Dec. 2010. DOI: 10.1109/tciaig.2010.2083662.

[25] T. Romstad, M. Costalba, and J. Kiiski. (Apr. 17, 2022). "Stockfish - Open Source Chess Engine," [Online]. Available: https://stockfishchess.org/.

[26] Computer Shogi Association. (Apr. 17, 2022). "Results of the 27th world computer shogi championship," [Online]. Available: http://www2.computer-shogi.org/wcsc27/index_e.html.

[27] R. Lake, J. Schaeffer, and P. Lu, *Solving large retrograde analysis problems using a network of workstations*, ser. Technical report / Department of Computing Science, University of Alberta 93,13. Edmonton, Alberta: Dep. of Computing Science, Univ., 1993, 32 pp.

[28] H. Bal and V. Allis, "Parallel retrograde analysis on a distributed system," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '95*, ACM Press, 1995. DOI: 10.1145/224170.224470.

[29] C. Gao, R. Hayward, and M. Muller, "Move prediction using deep convolutional neural networks in hex," *IEEE Transactions on Games*, vol. 10, no. 4, pp. 336–343, Dec. 2018. DOI: 10.1109/tg.2017.2785042.

[30] C. Gao, M. Müller, and R. Hayward, "Three-head neural network architecture for monte carlo tree search," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, International Joint Conferences on Artificial Intelligence Organization, Jul. 2018. DOI: 10.24963/ijcai.2018/523.

[31] K. Young, R. Hayward, and G. Vasan, *Neurohex: A deep q-learning hex agent*, 2016. DOI: 10.48550/ARXIV.1604.07097.

[32] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. DOI: 10.1038/nature14236.

[33] K. Takada, H. Iizuka, and M. Yamamoto, "Reinforcement learning to create value and policy functions using minimax tree search in hex," *IEEE Transactions on Games*, vol. 12, no. 1, pp. 63–73, Mar. 2020. DOI: 10.1109/tg.2019.2893343.

[34] S. Russell, *Artificial intelligence : a modern approach*. Upper Saddle River, N.J: Prentice Hall/Pearson Education, 2003, ISBN: 9780137903955.

[35] T. Cazenave, Y.-C. Chen, G.-W. Chen, S.-Y. Chen, X.-D. Chiu, J. Dehos, M. Elsa, Q. Gong, H. Hu, V. Khalidov, C.-L. Li, H.-I. Lin, Y.-J. Lin, X. Martinet, V. Mella, J. Rapin, B. Roziere, G. Synnaeve, F. Teytaud, O. Teytaud, S.-C. Ye, Y.-J. Ye, S.-J. Yen, and S. Zagoruyko, "Polygames: Improved Zero Learning," *International Computer Games Association Journal*, vol. 42, no. 4, pp. 244–256, 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03117499.

[36] International Computer Games Association. (Apr. 17, 2022). "Computer Olympiad 2020," [Online]. Available: https://icga.org/?page_id=3131.

[37] Q. Cohen-Solal and T. Cazenave, "Minimax strikes back," Dec. 19, 2020. arXiv: 2012.10700 [cs.AI].

[38] International Computer Games Association. (Apr. 17, 2022). "Computer Olympiad 2021," [Online]. Available: https://icga.org/?page_id=3216.

[39] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975. DOI: 10.1016/0004-3702(75)90019-3.

[40] J. Pawlewicz, R. Hayward, P. Henderson, and B. Arneson, "Stronger virtual connections in hex," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 02, pp. 156–166, Apr. 2015, ISSN: 1943-0698. DOI: 10.1109/TCIAIG.2014.2345398.

[41] HexWiki. (). "Hex gtp," [Online]. Available: https://www.hexwiki.net/index.php/GTP (visited on 08/12/2021).

[42] G. Farnebäck. (Aug. 12, 2021). "Gtp - go text protocol version 1," [Online]. Available: http://www.lysator.liu.se/~gunnar/gtp/.

[43] B. Arneson, P. Selinger, and R. B. Hayward. (Apr. 10, 2020). "Gui for board game hex (and y)," [Online]. Available: https://github.com/selinger/hexgui/.

[44] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012. DOI: 10.1109/tciaig.2012.2186810.

[45] D. Michie, "Game-playing and Game-learning Automata," in *Advances in Programming and Non-Numerical Computation*, Elsevier, 1966, pp. 183–200. DOI: 10.1016/b978-0-08-011356-2.50011-2.

[46] B. W. Ballard, "The $*$-minimax search procedure for trees containing chance nodes," *Artificial Intelligence*, vol. 21, no. 3, pp. 327–350, Sep. 1983. DOI: 10.1016/s0004-3702(83)80015-0.

[47] W. Jakob. (Aug. 11, 2021). "Pybind11 — seamless operability between c++11 and python," [Online]. Available: https://github.com/pybind/pybind11.

[48] M. Lanctot. (Aug. 11, 2021). "Openspiel — available games," [Online]. Available: https://github.com/deepmind/open_spiel/blob/master/docs/games.md.

[49] M. Lanctot. (Aug. 11, 2021). "Openspiel — available algorithms," [Online]. Available: https://github.com/deepmind/open_spiel/blob/master/docs/algorithms.md.

[50] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[51] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[52] M. J. Osborne and A. Rubinstein, *A Course in Game Theory*. MIT Press Ltd, Jul. 12, 1994, 368 pp., ISBN: 0262650401.

[53] Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. USA: Cambridge University Press, 2008, ISBN: 0521899435.

[54] J. K. Terry, B. Black, N. Grammel, M. Jayakumar, A. Hari, R. Sulivan, L. Santos, R. Perez, C. Horsch, C. Dieffendahl, N. L. Williams, Y. Lokesh, R. Sullivan, and P. Ravi, "Pettingzoo: Gym for multi-agent reinforcement learning," *arXiv preprint arXiv:2009.14471*, 2020.

[55] Association for Computing Machinery. (Aug. 24, 2020). "Artifact review and badging version 1.1," [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current.

[56] M. Lanctot. (Nov. 16, 2020). "Openspiel — alphazero," [Online]. Available: https://github.com/deepmind/open_spiel/blob/master/docs/alpha_zero.md (visited on 08/12/2021).

[57] M.-R. Daliri. (Jul. 21, 2020). "OpenSpiel PR 307: AlphaZero C++ via TensorflowCC," [Online]. Available: https://github.com/deepmind/open_spiel/pull/307 (visited on 08/12/2021).

[58] C. Jans. (Mar. 15, 2021). "OpenSpiel PR 319: Libtorch C++ AlphaZero," [Online]. Available: https://github.com/deepmind/open_spiel/pull/319 (visited on 08/12/2021).

[59] B. Obi Tayo. (Oct. 30, 2019). "Model Parameters and Hyperparameters in Machine Learning — What is the difference?" [Online]. Available: https://towardsdatascience.com/model-parameters-and-hyperparameters-in-machine-learning-what-is-the-difference-702d30970f6.

[60] L. N. Smith, "A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay," Mar. 26, 2018. arXiv: 1803.09820 [cs.LG].

[61] A. Prasad. (Jun. 20, 2018). "Lessons from alphazero (part 3): Parameter tweaking," [Online]. Available: https://medium.com/oracledevs/lessons-from-alphazero-part-3-parameter-tweaking-4dceb78ed1e5 (visited on 08/11/2021).

[62] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Networks*, vol. 12, no. 1, pp. 145–151, Jan. 1999. DOI: 10.1016/s0893-6080(98)00116-6.

[63] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016. DOI: 10.1038/nature16961.

[64] C. Jans, *Training DeepMind's OpenSpiel AlphaZero Algorithm to Play Clobber*, Aug. 31, 2020.

[65] W. T. Soh. (Apr. 13, 2019). "From-scratch implementation of alphazero for connect4," [Online]. Available: https://towardsdatascience.com/from-scratch-implementation-of-alphazero-for-connect4-f73d4554002a (visited on 08/26/2021).

[66] L. V. Allis, "Searching for solutions in games and artificial intelligence," Ph.D. dissertation, University of Limburg, Maastricht , The Netherlands, 1994, ISBN: 90-9007488-0.

[67] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 06, no. 02, pp. 107–116, Apr. 1998. DOI: 10.1142/s0218488598000094.

[68] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2016. DOI: 10.1109/cvpr.2016.90.

[69] Compute Canada. (Mar. 19, 2021). "Cedar – Compute Canada Doc," [Online]. Available: https://docs.computecanada.ca/wiki/Cedar (visited on 08/31/2021).

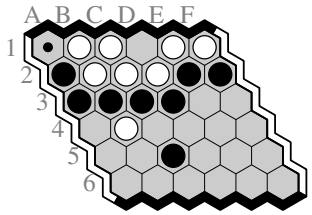
[70] W. S. Cleveland, "Robust locally weighted regression and smoothing scatterplots," *Journal of the American Statistical Association*, vol. 74, no. 368, pp. 829–836, Dec. 1979. DOI: 10.1080/01621459.1979.10481038.

Appendix A: Test Positions for Hex 6×6 and Hex 8×8

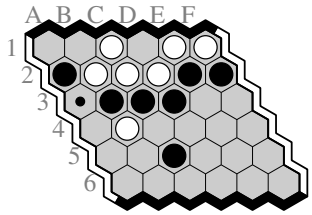
In this appendix, we list the visual representations of our test suites for of 200 test positions for Hex 6×6 and 500 test positions for Hex 8×8. The test suites are available online in JSON format at https://github.com/mrdaliri/hex-testsuite.

A.1 Hex 6×6

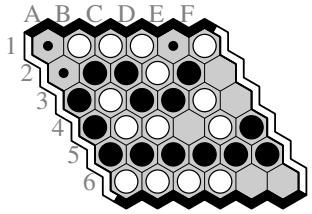
A.1.1 Black-to-play Test Positions



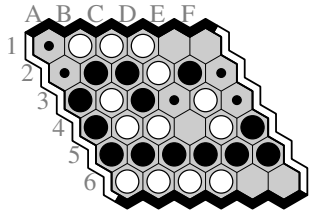
(1)



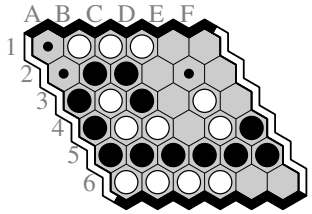
(2)



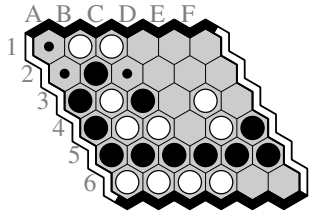
(3)



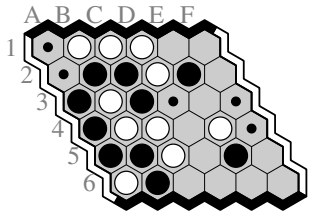
(4)



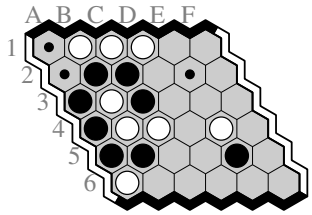
(5)



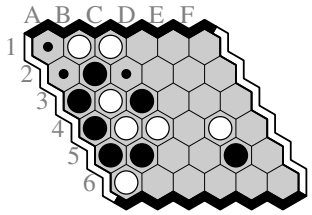
(6)



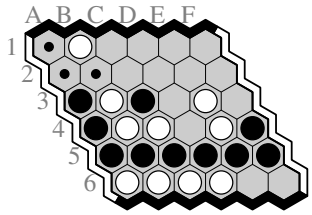
(7)



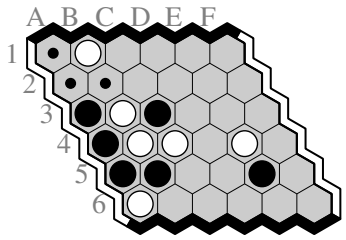
(8)



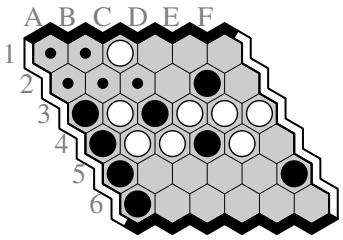
(9)



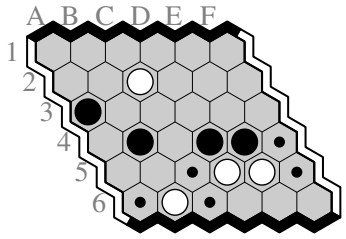
(10)



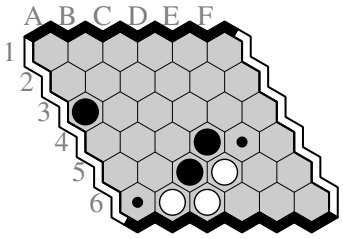
(11)



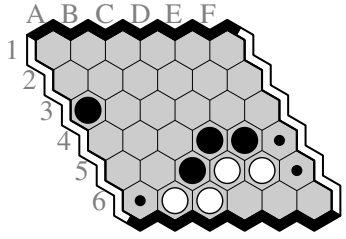
(12)



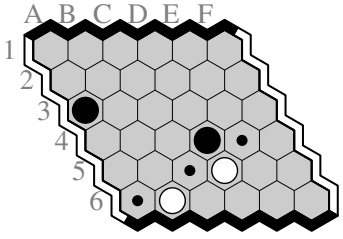
(13)



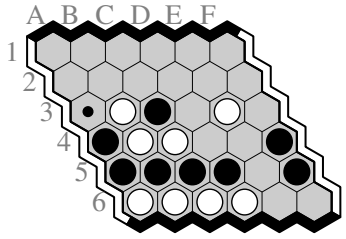
(14)



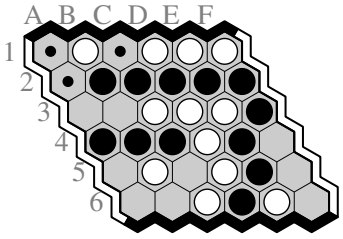
(15)



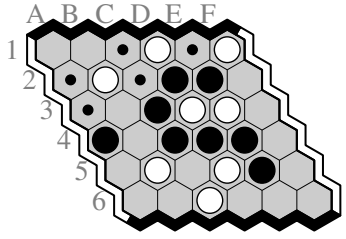
(16)



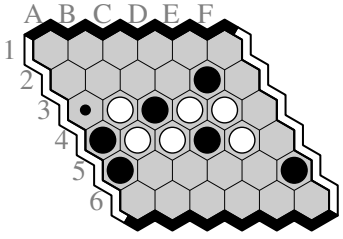
(17)



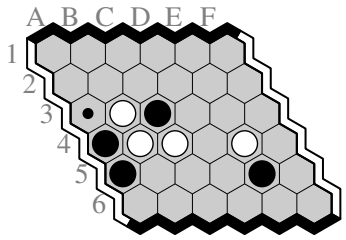
(18)



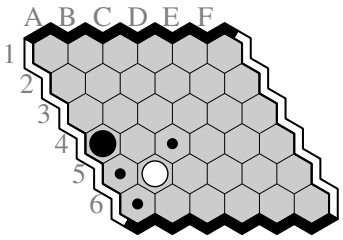
(19)



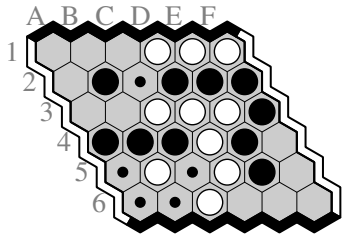
(20)



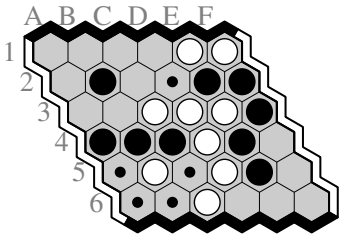
(21)



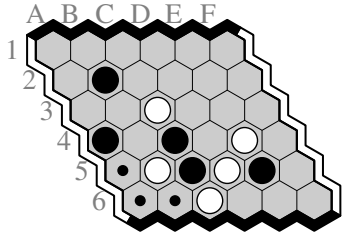
(22)



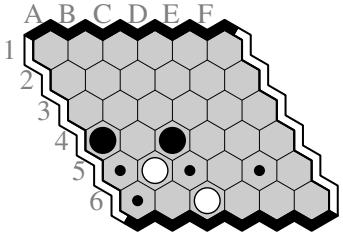
(23)



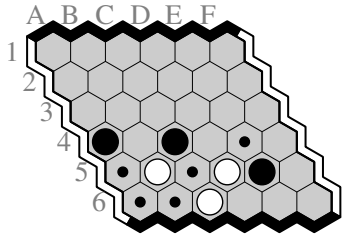
(24)



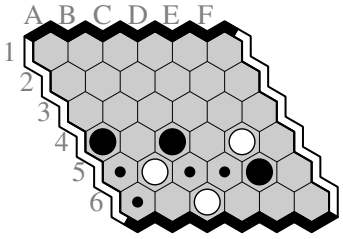
(25)



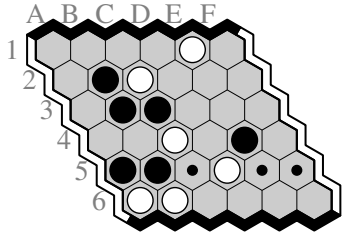
(26)



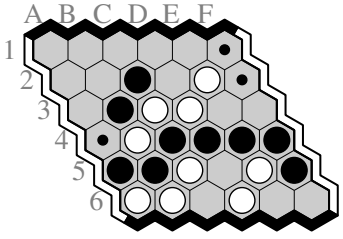
(27)



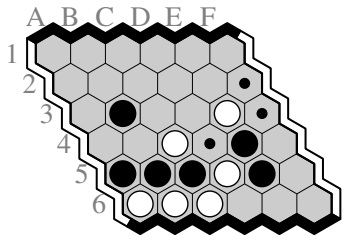
(28)



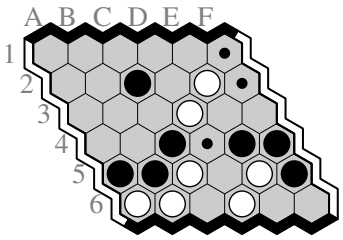
(29)



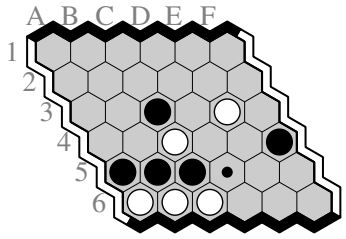
(30)



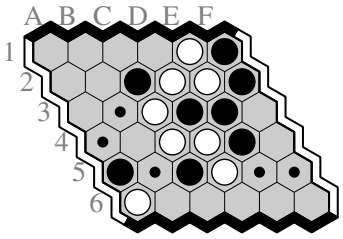
(31)



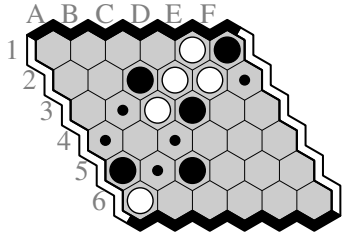
(32)



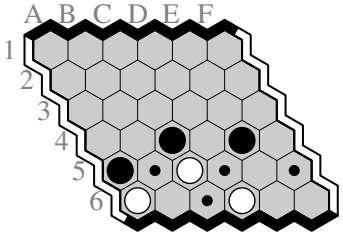
(33)



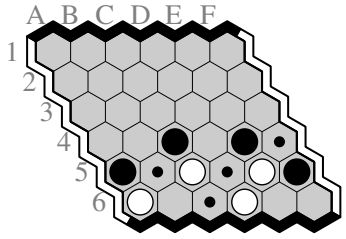
(34)



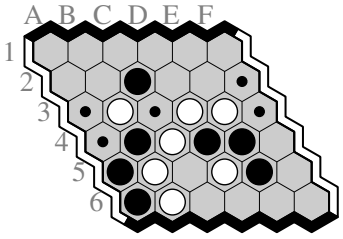
(35)



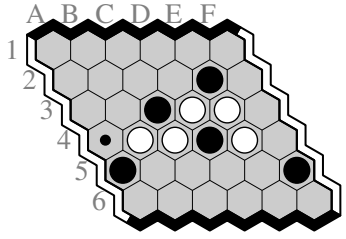
(36)



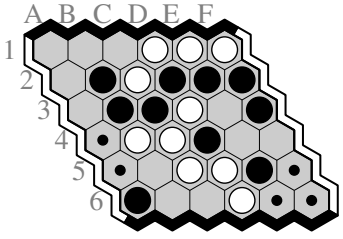
(37)



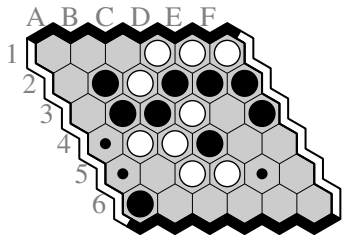
(38)



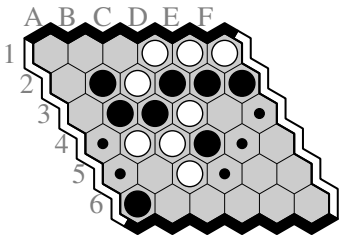
(39)



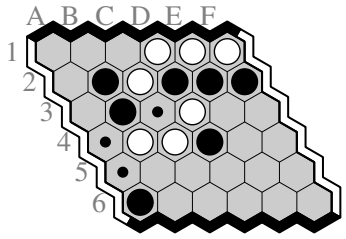
(40)



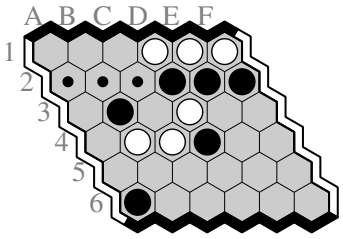
(41)



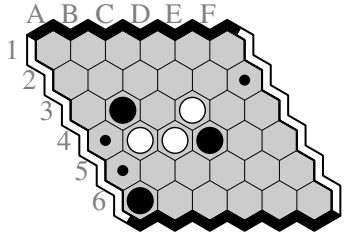
(42)



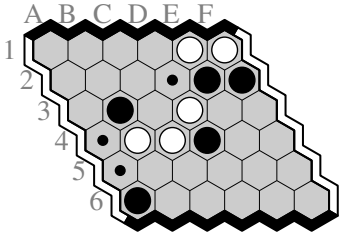
(43)



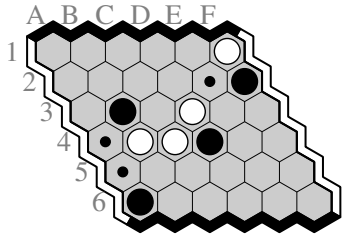
(44)



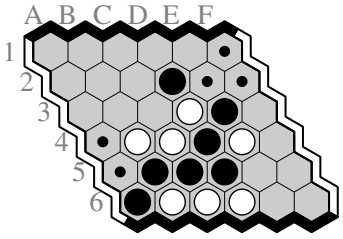
(45)



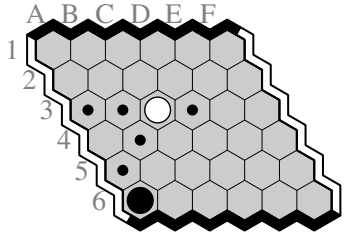
(46)



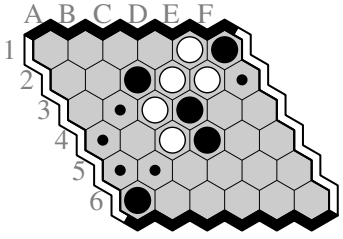
(47)



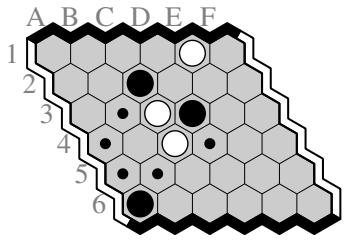
(48)



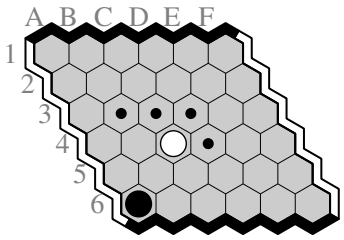
(49)



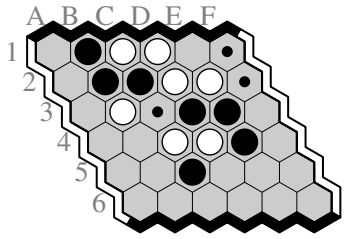
(50)



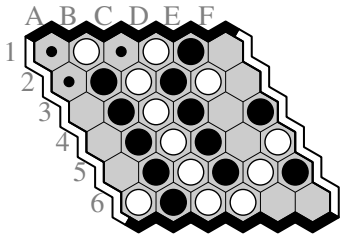
(51)



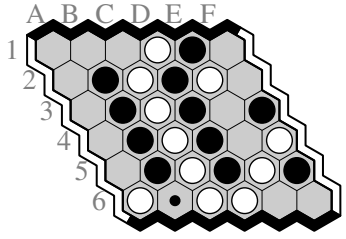
(52)



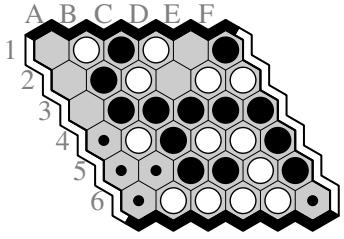
(53)



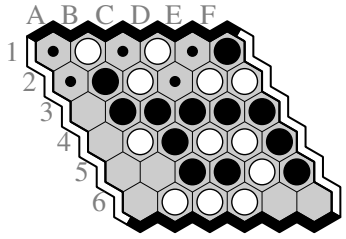
(54)



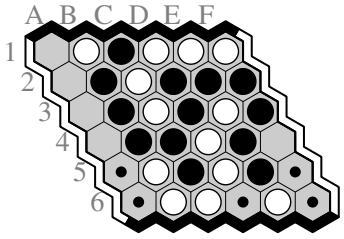
(55)



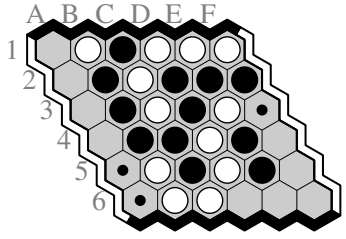
(56)



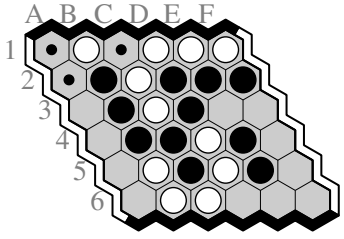
(57)



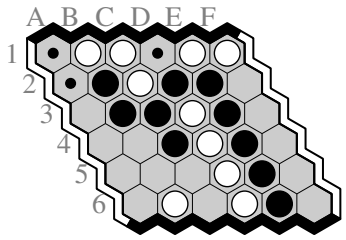
(58)



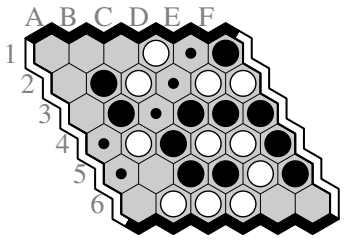
(59)



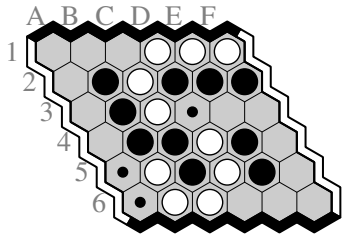
(60)



