Crafting a Hex Solver with Connection-Search and Expected-Work-Search

by

Xinyue Chen

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

Master of Science

Department of Computer Science
University of Alberta

Abstract

Hex is the 2-player alternate-turn perfect-information board game created by Piet Hein in 1942. The board is an $m \times n$ array of hexagonal cells that forms a parallelogram, or a rhombus when m equals n. Each player owns two opposing edges of the board. On a turn, a player colors an empty cell. The winner is whoever joins their two edges with a path of their color.

Connection Search refers to any hex-specific method to find connections. Threat Pattern Search, proposed by Jack van Rijswijck in 2000, finds cells belonging to opponent threats. Hiearchical Search, proposed by Vadim Anshelevich in 2002, finds point-to-point connections.

Expected Work Search (EWS) is the 2-player game tree search algorithm created by Owen Randall et alia in 2024. Combining features of Monte Carlo Tree Search and Proof Number Search, EWS guides search by considering both expected win rate and expected proof tree size.

In this thesis, we show how to craft a simple hex solver based on Connection Search and EWS. We start with a minimax solver that prunes moves based on move ordering, inferior move detection, and safe cell-to-cell connection strategies. We measure the importance of these features and add them to a game-agnostic version of EWS. The result is a 1,000 line Python program that solves 6×6 hex states.

Acknowledgements

I would like to thank Owen Randall for many helpful conversations and for sharing his EWS game-independent Python code. I also thank Ryan Hayward for many helpful conversations and for help with writing and editing. I thank Martin Müller and Paul Lu for being on my committee and taking the time to read this thesis. I also thank my family for their support while I was working on this thesis.

Table of Contents

List of Tables

List of Figures

Abbreviations

EWS	Expected work search.
MCTS	Monte Carlo tree search.
PNS	Proof number search.

Glossary of Terms

game graph from a game state, graph of all possible continuations of the game (no duplicate nodes).

game tree from a game state, tree of all possible continuations of the game (possible duplicate nodes).

game state a game position together with the player-to-move.

player-to-move from a game state, the player who moves next.

safe-join virtual-connection.

semi-connection for a player and two end-points, a connection strategy that holds if player moves next.

virtual-connection for a player and two end-points, a connection strategy that holds if opponent moves next.

weak-join semi-connection.

Chapter 1

Introduction

In this thesis we craft a hex solver based on connection search and the Expected Work Search algorithm of Randall et al. [Ran+24].

1.1 Hex

Hex is the 2-player alternate-turn perfect-information board game created in 1942 in Denmark by Piet Hein. Here are the rules. The hex board is an $m \times n$ array of hexagonal cells that forms a parallelogram, or a rhombus when m equals n. Each player owns two opposing edges of the board. On a turn, a player colors an empty cell. The winner is whoever joins their two edges with a path of their color. We call the players Black and White. Unless we say otherwise, Black plays first. See Figure 1.1. (A common hex variant is swap-rule hex: after Black's first move, White has the option of switching identities and playing as Black. In this thesis we do not consider swap-rule hex.)

In 1943 hex became popular in Denmark thanks to a regular hex column by Hein in

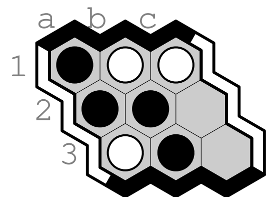


Figure 1.1: A finished 3×3 hex game. Black wins.

the Copenhagen newspaper *Politiken.* Most columns included a hex puzzle composed by chess expert Jens Lindhard, who later worked in the lab of Neils Bohr. In 1957 Martin Gardner wrote a column on hex for *Scientific American.* For more on hex, see *Hex, the full story* by Hayward and Toft [HT19] and *Hex, a playful introduction* by Hayward [Hay22].

Hex has properties that make it suitable for testing 2-player game algorithms that apply to games such as chess and go:

- Hex is easy to implement: to make a move, you find an empty cell and color it. Unlike with go, a cell's color never changes. Unlike with chess, the winning condition is easy to describe.
- Hex can be played on any $m \times n$ board. This also holds for go but not for chess. This property allows algorithms to be tested on gradually increasing board sizes.
- Hex cannot end in a draw. For a proof, see *Hex, the full story* or *Hex, a playful introduction.* [HT19; Hay22]. Both chess and go (with komi 0) allow draws. Designing algorithms for win/loss games is easier than for win/loss/draw games.

1.2 Connection search

Figure 1.2 shows a hex position. Assume that Black plays next: can you find a winning black strategy? Answer on the next page.

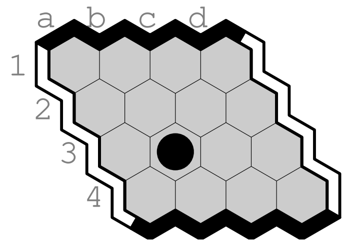


Figure 1.2: White to play. Find a winning black strategy.

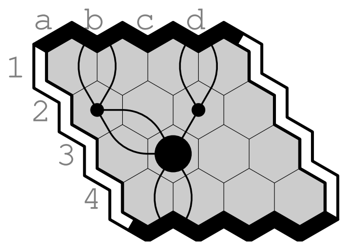


Figure 1.3: A winning 2nd-player strategy for Black.

Notice in Figure 1.3 that Black threatens with cell a4 or cell b4 to safely join the stone at b3 to the bottom edge. On her next move White can block at most one of these two threats. We say that Black has a *virtual-connection*, or *safe-join*, from b3 to the bottom edge. Similarly, as shown in the figure, Black has a safe-join from b3 to the top edge. Black has two threats: play at a2 and then use cells a1, b1, a3, b3, or play at c2 and then use cells c1 and d1. One threat uses cell set {a1, a2, a3, b1, b2}, the other uses cell set {c1, c2, d1}. These two sets do not intersect, so with her next move White can block at most one of these threats, so Black has a b3-to-top safe-join. The cell set {a4, b4} of the b3-to-bottom safe-join does not intersect the cell set {a1, a2, a3, b1, b2, c1, c2, d1} of the b3-to-top safe-join, so Black can follow them both, giving a top-to-bottom safe-join. Following this strategy, Black can win.

Definition 1 *For a player and two specified end-points, a* safe-join*, also called* virtual-connection*, is a 2nd-player strategy that joins the points, and a* weak-join*, also called* semi-connection *or* semi-virtual-connection*, is a 1st-player strategy that joins the points. For a player* P*, a* key *of a weak-join* W *is a cell that when* P*-colored transforms* W *into a safe-join. A* carrier *of a strategy is is the set of cells used.*

For example, for the position in Figure 1.2, the cell set {c1, c2, d1} is a carrier of a weak-join with key c2 that joins b2 to the top edge.

Connection search is the process of finding weak-joins or safe-joins. Connection search plays a key role in hex solvers.

1.3 Game tree search

Game tree search algorithms explore all or part of the game tree in order to either play a move or solve the game. Here are the relevant definitions.

Definition 2 *We consider two-player games. A* position *is, for each cell on the board, the color of that cell: for hex, the colors are empty, black or white. A* state *is a position and the player-to-move. For a state, the* game tree *is the tree of all possible continuations of the game: the start state is the root of the tree, the states reachable after one move are the children of the root, and so on. If we combine duplicate nodes in the tree to a single node, we have the* game graph*. For a player P and opponent Q, a* strategy *is a subtree of the game tree such that each subtree node with P to move has exactly one child (the strategy move), and each subtree node with Q to move has all possible children. A* winning strategy *(win-strat) is a strategy that wins against all possible opponent strategies. We say that a player* wins a state *if they have a winning strategy for that state. The* value *of a state is the minimax value of the state for the player-to-move(PTM): for hex, this will be 1st-player-win or 1st-player-loss. To* (ultra-weakly) solve *a state is to find its value. To* strongly solve *a state is to find its value and to find a win-strat. For a state, a* proof tree *is a subtree of the game tree that includes some win-strat.*

1.4 Expected work search

In 1994, Victor Allis introduced the Proof Number Search (PNS) and used it to solve Connect-4 [AMH94]. PNS guides search by considering the expected size of a proof tree. In 2006, Rémi Coulom used Monte Carlo Tree Search (MCTS) to play 9×9 go in a computer competition [Cou06b]. In 2016, AlphaGo used MCTS enhanced with deep convolutional neural nets to play 19×19 go against Lee Sedol [Sil+16].

In 2009, Broderick Arneson, Philip Henderson and Hayward wrote Solver, a PNS hex solver that found all winning 1st moves on the 8×8 board [HAH09b]. In 2013, Jakub Pawlewicz and Hayward parallelized Solver and found all winning 9×9 1st moves and 2 winning 10×10 1st moves [PH13].

In 2024 Owen Randall, Müller, Ting-Han Wei and Hayward created Expected Work Search (EWS), which combines features of MCTS and PNS [Ran+24]. EWS guides search by considering both expected win rate and expected proof tree size.

1.5 Thesis

The parallel version of Solver has about 28,000 lines of C++ code. Motivated by the desire to make games programming more accessible, in this thesis we design a simple 1,000-line Python hex solver.

In Chapter 2, we give background information. In Chapter 3, we show how to craft our hex solver. In Chapter 4, we present experimental results. In Chapter 5, we suggest future work.

Chapter 2

Game tree search background

Here we give background information on game tree search. We discuss four algorithms: Minimax Search, Proof Number Search (PNS), Monte Carlo Tree Search (MCTS), and Expected Work Search (EWS). See §1.3 for game tree search definitions.

2.1 Minimax search

We assume that the reader is familiar with minimax. The listing in Table 2.1 is based on the hex solvers in the github repository of Hayward's undergraduate course Games/Puzzles/Algorithms (GPA) [Hay25]. This is the negamax form of minimax: rather than consider two players Max and Min, we consider only Player-to-Move. This is also the boolean win/loss form of minimax: hex has no draws, so on line 6 we return as soon as a win is detected [Hay25]. We will craft a simple minimax hex solver in the next chapter.

Table 2.2 shows solving time data for vanilla boolean negamax program `hex_-`

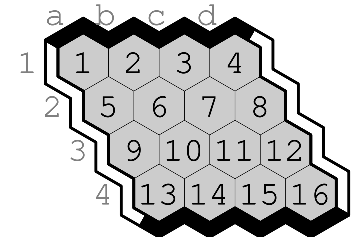


Figure 2.1: Default move order. For each player, all winning 1st moves.

```
# return winning move if ptm wins else empty string
def mmx_move(psn, ptm): # assumes no winner yet
  optm = oppCH(ptm) # opponent of player-to-move
  for k in CELLS: # for every cell on the board
    if is_empty_cell(k, psn):
      new_psn = color_cell(k, psn, ptm)
      if has_win(new_psn, ptm): # did ptm win?
        return k
      # if not, continue from new_psn with optm
      optm_wins = mmx_move(new_psn, optm)
      # if optm has no winning move, then
      #   ptm move at k is a winning move
      if not optm_wins:
        return k
  return '' # no winning move
```

Table 2.1: Boolean negamax hex listing.

program	move order	fn calls	tt lookups	time (s.)	move
`simple`	default	–	n/a	> 1800	–
	A	532,702,892	n/a	1716.5	d1
	B	7,034,997	n/a	22.8	c2
`tt`	default	3,479,112	2,208,656	6.1	d1
	A	2,214,621	1,434,778	3.8	d1
	B	183,896	96,061	0.4	c2

Table 2.2: Minimax 4×4 empty-board solving data.

`simple.py` and `hex_tt.py`, which has a TT. Figure 2.1 shows default move order and all winning first moves. Figure 2.2 shows alternate move orders A and B. Notice how a TT and a good move order each reduce solving time.

2.2 Proof number search

Proof number search is a best-first search created by Allis that uses proof number and disproof number to guide the search for a proof tree [AMH94]. Starting from the root of the game tree, PNS grows a subtree by repeated leaf expansion (at the node

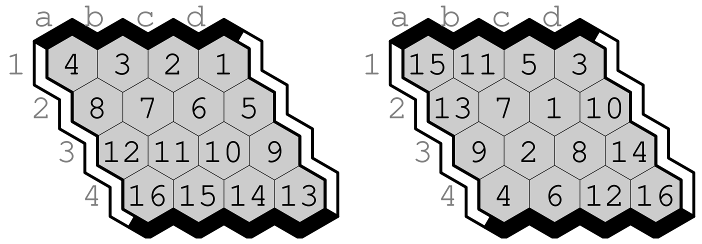


Figure 2.2: Move orders A (left) and B.

that is a leaf in the subtree, add all children of that node from the game tree). PNS stops when time runs out or a proof tree is found.

