Solving $8 \times 8$ Hex

Philip Henderson
Dept. of Computing Science
University of Alberta
ph@cs.ualberta.ca

Broderick Arneson
Dept. of Computing Science
University of Alberta
broderic@cs.ualberta.ca

Ryan B. Hayward
Dept. of Computing Science
University of Alberta
hayward@cs.ualberta.ca

Abstract

Using efficient methods that reduce the search space, we design an algorithm strong enough to solve all $8 \times 8$ Hex openings.

1 Introduction

We design an algorithm strong enough to solve all $8 \times 8$ Hex openings. Our algorithm builds on that of Hayward et al. [2005], which uses depth first search with a virtual connection engine (VCE) and some inferior cell analysis. Our new contributions are graph theoretic inferior cell methods, combinatorial decompositions, proof set shrinking, and transposition deductions. Our improvements to Hayward et al.’s algorithm include an inferior cell engine (ICE) with over 250 inferior cell patterns, and the use of precomputed connections in H-search.

1.1 Previous work

Hex is the classic two-player board game invented by Piet Hein [1942] and John Nash [1952]. The board is an $n \times n$ rhombus of hexagonal cells. Players alternately place a stone of their color on any empty cell. To win, a player connects her two opposing sides with her stones. Draws are not possible.

Nash proved the existence of a first-player-win strategy [1952] for zero-stone $n \times n$ Hex states (positions), but no explicit strategy that holds for all $n$ is known. Indeed, solving arbitrary states is PSPACE-complete [Reisch, 1981].

For even relatively small boards, solving Hex states by brute force search is infeasible. The number of internal nodes in the search tree is at most the number of distinct board states corresponding to legal move sequences. A conservative estimate of the latter number is the number of distinct board states in which the board is at most half full. This estimate includes some invalid states: those in which one player already has a winning path. However, it omits (probably more) valid states in which the board is more than half full. See Fig. 1 and the formula by John Tromp on page 6 of Browne’s book [2000].

Hayward et al. [2005] gave an algorithm strong enough to solve any $7 \times 7$ state; they solved all 49 $7 \times 7$ openings (one-stone states) with search trees totalling 14.2 million nodes. Rasmussen et al. [2007; 2008] verified these results more efficiently, requiring fewer than 0.5 million nodes, but were unable to solve any $8 \times 8$ states; Rasmussen estimates his algorithm would take 9 years to solve all $8 \times 8$ openings. By hand, Yang [2002] and Noshita [2005] solved one $8 \times 8$ opening, Yang [2003] solved the center $9 \times 9$ opening. See Fig. 2.

Figure 1: Number of at most half full $n \times n$ Hex states.

<table>
<thead>
<tr>
<th>n \times n</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 \times 1</td>
<td>1</td>
</tr>
<tr>
<td>2 \times 2</td>
<td>9</td>
</tr>
<tr>
<td>3 \times 3</td>
<td>554</td>
</tr>
<tr>
<td>4 \times 4</td>
<td>7.6e5</td>
</tr>
<tr>
<td>5 \times 5</td>
<td>4.0e9</td>
</tr>
<tr>
<td>6 \times 6</td>
<td>4.0e14</td>
</tr>
<tr>
<td>7 \times 7</td>
<td>1.5e20</td>
</tr>
<tr>
<td>8 \times 8</td>
<td>1.0e27</td>
</tr>
</tbody>
</table>

Figure 2: Previously solved opening moves. Color shows winner if Black opens there. $8 \times 8$ results were found by hand.

Fig. 1 suggests that solving $8 \times 8$ states is orders of magnitude harder than solving $7 \times 7$ states. Van den Herik et al. observed [2002], “Solving the $8 \times 8$ game is entirely intractable without fundamental breakthroughs.” We describe such breakthroughs, yielding an algorithm strong enough to solve all $8 \times 8$ openings in a total runtime of less than two weeks.

2 Connections and mustplay

In this section we describe the connection concepts underlying Hex solvers. Like many connection games, Hex has the
“sudden death” property. In a typical Hex state, a player’s opponent has one or more immediate winning threats. The player must play at a cell that interferes with each of these threats to avoid an immediate loss. Consider Fig. 3. In the top two diagrams, the shaded cells indicate immediate Black winning threats. Thus, as indicated in the bottom diagram, White must play in the intersection of these two threats.