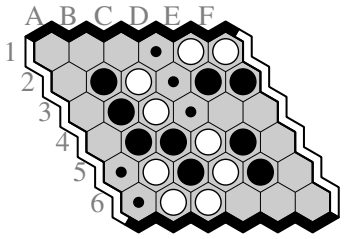
(61)



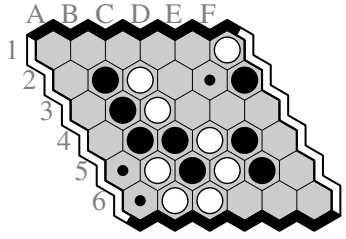
(62)



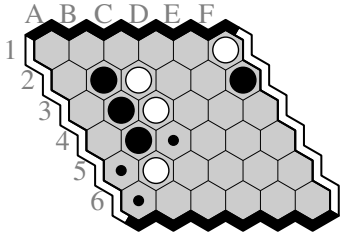
(63)



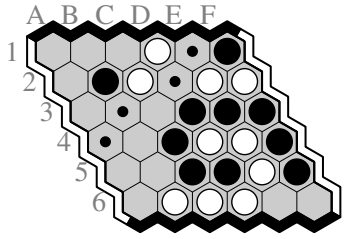
(64)



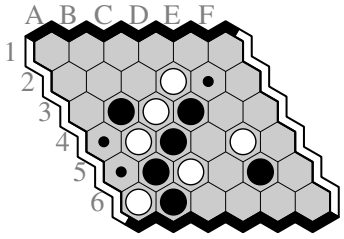
(65)



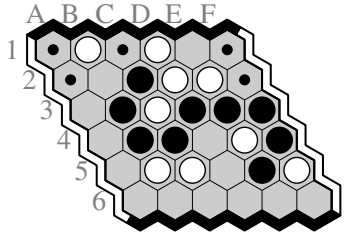
(66)



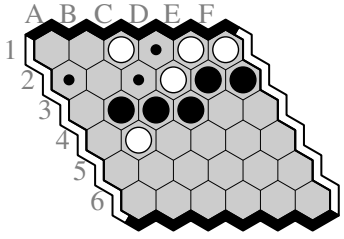
(67)



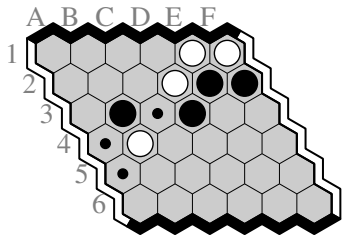
(68)



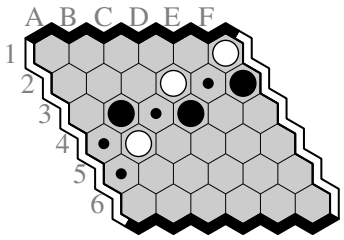
(69)



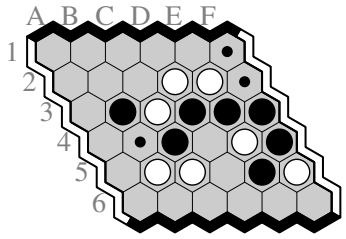
(70)



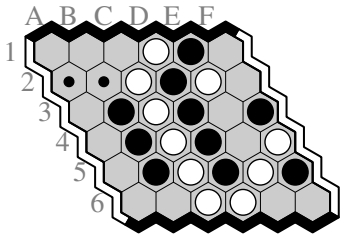
(71)



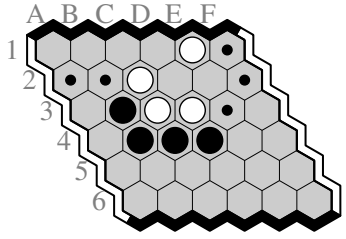
(72)



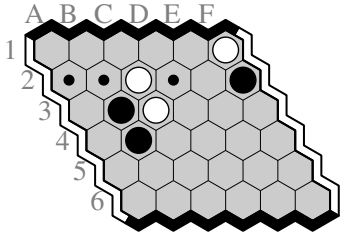
(73)



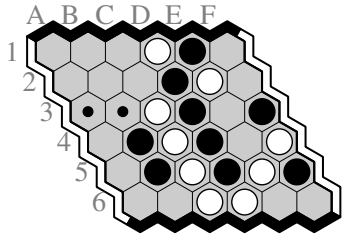
(74)



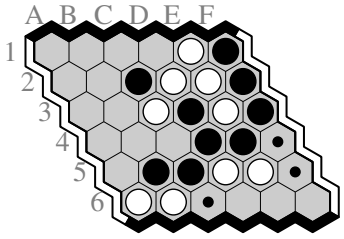
(75)



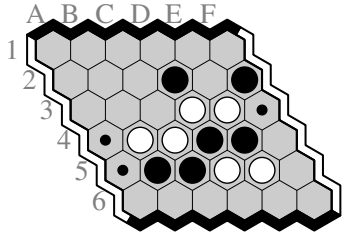
(76)



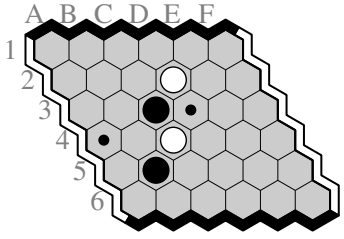
(77)



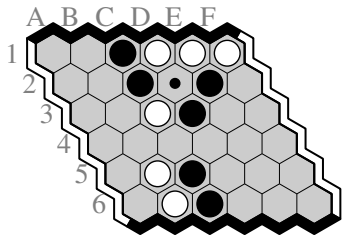
(78)



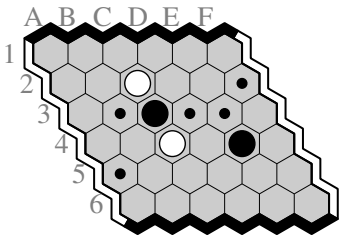
(79)



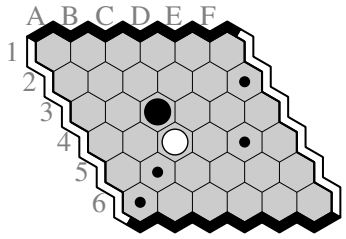
(80)



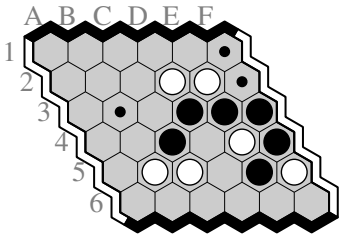
(81)



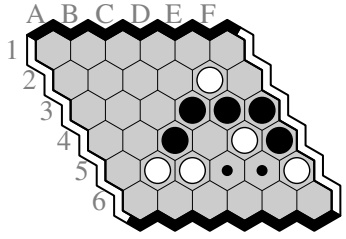
(82)



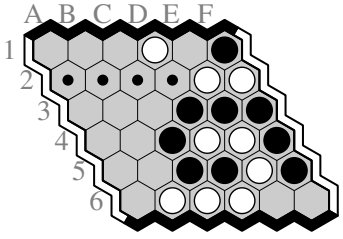
(83)



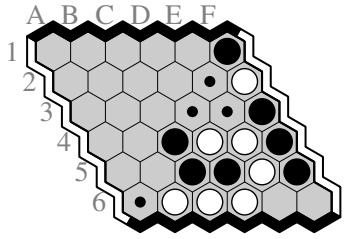
(84)



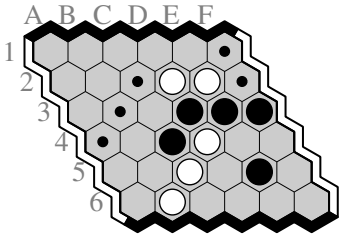
(85)



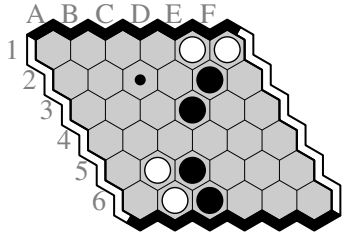
(86)



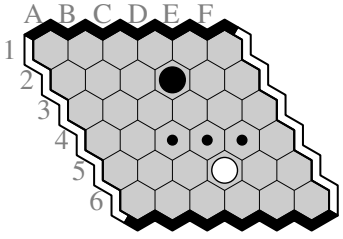
(87)



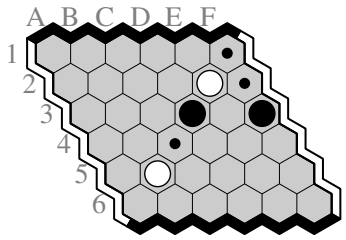
(88)



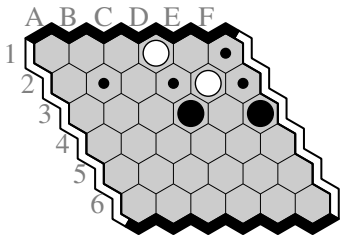
(89)



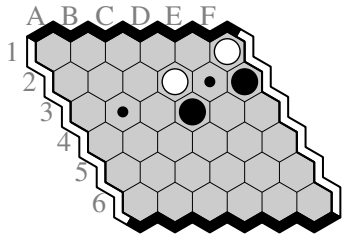
(90)



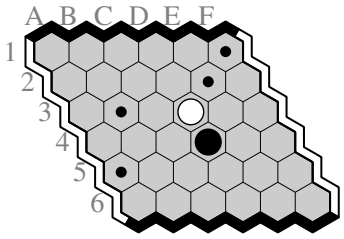
(91)



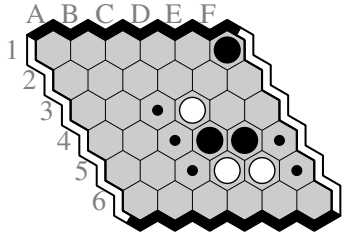
(92)



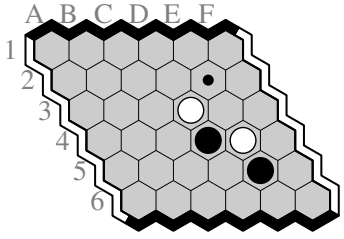
(93)



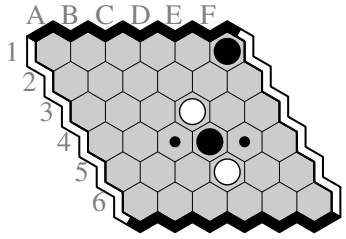
(94)



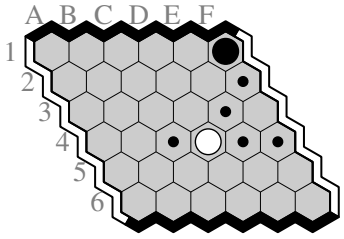
(95)



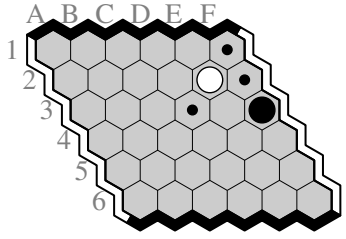
(96)



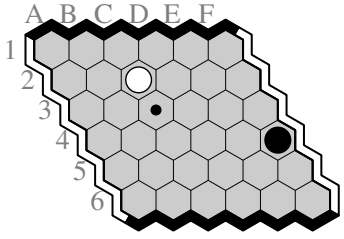
(97)



(98)

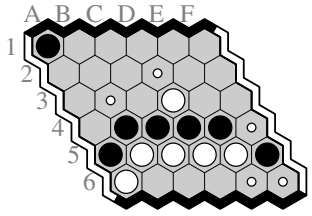


(99)

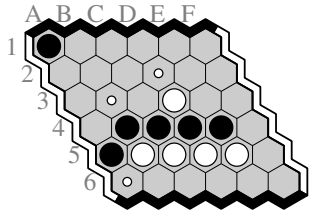


(100)

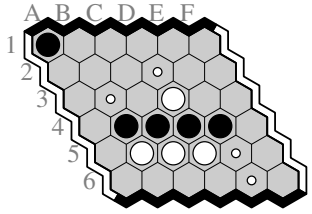
A.1.2 White-to-play Test Positions



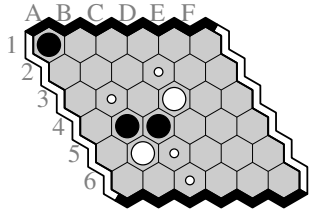
(1)



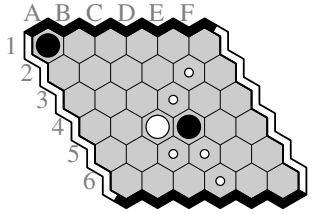
(2)



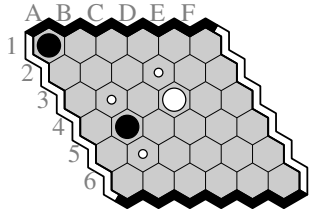
(3)



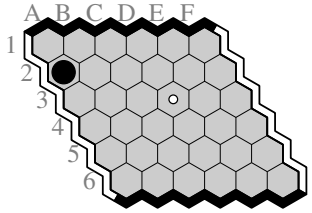
(4)



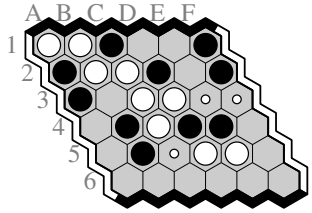
(5)



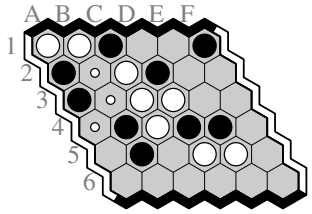
(6)



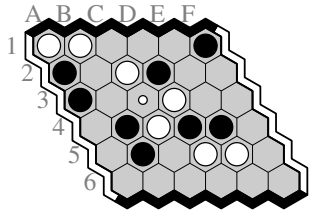
(7)



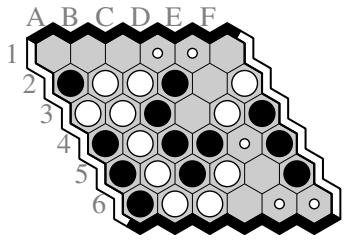
(8)



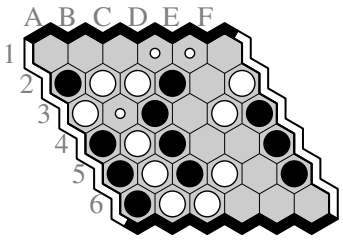
(9)



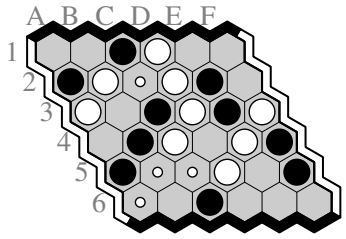
(10)



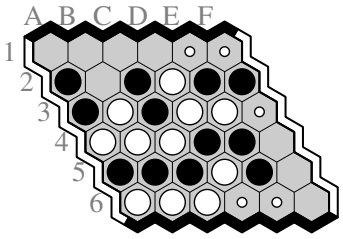
(11)



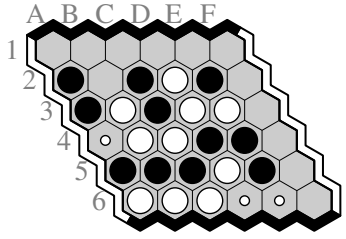
(12)



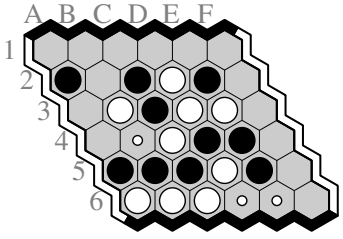
(13)



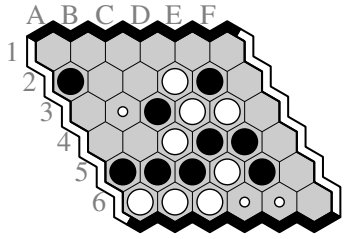
(14)



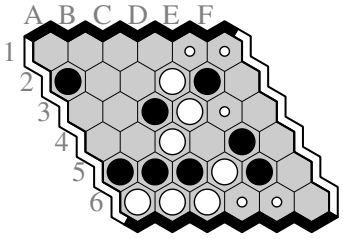
(15)



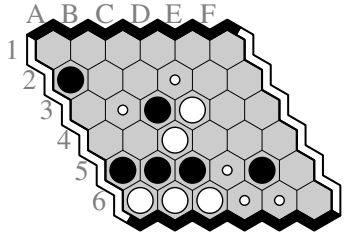
(16)



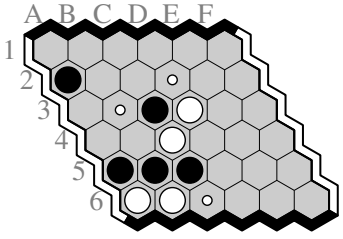
(17)



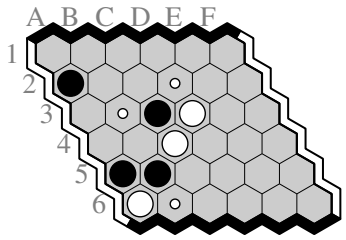
(18)



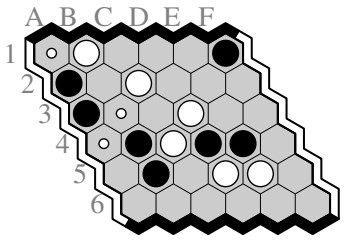
(19)



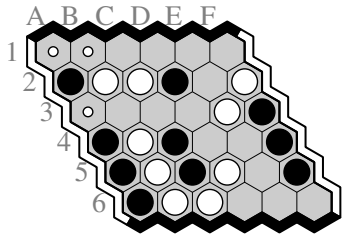
(20)



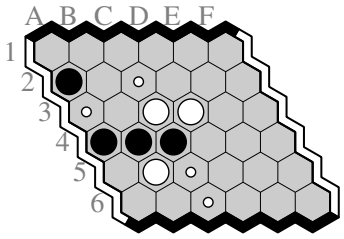
(21)



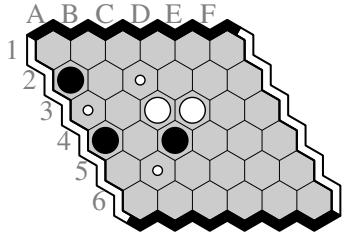
(22)



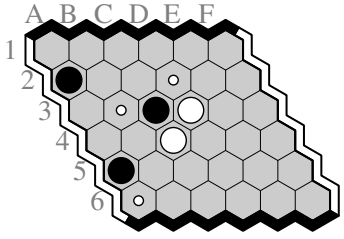
(23)



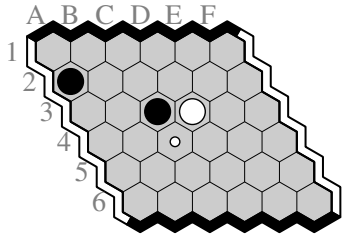
(24)



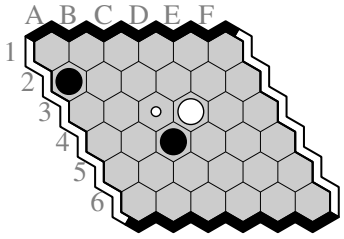
(25)



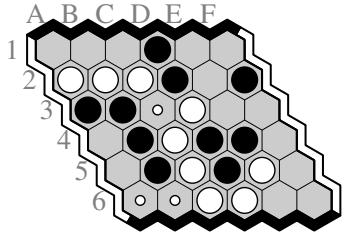
(26)



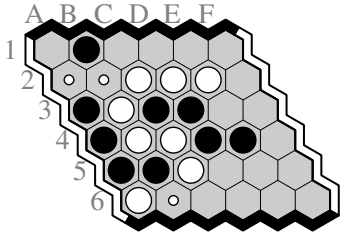
(27)



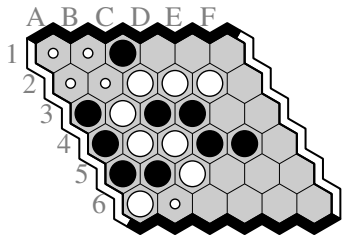
(28)



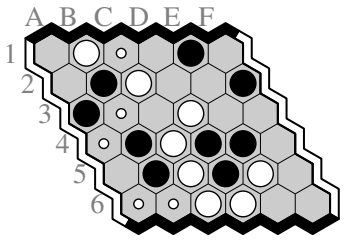
(29)



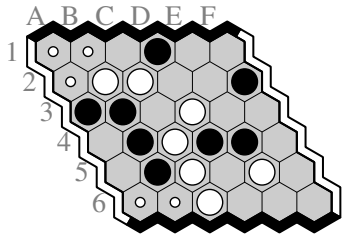
(30)



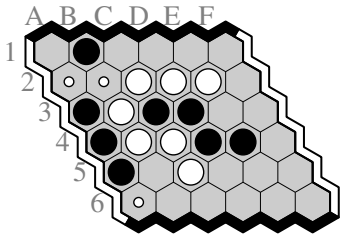
(31)



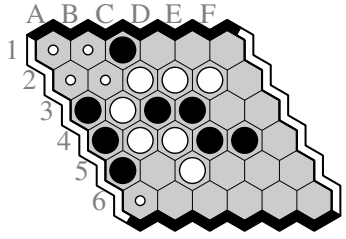
(32)



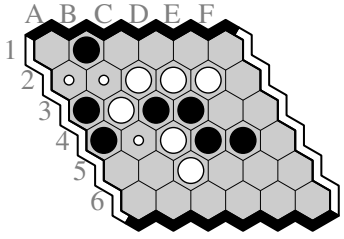
(33)



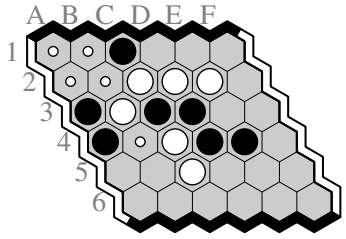
(34)



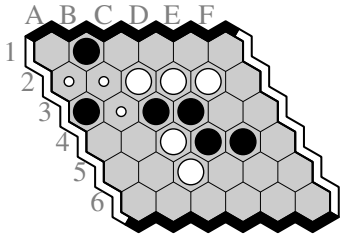
(35)



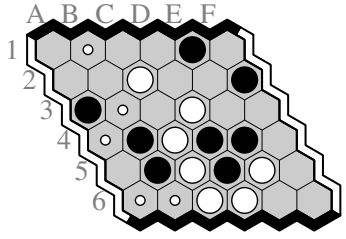
(36)



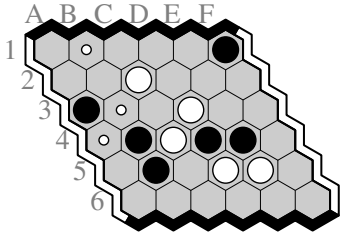
(37)



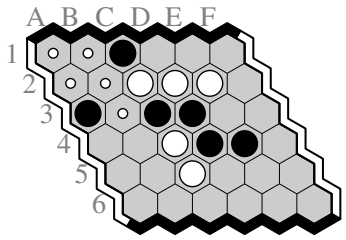
(38)



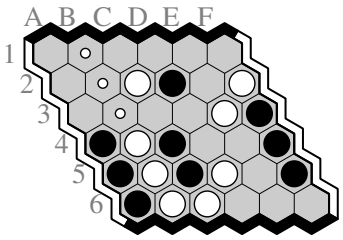
(39)



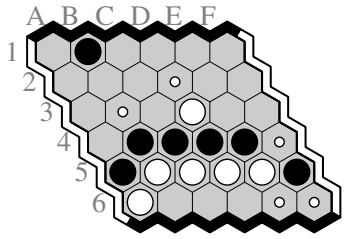
(40)



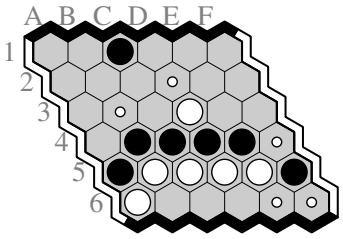
(41)



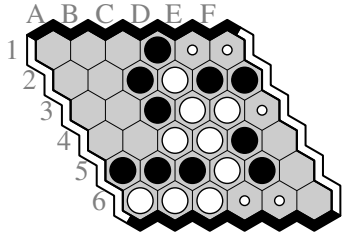
(42)



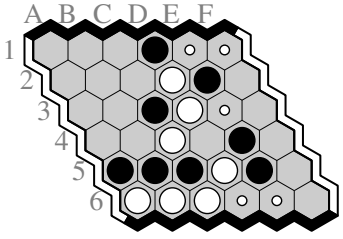
(43)



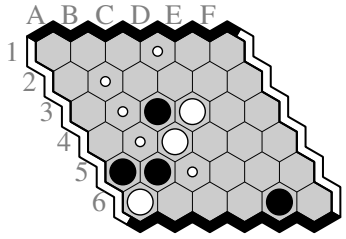
(44)



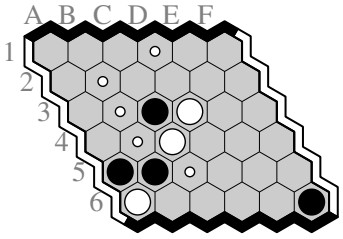
(45)



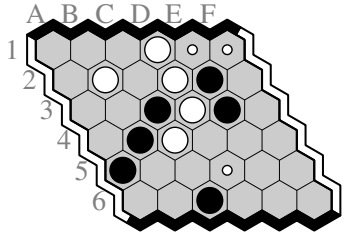
(46)



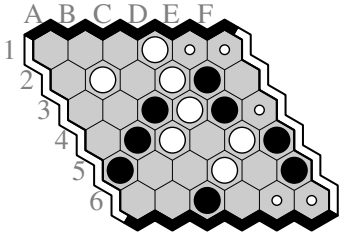
(47)



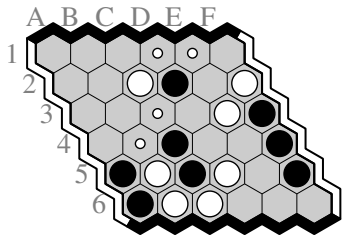
(48)



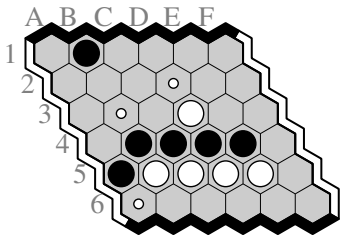
(49)



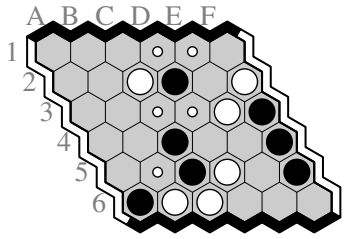
(50)



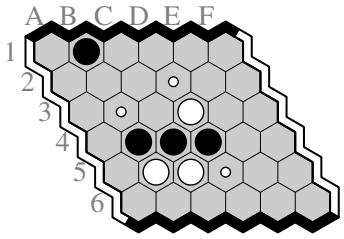
(51)



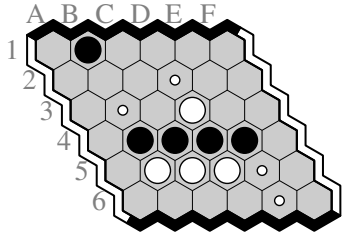
(52)