Definition 3 *For a subtree of a game tree that includes the root, the* proof number *(resp.* disproof number*) of a non-terminal leaf is the minimum number of non-terminal leaves that must evaluate to win (lose) for the game to evaluate to win (lose). A leaf of such a subtree is* most-proving *if its evaluation as win decreases the proof number and its evaluation as lose decreases the proof number.*

PNS always expands a most-proving leaf. Allis gave an efficient algorithm to find a most-proving leaf. For more on PNS, see the survey by Kishimoto et al. [Kis+12]. In 2009 Arneson, Henderson and Hayward wrote a PNS hex solver and solved all 1st moves on the 8×8 board [HAH09b].

In 2013, Jakub Pawlewicz and Hayward created Scalable Parallel Depth First Proof Number Search (SPDFPNS), added it to Benzene, and solved all 9×9 1st-moves and two winning 10×10 1st-moves [PH13].

2.3 Monte Carlo tree search

Monte Carlo Tree Search (MCTS) is a best-first search based on random-move playouts. The Upper Confidence Tree (UCT) version uses the bound of Kocsis and Szepesvári [KS06] to balance exploitaion (best-first search) with exploration (broader search). MCTS become popular after 2006, when Rémi Coulom's MCTS player Crazystone won gold at the International Computer Games Olympiad [Cou06a]. For more on MCTS see the survey by Browne et al. [Bro+12]. For more on MCTS see the survey by Browne et al. [Bro+12]. Here is the main loop the Python MCTS implementation at `geeksforgeeks.org` [gee]:

- **select** From the root, repeat this process until reaching a leaf: if some child has no simulations then return it; otherwise, among all children, select a best child according to the UCB exploitation-exploration balancing formula.
- **expand** Generate all children of the leaf and descend to one.
- **simulate** Perform a random-move playout from the child.
- **backpropagate** Update the simulation result at the child and all ancestors.

MCTS is usually used as a player rather than a solver, but by backing up true win/loss results at terminal nodes it can be transformed into a move-guided version of minimax search [WBS08].

2.4 Expected work search

In 2024, Owen Randall proposed Expected Work Search (EWS), a new solving algorithm [Ran+24]. EWS combines win rates as in Monte Carlo Tree Search (MCTS) with proof numbers as in Proof Number Search (PNS) to estimate the computation required to solve a state. Selecting a node with minimum expected work (EW) will hopefully reduce overall search time. The EWS main loop has three steps: **select**, **expand**, **backpropagate**. Simulate is now part of the expand step. Select and backpropagate are as in MCTS. The expand step is different: it include **simulate** and expands many nodes in one step.

During backpropagation, if a node is unsolved, its children are ordered based on their EW values. The EW of a child X is calculated recursively with this formula:

$$\text{EWloss}(X) = \sum_{i=1}^{n} X_i \text{EWwin}(C_i)$$

$$\mathrm{EWwin}(X) = \sum_{i=1}^{n} \left(\mathrm{EWloss}(C_i) \cdot \prod_{j=1}^{i-1} \mathrm{WR}(C_j) \right)$$

The win rate of X estimates the probability that X wins. Leaf nodes are initialized with a heuristic formula. Move order does not matter when X is losing, since all child nodes must be proved to prove that X loses. But move order matters when X is winning, since only one child must be proved to prove that X wins. So in computing EWwin(X), children of X are searched in order starting with C_1. EWS has a negamax format: when selecting from a set of children, EWS prioritizes child nodes with small EW and low win rate.

Chapter 3

Hex background

Piet Hein created hex, originally called Polygon, sometime before fall 1942. He introduced the game to the public through columns in the Copenhagen newspaper *Politiken* from December 26, 1942 to August 11, 1943 [Hei42].

Charles Titus and Craige Schensted created Y, also called Triangle: the rules are the same as for hex, except that the board is a triangular array of hexagons, and the winner is whoever joins all three sides. Figure 3.1 shows a Y board after a game won by black. The game is called Y because winning set cells often form a y-shape, as in the figure. Y generalizes hex: for any hex state there is an associated Y state such that the two game trees are isomorphic.

In 1975, Titus and Schensted wrote a book on Y with tips on how to play: since Y generalizes hex, these tips also apply to hex [ST75]. Claude Berge wrote an unpublished manuscript on how to play hex [Ber77; Hay03]. Cameron Browne wrote the first book on hex [Bro00]. Hayward and Toft wrote a history of hex [HT19]. Hayward

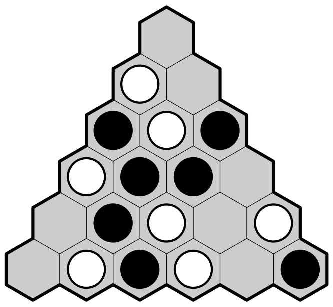


Figure 3.1: A Y game won by Black.

wrote an introduction to hex and math [Hay22]. Matthew Seymour wrote an online strategy guide and a collection of puzzles [Sey20b; Sey20a].

Hex is an example of the class of alternate-turn 2-player *set coloring* games: on a turn, a player colors an empty cell with their color; the winner is whoever first colors a set of cells that satisfies the particular game-specific winning condition. Tic-tac-toe, Y and the board game Havannah are set coloring games. Many of the ideas in this chapter apply not just to hex in particular but to set coloring games in general. For more on this, see Jack van Rijwijck's Ph.D. thesis [Rij06].

3.1 Hex solving theorems

Here are some well-known theorems relating to solving hex states:

Theorem 4 See [HT19] or [Hay22] for proofs.

- ***extra cells help.*** *Consider a state S that player P wins. Alter the state by P-coloring an empty cell without changing the player-to-move. Then P wins the altered state.*

- ***1st-player wins.*** *For all positive n, 1st-player wins $n{\times}n$ hex.*

- ***closer edges helps.*** *For all positive m and all $n > m$, Black (whose edges are closer together than White's) wins $m{\times}n$ hex.*

- ***dead cells are useless.*** *In a position, an empty cell is* dead *if it is not in a minimal set of empty cells that, when colored, join a player's two edges. A cell is* live *if it not dead. For a state with at least one live cell, there is a winning move to a live cell.*

- ***hex is PSPACE-complete.*** *Solving arbitrary hex states is PSPACE-complete.*

3.2 Hex solving hardness

The previous theorem refers to the complexity of solving hex problems as a function of the board size, so as board size increases, but does not say anything about the hardness of solving hex on any one particular board. One measure of hardness on a fixed size board is the size of the game graph. Henderson et al. conservatively estimate $n \times n$ empty-board game graph size as the number of positions corresponding to move sequences that fill only half of the board [HAH09a]. See Table 3.1.

4x4	5x5	6x6	7x7	8x8
7.6e5	4.0e9	4.0e14	1.5e20	1.0e27

Table 3.1: Number 1/2-full hex states (Henderson et al.)

3.3 Dead, captured, inferior

Definition 5 [Hay22] *For a player* P*, opponent* Q*, and state* S *with position* X*,*

- *a* winset *is a set of empty cells that join* P*'s edges when* P*-colored,*
- *a winset is* minimal *if any proper subset is not a winset,*
- *an empty cell is* dead *if it is not in any minimal winset, otherwise it is* live*,*
- *a colored cell is* dead *if it is dead after uncoloring,*
- *a set of empty cells* S *is* P-captured *if* P *has a 2nd-player strategy on* S *that leaves dead every cell of* S *that is empty or opponent-colored,*
- *a move to cell* j *is* inferior *to a move to cell* k *if either a* P*-move to* k *captures* j *or a* Q*-move to* k *leaves* j *dead.*

Consider for example Figure 3.2. In the 1st (from left) diagram, all empty cells are live. In the 2nd diagram, cells $\{a1, a2, b1\}$ are dead, all other empty cells are live. In the 3rd diagram, cells $\{a1, a2, c1\}$ are dead, all other cells are live. Thus the dotted cells in the 1st diagram are Black-captured: if White ever plays at one, Black can reply at the other, leaving White's cell dead.

Theorem 6 See [HR06] for proofs.

- ***dead and inferior cells can be pruned.*** *For a hex state with at least one live cell, there is a winning move to a live cell that is not inferior.*
- ***captured cells can be filled.*** *For a hex state with a* P*-captured set* S*,* P*-coloring* S *does not change the value of the state.*

In solving a hex position we can apply Theorem 6 by pattern matching. For example, after rotation, we can match the left pattern in Figure 3.3 with the Black cell, two dots and top of the board in Figure 3.2 and then Black-color the two dotted cells. For the empty 4×4 board, when searching for a winning Black 1st move, we can match the right pattern in Figure 3.3 to two locations at the top of the board and two locations at the bottom of the board, thus pruning all top-row and all bottom-row cells from our search: instead of considering 16 possible 1st moves, we restrict our search to 8 possible 1st moves.

3.4 Mustplay reasoning

In *L'art subtil du hex*, Berge explains the idea of mustplay [Ber77; Hay03], which we rephrase in our terminology. Consider the left position in Figure 3.4: where should White play? If White sees the Black win-threat (winning weak join) in the diagram, White knows she must interfere with this threat: otherwise, Black plays at the dot and wins.

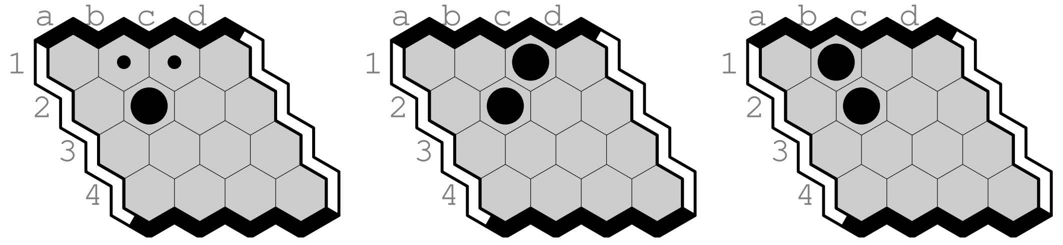


Figure 3.2: The dotted cells are Black-captured.

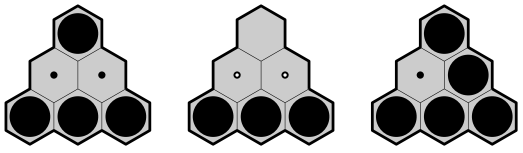


Figure 3.3: Capture, inferior and dead cell patterns.

Definition 7 *For a state with player-to-move* P *and opponent* Q*, a* win-threat *is a* Q*-winning weak-join. With respect to a set of win-threats,* P*'s* mustplay *is the intersection of the cell sets of* Q*'s win-threats.*

Next consider Figure 3.5. White can disrupt the first Black win-threat by playing at *d2*: does this win for White? No: Black has a second win-threat as shown. Once White sees this, she must find a move that blocks both win-threats, namely a move that is in the intersection of the associated two threats.

3.5 Opponent-oblivious winning strategies

Recall from Definition 2 that a strategy is defined as a certain kind of subtree of the game tree. In hex or any set coloring game, there is a winning strategy without specifying any opponent moves: we call such a strategy *opponent oblivious*. Hayward et al. used this idea to verify the correctness of a winning 7×7 hex strategy of Jing Yang [HAH06].

Theorem 8 *An oblivious 1st-player strategy is a move optionally followed by two or more oblivious 2nd-player strategies. An oblivious 2nd-player strategy is a set of*

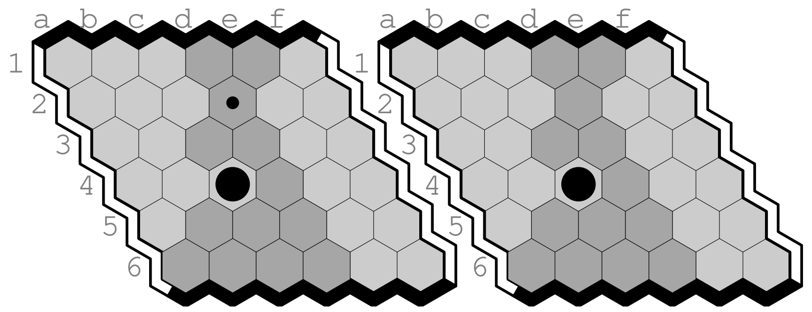


Figure 3.4: A black win-threat and associated white mustplay.