Formally, with respect to a Hex state, a player $P$, and a pair of destinations, a virtual connection (resp. semi connection) is a second-player (resp. first-player) inter-destination connection strategy for $P$. The set of empty cells used by a connection is its carrier. A connection strategy is winning if it connects its player’s two sides. With respect to a set of opponent winning semi connections, a player’s must-play is the intersection of the carriers of these connections. The top diagrams of Fig. 3 show the carriers of a Black winning semi connection; the right diagram shows the associated White mustplay.

![Figure 3: White to play. Each marked cell shows a Black winning threat that uses only the shaded cells. White must play at one of the cells (bottom) that interferes with each threat.](image)

To solve Hex states, it is critical to compute mustplays efficiently, as the discovery of a winning virtual connection allows the declaration of the winner long before either player has an absolute connection. One such algorithm is Anshelevich’s H-search [Anshelevich, 2000; Melis, 2003], which builds up connections by repeatedly applying combining rules. Following Hayward et al. [2005], we incrementally update connection lists using H-search augmented with a pre-computed list of connections to the sides [King, 2001]. Also, during our search for a winning connection, we use discovered carriers of child state connections to refine the mustplay [van Rijswijk, 2003]. If at some point we discover that $P$’s opponent $Q$ has a set of winning semi connections whose associated $P$-mustplay is empty, then our search is over: $Q$ has a winning virtual connection whose carrier is the union of the carriers semi connections.

In our solver, when no opponent winning semi connections are known, we order moves with an electric circuit model evaluation function [Anshelevich, 2000; Melis, 2003]. Once a nontrivial mustplay is detected, we order moves by must-play size, breaking ties with the circuit function.

### 3 Cell properties

Mustplay computation allows us to prune moves based on global information, namely a set of threatening opponent strategies. Another form of pruning follows by observing that moves into certain local board patterns are provably inferior.

The value of a move is the win/loss value of the resulting game state. A cell of a Hex state is inferior if there is some other cell whose value is at least as good. Inferior cells can often be pruned from the list of moves to consider. For example, a cell (empty or occupied) is dead if it is not on any minimal winning path. A cell is dead for one player if and only if it is dead for the other, so adding a stone (of either color) to a dead cell does not change a state’s value. A set of cells is $P$-captured if $P$ has a second player strategy that leaves every cell dead or with a $P$-stone. Filling in a $P$-captured set with $P$-stones does not change a state’s value. An empty cell is $P$-vulnerable if some $Q$-reply makes it dead; it is $P$-capture-dominated if a $P$-move to some other cell $P$-captures it. Any $P$-move to a cell that is dead, captured, or $P$-dominated is inferior [Björnsson et al., 2007]. See Fig. 4.

![Figure 4: In the top five patterns, the empty cell is dead. In the remaining patterns, the empty cell set is Black-captured.](image)

In solving $7 \times 7$ Hex, Hayward et al. [2005] pruned inferior cells using only one captured pattern. By contrast, our ICE matches 273 patterns computed as by Björnsson et al. [2007], including 5 dead, 12 captured, 34 vulnerable, and 219 capture-dominated. See Fig. 4. ICE also matches 3 induced-path-domination patterns in the acute corner, as defined by Henderson et al. [2008]. ICE iteratively fills in dead and captured cells until no further fill-in can be deduced, and then computes vulnerable and dominated cells for the player to move. See Fig. 5.

![Figure 5: A state with initial dead cells (top left), captured cells (top right), iteratively-computed dead and captured cells (middle), and subsequent Black-vulnerable/dominated (bottom left) and White-vulnerable/dominated cells (bottom right).](image)

In addition to pattern matching, ICE computes dead cells
using graph-theoretic properties of the Hex state. For a Hex state and a player $P$, consider the cell adjacency graph, in which vertices are adjacent if the corresponding empty cells touch or are connected by a path of $P$-stones. In this graph, the set of empty cells neighboring a (connected) group of $P$-stones forms a clique. Furthermore, any cell cut from either $P$-side by a $P$-clique is dead, since it cannot be on any minimal winning path. ICE efficiently checks for dead regions by determining whether the empty neighbors of each group isolate any cells. See Fig. 6.

ICE also deduces inferior cell information during the 1-ply search that is performed at each node for the purpose of move ordering the children. For example, any cell of a child state that becomes dead after fill-in is vulnerable in the parent state. In this manner ICE prunes many vulnerable and capture-dominated cells.

Figure 6: These regions are dead due to clique cutsets.