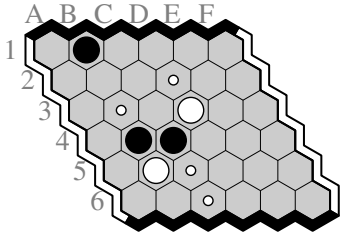
(53)



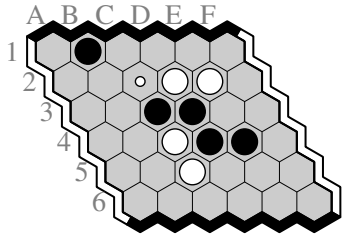
(54)



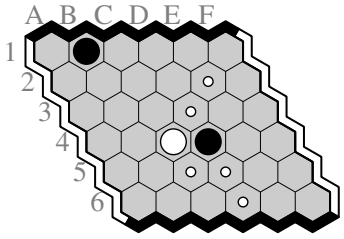
(55)



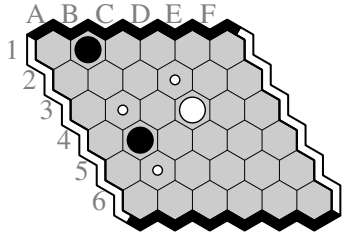
(56)



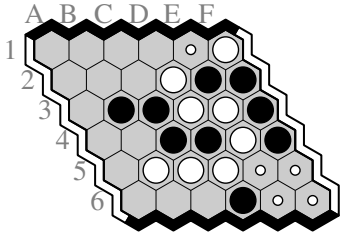
(57)



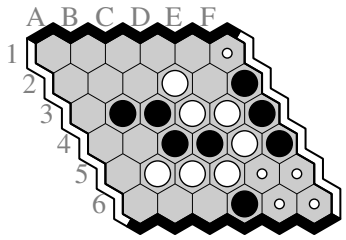
(58)



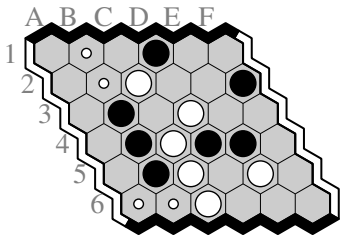
(59)



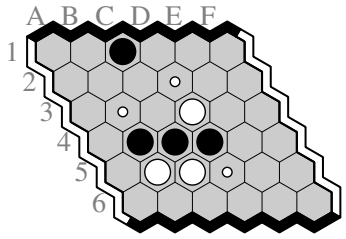
(60)



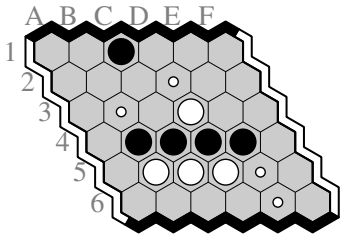
(61)



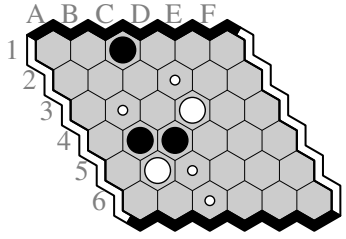
(62)



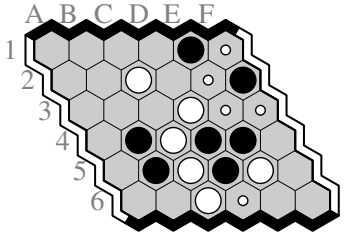
(63)



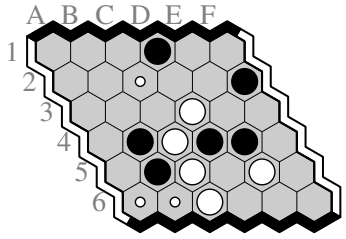
(64)



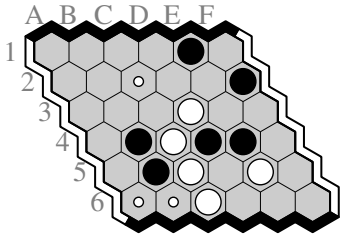
(65)



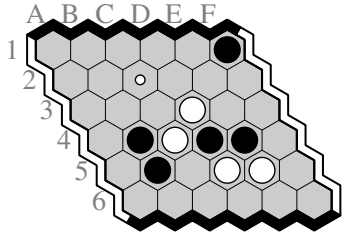
(66)



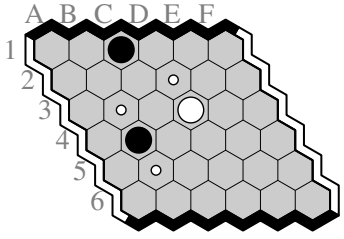
(67)



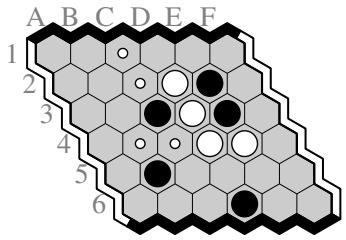
(68)



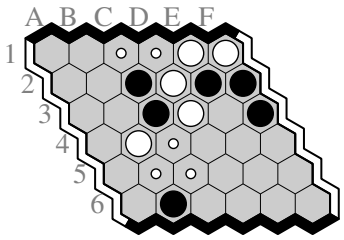
(69)



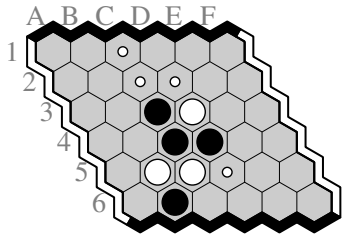
(70)



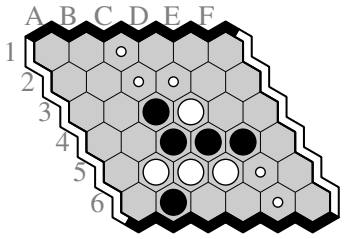
(71)



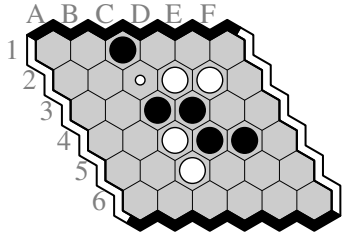
(72)



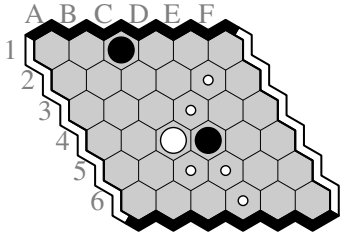
(73)



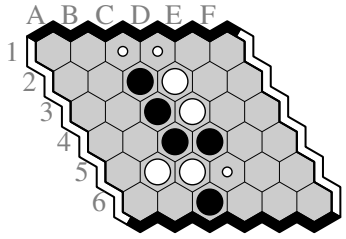
(74)



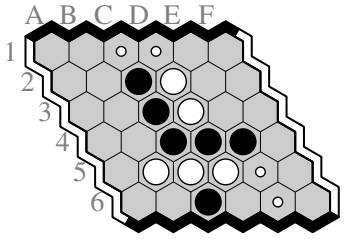
(75)



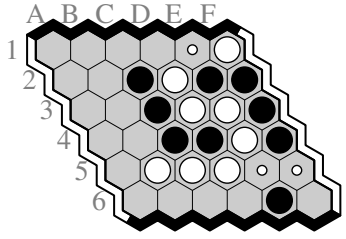
(76)



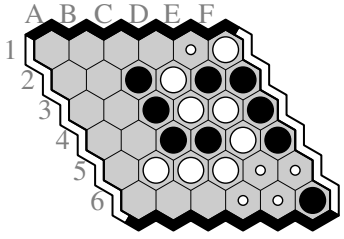
(77)



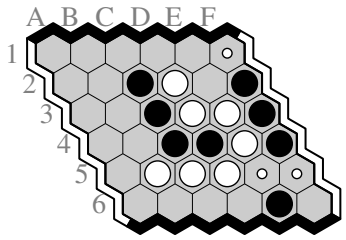
(78)



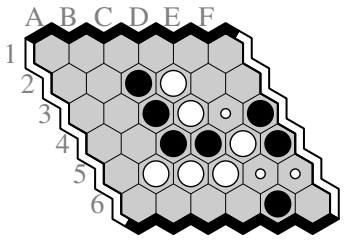
(79)



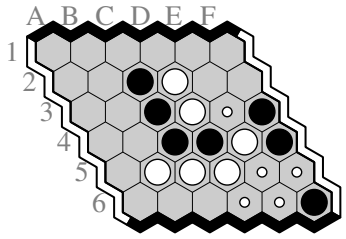
(80)



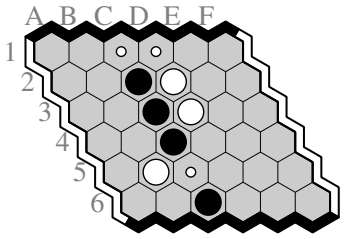
(81)



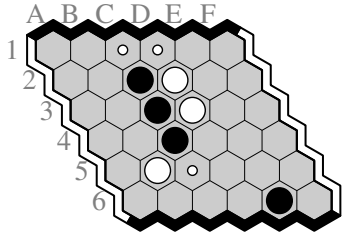
(82)



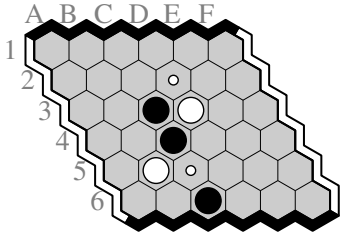
(83)



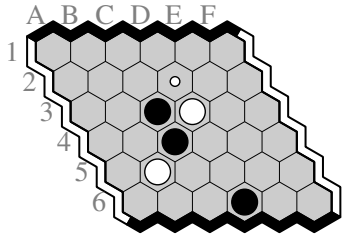
(84)



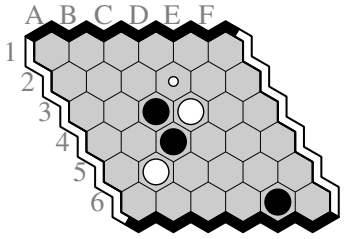
(85)



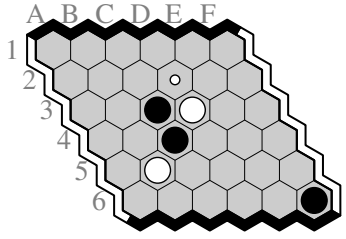
(86)



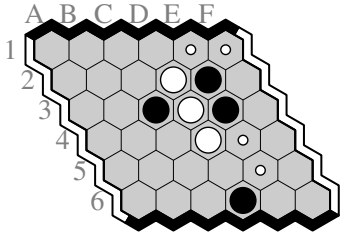
(87)



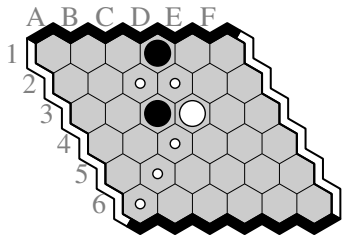
(88)



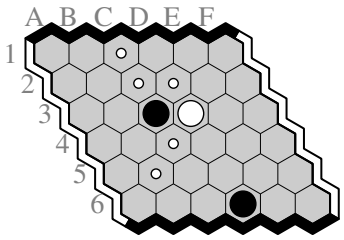
(89)



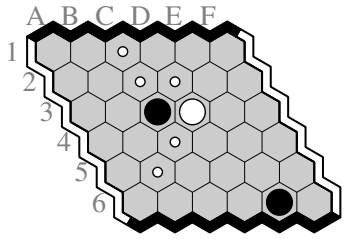
(90)



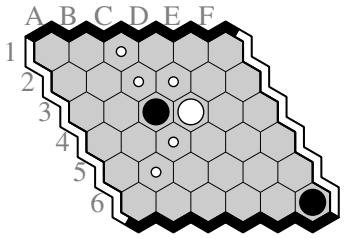
(91)



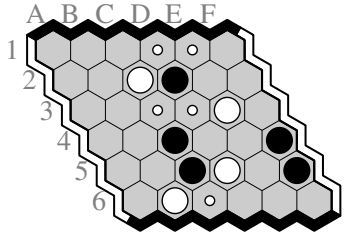
(92)



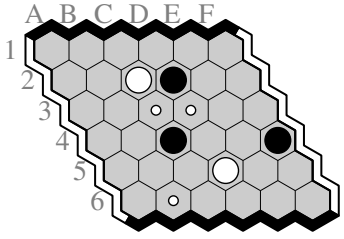
(93)



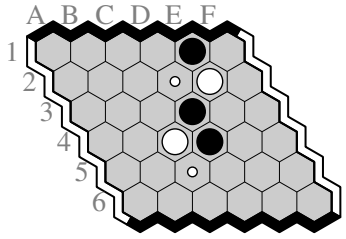
(94)



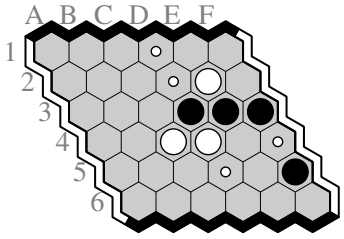
(95)



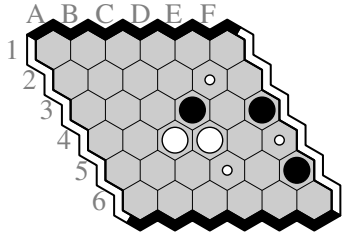
(96)



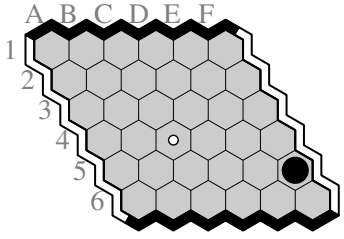
(97)



(98)



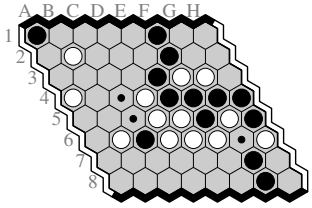
(99)



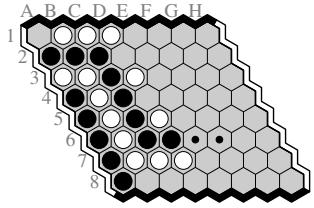
(100)

A.2 Hex 8×8

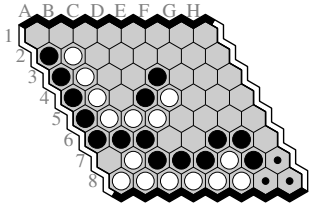
A.2.1 Black-to-play Test Positions



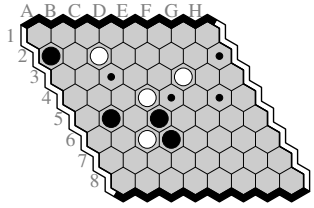
(1)



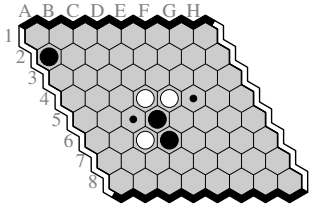
(2)



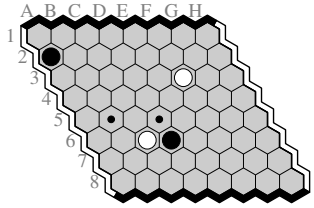
(3)



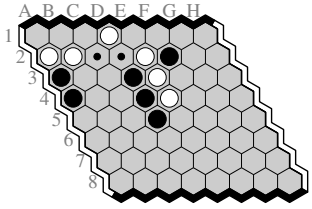
(4)



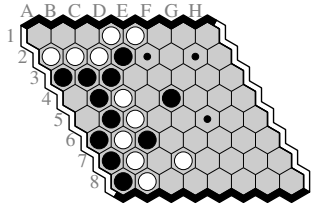
(5)



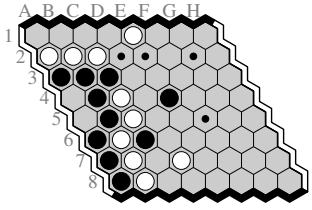
(6)



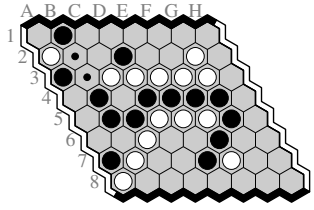
(7)



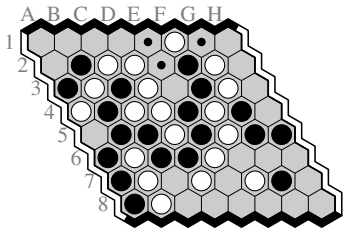
(8)



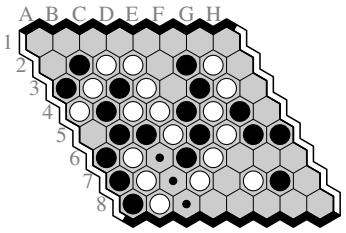
(9)



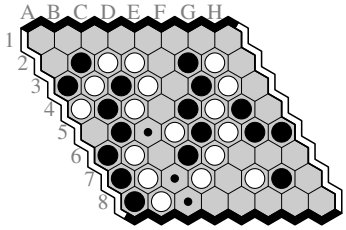
(10)



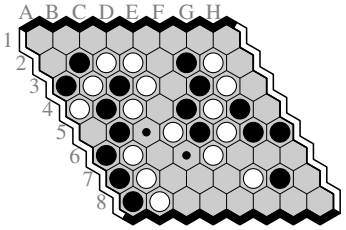
(11)



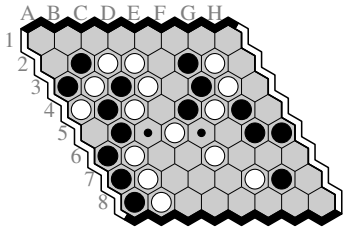
(12)



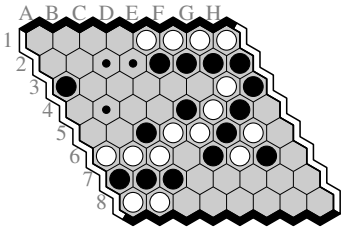
(13)



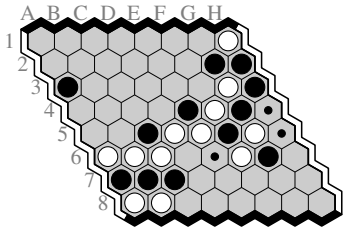
(14)



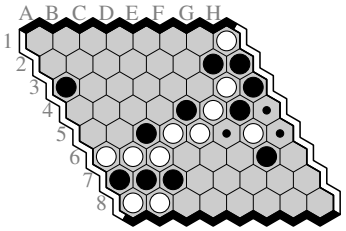
(15)



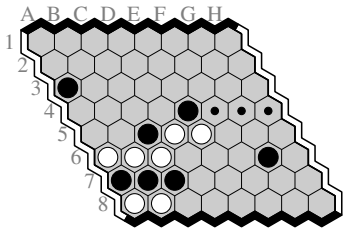
(16)



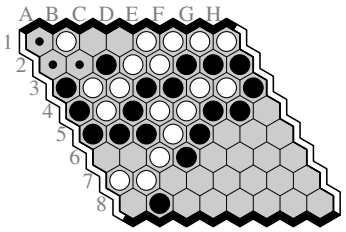
(17)



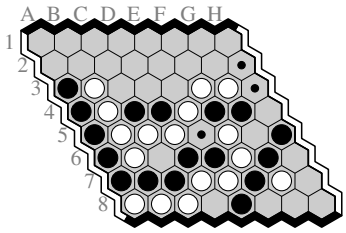
(18)



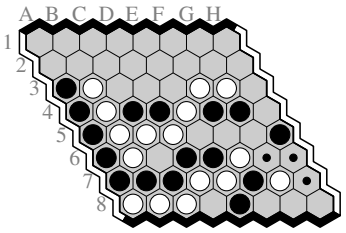
(19)



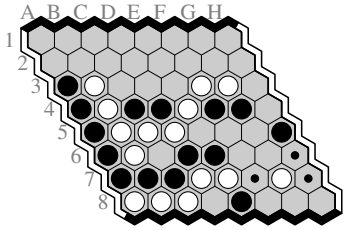
(20)



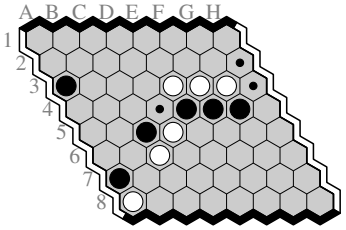
(21)



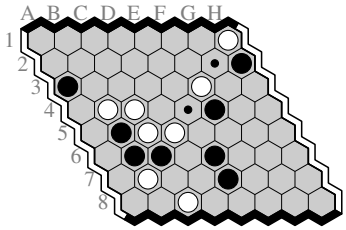
(22)



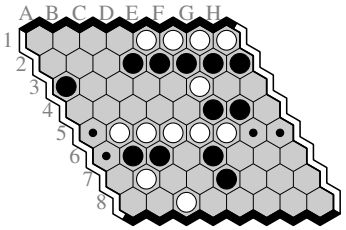
(23)



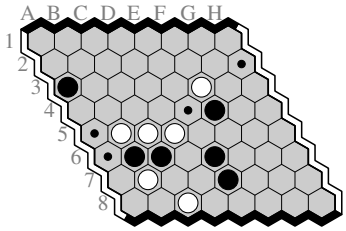
(24)



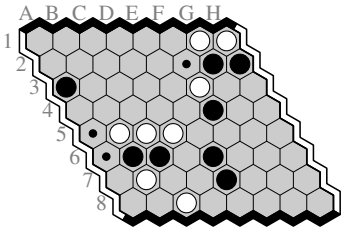
(25)



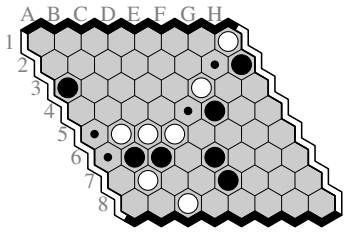
(26)



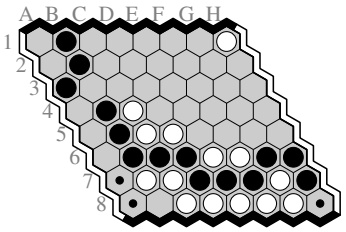
(27)



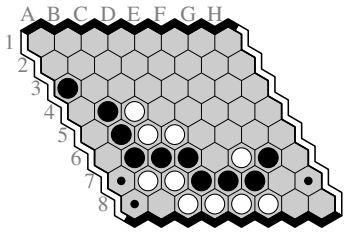
(28)



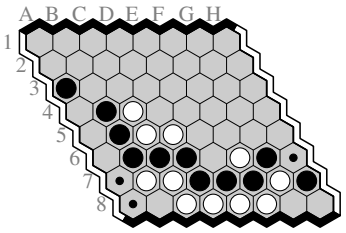
(29)



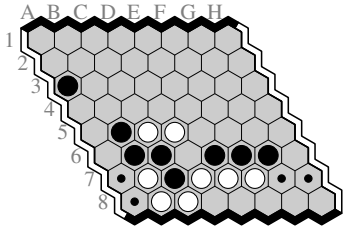
(30)



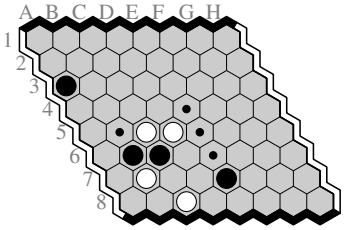
(31)



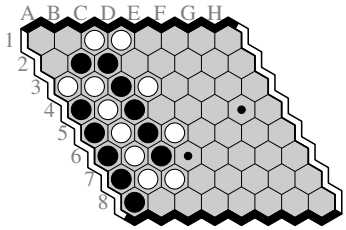
(32)



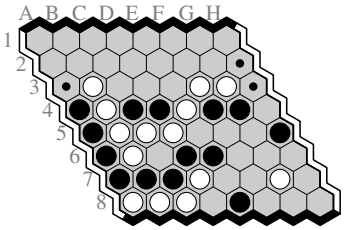
(33)



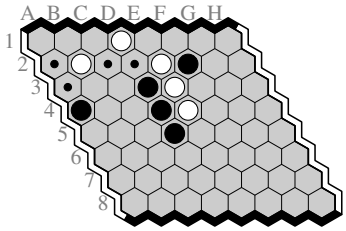
(34)



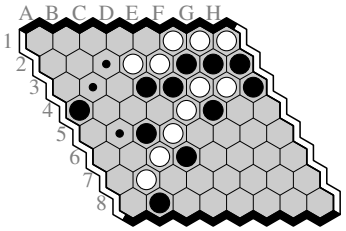
(35)



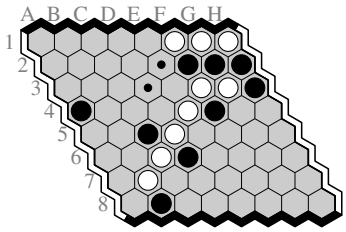
(36)



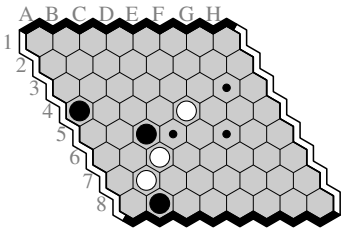
(37)



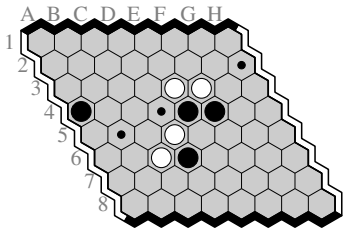
(38)



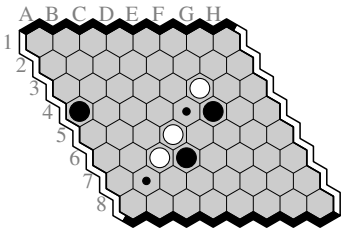
(39)



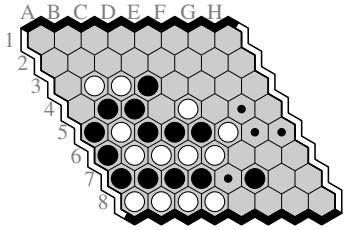
(40)



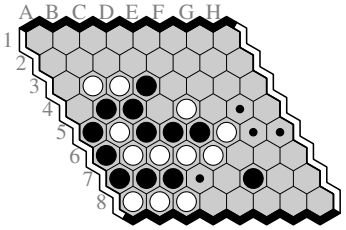
(41)



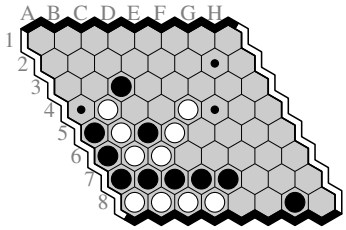
(42)



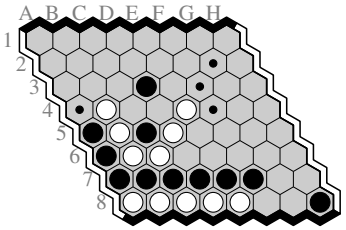
(43)