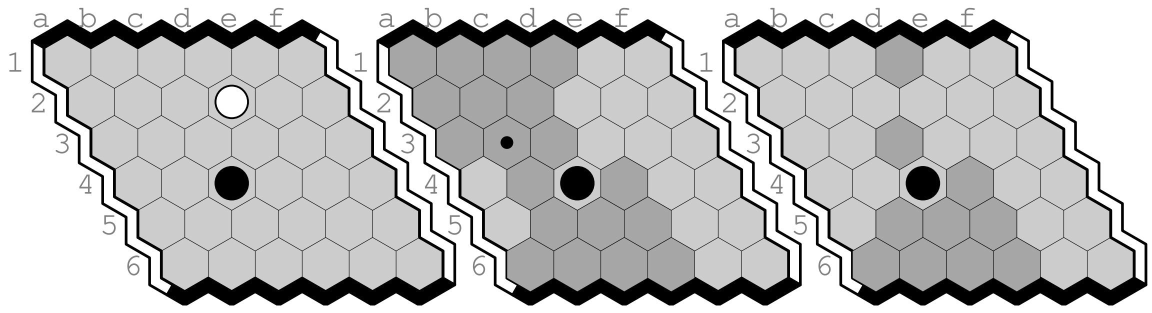


Figure 3.5: Does d2 win for White? No: another threat and mustplay.

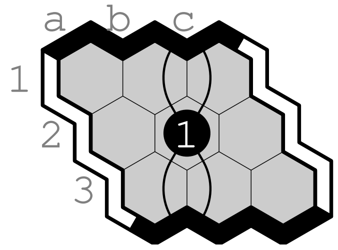


Figure 3.6: A winning 1st-player strategy.

two or more oblivious 1st-player strategies whose carrier intersection is empty. Every winning strategy can be represented as an oblivious strategy.

Consider for example the winning Black strategy shown in Figure 3.6. We can represent this strategy as `b2{b1{a3,b3}, c1{a3,b3}, a3{b1,c1}, b3{b1,c1}}` , namely Black plays b2 and then, after White's reply, plays any of the four remaining oblivious 2nd-player strategies that do not include the cell colored by White. A more compact representation of this strategy is `b2{b1,c1}{a3,b3}` .

Chapter 4

Connection search background

Here we give background information on connection search, any process that finds connections — either safe-joins or weak-joins — in a hex position. Human hex players pay particular attention to *edge templates*, namely safe-joins that connect a stone to a player's edge. For example, the left diagram in Figure 3.2 can be considered an edge template: it shows that a black stone in the 2nd from from a black edge with two empty cells beneath it can be safely joined to the edge. See Peter Selinger's hex templates webpage for more examples [Sel25].

4.1 Mustplay search

Here we show how Berge's definition of mustplay can be used to solve a hex state. Consider the empty 6×6 board with Black to play. We saw in Figures 3.4 and 3.5 that Black has two threats which reduce White's mustplay to 10 cells. Figure 4.1 shows five more Black threats that reduce the mustplay to just one cell, namely *c3*, so assume White plays there. After White plays at *c3*, Black checks for White threats and finds one, shown in Figure 4.2. Now Black's mustplay has 13 cells.

We leave the rest of this problem to you. From the state in Figure 4.2, can you find a winning move?

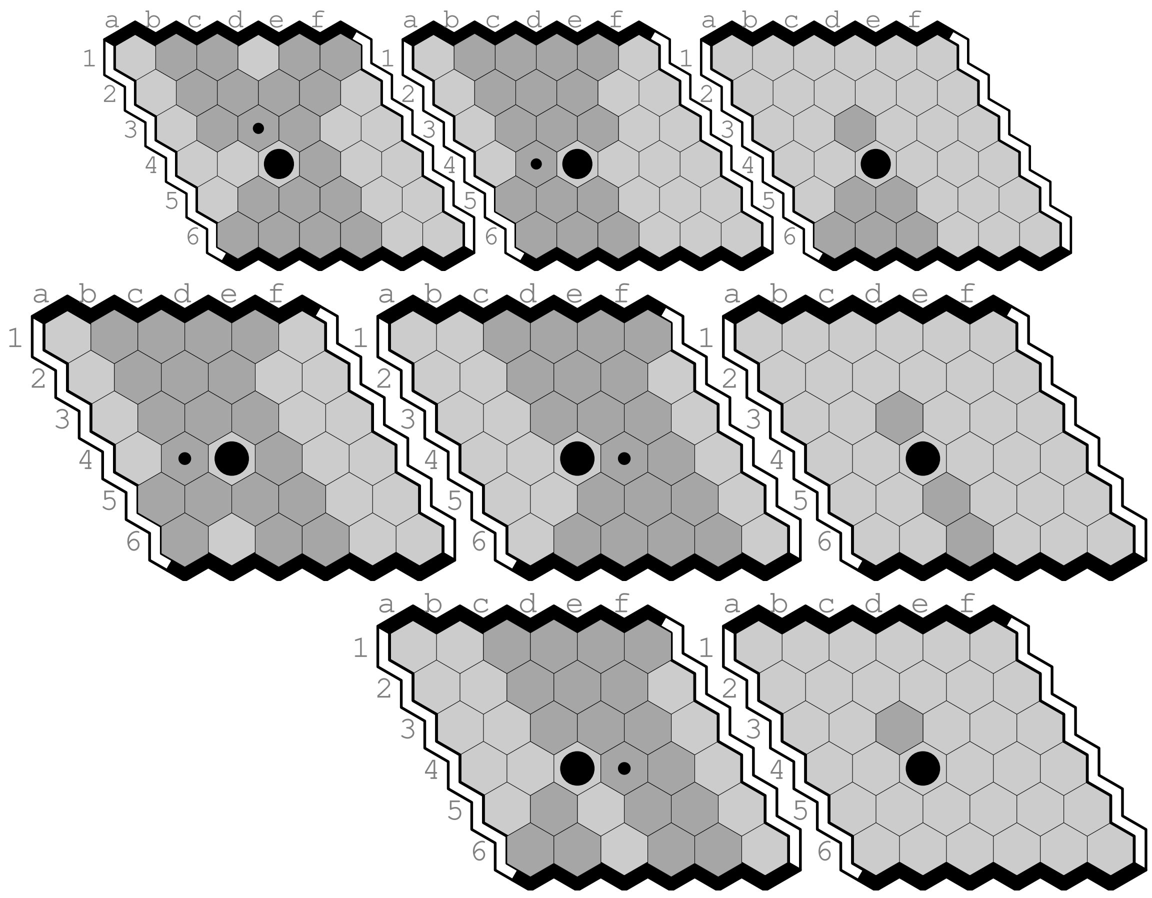


Figure 4.1: Five more win-threats with refined mustplays.

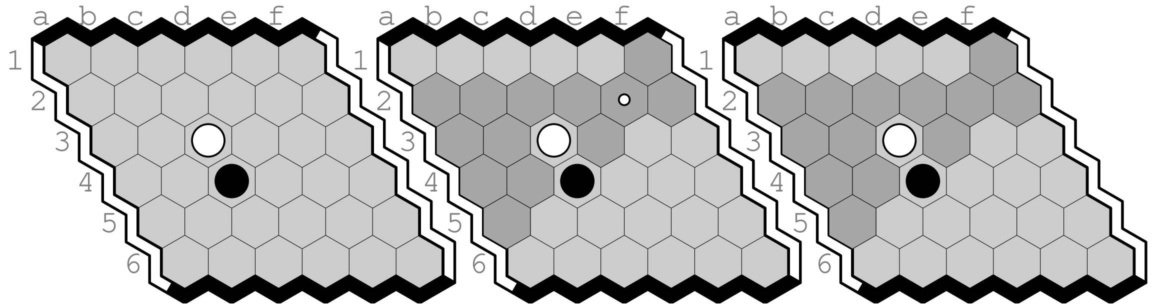


Figure 4.2: White's forced move, a win-threat and Black's mustplay.

4.2 Threat search

Here we discuss how recognizing opponent threats in hex can be used to prune the search tree. This discussion applies not only to hex but to set coloring games in general.

For a given subgame G and player-to-move P, a *threat* is an opponent-winning 1-strat. We call this a threat because, unless P blocks it, P will lose. For G and p and a set S of threats, the blockset (also called the mustplay region) is the intersection of the carriers of the threats in S. We call this the blockset B because, in order to block all threats, P's next move must in B: otherwise a threat is not blocked and the opponent can win.

Here is how carriers and threats can be used to prune the search for a winning move. At a node in the search tree, once a winning node for the player-to-move is detected, pass the carrier for the associated winning 1-strat back to the parent as a newly discovered threat. At the parent node, refine the current blockset by intersecting it with the new threat and prune any move not in the refined blockset.

If at a node z all moves are pruned, then z is a loss for the player-to-move P, and the union of all threats (Q-winning 1-strats) has empty intersection and so is a Q-winning 2-strat S: S is passed back to the parent y of z, where together with the move m at y it becomes a Q-winning 1-strat, and so $S' = \{m\}$ union S can immediately be passed back to y's parent x as a new threat.

These ideas are discussed in more detail in §8.5 of Jack van Rijswijck's *Computer Hex: are Bees better than fruitflies?* M.Sc. thesis, UAlberta, 2000. See also his Ph.D. thesis *Set coloring games* Ph.D. thesis, UAlberta, 2006.

Table 4.1 is based on program `hex_vc3.py` in Hayward's GPA github repo: with no transposition table, with move ordering B in Figure 2.2, it solves the empty 4×4 board for Black to play in 0.3 seconds.

```
# return winning move if any and union of winner's threats
0   def threat_search(psn, ptm, mustplay): # no winner yet
1     opt = oppCH(ptm) # opponent of player-to-move
1a    u_threats = set() # union of cells of opt threats
2     while mustplay: # mustplay is not empty
3       k = mustplay.pop() # remove cell k from mustplay
4       new_psn = color_cell(k, psn, ptm)
5       if has_win(new_psn, ptm): # did ptm win?
6         return k, set([k])
7       # if not, continue from new_psn with opt
7a      opt_mp = emptycells(new_psn)
8       opt_mv, threat = threat_search(new_psn, opt, opt_mp)
9       # if opt has no winning move, then
10      #   ptm move at k is a winning move
11      if not opt_mv: # opt loses
11a       threat.add(k)
12        return k, threat
12a     mustplay = mustplay.intersection(threat)
12b     u_threats = u_threats.union(threat)
13    return '', u_threats # no winning move
```

Table 4.1: Threat search listing

4.3 Decomposition search

By hand, Jing Yang found winning 1st-move hex strategies for the 7×7, 8×8 and 9×9 boards by decomposing the board into local searches, for example between a cell and an edge [YLP01; YLP02; YLP07]. Yang's strategies allow a compact opponent-oblivious representation as described at the end of §3.5. Kohei Noshita also found a winning 1st-move 8×8 strategy [Nos05]. Hayward et al. verified one of Yang's strategies [HAH06]. David Pankratz implemented a web-visualizer of Yang's 9×9 strategy [Pan19].

4.4 H-search

Vadim Anshelevich proposed Hierarchical Search (H-search), a bottom-up method to find connections [Ans02]. Recall that for any point-to-point safe-join or weak-join, the carrier is the set of empty cells used by the connection.

Definition 9 *A safe-join is* trivial *if its carrier has no cells. A weak-join is* trivial *if its carrier has exactly one cell.*

For example, if two cells are adjacent, then there is a trivial safe-join between them; if two cells have a common empty neighbor, then there is a trivial weak-join between them. H-search starts with trivial joins and repeatedly finds larger ones according to these series/parallel rules:

- **AND rule.** Assume that a position has a safe-join (x,A,y) between x and y with carrier A, and also a safe-join (y,B,z) between y and z with carrier B, and x does not equal z, and A and B do not intersect:
  - if y is empty then (x, A∪{y}∪B, z) is a weak-join with key y
  - if y is P-colored then (x, A∪B, z) is safe-join for player P.

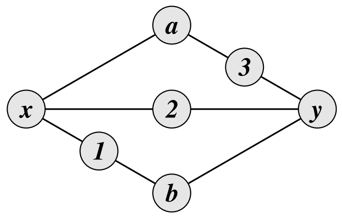


Figure 4.3: The braid, an a-to-b weak-join not found by H-search.