4 Combinatorial decompositions

In games such as Hex, Go, and Amazons, groups eventually divide the board so that the search space decomposes combinatorially. An example in Hex is when a group of $P$-stones touches both of $Q$’s sides. This particular decomposition is rare. We have identified a more general form of Hex decomposition based on the graph theoretic properties of an opposite-color bridge, namely a bridge between opposite-color stones. See Fig. 7.

**Lemma 1.** For a Hex state with an opposite-color bridge, deleting the direct adjacency between the two empty cells does not change the game theoretic value of the state.

**Proof.** Deleting an adjacency can destroy (but not create) a winning path. This does not change the game’s outcome: if player $P$ has a winning path that uses the deleted edge, the path can be rerouted through the bridge’s $P$-colored endpoint, since it is a common neighbor to both cells.

Two opposite-color stones crowd if they touch or form a bridge; a $P$-group crowds a $Q$-group if some $P$-stone in the former crowds some $Q$-stone in the latter. See Fig. 8.

Figure 7: A bridge between opposite-color stones.

**Theorem 1.** Assume a Hex state has a $P$-group $G$ that crowds both $Q$ sides. Then $P$ wins if and only if $P$ wins the two subgames obtained by splitting the board along $G$ and any associated opposite-color bridges.

**Proof.** (sketch) Apply Lemma 1: for each opposite-color bridge from $G$ to either of $Q$’s sides, delete the adjacency. Now $G$ separates these two halves of the board, so any winning path for $P$ goes through $G$.

Figure 8: The Black group crowds both White sides (middle), so by Thm. 1 Black wins this game if and only if Black wins both subgames (bottom).

In Hex, it can be advantageous to probe a virtual connection, even if the connection is maintained. As noted in Theorem 1, a probe into one subgame of a decomposition has no effect on the other subgame; it matters only whether there is a strategy to connect the region’s boundaries. This theorem can be extended as follows:

**Theorem 2.** Assume a Hex state has a 4-cycle of alternating $P$- and $Q$-groups, such that consecutive groups crowd and that there is a virtual connection between the two $P$-groups whose carrier is a subset of the 4-cycle’s interior region. Then the winner of the state is unchanged by filling the interior region with $P$-stones.

Cells that are captured by such 4-cycle decompositions are filled in, and thus pruned from consideration. See Fig. 9.

Figure 9: A 4-cycle of alternating consecutively crowding groups with an interior Black virtual connection (left). By Thm. 2, Black captures the region (right).

5 Proof sets

In Hex, one can often use a discovered winning strategy against multiple opponent moves. A proof set is the set of empty cells used in a state’s winning strategy; such sets can be efficiently computed recursively. Proof set pruning is the pruning of any opponent moves that do not intersect the discovered proof set. This is just the special case of mustplay pruning once a winning strategy has been found.

Since smaller proof sets yield more pruning, we developed an efficient algorithm to deduce the existence of smaller proof sets (corresponding to winning strategies that use fewer cells). This proof set reduction algorithm is as follows: given a proof set $S$ for a player $P$, assign all empty cells outside $S$ to $Q$, and then use inferior cell analysis to compute dead
and \(Q\)-captured cells; assign the remaining empty cells to the reduced proof set \(S\). See Fig. 10.

![Figure 10: A White proof set (left) and its proof set reduction.](image)

**Theorem 3.** The proof set reduction algorithm produces a valid proof set.

**Proof.** (sketch) Assigning empty cells outside proof set \(S\) to \(Q\) does not affect \(P\)'s winning strategy on \(S\), so the first step does not change the state’s game theoretic value. Also, inferior cell theory guarantees that filling-in dead and captured cells does not change the value. Thus \(P\) still has a winning strategy on the reduced proof set \(S' \subseteq S\).

6 Transposition deductions

To avoid re-solving equivalent states, it is common practice to store solved state values. We extend this practice by storing not only the current state, but also any additional states whose value can be deduced from its result. We perform these deductions only for states with at most \(k\) stones; these values are permanently stored in a database. For states with more than \(k\) stones, we store values in a separate transposition table with \(2^{20}\) entries; hash collisions overwrite older entries.

In addition to 180 degree board rotation, two new types of transposition deduction are used in our solver: proof set transpositions and player exchange transpositions. Proof set transpositions use the fact that any cells outside the proof set can be assigned to the losing player without affecting the result, as mentioned in §5. Thus, if a proof set omits \(i\) empty cells in a state with \(j\) stones for the losing player, then we can identify \((i+j)\) states with the same game theoretic value, and record all of them in the database. Note that proof set reduction also increases the number of proof set transpositions.