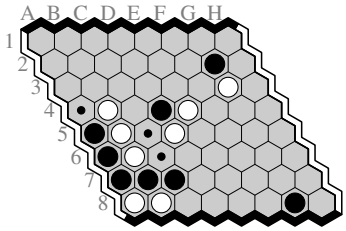
(44)



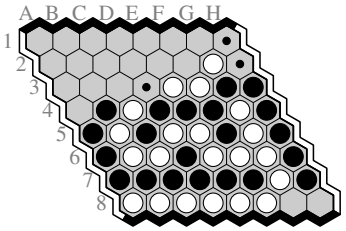
(45)



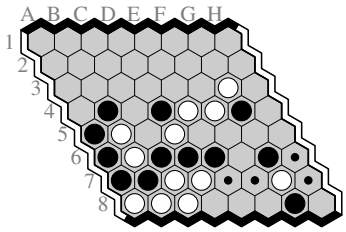
(46)



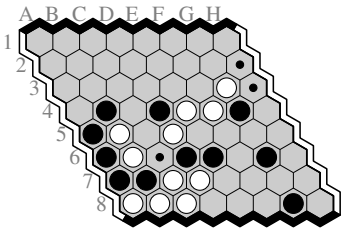
(47)



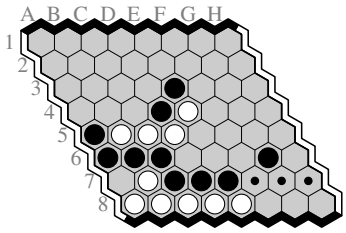
(48)



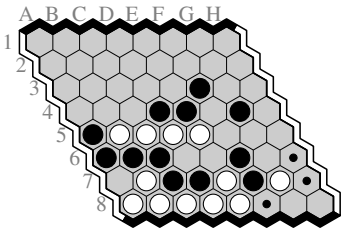
(49)



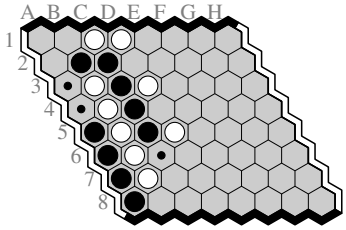
(50)



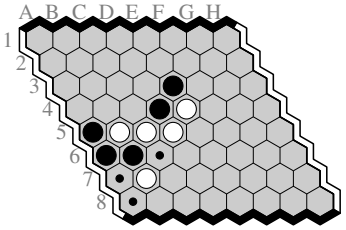
(51)



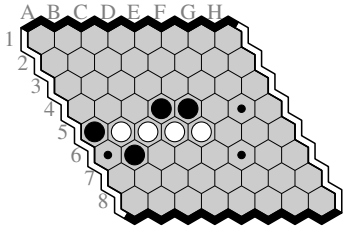
(52)



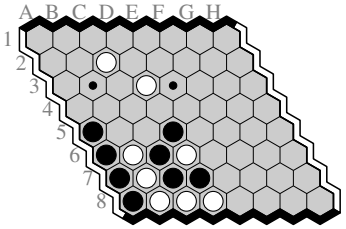
(53)



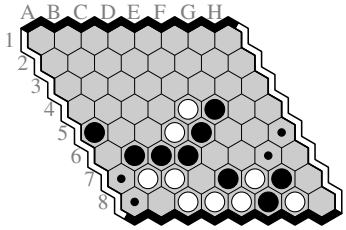
(54)



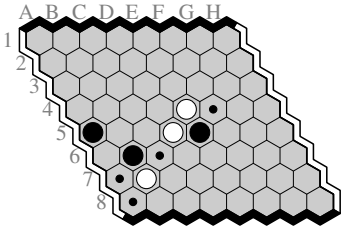
(55)



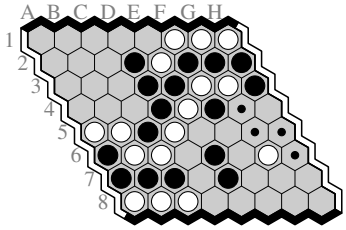
(56)



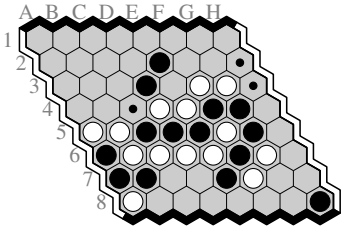
(57)



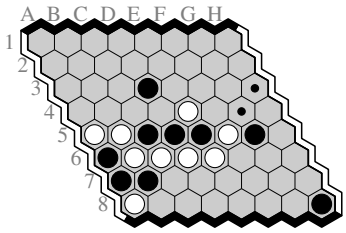
(58)



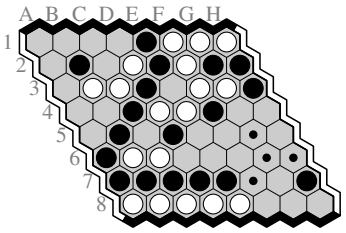
(59)



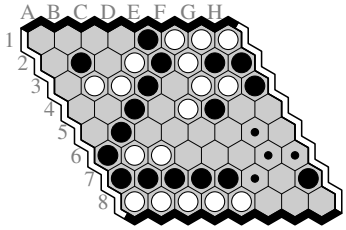
(60)



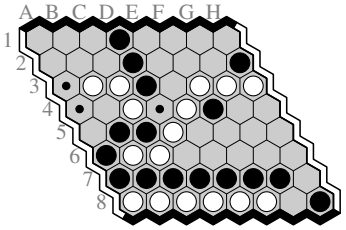
(61)



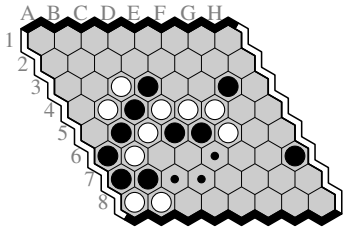
(62)



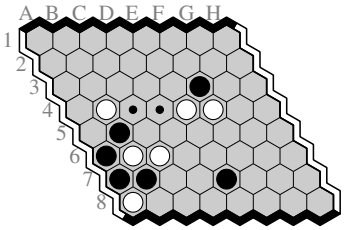
(63)



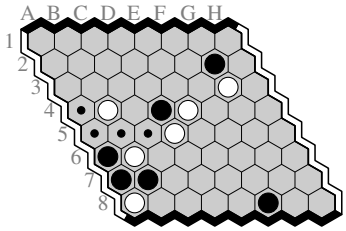
(64)



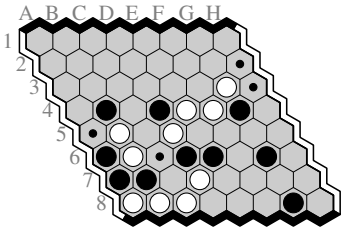
(65)



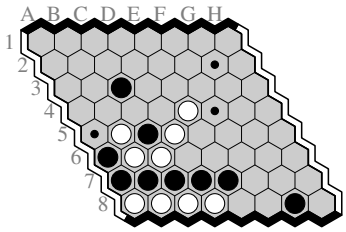
(66)



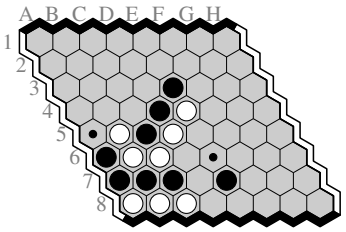
(67)



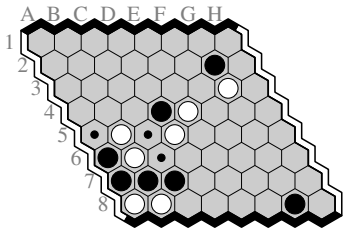
(68)



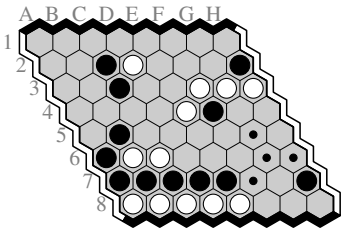
(69)



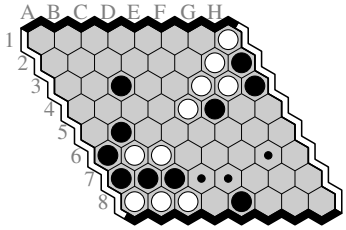
(70)



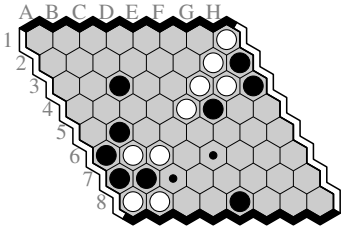
(71)



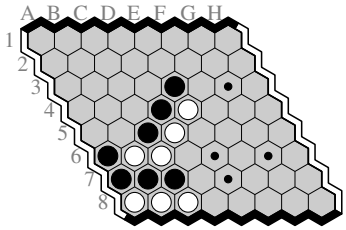
(72)



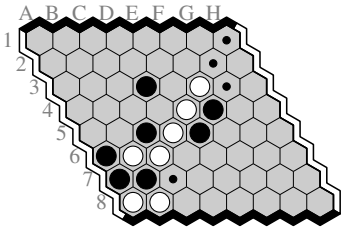
(73)



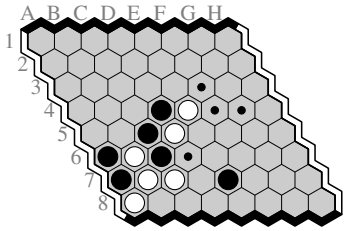
(74)



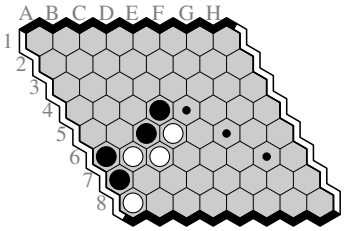
(75)



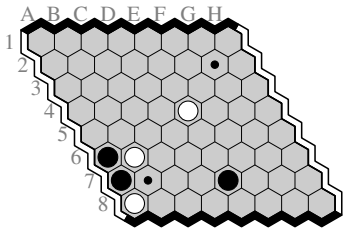
(76)



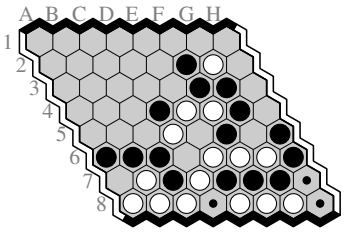
(77)



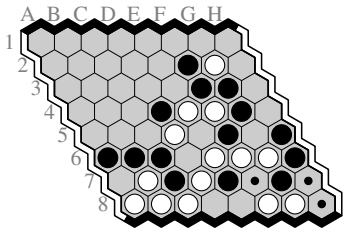
(78)



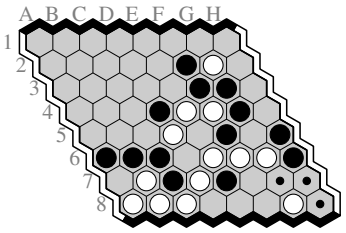
(79)



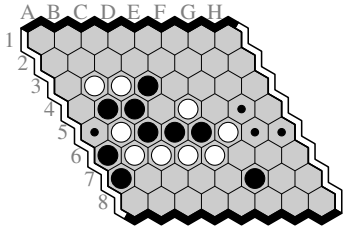
(80)



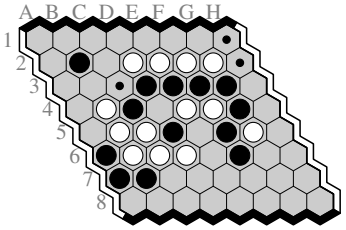
(81)



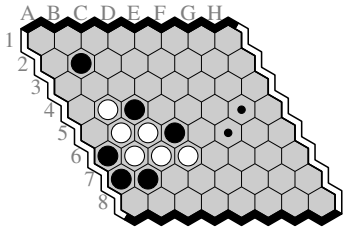
(82)



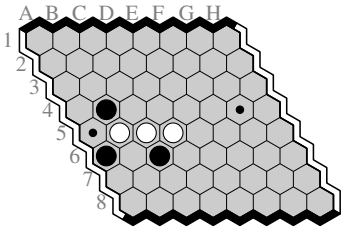
(83)



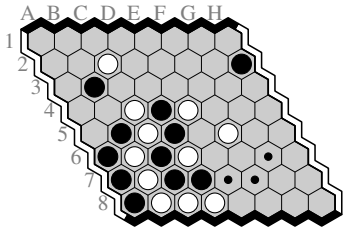
(84)



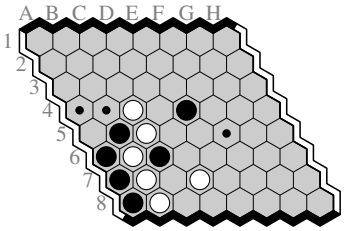
(85)



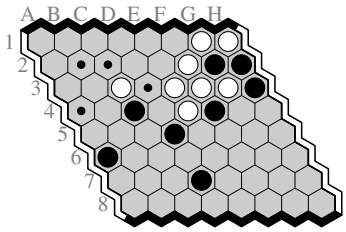
(86)



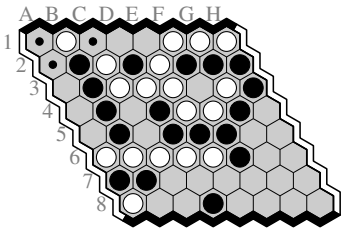
(87)



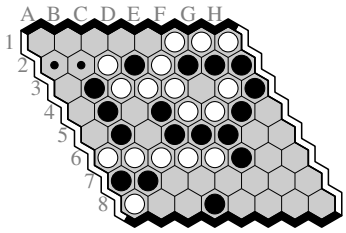
(88)



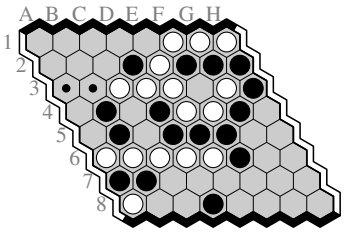
(89)



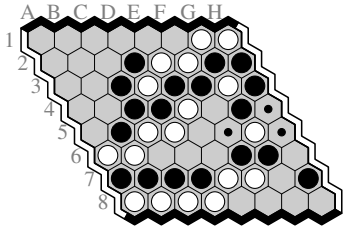
(90)



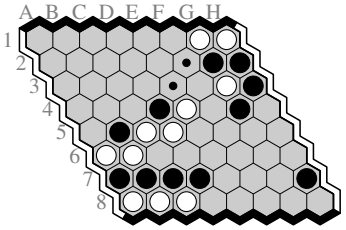
(91)



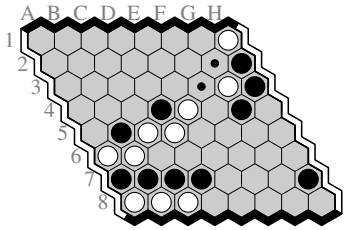
(92)



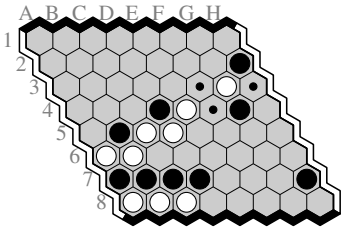
(93)



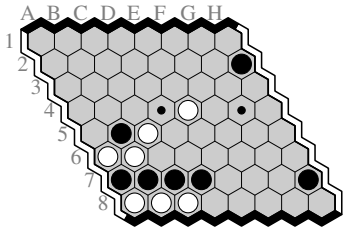
(94)



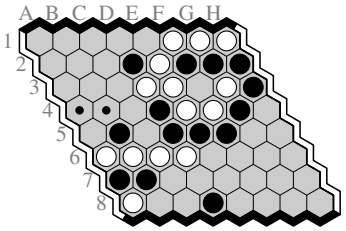
(95)



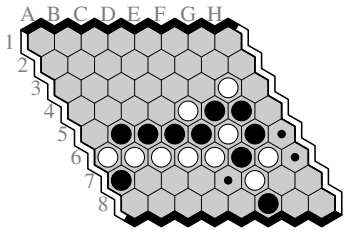
(96)



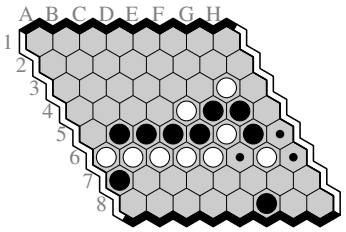
(97)



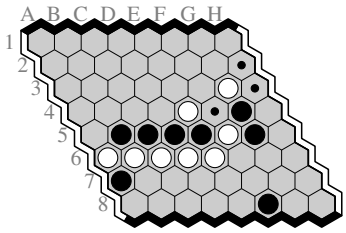
(98)



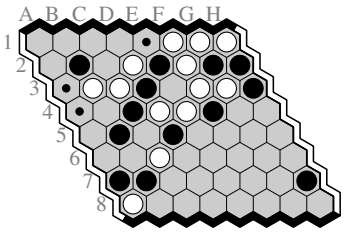
(99)



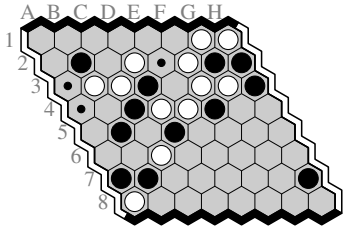
(100)



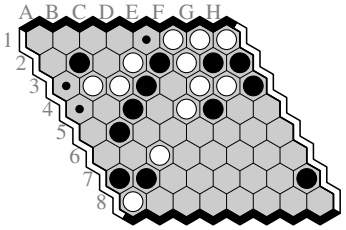
(101)



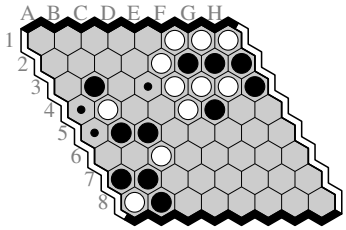
(102)



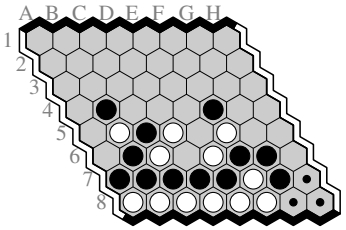
(103)



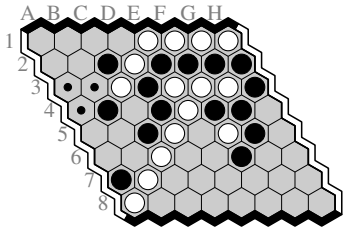
(104)



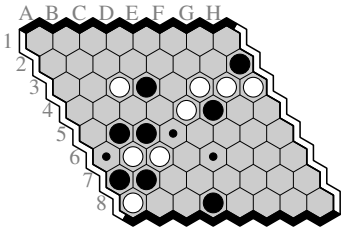
(105)



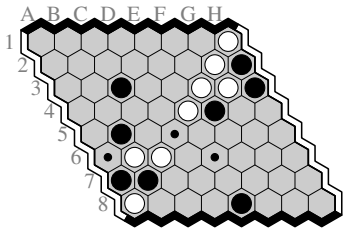
(106)



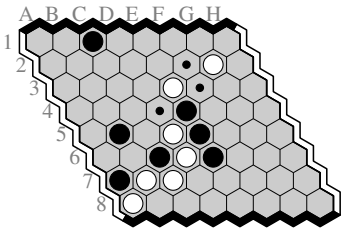
(107)



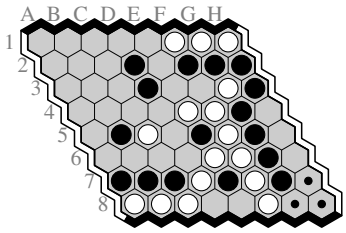
(108)



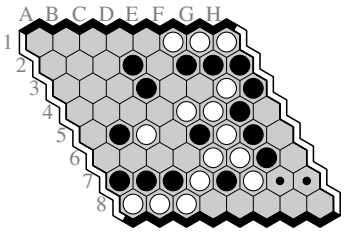
(109)



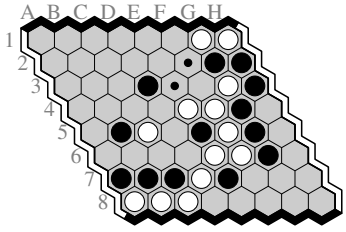
(110)



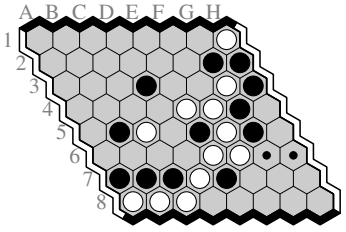
(111)



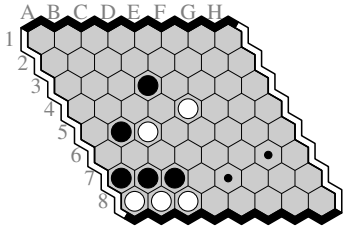
(112)



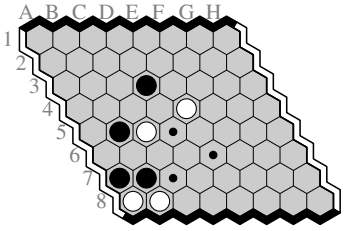
(113)



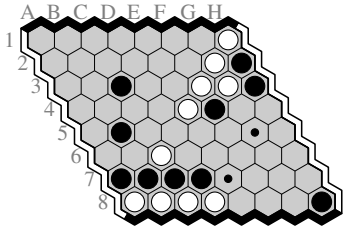
(114)



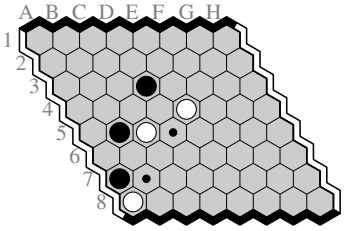
(115)



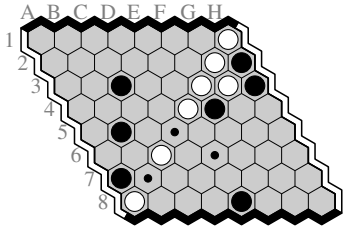
(116)



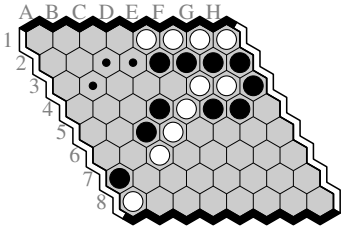
(117)



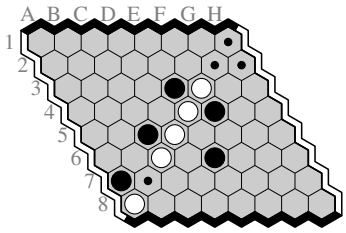
(118)



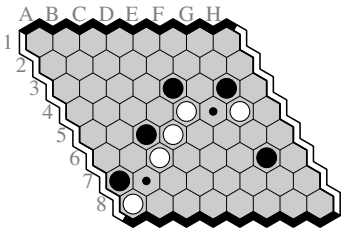
(119)



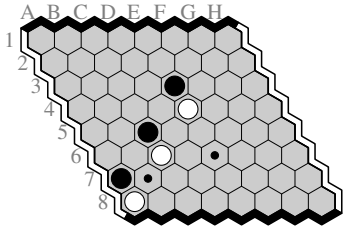
(120)



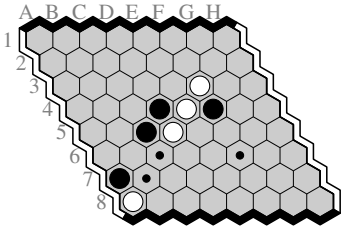
(121)



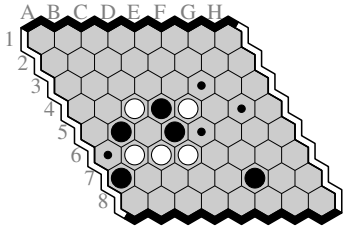
(122)



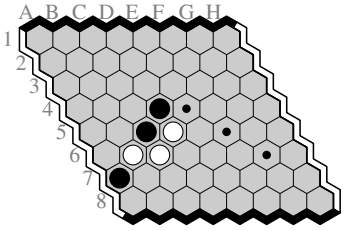
(123)



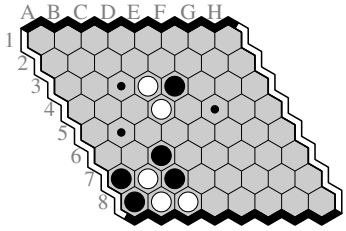
(124)



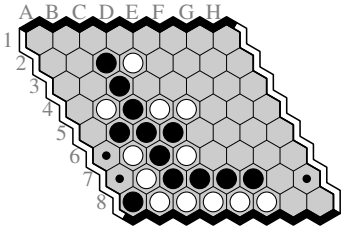
(125)



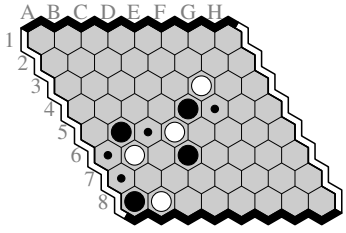
(126)



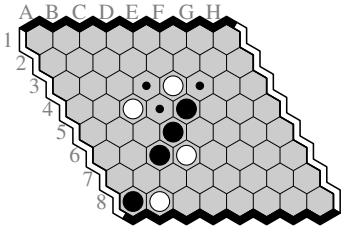
(127)



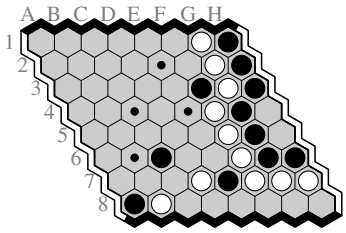
(128)



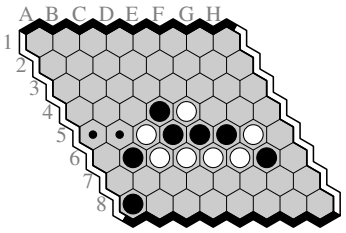
(129)



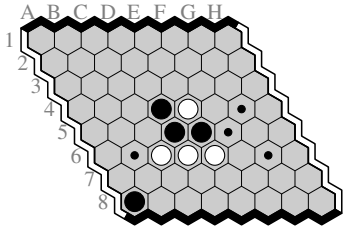
(130)



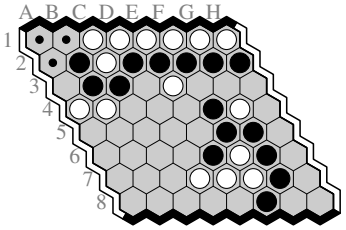
(131)



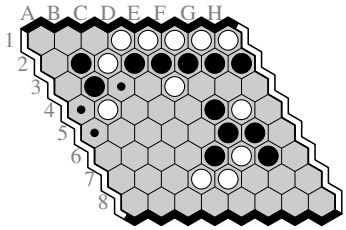
(132)



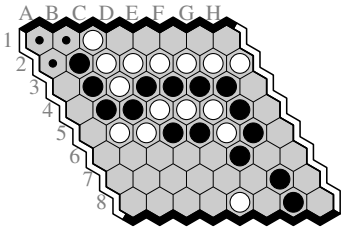
(133)



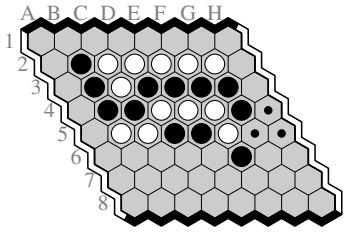
(134)