- **OR rule.** Assume for some $t \geq 2$ that position has safe-joins (x, A_1, y), $\ldots$, (x, A_t, y) and that $A_1 \cap \ldots \cap A_t$ is empty. Then $(x, A_1 \cup \ldots \cup A_t, y)$ is a safe-join.

By default, H-search stops only when no new connections are found. In practise, when H-search is used inside a hex player or solver, H-search is modified to stop early. A common variation of H-search is to use the **OR-k rule**, which limits the number of weak-joins that are combined to k.

The H-search combining rules do not find all connections. For example, see the braid in Figure 4.3. Henderson et al. proposed XH-search, a generalization of H-search that finds this and other connections [HAH09a].

Pawelwicz et al. proposed the FastVC Search, which finds connection by changing many aspects of the AND/OR rules. One change is to store connections more efficiently, in particular storing only minimal carriers. which allows many search optimizations. Another change is to apply the OR-rules so that all safe-joins between two fixed endpoints are created by an operation called semis-combiner. Rather than considering separately all weak-join subsets that might give rise to new safe-joins, semis-combiner acts on all input safe-joins at once [Paw+15].

4.5 Computer hex solvers

The first computer hex player was a machine built by Claude Shannon's team around 1950 [Sha50]. The first computer hex solver was by Bert Enderton, who solved all 6×6 and three 7×7 1-move openings by 1995. By 2001 Jack van Rijswijck solved all 6×6 2-move openings [HT19]. By 2003 Hayward et al. solved all 7×7 1-move openings using the 6-cell capture pattern from Theorem 6 [Hay+05]. By 2008, using proof number search, enhanced H-search, a large set of capture and inferior cell patterns, and new hex decomposition observations, Henderson et al. solved all 8×8 1-move openings [HAH09a]. Fabiano and Hayward found more fill and prune patterns [FH19].

By 2008 Martin Müller's research group started work on Fuego, an MCTS platform for go and other games [Enz+10]. Soon after, Arneson et al. modified their hex player/solver package Benzene to run on top of Fuego and built the MCTS player MoHex [AHH10]. Aja Huang and Jakub Pawlewicz later strengthened MoHex [Hua+13; Paw+15]. Chao Gao implemented a neural net player [GHM18].

Pawlewicz designed Scalable Parallel Depth First Proof Number Search (SPDF-PNS) and added it to Benzene. By 2013 SDDFPNS had solved all 9×9 and 2 10×10 openings. Benzene currently has about 80,000 lines of C++ code.

In Python, Masoud Moghadam implemented an MCTS hex player but not a solver [Mog20a; Mog20b].

In 2024 Randall et al. implemented a game-independent `C++` version of EWS and solved go and hex states. With no hex knowledge, this solved the empty 6×6 board in 1520 seconds using 93,963,192 nodes. Adding hex knowledge reduced this to .0005 seconds and 26 nodes [Ran+24].

4.6 EWSPyHex

In the next chapters we show how to craft EWSPyHex, a hex solver based on connection search and EWS. We focussed our attention on solving 6×6 states, as this is the board size where humans start to find positions hard to solve, and the game graph size estimate jumps from $4{\times}10^9$ to $4{\times}10^{14}$. EWSPyHex solves the empty 6×6 board in 12 seconds and each 1-move opening in at most 1.42 hours.

Chapter 5

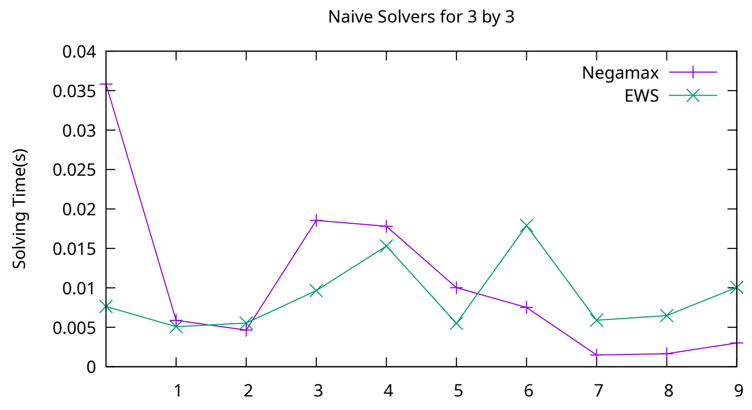
Crafting an EWS hex solver

We start with an EWS solver. We add a minimal set of features so that we can solve all 6×6 hex opening moves in a reasonable amount of time.

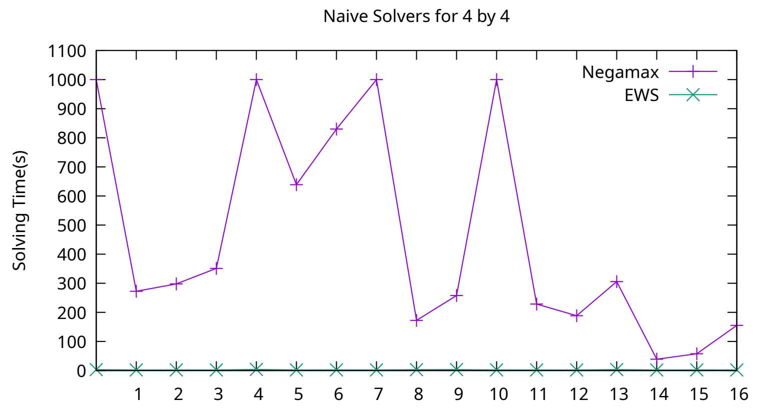
5.1 EWS

Our EWS code is based on EWS0, a game-generic featureless EWS implementation Python implementation by Owen Randall. Our code is stored in a public github repo https://github.com/ChenXinyue11/EWXPyHEX.git

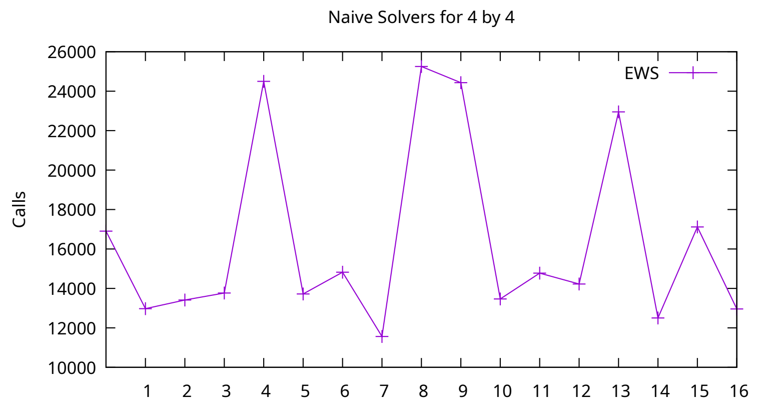
The plot below compares EWS0 and a boolean negamax version of our solver with default move ordering on the empty state and all 1-move openings on the 3×3 and 4×4 boards. In our plots, the leftmost horizontal point corresponds to the empty board; horizontal labels 1, 2, . . . , correspond to cells a1, b1, . . . respectively.



For 3×3 board positions, both solvers can solve all positions.



For 4×4 board positions, in 1000 seconds or less, EWS can solve all positions, and Negamax can solve 13 out of 17 positions. EWS performs better; on average, it takes 2.5 seconds to solve a position. For negamax, including the position that it cannot solve within 1000 seconds, it takes 458 seconds on average. This indicates that moves selected by EW are easier to solve.

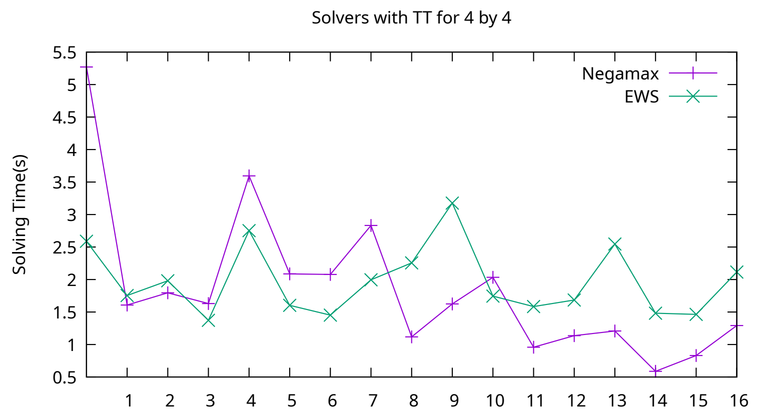


5.2 Transposition tables

We use two transposition tables in Negamax and EWS:

- **Win/loss** We store the win/loss result.
- **H-search transposition** After calling H-search on a state with player-to-move P, we store P's mustplay and – if the solver version uses them – child H-search heuristics.

We first add the Win/loss TT to both solvers.

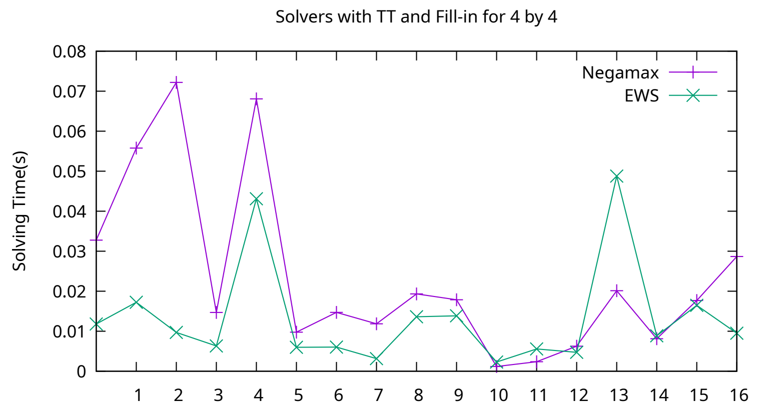


For 10 of the 17 positions, Negamax_TT took about the same time as EWS_TT, with average solving times 1.9 seconds and 2.0 seconds respectively.

5.3 Handling inferior or captured cells

Our solver prunes inferior moves and fills dead or captured cells based only on the three patterns shown in Figure 3.3.

- For negamax, we handle the capture and inferior cells every time we make a move.
- For EWS, we handle the capture and inferior cells both in the expand step and the rollout step.



Appling the capture/inferior/dead patterns reduces solving time for both MmxTTF and EWSTTF: both solve all positions in under one second.

5.4 H-search and variants

H-search of a state with player-to-move P returns this information:

- if solved, the cell set of a winning connection,
- if unsolved, P's current mustplay and, for each cell in the set, a heuristic connection-based score.

In this thesis, we compare different H-search variants of H-search and measure their effect on solver runtime.

5.4.1 Border as midpoint

When using the H-search to find connections, a midpoint is used in the AND-rule to form new weak-join and safe-joins.

A border can be seen as a middle cell where the color is the color of the border. With the border as the middle cell, there are more connections found through the border. More connections can help the solver or player program to heuristic the position more accurately.

5.4.2 Incremental H-search

When using H-search to solve a position, the connections that H-search finds are sometimes not enough to give the final result. Then, the search must go through the position's children. H-search will be used on those children again to help find the result.

Instead of recalculating all connections from scratch, Incremental H-search reuses and updates the previous connections. It is used in Mohex and mentioned in Henderson's thesis: Here we will explain how it works in detail. New cell refers to the cell just colored.

Step 1a: new cell's color differs from player-to-move's color

In this case, the added cell will not create new safe-joins/weak-joins for the H-search to compute. Instead, the cell will block some of the already computed safe-joins/weak-joins from the previous position. The steps are as follows:

1. Go through each safe-join/weak-join computed in the previous position.

2. If the added cell is used as an endpoint, delete the entire group of connections associated with that endpoint.

3. If the added cell is part of the carrier of a safe-join/weak-join delete that single connection.

Step 1b: new cell's color is player-to-move's color

In this case, the new cell leads to new connections:

1. Go through the weak-joins computed in the previous position.

2. If there is an weak-join that uses the added cell as a key, this weak-join becomes a safe-join . The carrier of this new safe-join is the carrier of the original weak-join minus the key.

3. If using cell block as the ends of safe-joins/ weak-joins, then the following needs to be considered. If the added cell forms a new group of cells with other groups, this new group will inherit all the connections of the old groups. The connections between the old groups are deleted, as they have become one block and will be treated as a single endpoint in the H-search.