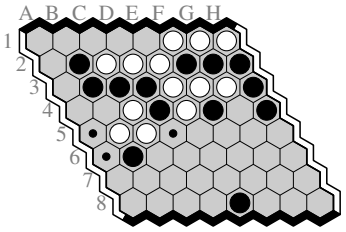
(135)



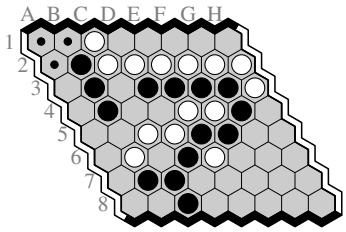
(136)



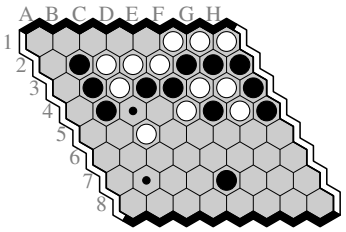
(137)



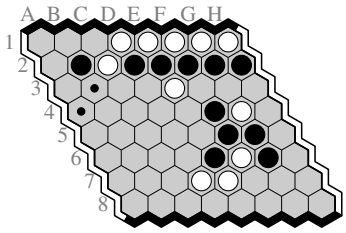
(138)



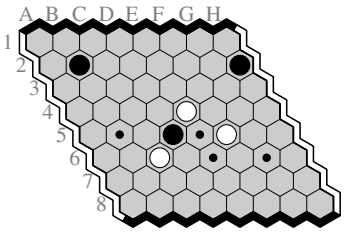
(139)



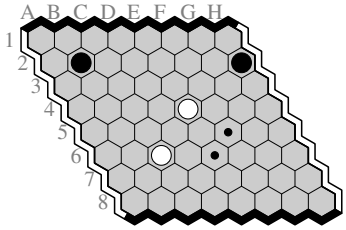
(140)



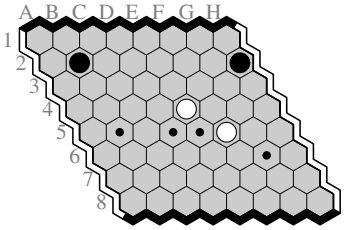
(141)



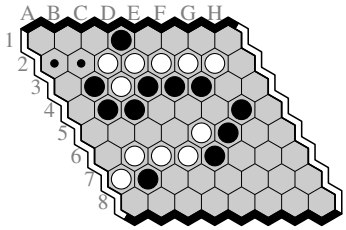
(142)



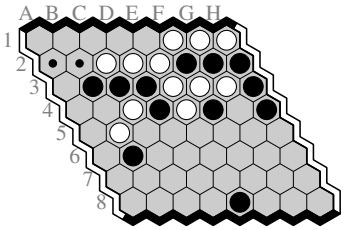
(143)



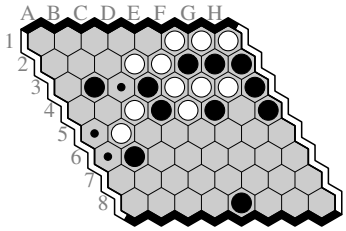
(144)



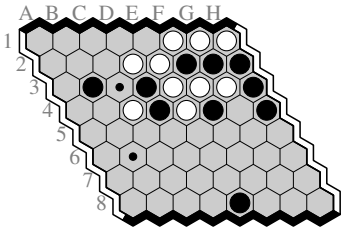
(145)



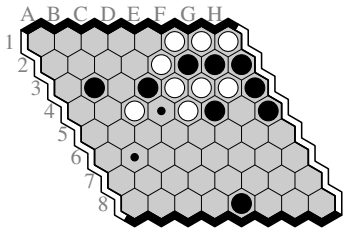
(146)



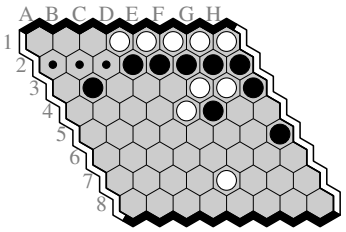
(147)



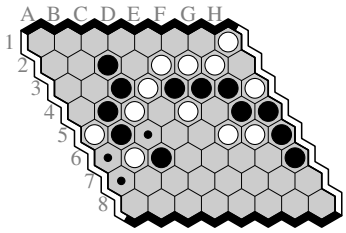
(148)



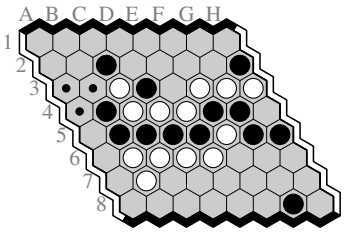
(149)



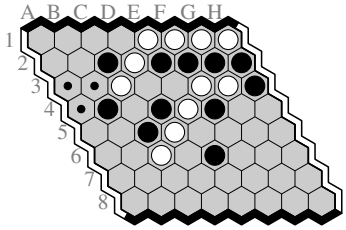
(150)



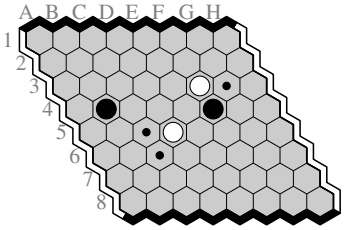
(151)



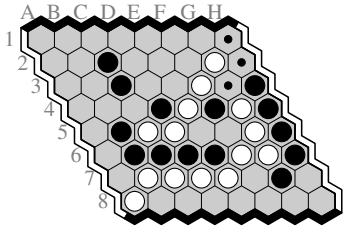
(152)



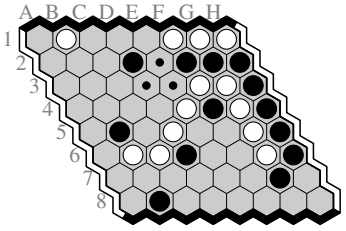
(153)



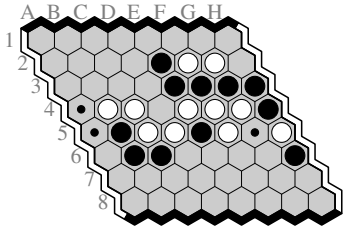
(154)



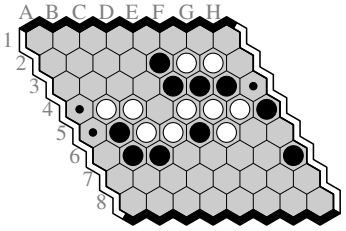
(155)



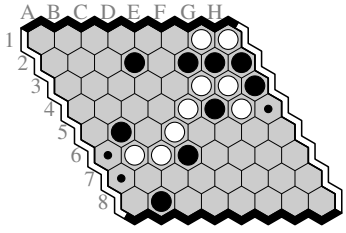
(156)



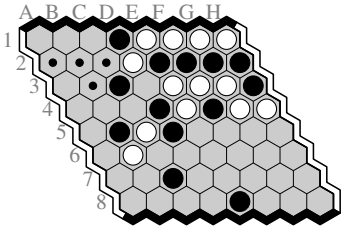
(157)



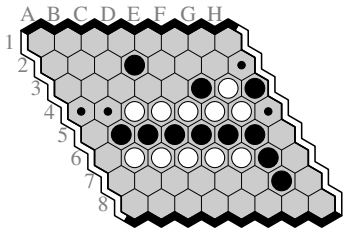
(158)



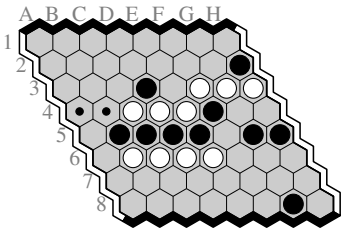
(159)



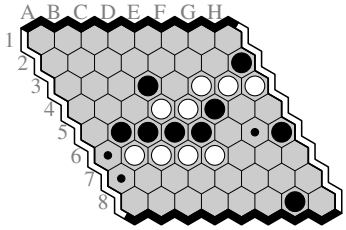
(160)



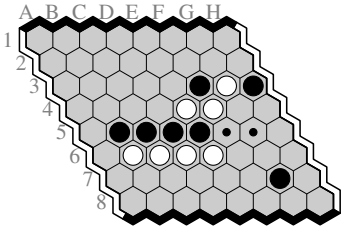
(161)



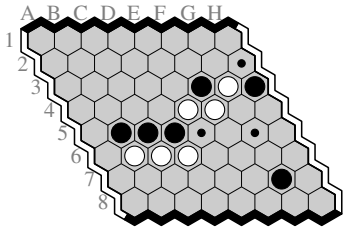
(162)



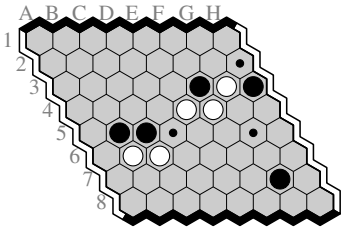
(163)



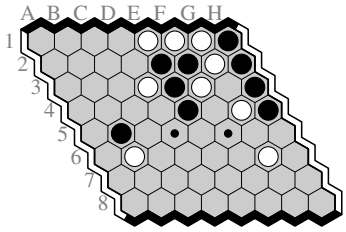
(164)



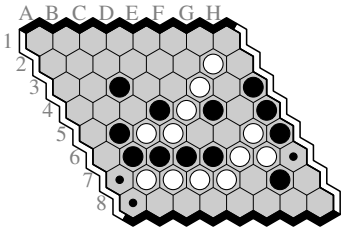
(165)



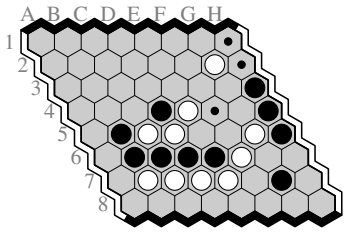
(166)



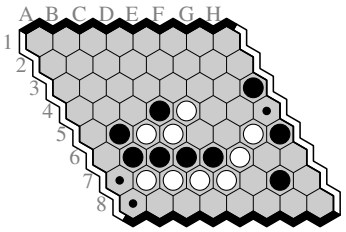
(167)



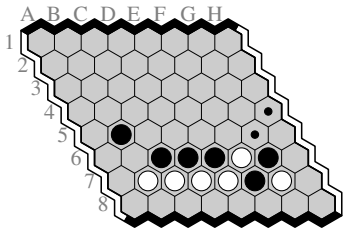
(168)



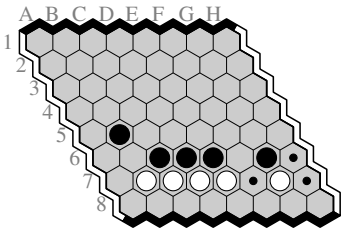
(169)



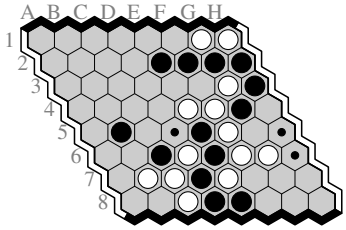
(170)



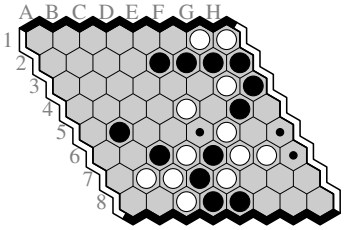
(171)



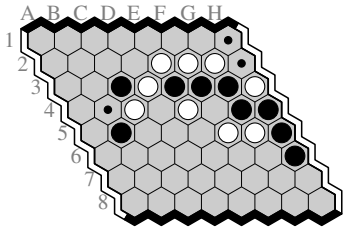
(172)



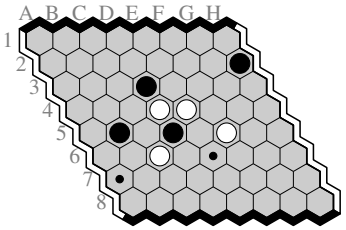
(173)



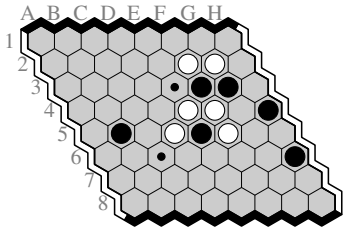
(174)



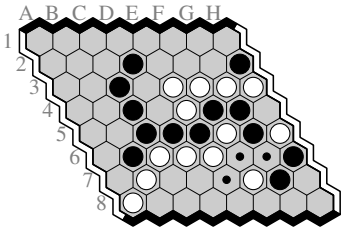
(175)



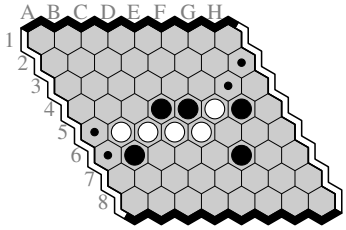
(176)



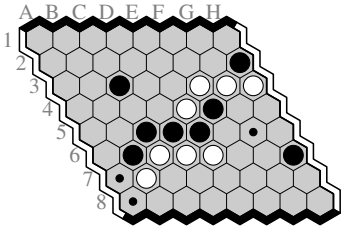
(177)



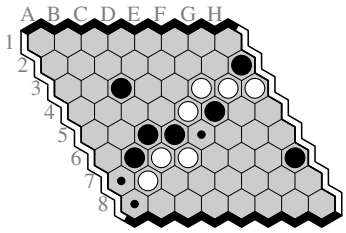
(178)



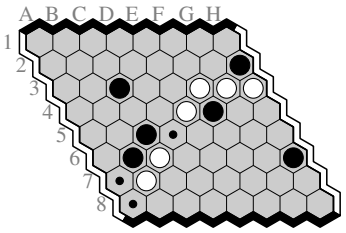
(179)



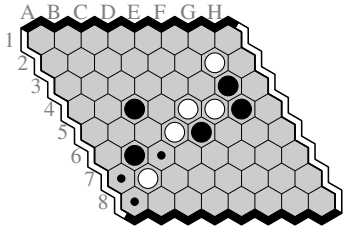
(180)



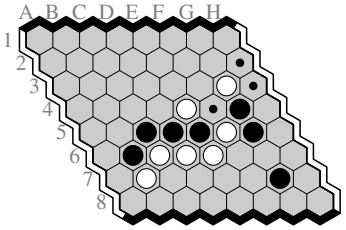
(181)



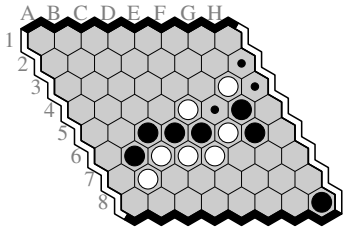
(182)



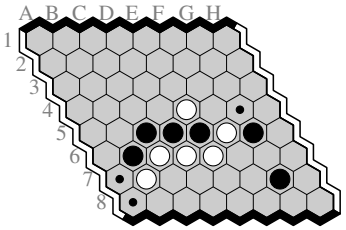
(183)



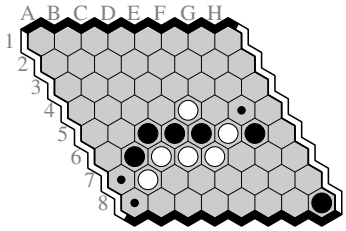
(184)



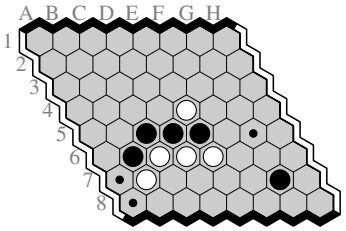
(185)



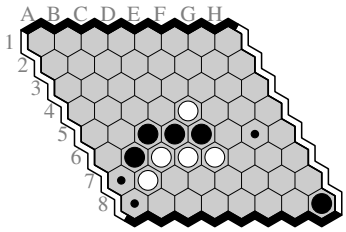
(186)



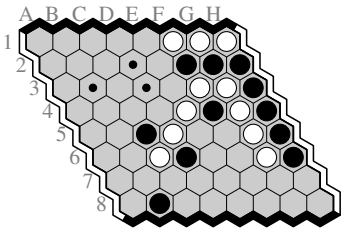
(187)



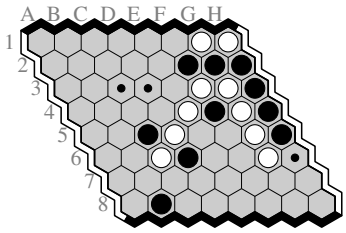
(188)



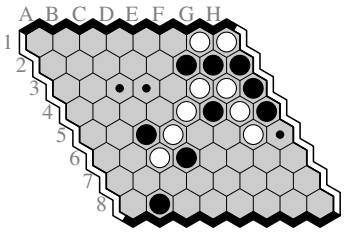
(189)



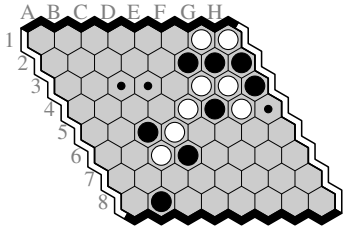
(190)



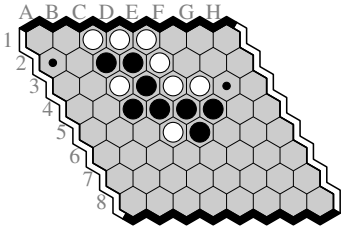
(191)



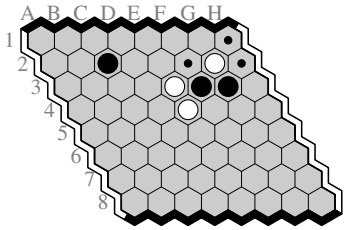
(192)



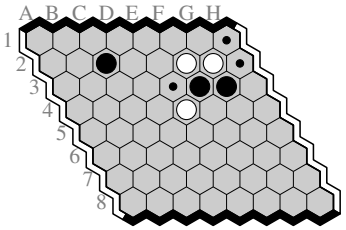
(193)



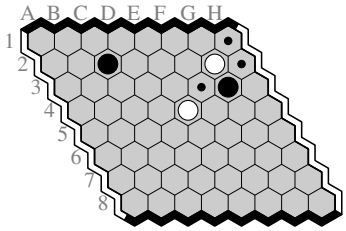
(194)



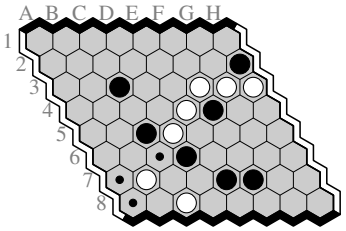
(195)



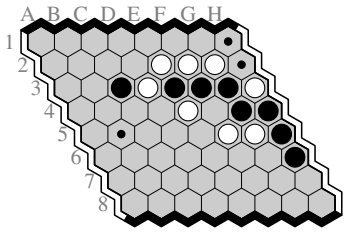
(196)



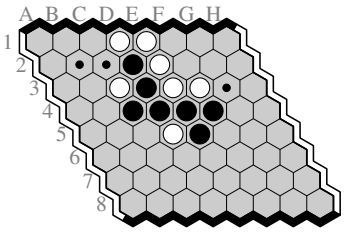
(197)



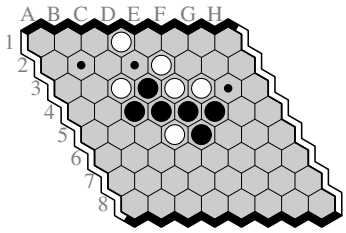
(198)



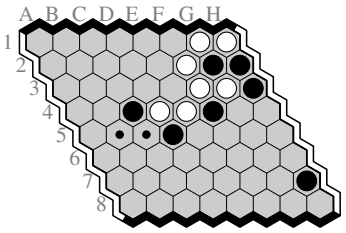
(199)



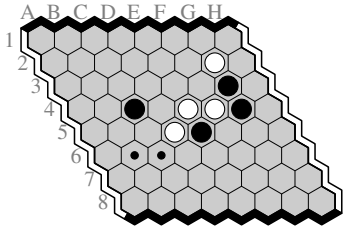
(200)



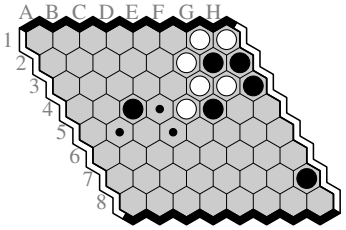
(201)



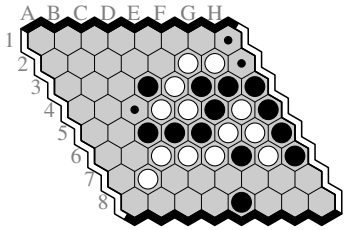
(202)



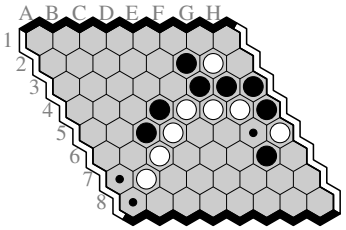
(203)



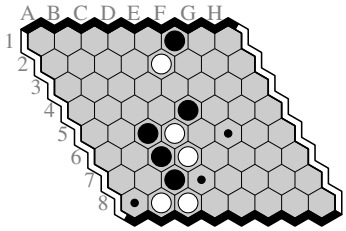
(204)



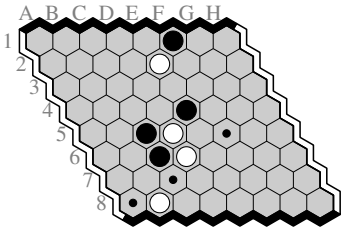
(205)



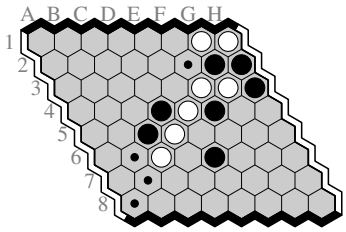
(206)



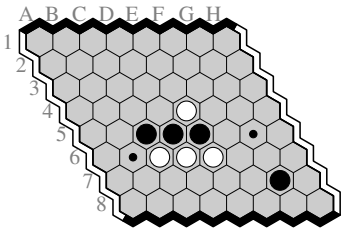
(207)



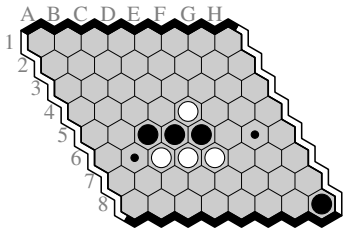
(208)



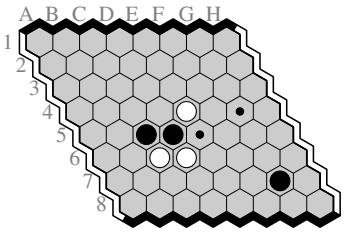
(209)



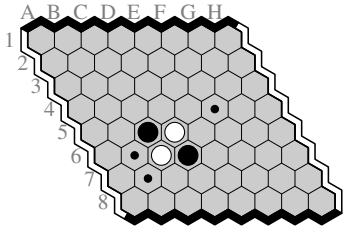
(210)



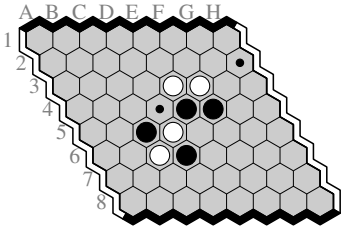
(211)



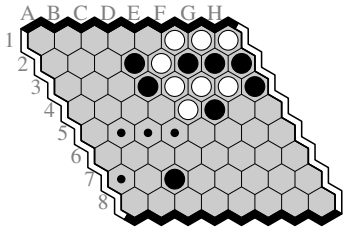
(212)



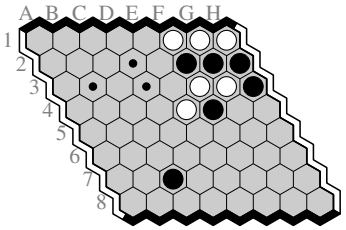
(213)



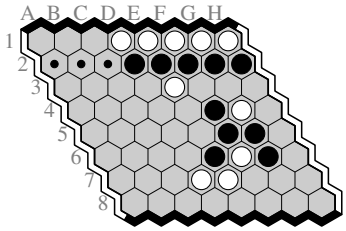
(214)



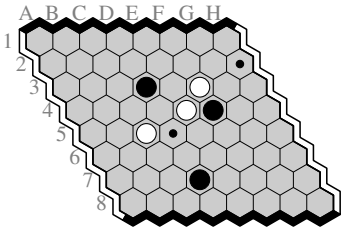
(215)



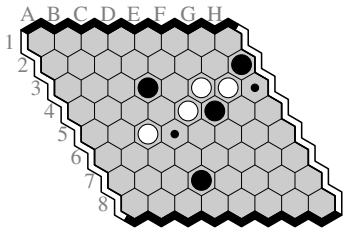
(216)



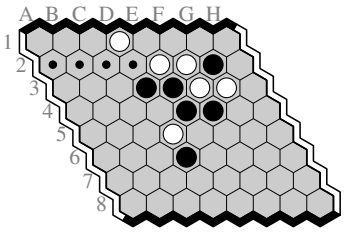
(217)



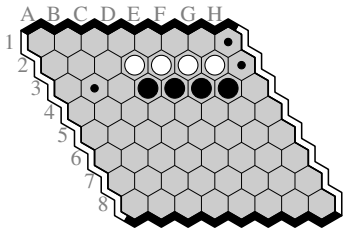
(218)



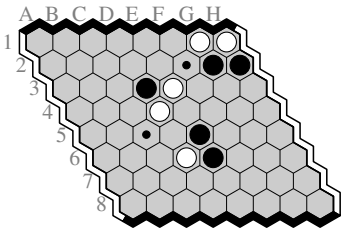
(219)



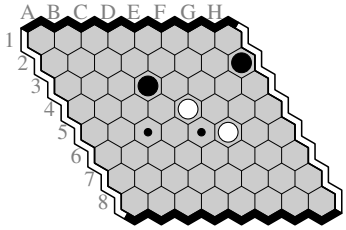
(220)



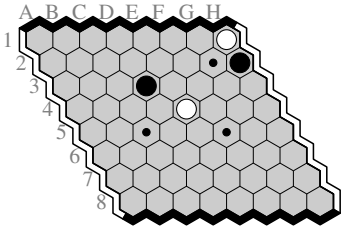
(221)



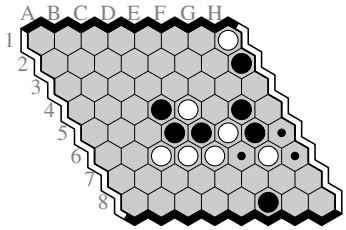
(222)



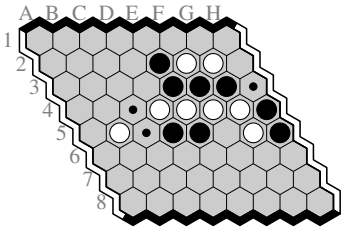
(223)



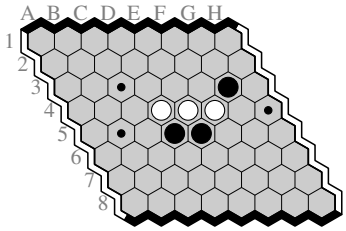
(224)



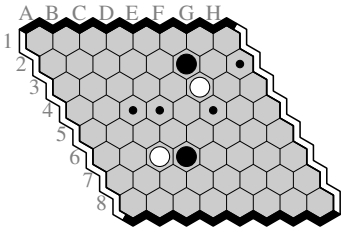
(225)



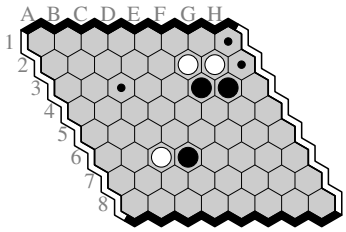
(226)



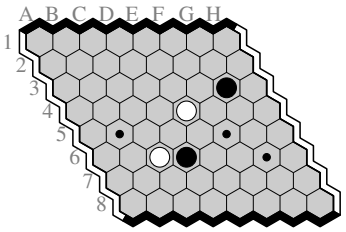
(227)



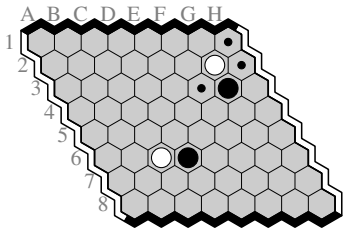
(228)



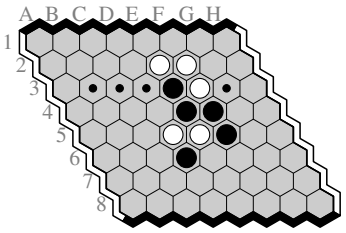
(229)



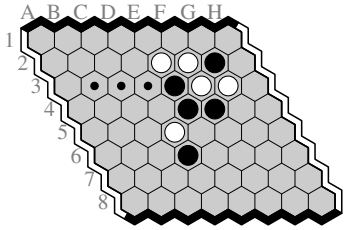
(230)



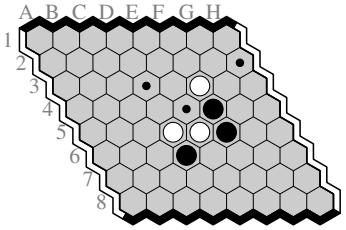
(231)



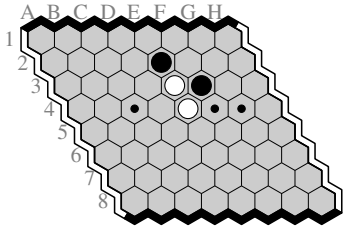
(232)



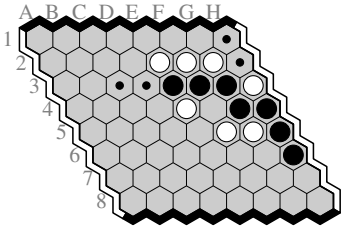
(233)



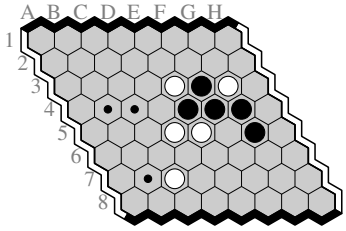
(234)



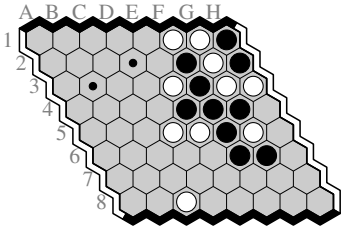
(235)



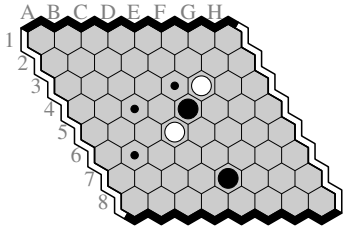
(236)



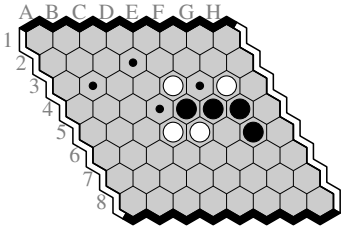
(237)



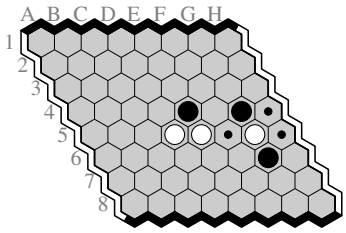
(238)



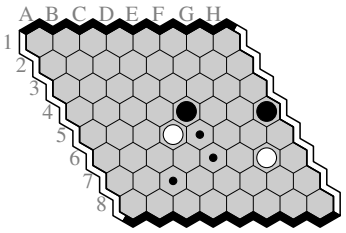
(239)



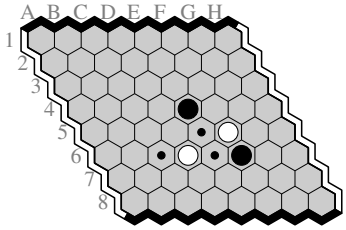
(240)



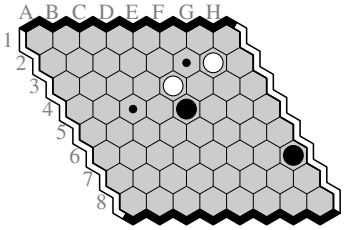
(241)



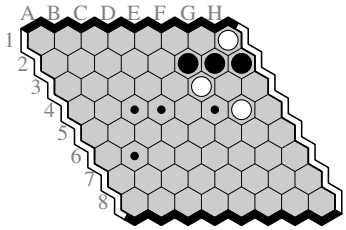
(242)



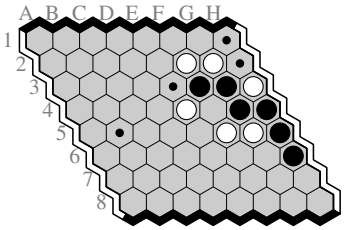
(243)



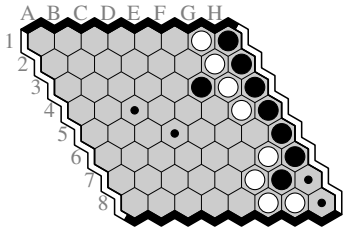
(244)



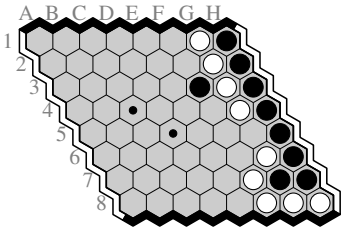
(245)



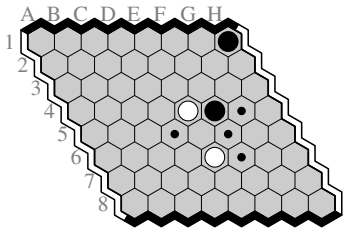
(246)



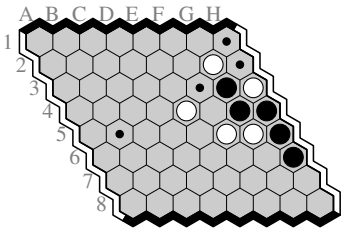
(247)



(248)

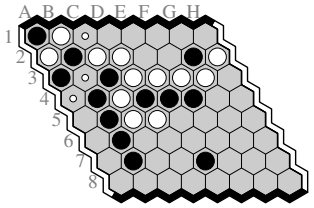


(249)

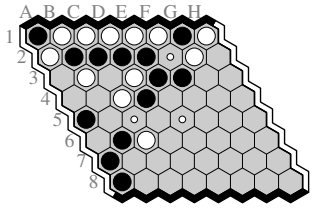


(250)

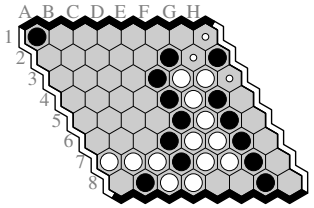
A.2.2 White-to-play Test Positions



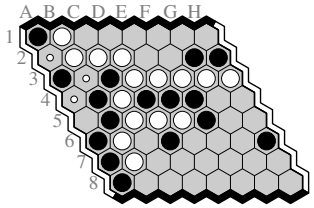
(1)



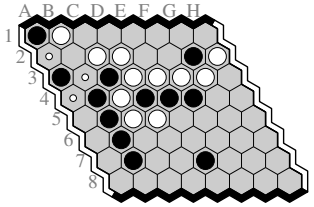
(2)



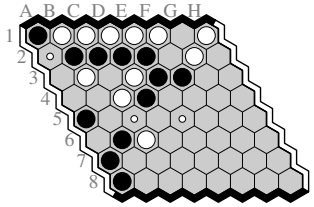
(3)



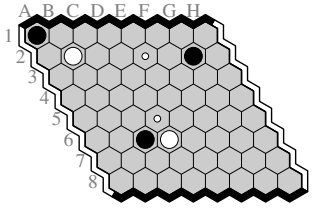
(4)



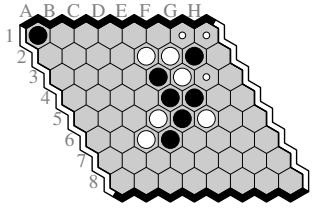
(5)



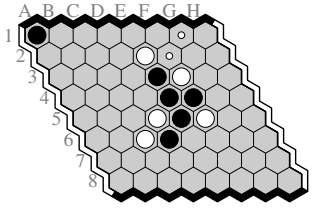
(6)



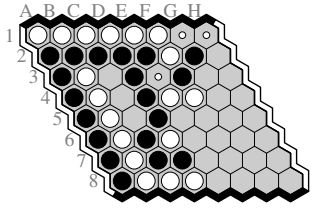
(7)



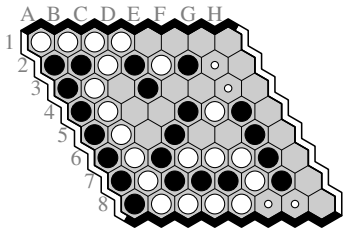
(8)



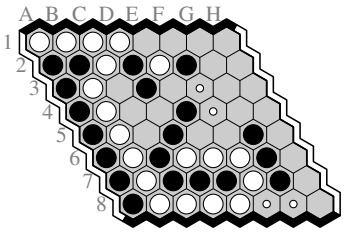
(9)



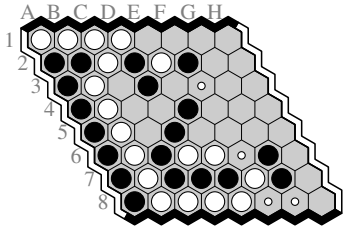
(10)



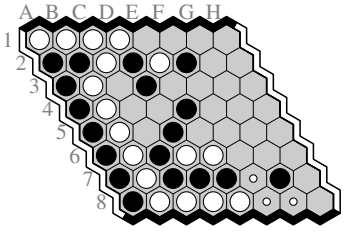
(11)



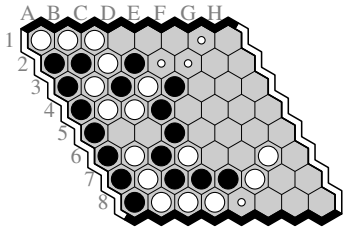
(12)



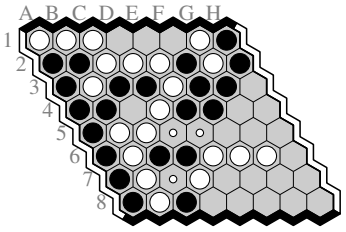
(13)



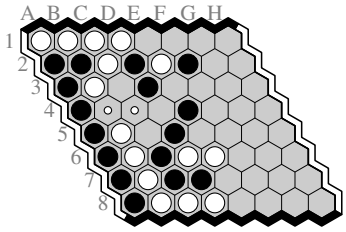
(14)



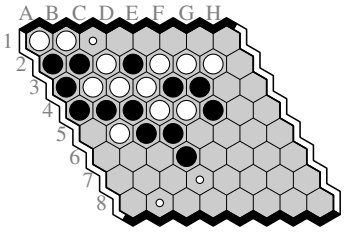
(15)



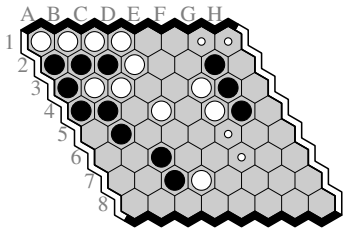
(16)



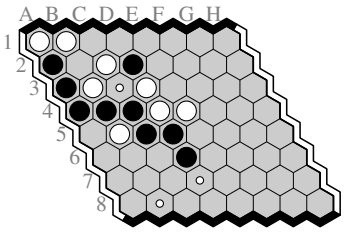
(17)



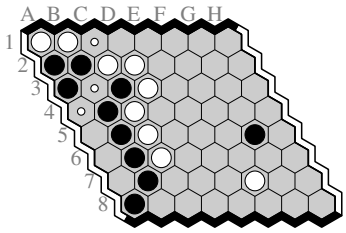
(18)



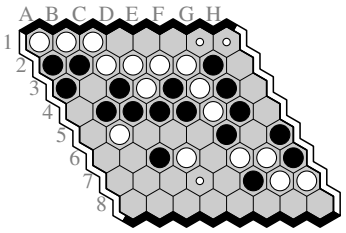
(19)



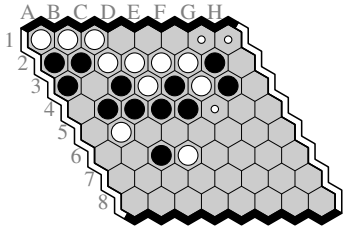
(20)



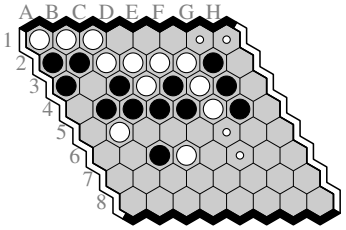
(21)



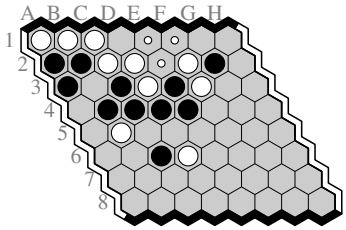
(22)



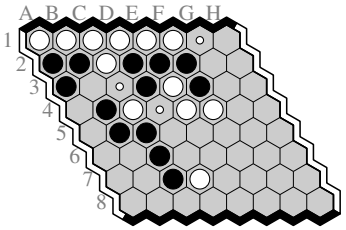
(23)



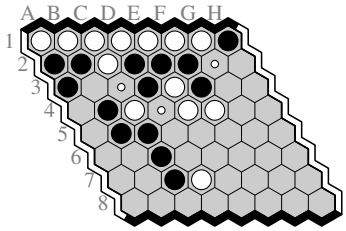
(24)



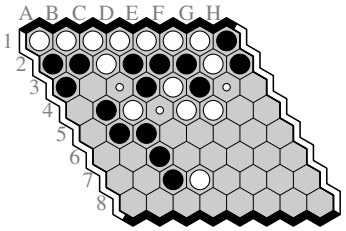
(25)



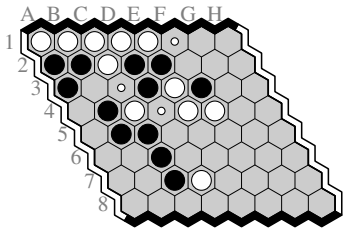
(26)



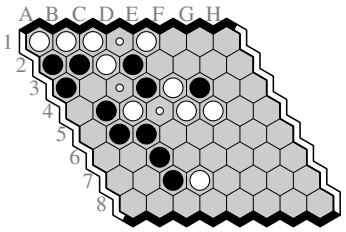
(27)



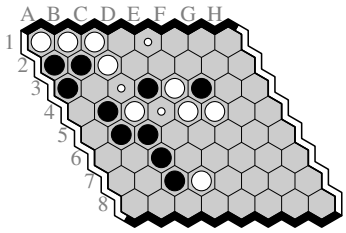
(28)



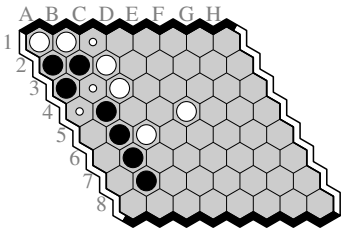
(29)



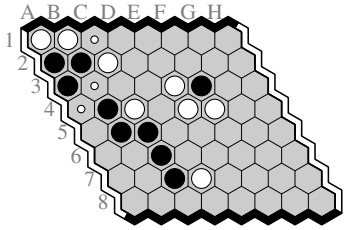
(30)



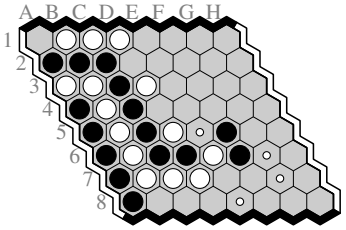
(31)



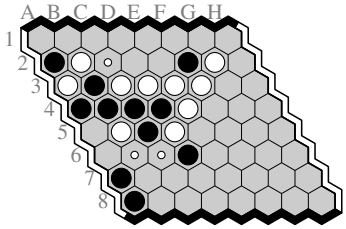
(32)



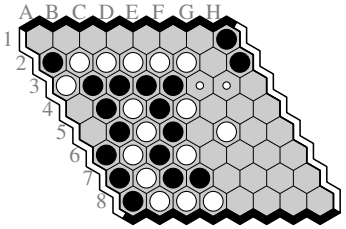
(33)



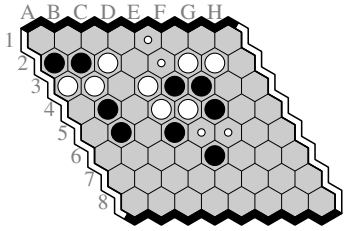
(34)



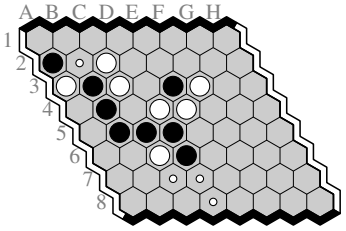
(35)



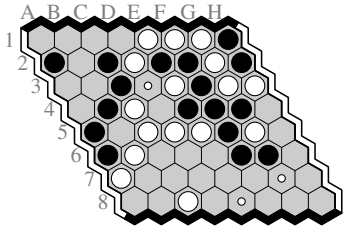
(36)



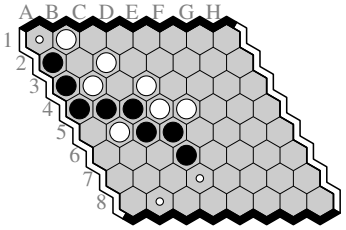
(37)



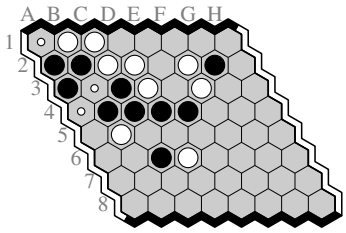
(38)



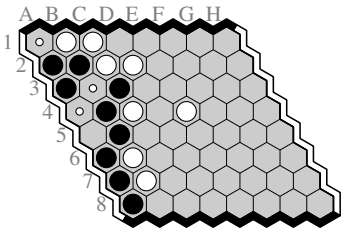
(39)



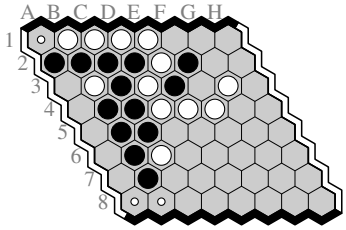
(40)



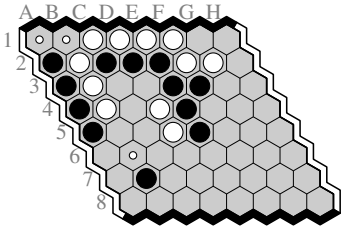
(41)



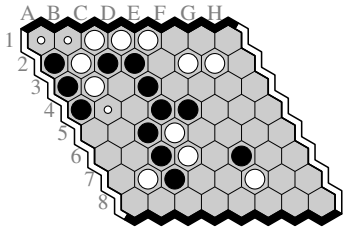
(42)



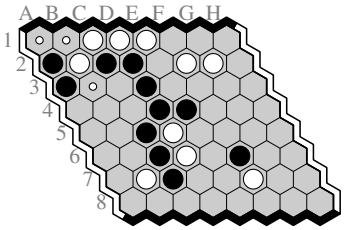
(43)



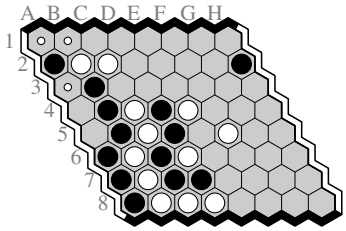
(44)



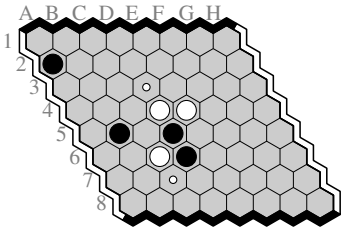
(45)



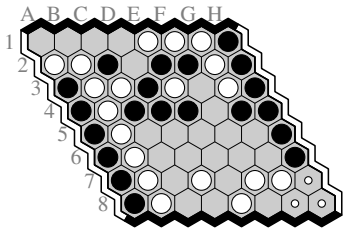
(46)



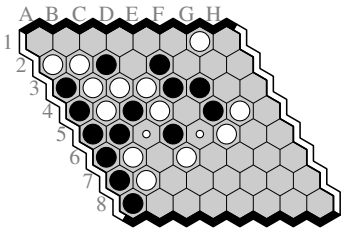
(47)



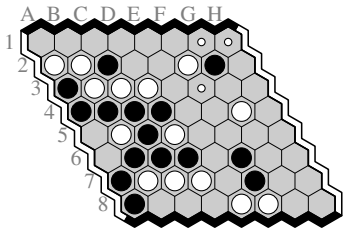
(48)



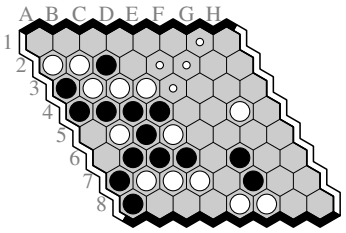
(49)



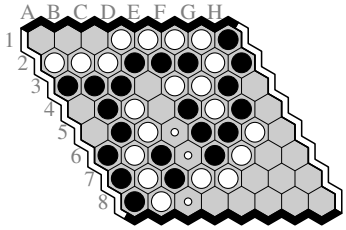
(50)



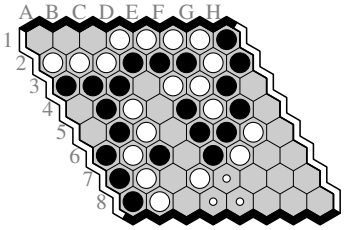
(51)



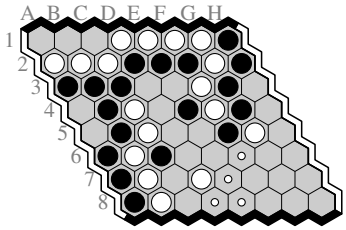
(52)



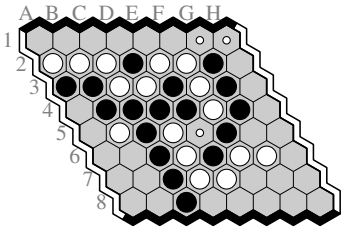
(53)



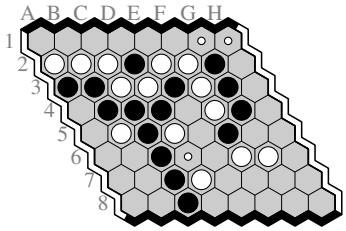
(54)



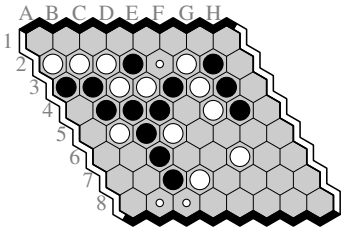
(55)



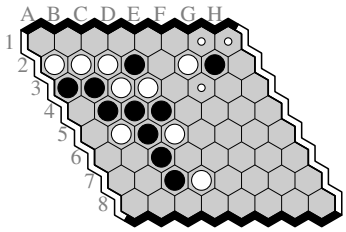
(56)



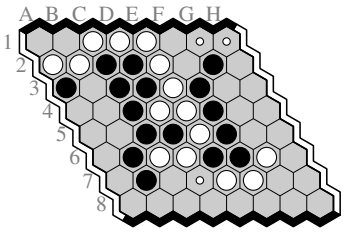
(57)



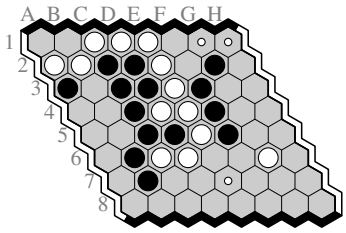
(58)



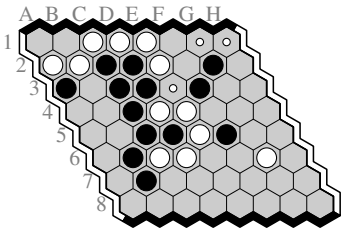
(59)



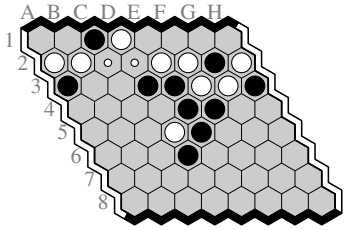
(60)



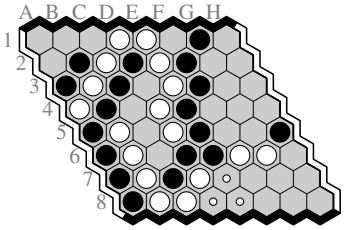
(61)



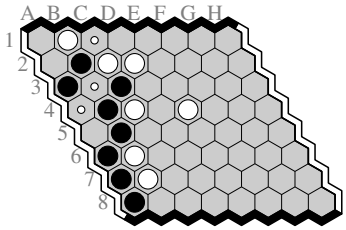
(62)



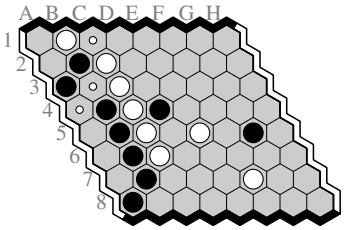
(63)



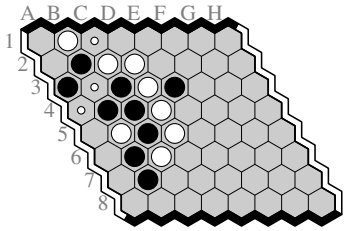
(64)



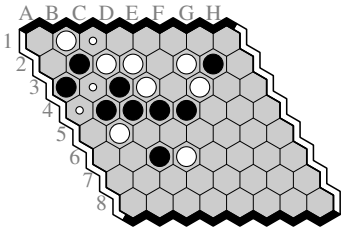
(65)



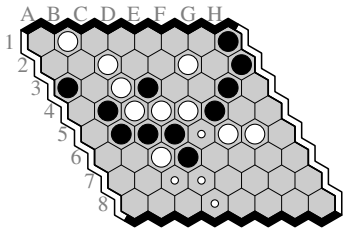
(66)