Final Step: Apply AND/OR Rules

After Step 1, we apply the AND/OR rules to compute any new connections for the updated position.

5.4.3 K-OR rule

In Anshelevich's original algorithm, when using the OR rule to form safe-joins, all combinations of weak-joins are considered. A safe-join will be formed no matter how many weak-joins are involved. In this way, all safe-joins that can be formed from weak-joins are found. However, it is slow to iterate over all the possible combinations.

The K-OR rule reduces H-search runtime by limiting the number of new connections created, finding only those connections that can be formed with k or fewer weak-joins.

5.4.4 Early stop

While using H-search to solve the game's result, sometimes it is not necessary to find all possible connections.

One easy change is that, based on the current player, stop the H-search early once the Win/Lose can be determined.

- If we search for connections of player-to-move the H-search can stop as soon as an weak-join that connects both corresponding edges is formed. In this case, the current player will make the next move on the key of that weak-join and from a safe-join and win the game.

- If we search for connections for the next player, the H-search can stop as soon as a safe-join that connects both corresponding edges is formed. In this case, no matter what the current player plays, the next player will play on the safe-join and win the game.

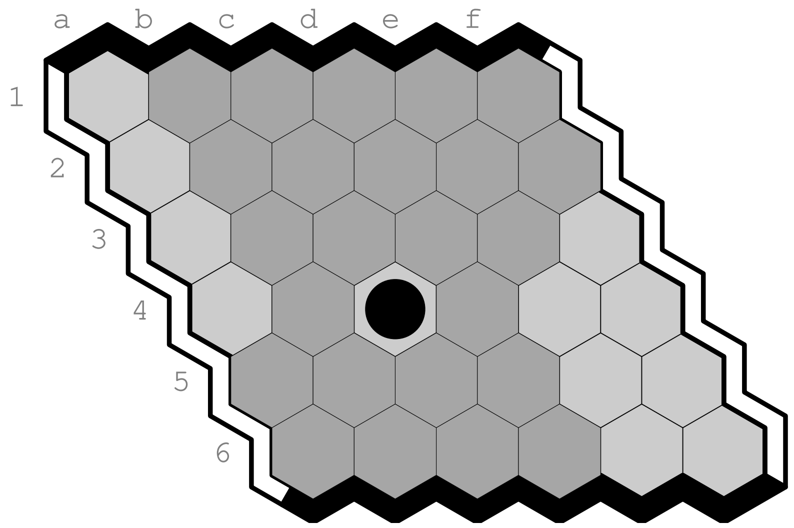


Figure 5.1: Area limited H-search

5.4.5 Optimization

The more connections a position has, the slower the running time to perform H-search is. We only want the important connection that can help the position to find a result or a mustplay set.

Therefore, we reduce the number of connections found by each iteration of the AND-rule and the OR-rule. If there is already a safe-join between the two ends, then when performing the And rule, we skip the step of forming an weak-join for the two ends. After each iteration of forming connections, we remove the weak-joins for which there is already at least a safe-join between two ends. Fewer weak-joins will make the safe-joins form by the OR-rule become fewer, and fewer safe-joins forms will result in fewer weak-joins by the AND-rule.

5.4.6 Area-limited H-search

For a player, start with all empty cells that neighbor a player's color cell. Then add any empty cells in a shortest path from a neighbor cell to the player's edge. Then perform H-search on this cell set. See Figure 5.1.

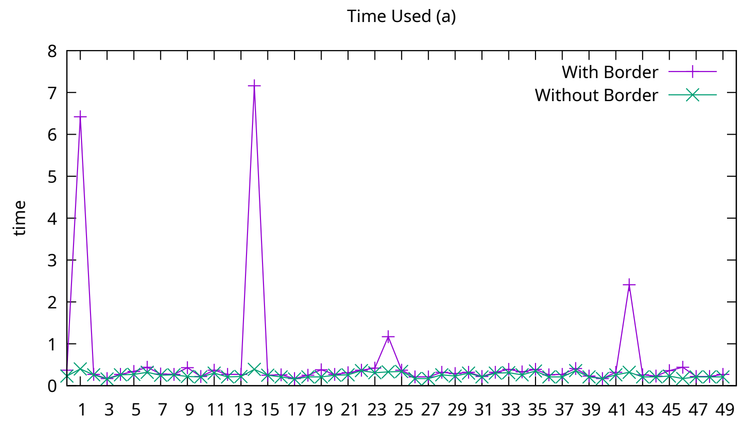
Another idea is area-limited-number H-search: limit the number of safe-joins and weak-joins for cells that are not included in the area when performing an H-search.

5.5 H-search variants comparison

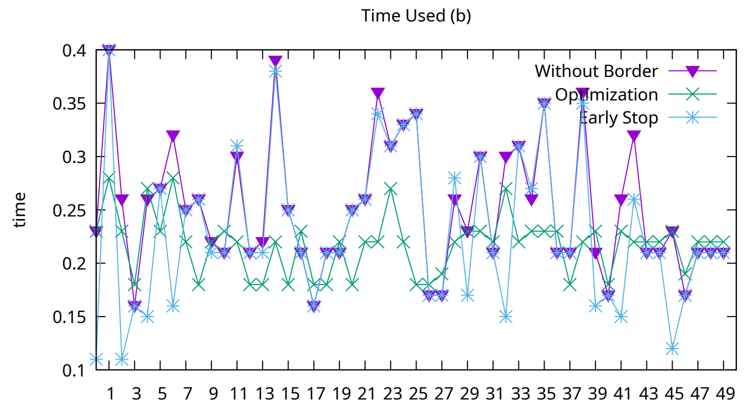
For this section, we compare the different variants' running times and number of connections found.

5.5.1 H-search time

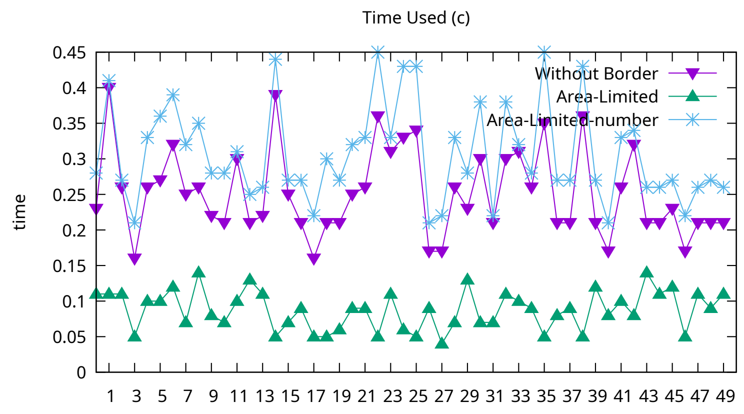
The plots below show the run-time of different H-search variants for 50 3-move opening 6×6 positions. The H-search using the 3-OR rule, without the border as the middle points, is plotted in every plot as a base case.



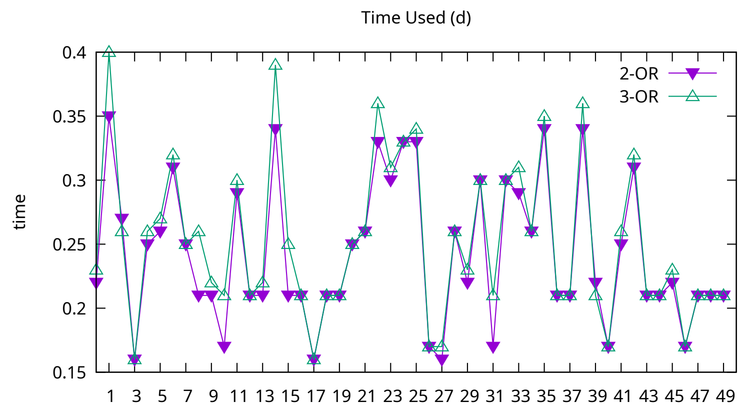
The plot (a) shows the runtime for H-search with the border as the middle points of connections, H-search without the border as the middle points.



The plot (b) shows the runtime for H-search without the border as the middle points, H-search with optimization, and H-search with early stop.



The plot (c) shows the runtime for H-search without the border as the middle points, and Area-limited H-searches.



The plot (d) shows the runtime for H-search without the border as the middle points using the 2-OR rule and the 3-OR rule. For the base case, on 50 positions, it takes 0.25 seconds to perform an H-search on average.

From the plots, we can see that the H-search with the border as the middle point is slower in some positions. On average, it takes 0.62 seconds per position, more than two times as long as the base case. Therefore, we do not use this version in the solver.

Since the XH-search relies on using the border as a middle point for connection to apply the crossing rule and find stepping cells, it finds more connections and also takes more time. We will not consider XH-search in our solver.

Running times for optimization and Early Stop are always no more than the base case. On average, Optimization takes 0.21 seconds per position, and Early stop takes 0.23 seconds per position.

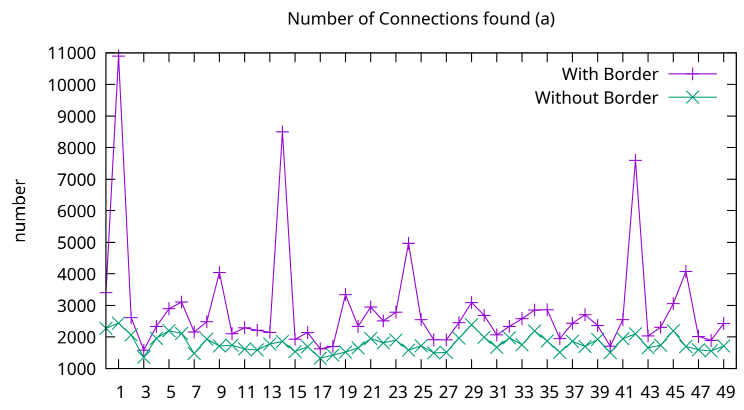
The 2-OR rule is quicker than the 3-OR rule in all positions; it takes 0.24 seconds per position.

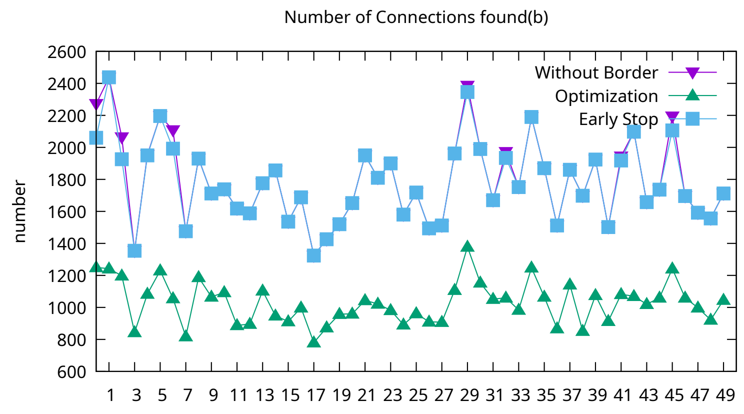
Area-limited H-search takes 0.08 seconds per position.

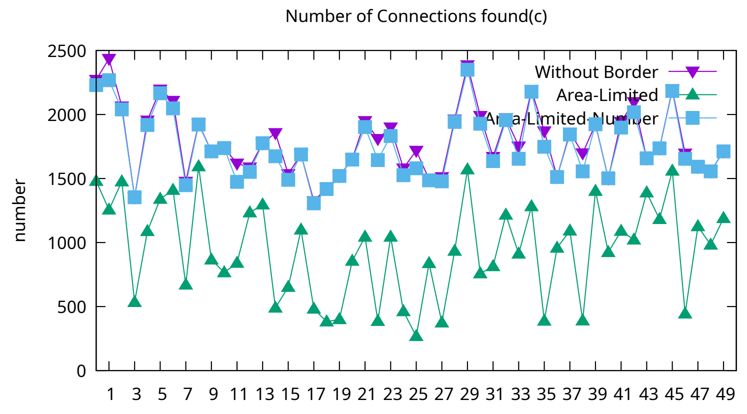
Area-limited-number H-search is always above the base case H-search time. It takes 0.30 seconds per position. The connections the variant finds will be no more than the base case. That makes the variant waste unnecessary time. We will not consider it in our solver.

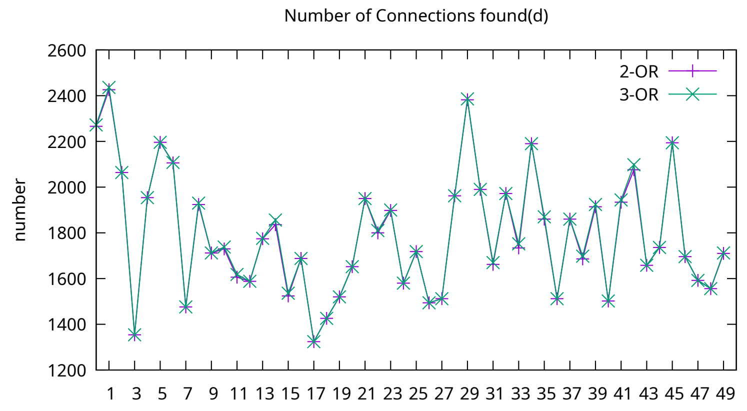
5.5.2 H-search connections found

The variants are plotted in the same way as in the last section.









On average, base case finds 1793 connections per position, H-search with 2-OR rule finds 1789 connections per position, H-search with border as middle point finds 1907 connections per position, Optimization finds 1030 connections per position, Early stop finds 1780 connections per position, Area-limited finds 945 connections per position, Area-limited-number finds 1751 connections per position,

By combining the data plotted above, we see that in most cases, if a variant has a shorter runtime, it will find fewer connections. The question is how the missing connections affect the time needed to solve a position and whether the missing connections are important. One important type of connection is one that connects both edges. They will be used in solving to find a direct win or reducing the number of mustplay cells.

For example, if we apply H-search on a 6×6 board with a black cell on c4 with the

2-OR rule and the 3-OR rule. The 2-OR rule finds 14 weak-joins for the Black player from top to bottom, and the 3-OR rule finds 16. Using the weak-joins, find mustplay cells for the White player, the 2-OR rule finds 4 mustplay cells, and the 3-OR rule finds only 1 mustplay cell.

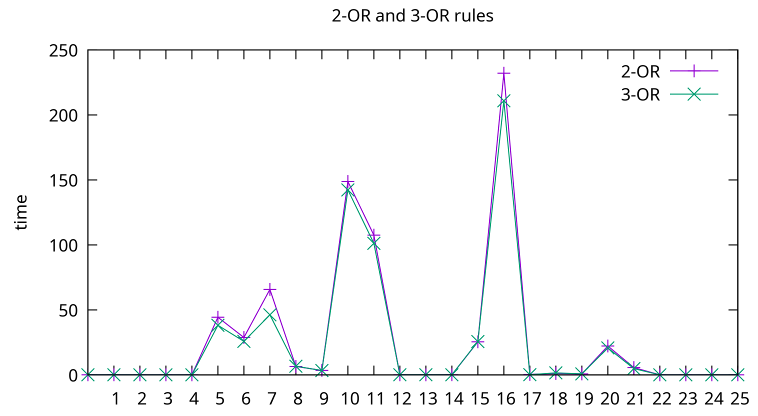
Therefore, the 2-OR rule's searching branch will be larger for this position, which may slow down the search. The trade-off is that if the variant's runtime is fast, will the missing connections slow down the solving time, making it not worth performing?

5.5.3 Solvers with H-search

We apply H-search to NegaMax. When a position is reached, we perform H-searches for the position for both players. If H-search finds an outcome by connections, the position is ended. If the position remains unsolved, use weak-joins for the opponent player to find mustplay cells. Only mustplay cells will be considered as a valid next move.

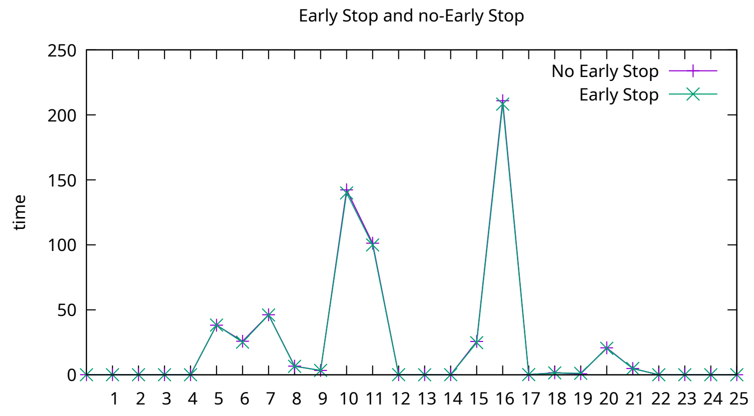
We apply H-search to EWS. When a node expands, we perform H-searches for the position for both players. If the position remains unsolved, only mustplay cells are added as children.

We are using negamax with fill-in as the solver to test how the variants of H-search perform on 5×5 all openings.



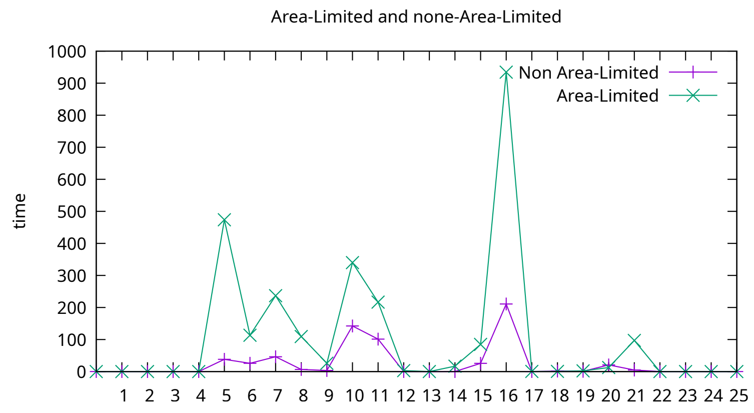
We compare the solving time used among all the variants with the H-search 3-OR rule without middle points.

The H-search with the 3-OR rule perform better than the H-search with the 2-OR rule. On average, the 3-OR rule takes 24.21 seconds to solve a position, and the 2-OR rule takes 26.69 seconds. When solving a position, the 3-OR rule gives narrower mustplay cells for some positions than the 2-OR rule, like the example in the last section. And the extra time token by the 3-OR rule is worth it.



Early Stop is slightly faster, with 23.89 seconds to solve a position.

Early Stop is safe to use, as it will not prune connections until a winning connection is found. In that case, we solve the position and the mustplay cells are not needed. Early Stop works well if a connection from edge to edge is easier to find. For the cases where no winning connections are found, it will not make it any worse.



Area-limited H-search works on the latter board if the positions are easy to find an edge-to-edge connection. However, they do not work in many cases since the local part misses out the important connections and makes the search branch unnecessarily large.

Overall, we prefer an H-search that can find more connections in a reasonable time rather than just finding some connections quickly.

Therefore, based on our data, we prefer the version of H-search without border as the middle point, 3-OR rules, and early stops.

If we perform Optimization, then some of the weak-joins are pruned. When we use the safe-joins and weak-joins for the next position, many connections are lost. For example, if we perform incremental H-search on an empty 6×6 board to a black stone placed on c4, the mustplay size is one. With Optimization, the size becomes four since some weak-joins from the top to the bottom edge are not found.

For the later section, we will compare Optimization and Incremental with both solvers on some 6×6 positions.

5.6 H-search transposition table

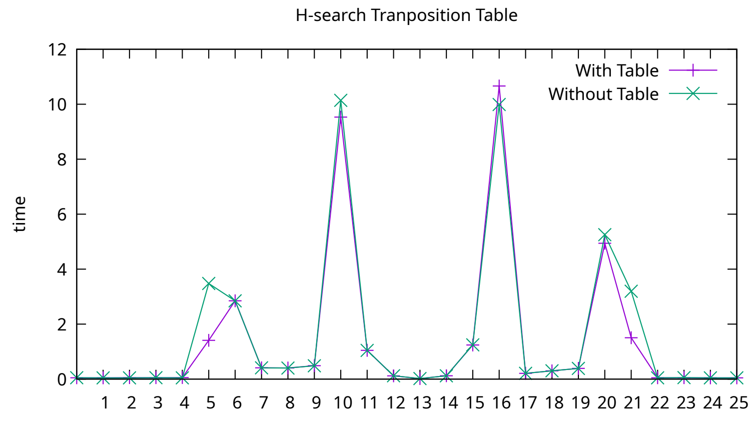
After performing an H-search for a position, we store the mustplay and H-search heuristic score. If the same position is met again, instead of performing H-search again, we look up the mustplay set and H-search heuristic score.

- For Negamax, H-search transposition table is not necessary. If we encounter a position, it will already be solved and stored in the Result transposition table.
- For EWS, we look up the position before performing H-search.

There is one position where using H-search transposition table takes longer to solve.

On average, solving a position with the H-search transposition table takes 1.39 seconds and 1.54 seconds without it.

Without having the H-search transposition table, EWS takes 1.1 times the runtime of having one. There are positions that EWS already processes, but the result of the games remains unknown. Having an H-search transposition Table saves time from performing an H-search again once we expand the same position.



5.7 Move ordering

After performing an H-search, we collect connection information for each empty cell. We use this information not only to determine the mustplay but also to determine move ordering. If no win/loss is found, player-to-move has no edge-to-edge weak-join or safe-join: we then score cells depending on whether they safely join one edge and weakly join the other. Moves are ordered as follows.

For each cell, we find the distance to and number of connections to the player-to-move's edges. First, we assign an initial score of 0 to each empty cell. Then we see if there are any safe-joins connecting the empty cell to player-to-move's edges(either one of the two edges). If not, we keep the score 0. If there is, we get the distance of the empty cell to the edge and add the distance to the score. Then we see if any weak-joins connects the empty cell to the other edge. If there are, we add a value to the score based on how many weak-joins there are. Then check the other edge, if there is also a safe-joins to that edge, do the same thing. We keep the larger score. After all empty cells are checked, we sort the empty cells in descending order according to the score.

```
def move_order(cellist,board,vcs,svcs,player):
```

```
move_score_list = [] #init empty list for move scores pairs
edge1 = Edge[0] # player edge, top/black, left/white
edge2 = Edge[1] # player edge, bottom/black, right/white
for cell in cellist:
  score1 = 0
  score2 = 0
  key1 = edge1+cell # key for first edge.
  key2 = edge2+cell # key for second edge.
  if len(vcs[key1]) >0:
    # dist. to 1st edge if cell has safe join to 1st edge
    score1 = Distance_to_edge1(cell,player)
    # if cell has weak join to 2nd edge, add number joins
    if len(svcs[key2]) >0:
      score1+= len(svcs[key2])*0.5
  if len(vcs[key2]) >0:
    # dist. to 2nd edge if cell has strong join to 2nd edge
    score2 = Distance_to_edge2(cell,player)
    # if cell has weak join to 1st edge, add number joins
  if len(svcs[key1]) >0:
    score2+= len(svcs[key1])*0.5
  score = max(score1,score2)
  move_score_list.append((cell,score))
#give a score for each cell based on dist. to edges
move_score_list = center_weight(move_score_list)
# sort move score list by score.
move_score_list.sort(key=lambda a:a[-1],reverse=True)
return move_score_list
```

Center weight

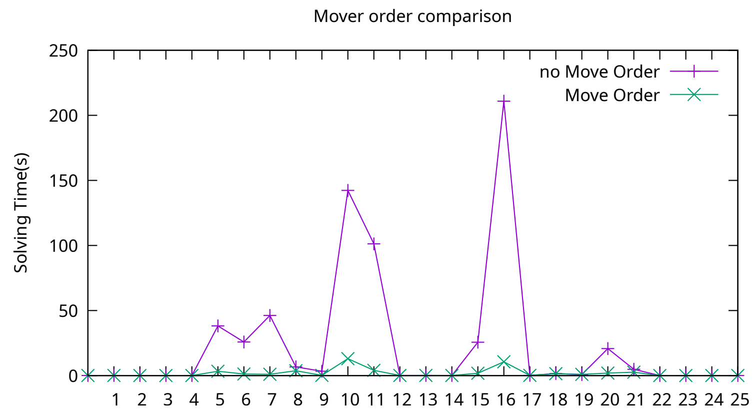
For move ordering, we give more weight to the center cells. We label cells by row and column number, counting from 0.

$$\text{weight}(r,c) = \frac{\min(r, n-(r+1)) + \min(c, n-(c+1))}{\text{scaling factor}} + \text{boost}$$

We use scaling factor 10 give a boost of 1 to each cell on the main diagonal (obtuse-corner to obtuse-corner). The weight is added to each cell's score.