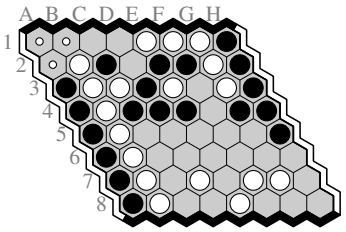
(67)



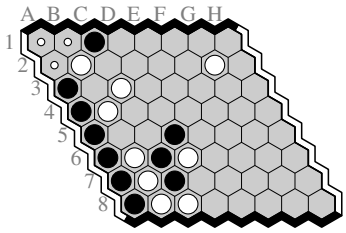
(68)



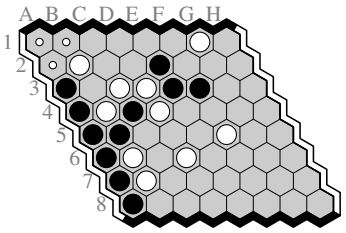
(69)



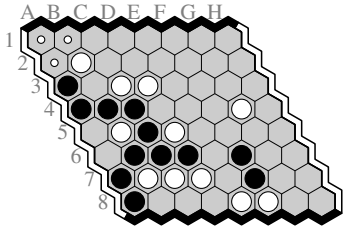
(70)



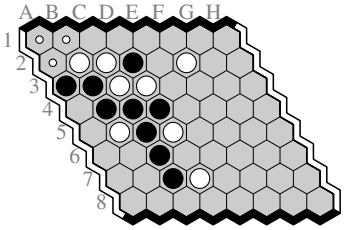
(71)



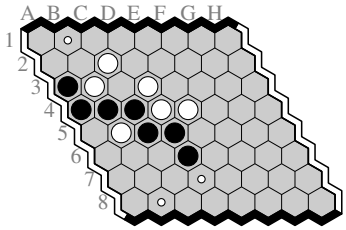
(72)



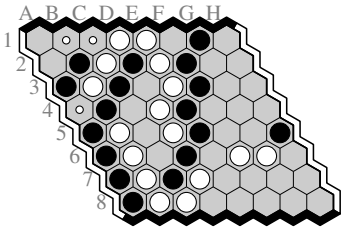
(73)



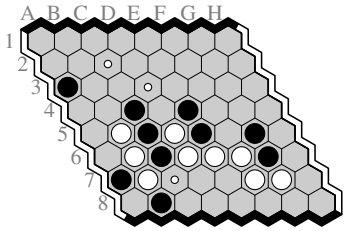
(74)



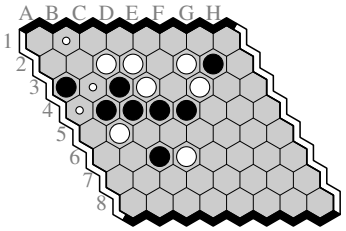
(75)



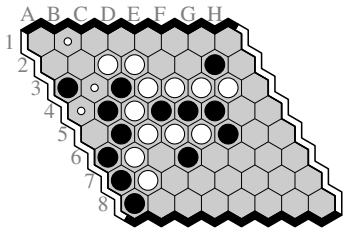
(76)



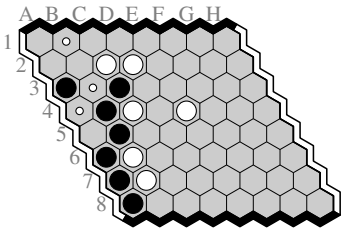
(77)



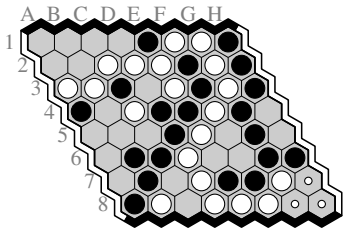
(78)



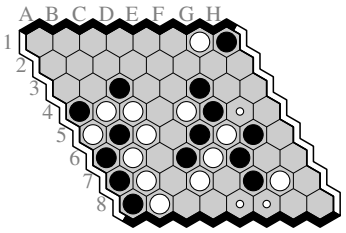
(79)



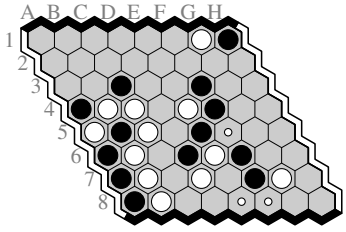
(80)



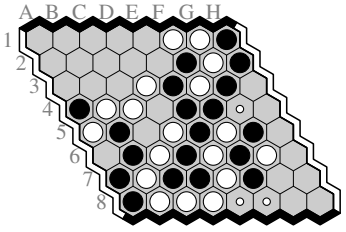
(81)



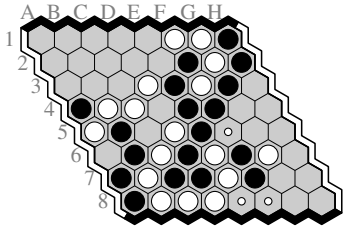
(82)



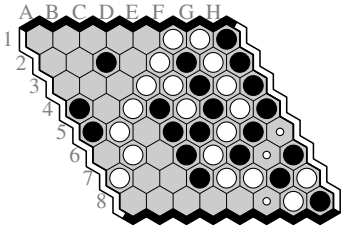
(83)



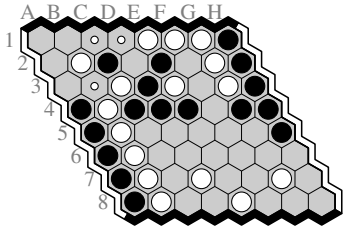
(84)



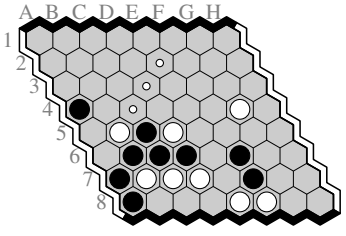
(85)



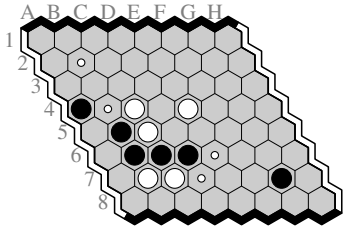
(86)



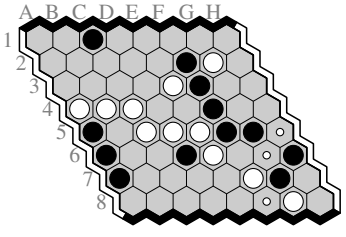
(87)



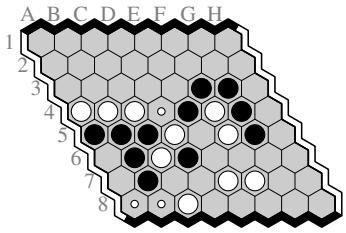
(88)



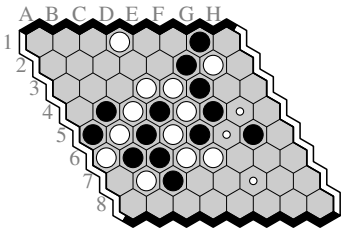
(89)



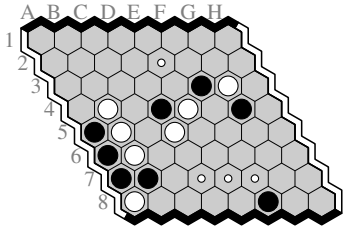
(90)



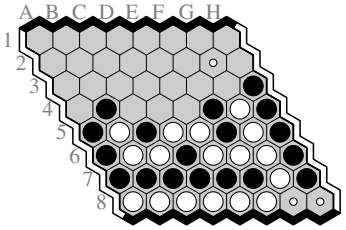
(91)



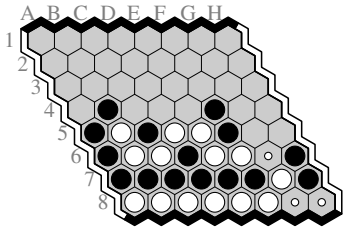
(92)



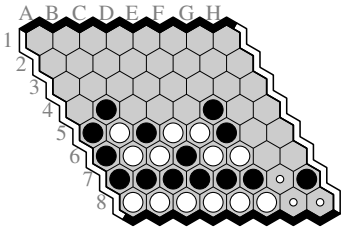
(93)



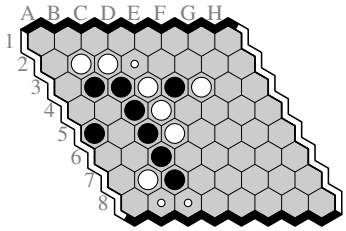
(94)



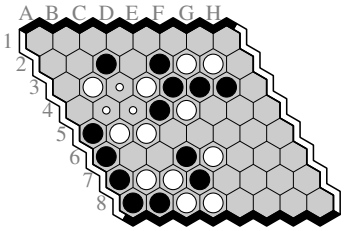
(95)



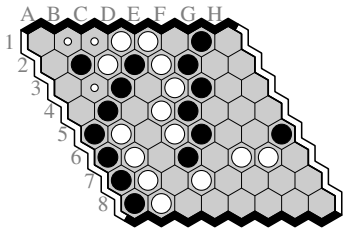
(96)



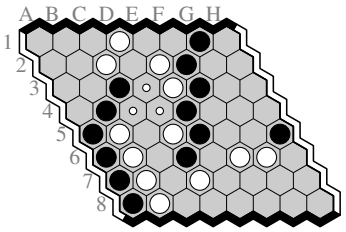
(97)



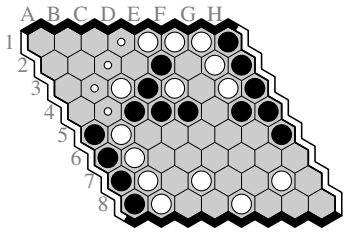
(98)



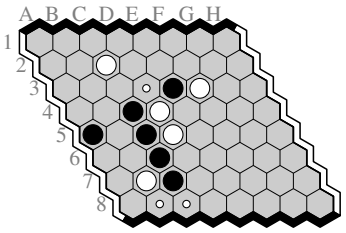
(99)



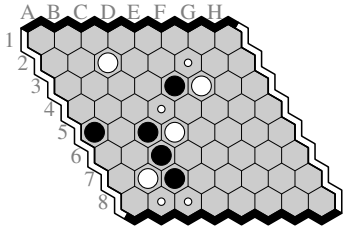
(100)



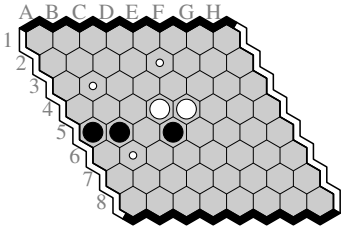
(101)



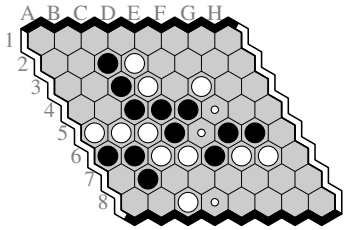
(102)



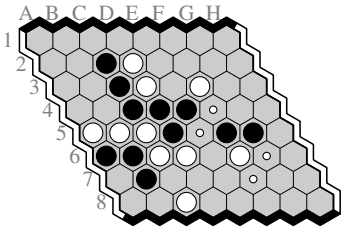
(103)



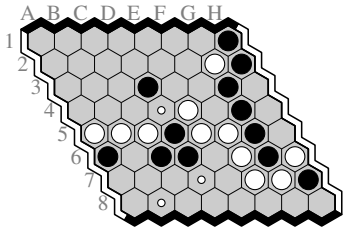
(104)



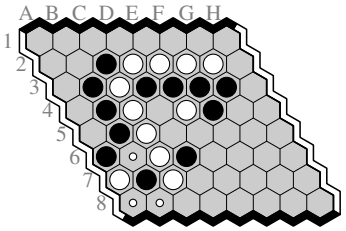
(105)



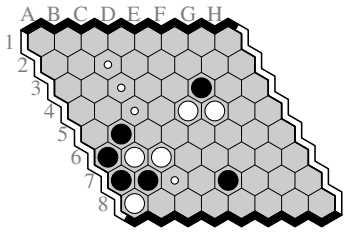
(106)



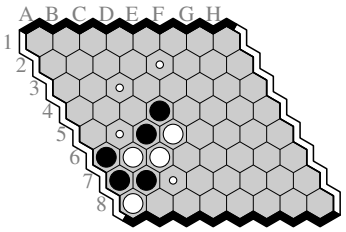
(107)



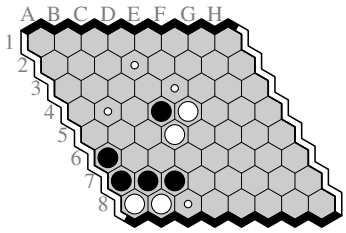
(108)



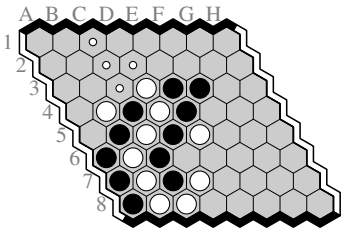
(109)



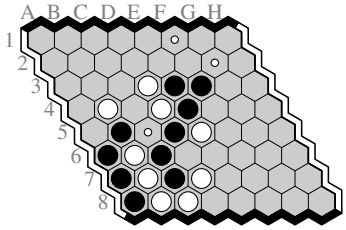
(110)



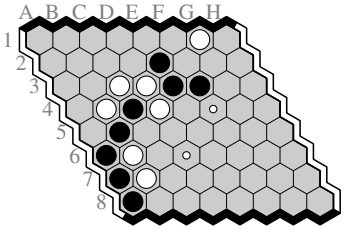
(111)



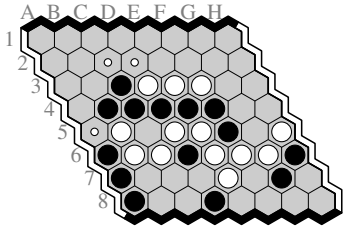
(112)



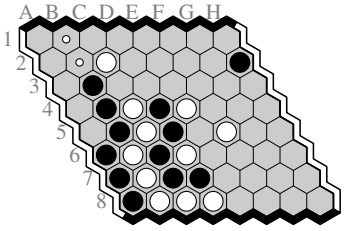
(113)



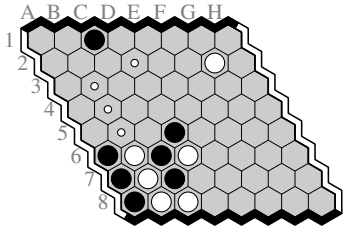
(114)



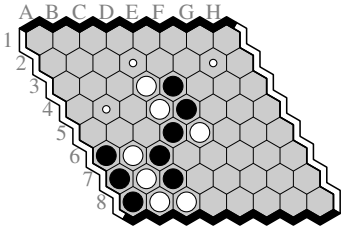
(115)



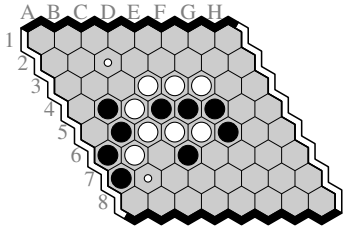
(116)



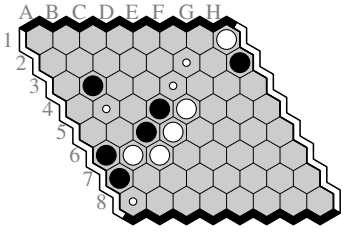
(117)



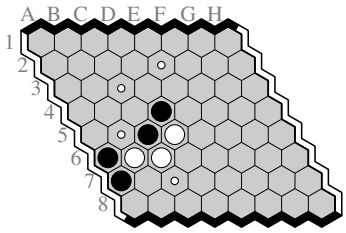
(118)



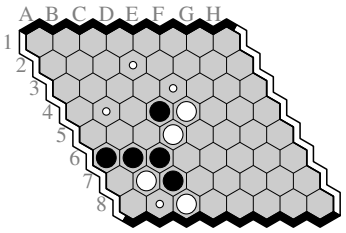
(119)



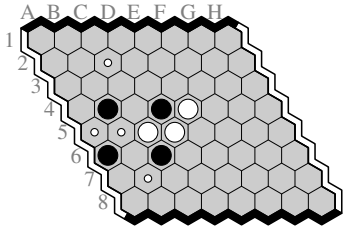
(120)



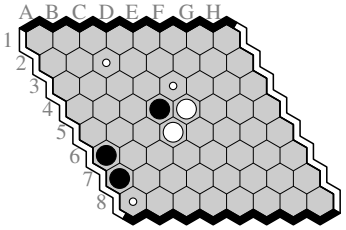
(121)



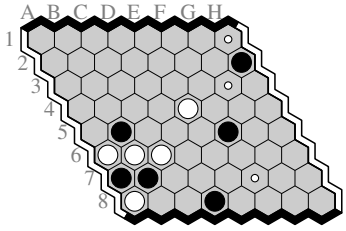
(122)



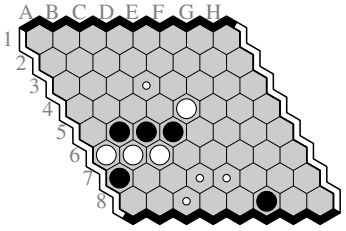
(123)



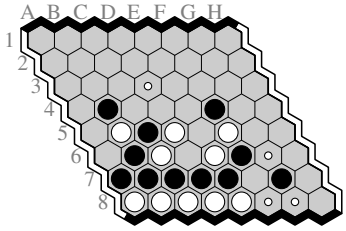
(124)



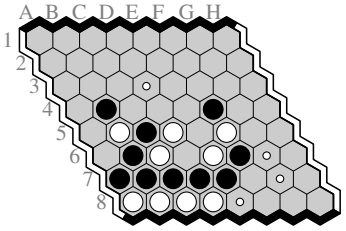
(125)



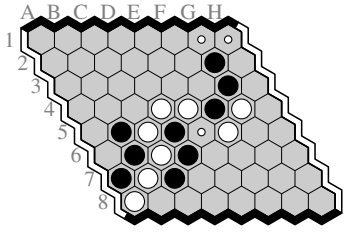
(126)



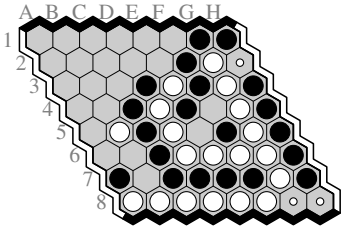
(127)



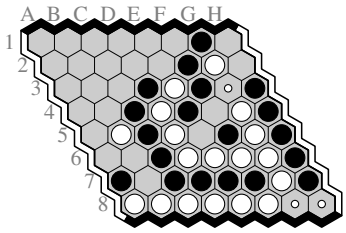
(128)



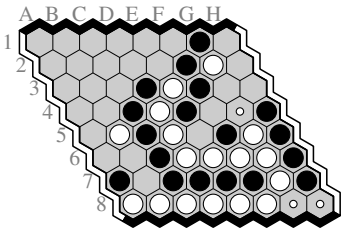
(129)



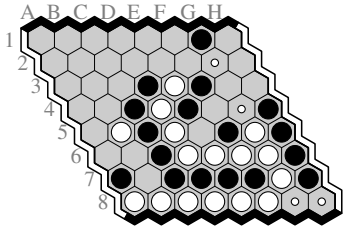
(130)



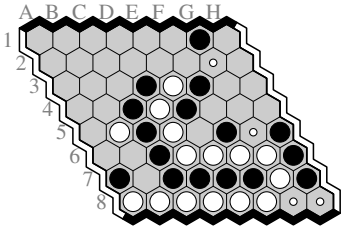
(131)



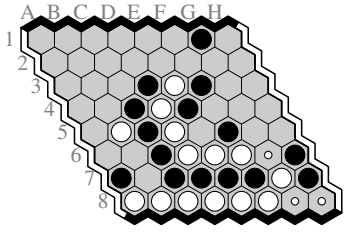
(132)



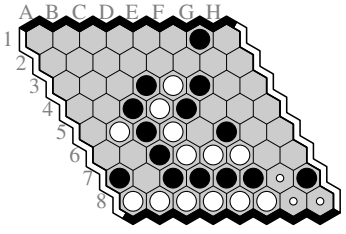
(133)



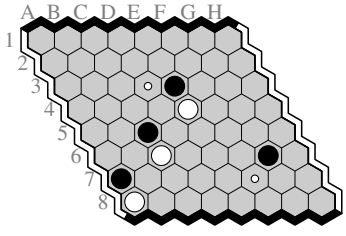
(134)



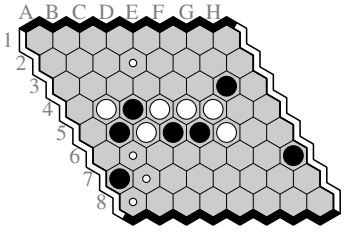
(135)



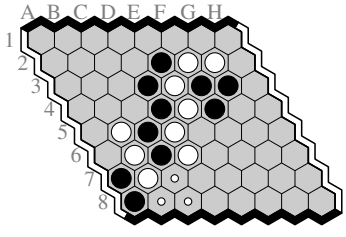
(136)



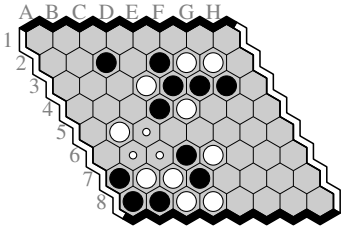
(137)



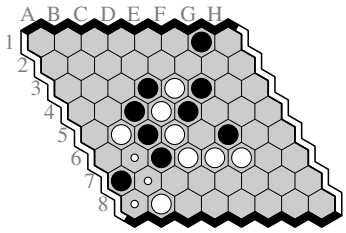
(138)



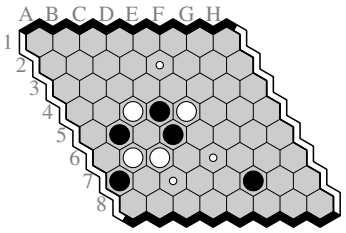
(139)



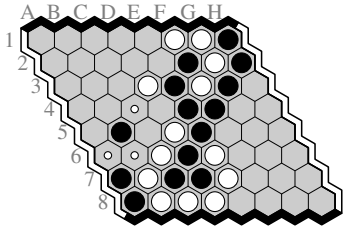
(140)



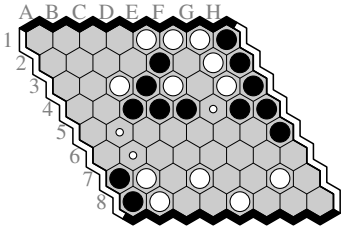
(141)



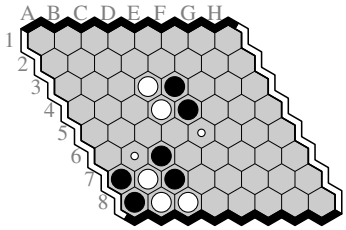
(142)



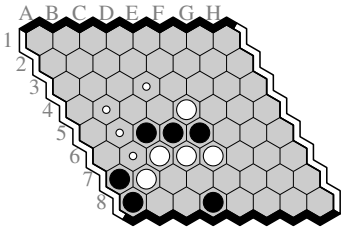
(143)



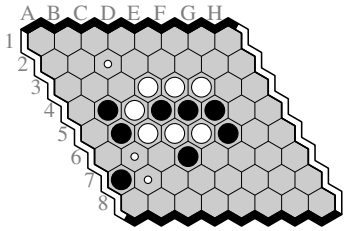
(144)



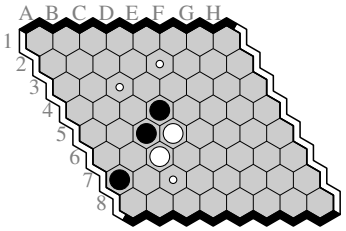
(145)



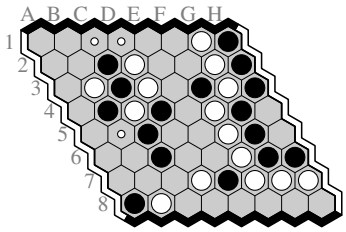
(146)



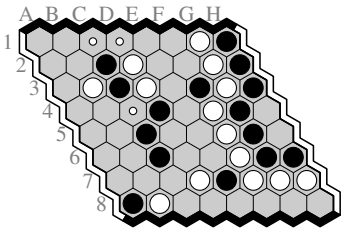
(147)



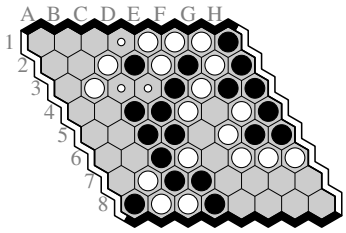
(148)



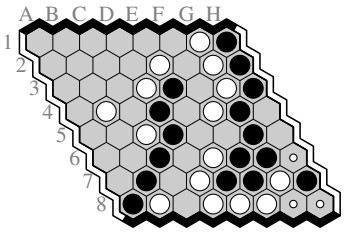
(149)



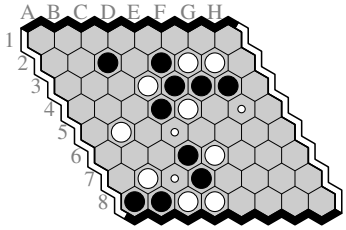
(150)



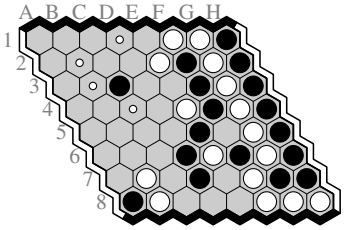
(151)



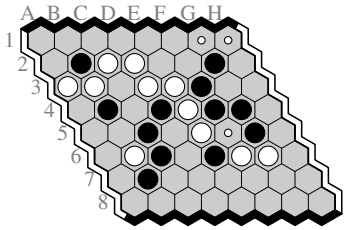
(152)



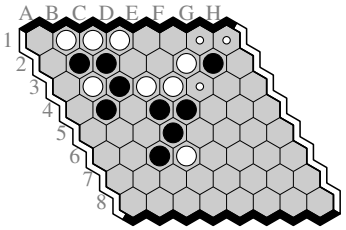
(153)



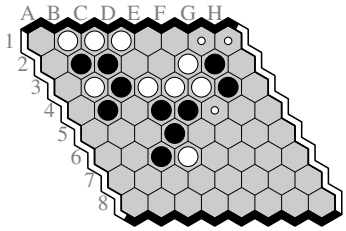
(154)



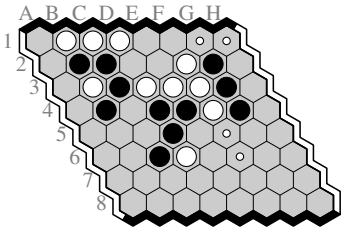
(155)



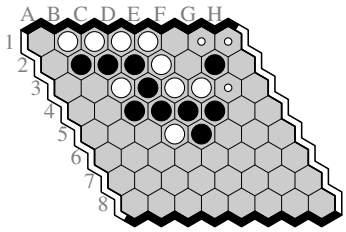
(156)



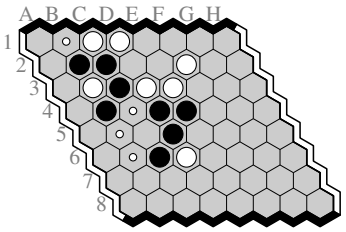
(157)