5.8 Negamax move orders

We use the move order function with the negamax solver to determine which cell to search next. We test on empty and all one-move opening 5×5 positions for the negamax solver using 3-OR, with/without move order.



With the default move order, it takes 24.21 seconds to solve a position, whereas with the move order, it takes 1.81 seconds per position.

For a position that has a large size of mustplay like an empty board, move ordering is most likely to make the solving time quicker. If the position is losing, no matter what the order of search is, all cells in the mustplay will be searched. However, the children can also benefit from the order.

For example, on an empty 6×6 board, the solving time without moving orders takes more than 1000 seconds, and with moving orders, it takes 8 seconds.

5.9 Heuristic score in EWS calculation

After performing H-search, we get the connection heuristic score for each child. We introduce three new variables to the EWS formula:

- **MaxScore:** The maximum score among all children,
- **Score:** The child's score. Since EWS select the child with the lowest EW first, we use Maxscore minus the child's score as Score.
- **W:** Weight to control the effect of heuristic scores.

In calculating expected work (EW), we add Weighted MaxScore to the number of visits and Weighted Score to the number of wins:

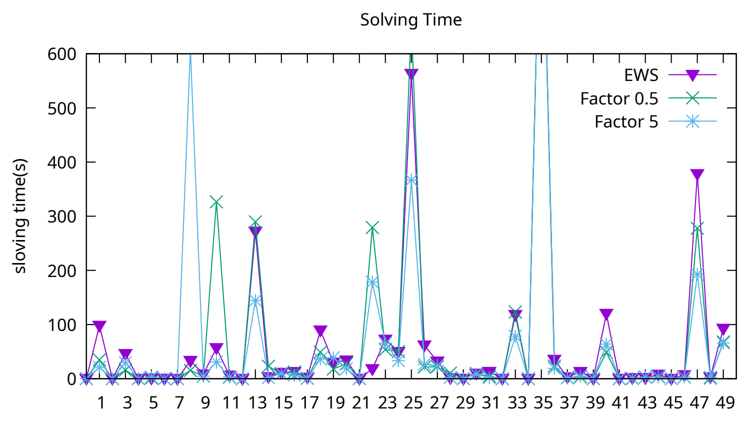
$$\mathrm{WR}(X) = \frac{(\mathrm{WINS} + \mathrm{W^*SCORE})}{(\mathrm{VISITS} + \mathrm{W^*MAXSCORE})}$$

Notice H-scores have a significant influence on child selection When visit numbers are small, this influence decreases as the search continues and the number of visits increases.

We test the solvers on 3-move openings 6×6 board.

Without H-scores, it takes 66.16 seconds per position.

For W at 0.5, it takes an average of 69.98 seconds to solve a position. For W at 5, it takes an average of 62.15 seconds to solve a position.

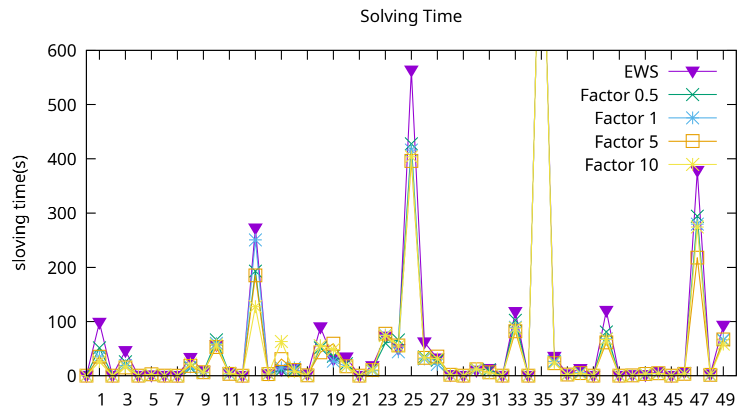


Although, on average, with W at 5, the EWS solver solves faster than without H-scores, there are positions that are solved slower.

For example, for the position at label 8 (e6, f2, a1), EWS finds a winning move at b5, and EWS with H-scores finds a winning move at c4. The H-scores added to the EW formula keep the solver choosing hard-to-solve children.

Instead of the H-scores, we use the order found by the H-scores.

- **MaxScore:** Number of children
- **Score:** The index of the child in the ordered move.
- **W:** Weight to control the effect of heuristic scores.

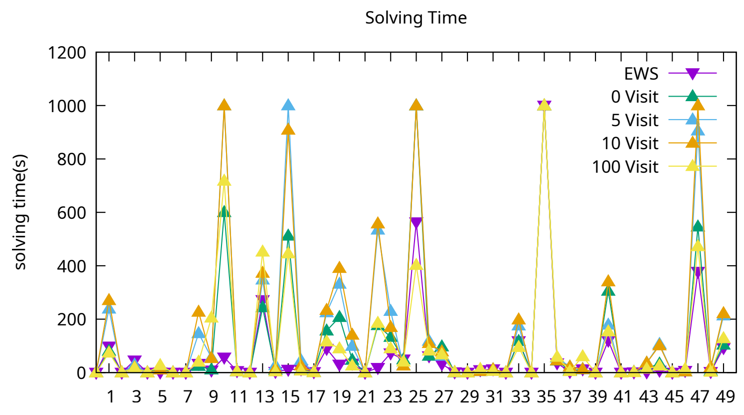


For W at 0.5 (resp. 1, 5, 10), it takes 54.90 (53.74, 51.78, 52.30) seconds to solve a position. At W at 5, the EWS perform best; we use this W value in our solvers.

There are still two positions that EWS solves quicker without H-scores, but the differences are not as significant as using H-scores directly.

5.10 H-search based on visits

H-search is computationally expensive and does not always prune the mustplay set. For this reason, we perform H-search at a node only if the number of visits exceeds a certain threshold, and then perform H-search during selection but not during expansion. If H-search solves the position, we return the win/loss result. Otherwise, we prune children not in the mustplay.



When we perform H-search at visit 0 in backpropagation, it takes 113.70 to solve a position. Performing H-search when a node reaches 5 (resp. 10, 100) visits takes 165 (172, 102) seconds per position.

From the above plot, we can see that performing H-search after some number of visits is not always worthwhile.

For most positions, if we do not perform H-search early, we will waste time on the nodes that are supposed to be pruned by the mustplay that H-search finds out.

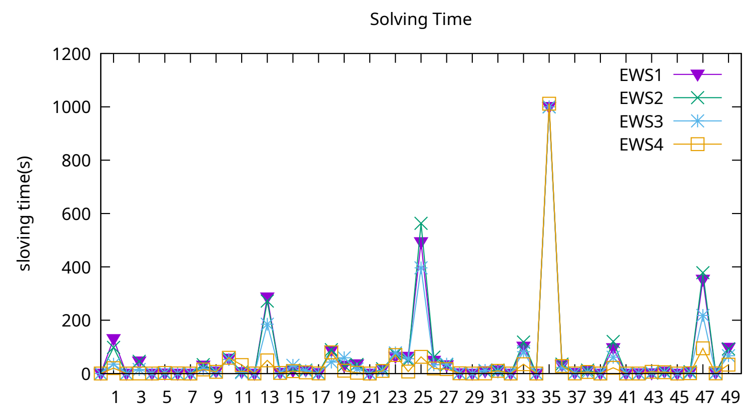
Overall, the time that H-search uses to prune nodes and end game early outweighs the time that we use more simulation to find a good node to perform H-search.

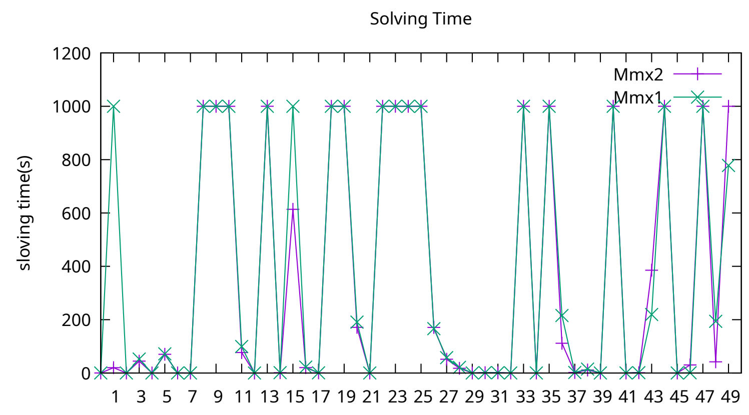
5.11 Negamax versus EWS: 6x6 data

Here we compare versions of Negamax and versions of EWS using 50 3-move openings on 6×6 board:

- Mmx1: boolean negamax with result tranposition table, fillin, H-search without border as middle point with 3-OR rule, early stop, optimization, and move ordering.

- Mmx2: boolean negamax with result tranposition table, fillin, H-search without border as middle point with 3-OR rule, Early stop, Incremental, and move ordering.
- EWS1: EWS with result Tranposition table, fillin, H-search without border as middle point with 3-OR rule, Early stop, Incremental.
- EWS2: with Tranposition tables, fillin, H-search without border as middle point with 3-OR rule, Early stop, Optimization.
- EWS3: with Tranposition tables, fillin, H-search without border as middle point with 3-OR rule, Early stop, Optimization, H-scores order.
- EWS4: with Tranposition tables, fillin, H-search without border as middle point with 3-OR rule, Early stop, Incremental, H-scores order.

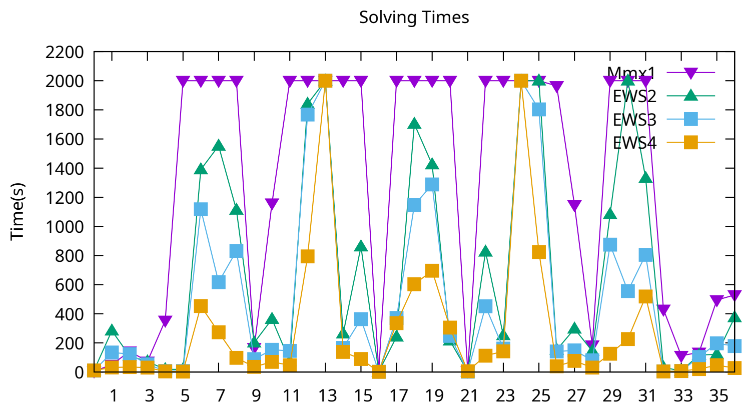




For boolean negamax solvers, Mmx2 solves 34 out of 50 positions, whereas Mmx1 solves 33. Both solvers solve positions that the other one can't solve. Therefore, it is unclear which is better. Both of them are slower than EWS solvers.

For EWS solvers, all four solve 49 positions in under 1000 seconds. EWS4 performs best at 14.85 seconds per position, excluding the unsolved position. EWS1 takes (resp. EWS2, EWS3) 44.15 seconds (47.10, 32.36), so 3.0 (3.2, 2.2) times the runtime of EWS4.

5.12 Solving 6x6 empty board and 1-move openings



For Mmx1, it solves 17 out of 37 positions under 2000 seconds.

EWS2 (resp. EWS3, EWS4) solves 33 (35, 35) positions under 2000 seconds.

Excluding the four unsolved positions that EWS2 cannot solve, EWS2 (resp. EWS3, EWS4) took 500.00 (357.35, 157.75) seconds. EWS2 take 3.2 times the runtime of EWS4, EWS3 take 2.3 times the runtime of EWS4.