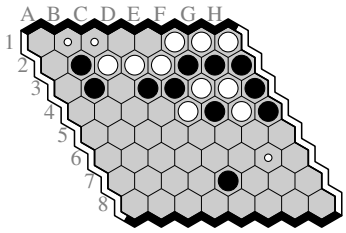
(158)



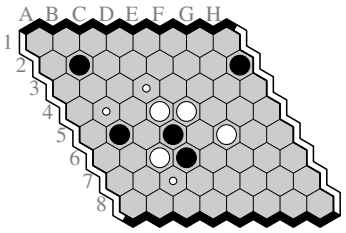
(159)



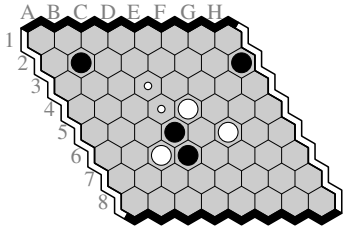
(160)



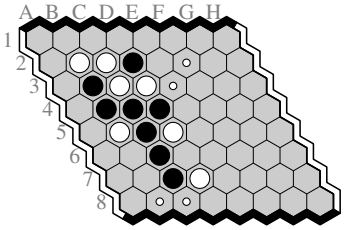
(161)



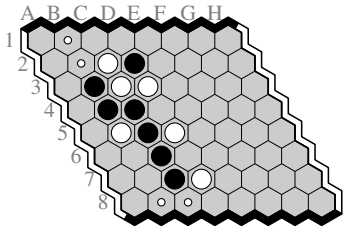
(162)



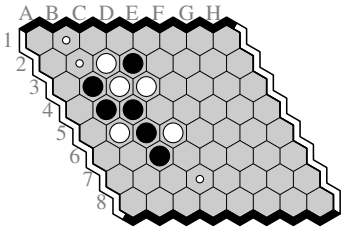
(163)



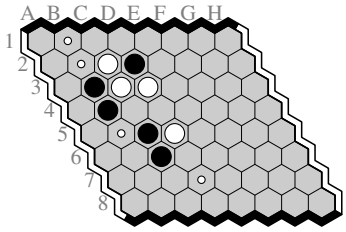
(164)



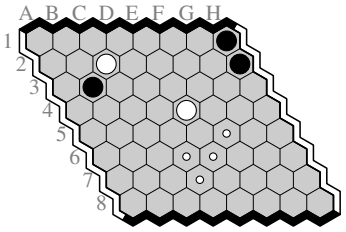
(165)



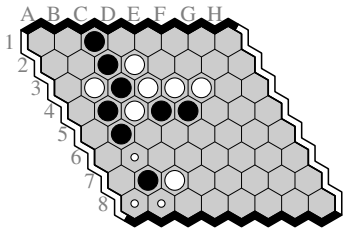
(166)



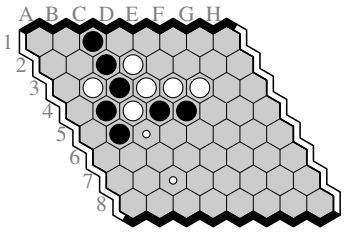
(167)



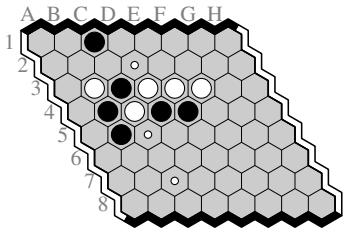
(168)



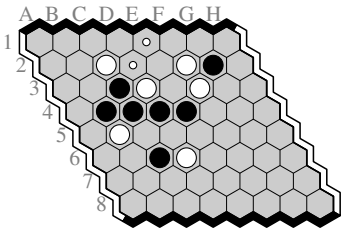
(169)



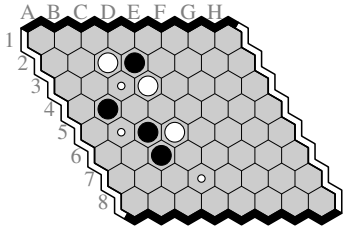
(170)



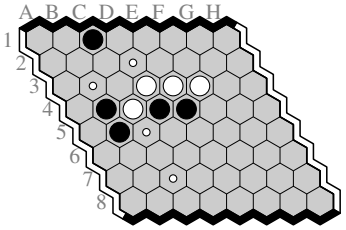
(171)



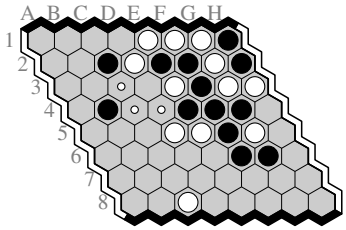
(172)



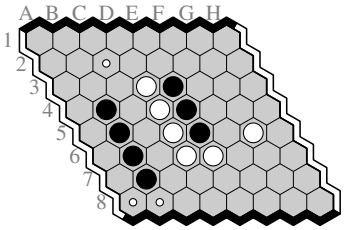
(173)



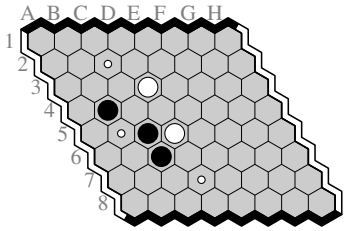
(174)



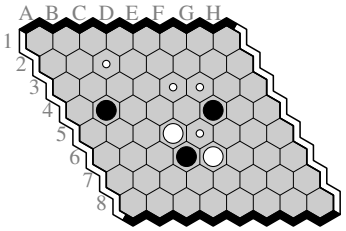
(175)



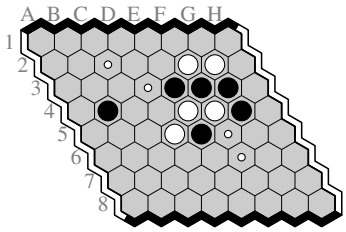
(176)



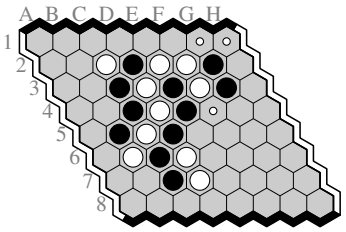
(177)



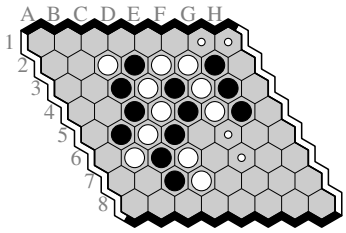
(178)



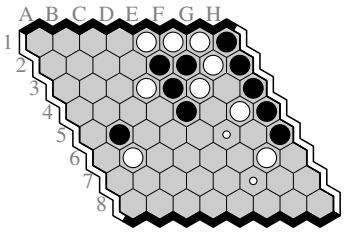
(179)



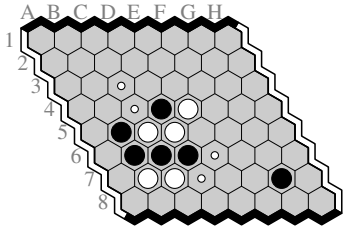
(180)



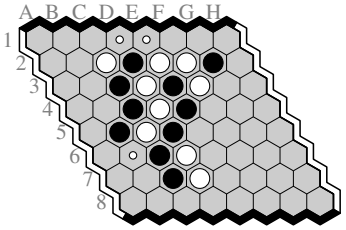
(181)



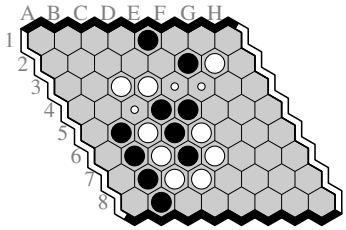
(182)



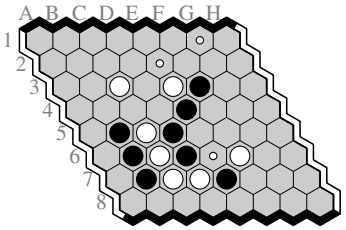
(183)



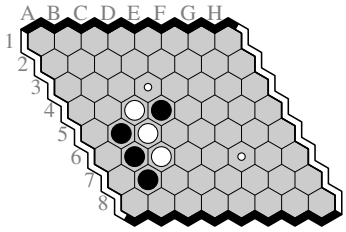
(184)



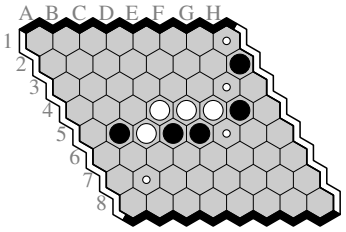
(185)



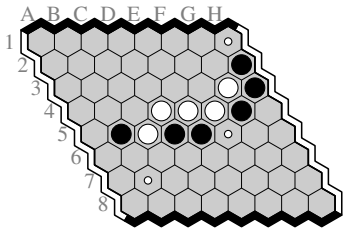
(186)



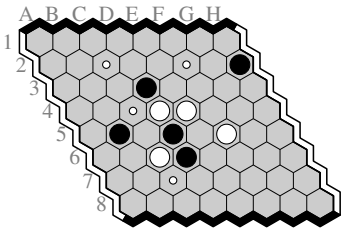
(187)



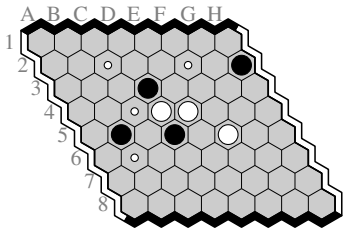
(188)



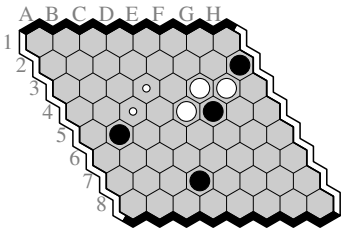
(189)



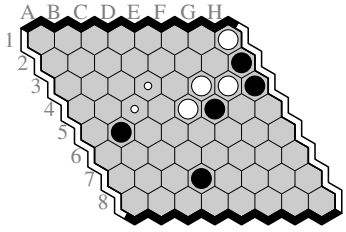
(190)



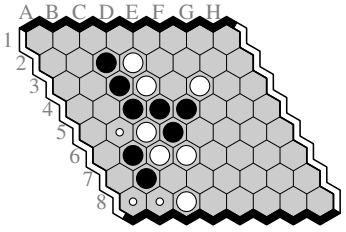
(191)



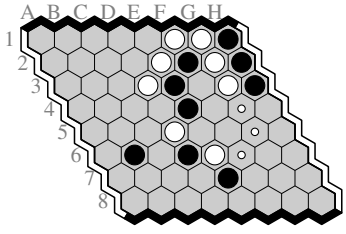
(192)



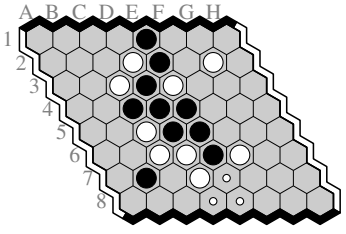
(193)



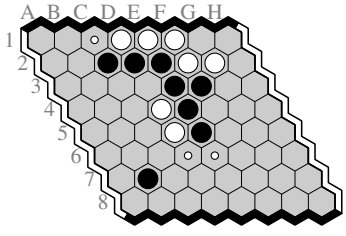
(194)



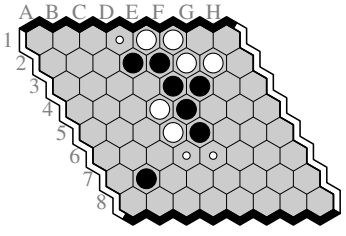
(195)



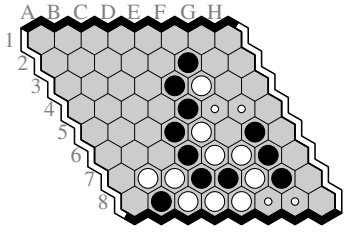
(196)



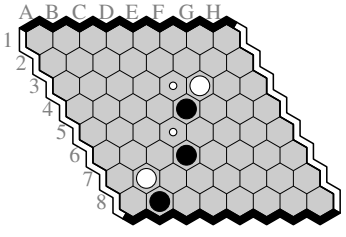
(197)



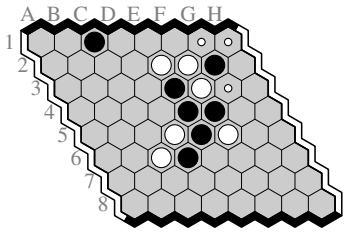
(198)



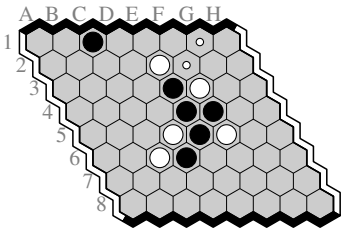
(199)



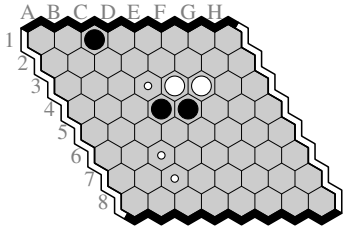
(200)



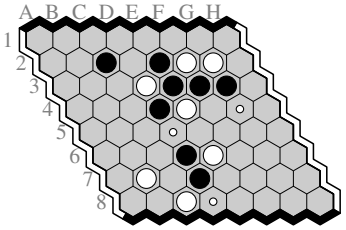
(201)



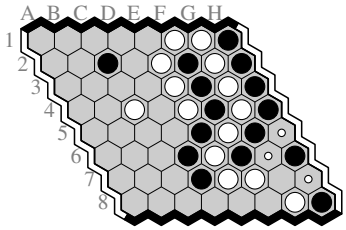
(202)



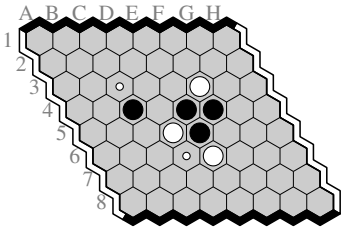
(203)



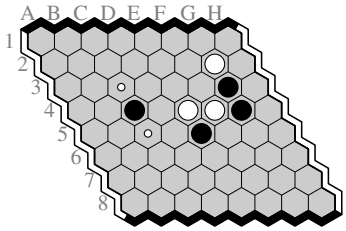
(204)



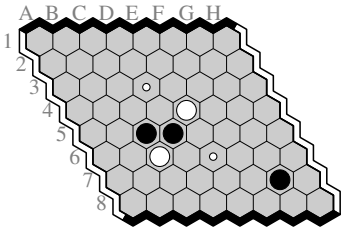
(205)



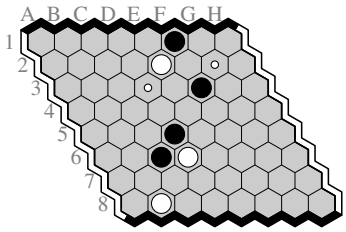
(206)



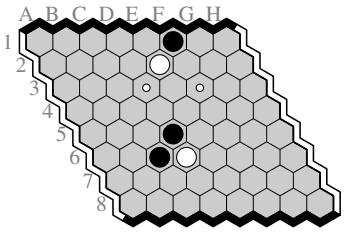
(207)



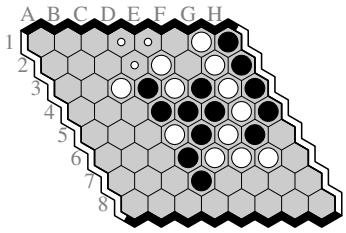
(208)



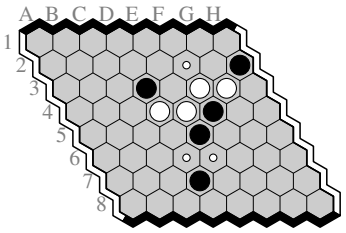
(209)



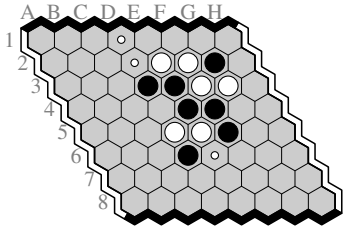
(210)



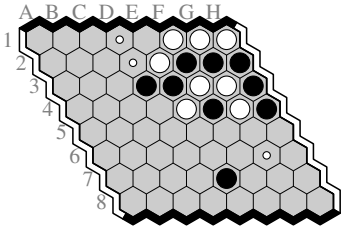
(211)



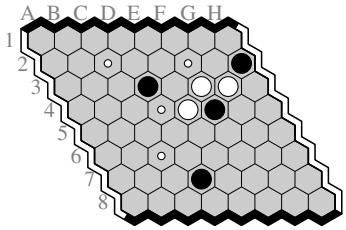
(212)



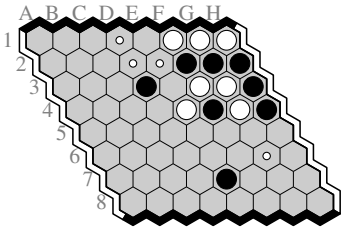
(213)



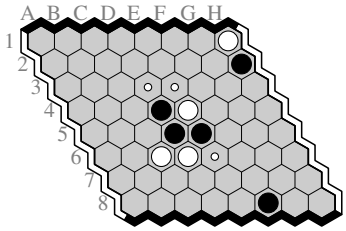
(214)



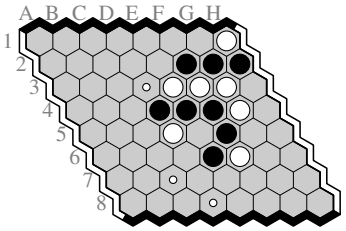
(215)



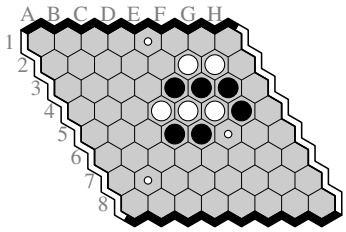
(216)



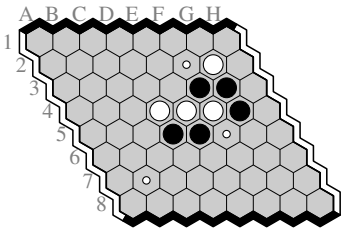
(217)



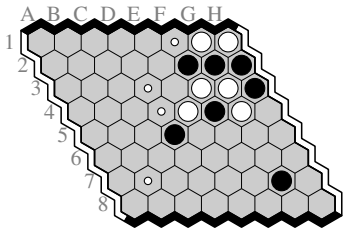
(218)



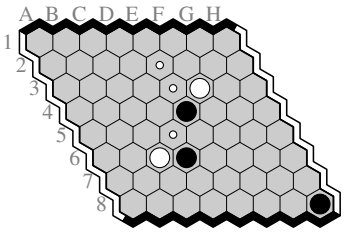
(219)



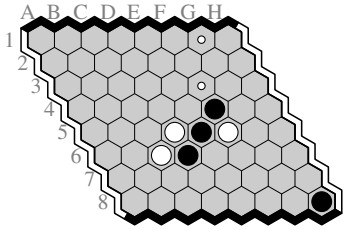
(220)



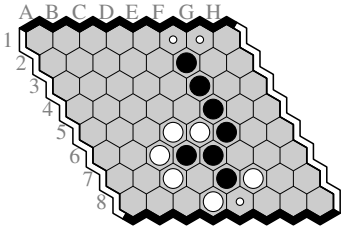
(221)



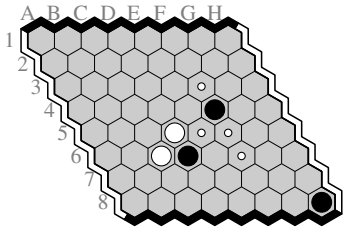
(222)



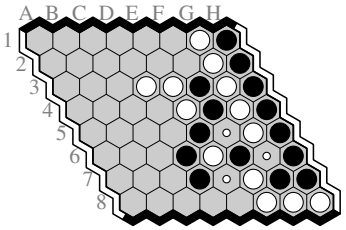
(223)



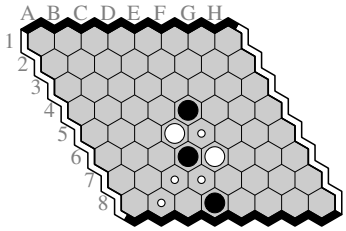
(224)



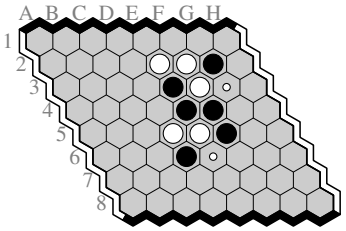
(225)



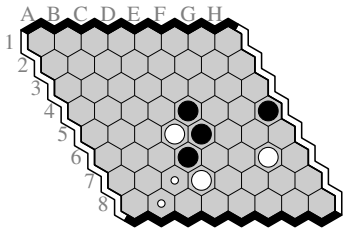
(226)



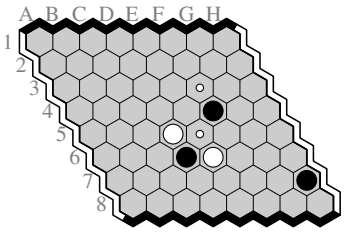
(227)



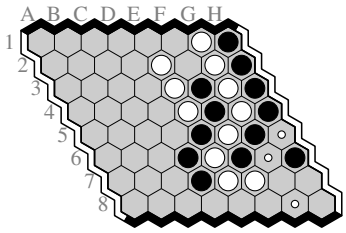
(228)



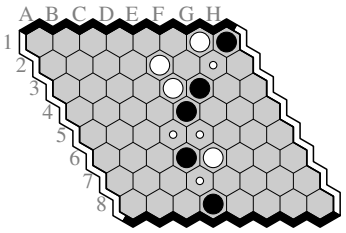
(229)



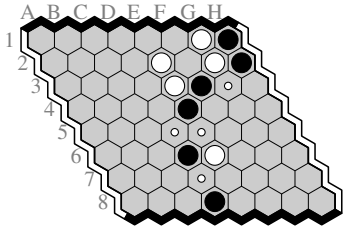
(230)



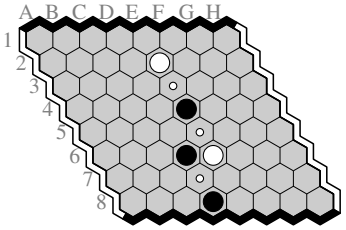
(231)



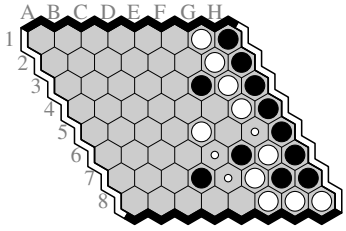
(232)



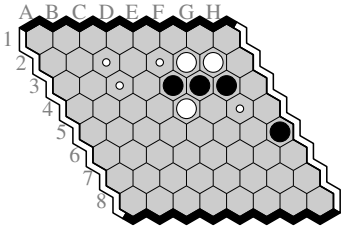
(233)



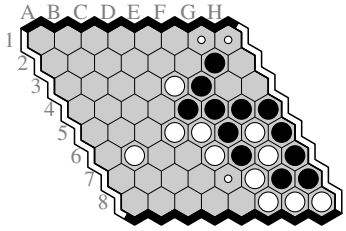
(234)



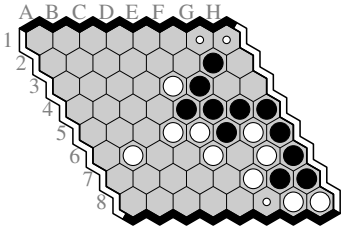
(235)



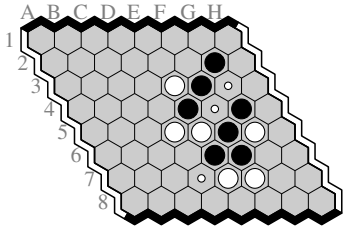
(236)



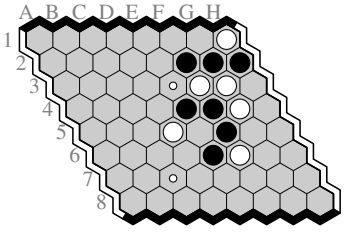
(237)



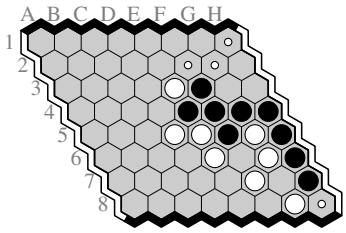
(238)



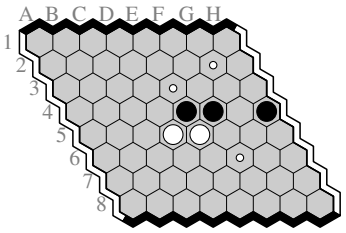
(239)



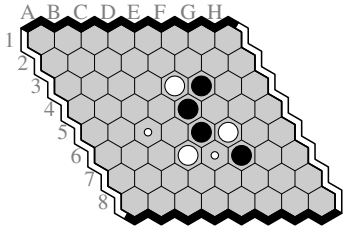
(240)



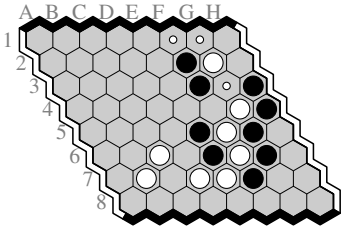
(241)



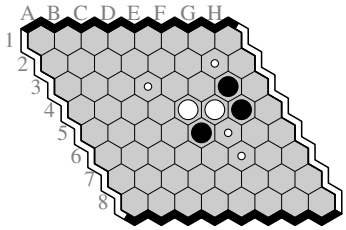
(242)



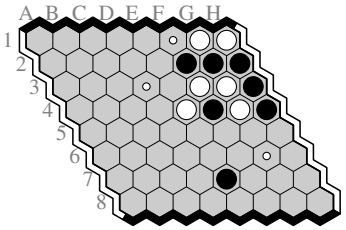
(243)



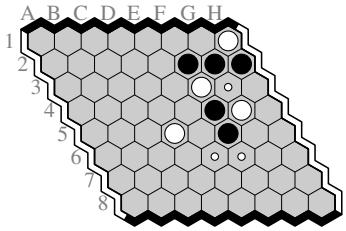
(244)



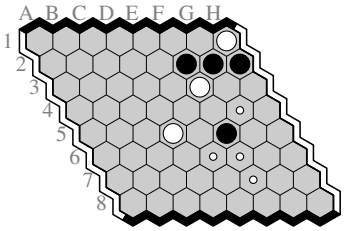
(245)



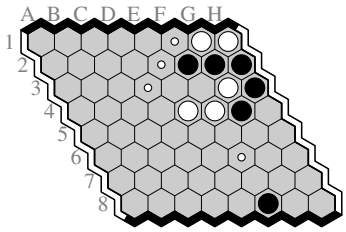
(246)



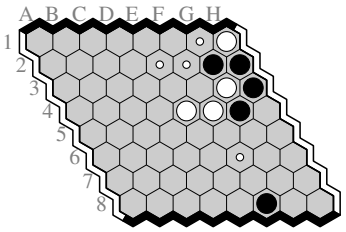
(247)



(248)



(249)



(250)