Excluding the two unsolved positions, EWS3 took 404.34 seconds per position, and EWS4 took 178.74 seconds per position. EWS3 take 2.3 times the runtime of EWS4.

e1, b4, d1	f2, c2, f6	a4, b5, e6	e5, f5, b3	a5, c2, b4
a1, b3, f4	e4, c3, d4	d5, e4, d3	e6, f2, a1	a3, b6, d6
a2, c6, a5	a5, f1, d2	d3, f2, c6	a3, f3, f1	f6, b1, d4
d4, b6, f3	c2, e4, f3	d3, c1, b3	f4, a1, b4	a4, d6, e5
b1, f4, c5	e6, b3, d2	e6, a1, f3	f3, c3, d3	a3, d6, c5
e1, a1, d2	e4, f4, d1	c6, f5, e4	e6, b2, a4	f1, d3, d1
a1, f5, b1	f5, d1, e4	a5, c2, d5	e4, a6, f3	a1, e4, f5
a3, f6, a1	e4, b3, b4	d1, b6, c1	c6, e6, c4	a5, c5, d2
b4, b6, f2	d1, c2, b5	d5, b2, f6	b5, f2, f5	c1, a4, f3
e6, e3, c6	e6, d6, d1	e4, f3, e6	f2, a3, b5	b3, f3, b1

Table 5.1: 50 3-move 6×6 openings

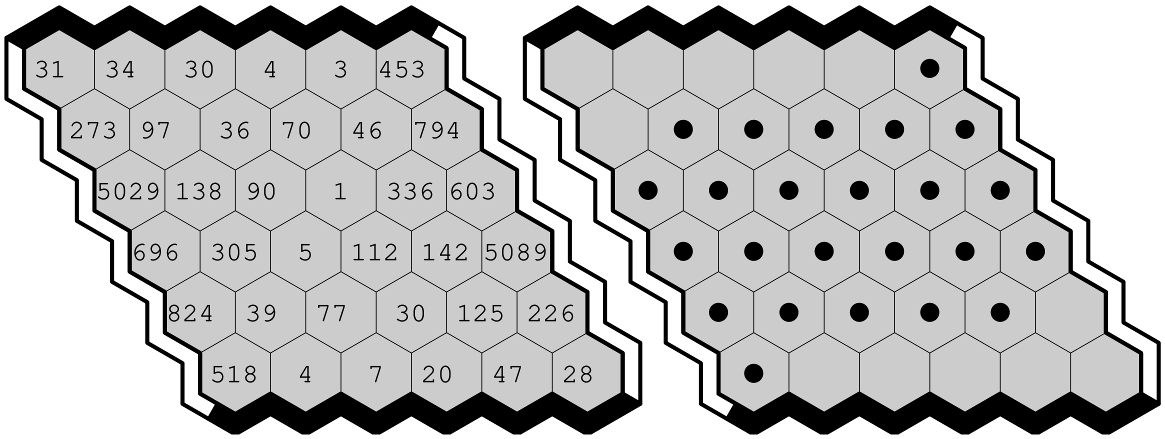


Figure 5.2: EWSPyHex solving time (seconds), all winning moves.

The best version of our EWS solver has these features:

- win/loss and H-search transposition tables
- capture and dead fill-in, inferior pruning,
- H-search with options 3-OR, early stop, incremental,
- H-scores order

This version solved the empty board and all 1-move openings in under 4500 seconds each. Opening moves `a3` and `f4` took the most time. See Figure 5.2.

Chapter 6

Conclusion

We showed that adding the hex-specific features of fill-in, H-search, H-scores and mustplay reasoning to EWS gives a Python hex solver that can solve all 6×6 1-move opening in at most 5100 seconds, with only 2 openings taking longer 2000 seconds. The best negamax version solved only 16 of the 36 openings in at most 2000 seconds.

We considered other features that did not reduce runtime of our EWS solver. We expected that limiting H-search runtime by restricting the number of connections found would reduce solving time, the opposite was true: spending a little more time to reduce the mustplay was worthwhile.

6.1 Future work

There is space to improve our solver. Having pre-calculated H-search information, such as connections and mustplay sets, will help reduce solving time. When choosing a move, we could prefer a bridge jump to connect to another group. Pre-storing more capture and inferior patterns can prune more moves and and reduce runtime. Backing up threats found at a terminal position might also reduce runtime.

H-score, the heuristic move score based on H-search computation, gives a high score to moves in more connections, but not necessarily to moves with least searching work. Instead of using the number of weak-joins for H-scores, we could try ordering the cells using the intersection size of weak-joins.

When doing H-search, we could focus less on the connections that are not in the mustplay set.

Running an H-search on a near-ended position to find a result might be slower than solving the position (with patterns like bridge connection) without using H-search. The idea is that once we have reached a certain depth, we stop performing the H-search.

Bibliography

[AHH10] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. “Monte Carlo Tree Search in Hex”. In: *IEEE Trans. Comput. Intell. AI Games* 2.4 (2010), pp. 251–258.

[AMH94] L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. “Proof-Number Search”. In: *Artif. Intell.* 66.1 (1994), pp. 91–124.

[Ans02] Vadim V. Anshelevich. “A hierarchical approach to computer Hex”. In: *Artif. Intell.* 134.1-2 (2002), pp. 101–120.

[Ber77] Claude Berge. *L’Art Subtil du Hex.* manuscript. 1977.

[Bro+12] Cameron Browne et al. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Trans. Comput. Intell. AI Games* 4.1 (2012), pp. 1–43.

[Bro00] Cameron Browne. *Hex Strategy: Making the Right Connections.* Natick: AK Peters, 2000.

[Cou06a] Rémi Coulom. “Crazystone wins 9×9 Go Tournament”. In: *International Computer Games Assoc.* 29.2 (June 2006), pp. 98–99.

[Cou06b] Rémi Coulom. “Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search”. In: *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers.* Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers. Vol. 4630. Lecture Notes in Computer Science. Springer, 2006, pp. 72–83.

[Enz+10] Markus Enzenberger et al. “Fuego - An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search”. In: *IEEE Trans. Comput. Intell. AI Games* 2.4 (2010), pp. 259–270.

[FH19] Nicolas Fabiano and Ryan Hayward. “New Hex Patterns for Fill and Prune”. In: *Advances in Computer Games - 16th International Conference, ACG 2019, Macao, China, August 11-13, 2019, Revised Selected Papers.* Ed. by Tristan Cazenave et al. Vol. 12516. Lecture Notes in Computer Science. Springer, 2019, pp. 79–90.

[gee] geeksforgeeks.org. *ML | Monte Carlo Tree Search.* URL: https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/ (visited on 05/15/2025).

[GHM18] Chao Gao, Ryan Hayward, and Martin Müller. “Move Prediction Using Deep Convolutional Neural Networks in Hex”. In: *IEEE Trans. Games* 10.4 (2018), pp. 336–343.

[HAH06] Ryan B. Hayward, Broderick Arneson, and Philip Henderson. “Automatic Strategy Verification for Hex”. In: *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers. Vol. 4630. Lecture Notes in Computer Science. Springer, 2006, pp. 112–121.

[HAH09a] Philip Henderson, Broderick Arneson, and Ryan B. Hayward. “Hex, Braids, the Crossing Rule, and XH-Search”. In: *Advances in Computer Games, 12th International Conference, ACG 2009, Pamplona, Spain, May 11-13, 2009. Revised Papers*. Ed. by H. Jaap van den Herik and Pieter Spronck. Vol. 6048. Lecture Notes in Computer Science. Springer, 2009, pp. 88–98.

[HAH09b] Philip Henderson, Broderick Arneson, and Ryan B. Hayward. “Solving 8x8 Hex”. In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier. 2009, pp. 505–510.

[Hay+05] Ryan Hayward et al. “Solving 7x7 Hex with domination, fill-in, and virtual connections”. In: *Theor. Comput. Sci.* 349.2 (2005), pp. 123–139.

[Hay03] Ryan Hayward. “Berge and the Art of Hex”. manuscript in honour of Claude Berge. 2003. URL: www.cs.ualberta.ca/~hayward/papers/pton.pdf.

[Hay22] Ryan Hayward. *Hex, a playful introduction*. Providence: MAA Press, 2022.

[Hay25] Ryan Hayward. *Games-puzzles-algorithms github repo*. UAlberta CMPUT 355. 2025. URL: https://github.com/ryanbhayward/games-puzzles-algorithms (visited on 05/25/2025).

[Hei42] Piet Hein. *Vil De laere Polygon? Politiken* newspaper, p4. Dec. 26, 1942.

[HR06] Ryan B. Hayward and Jack van Rijswijck. “Hex and combinatorics”. In: *Discret. Math.* 306.19-20 (2006), pp. 2515–2528.

[HT19] Ryan Hayward and Bjarne Toft. *Hex, the full story*. London: CRC Press, 2019.

[Hua+13] Shih-Chieh Huang et al. “MoHex 2.0: A Pattern-Based MCTS Hex Player”. In: *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers*. Ed. by H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat. Vol. 8427. Lecture Notes in Computer Science. Springer, 2013, pp. 60–71.

[Kis+12] Akihiro Kishimoto et al. “Game-Tree Search Using Proof Numbers: The First Twenty Years”. In: *J. Int. Comput. Games Assoc.* 35.3 (2012), pp. 131–156.

[KS06] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Vol. 4212. Lecture Notes in Computer Science. Springer, 2006, pp. 282–293.

[Mog20a] Masoud Masoumi Moghadam. *MCTS-agent-python*. 2020. URL: https://github.com/masouduut94/MCTS-agent-python (visited on 05/07/2025).

[Mog20b] Masoud Masoumi Moghadam. *MCTS: implementing RL in real-time game player*. 2020. URL: https://towardsdatascience.com/monte-carlo-tree-search-implementing-reinforcement-learning-in-real-time-game-player-a9c412ebeff5/ (visited on 05/07/2025).

[Nos05] Kohei Noshita. “Union-Connections and Straightforward Winning Strategies in Hex”. In: *J. Int. Comput. Games Assoc.* 28.1 (2005), pp. 3–12.

[Pan19] David Pankratz. *A web-visualizer of Jing Yang’s 9×9 Hex strategy*. 2019. URL: https://webdocs.cs.ualberta.ca/~hayward/355/asn/hexviz/ (visited on 05/15/2025).

[Paw+15] Jakub Pawlewicz et al. “Stronger Virtual Connections in Hex”. In: *IEEE Trans. Comput. Intell. AI Games* 7.2 (2015), pp. 156–166.

[PH13] Jakub Pawlewicz and Ryan B. Hayward. “Scalable Parallel DFPN Search”. In: *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers*. Ed. by H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat. Vol. 8427. Lecture Notes in Computer Science. Springer, 2013, pp. 138–150.

[Ran+24] Owen Randall et al. “Expected Work Search: Combining Win Rate and Proof Size Estimation”. In: *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*. ijcai.org, 2024, pp. 7003–7011.

[Rij06] Jack van Rijswijck. “Set Colouring Games”. PhD thesis. University of Alberta, 2006.

[Sel25] Peter Selinger. *Hex Templates*. https://www.mathstat.dal.ca/~selinger/templates/ [Accessed: 2025-05-09]. 2025.

[Sey20a] Matthew Seymour. *Hex puzzles*. 2020. URL: http://www.mseymour.ca/hex_puzzle/hexpuzzle.html (visited on 05/07/2025).

[Sey20b] Matthew Seymour. *Hex: a strategy guide*. 2020. URL: http://www.mseymour.ca/hex_book/hexstrat.html (visited on 05/07/2025).

[Sha50] Claude Shannon. “Programming a computer for playing chess”. In: *Philosophical Magazine*. 7th ser. 41.314 (Mar. 1950), pp. 256–275.

[Sil+16] David Silver et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search”. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489.

[ST75] Craige Schensted and Charles Titus. *Mudcrack Y and Poly-Y*. Peaks Island, Maine: Neo Press, 1975.

[WBS08] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. "Monte-Carlo Tree Search Solver". In: *Computers and Games, 6th International Conference, CG 2008, Beijing, China, Sept 29 — Oct 1*. Ed. by H. Jaap van den Herik et al. Vol. 5131. Lecture Notes in Computer Science. Springer, 2008, pp. 25–36.

[YLP01] Jing Yang, Simon X. Liao, and Miroslaw Pawlak. "A decomposition method for finding solution in game Hex 7×7". In: *Proceedings of the 1st International Conference on Application and Development of Computer Games in the 21st Century, held on 22-23 November 2001 in City University of Hong Kong*. Ed. by Cyril Tse Ning. City University of Hong Kong, 2001, pp. 96–111.

[YLP02] Jing Yang, Simon X. Liao, and Miroslaw Pawlak. "New Winning and Losing Positions for 7x7 HEx". In: *Computers and Games, Third International Conference, CG 2002, Edmonton, Canada, July 25-27, 2002, Revised Papers*. Ed. by Jonathan Schaeffer, Martin Müller, and Yngvi Björnsson. Vol. 2883. Lecture Notes in Computer Science. Springer, 2002, pp. 230–248.

[YLP07] Jing Yang, Simon X. Liao, and Miroslaw Pawlak. "Apply Heuristic Search to Discover a New Winning Solution in Hex Game". In: *Fourth International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2007, 24-27 August 2007, Haikou, Hainan, China, Proceedings, Volume 4*. Ed. by J. Lei. IEEE Computer Society, 2007, pp. 328–333.