Player exchange transpositions are more complex, and involve transforming a state by exchanging the roles of players \(P\) and \(Q\). If stone positions are mirrored (in either diagonal), and stone colors and the player to move are changed, then the new state’s winner is the opponent of the original state’s winner. However, the method as currently described is useless, since it produces only unreachable states due to constraints on the player to move and/or the number of stones present for each player. In order to produce reachable transposition deductions, after transforming the board we must either add a single first player stone or else remove a single second player stone. So as not to change the state’s value, we can only add winner stones, remove loser stones, or add loser stones outside the winner’s proof set. See Fig. 11.

These deductions often compute thousands of database entries from one solved state, but the extra computation time prevents this method from being feasible at all stages of solver’s search. Thus we limit the deductions to states with at most \(k\) stones, using \(k=4\) for \(7 \times 7\) and \(k=5\) for \(8 \times 8\).

7 Solving algorithm

Our solving algorithm combines the aforementioned methods as follows. Explore the Hex game tree in depth first order. If a state’s value is in the database/transposition table, return the value. Otherwise, use ICE fill-in and 4-cycle decompositions to compute an equivalent board state, and then use VCE to compute virtual/semi connections on this new state. If ICE (with a winning path) or VCE (with a winning virtual connection) solves the new state, then reduce the proof set, store the value (and any deduced values) in the database/transposition table, and return to the parent state. Otherwise, order all moves to consider – empty non-inferior cells in the mustplay – by mustplay size and electric circuit model evaluation. If the board decomposes into two independent parts, then split the search. If a child state’s value resolves that of its parent, then store the value and return; otherwise use the proof set to prune additional moves and proceed to the next child.

8 Solving 8x8

We ran our algorithm on an Intel Core 2 Quad Q9559 LGA775 (2.66GHz/1333FSB/12MB) desktop with 2GB RAM. Solving all \(8 \times 8\) states took 106,448,370 interior nodes and just under 301 hours. See Fig. 12.

In order to exploit the database, openings were solved in the hand-picked order e4,d4,c5,f3, e3,d3,c4,c3, g2,f2,e2,d2, c2,b3,b4,b5, b6,h1,a3,a4, a5,a6,a7, followed by the remaining capture-dominated and rotated openings. Fig. 13 shows the number of interior nodes in the search trees. Zero entries indicate instant solution by database lookup.

![Figure 12: 8\times8 opening moves solved by our algorithm.](image)
large gap between the improvement ratios for time and space indicates the significant time cost of this feature. The closeness of the time and space ratios for transposition deduction suggests that our selected maximum application depth was shallow enough to avoid a combinatorial explosion in the number of deduced entries. While proof set reduction had a slightly negative effect, we suspect that the scalability of this method (computational effort versus potential savings) makes it useful for larger board sizes or deeper database depths; we have not yet run tests to confirm this.

9 Feature contributions

To measure the contributions of the various features of our solver, we solved all 7×7 openings repeatedly, each time with a different feature turned off/on. The base run with all features on took just over 10 minutes with a search tree containing 5.5e7 internal nodes; the base run with all features off took just over 15 hours with a search tree containing 5.5e7 internal nodes. Fig. 14 summarizes the effects on both solving time and the number of internal nodes in the search tree, showing the ratio of each adjusted run with one feature off/on versus the base run with all features on/off.

For most of our features, computation time is polynomial in the board size, whereas search reduction seems exponential. The only exceptions are virtual connections (which are central to our algorithm and so never turned off) and transposition deductions, which take exponential time. For both these features we limit the computation process, in the former case by restricting the generality of the combining rules, and in the latter case by applying the feature only at shallow tree depth. We suspect that this polynomial computability is the key to our success in solving 8×8 states. By contrast, the template-matching techniques of Rasmussen et al. [2007] apparently require exponential time; this may be why their solver, which performs comparably to our solver on 7×7 states, cannot solve any 8×8 opening states.

In terms of both time and search space savings, our most important feature is mustplay move ordering. However, the large gap between the improvement ratios for time and space indicates the significant time cost of this feature. The closeness of the time and space ratios for transposition deduction suggests that our selected maximum application depth was shallow enough to avoid a combinatorial explosion in the number of deduced entries. While proof set reduction had a slightly negative effect, we suspect that the scalability of this method (computational effort versus potential savings) makes it useful for larger board sizes or deeper database depths; we have not yet run tests to confirm this.

The benefits of many features are interdependent: the time/space ratios when a single feature is off are not equal to the inverse of the time/space ratios when a single feature is on. An example of this is the improved ICE, which worsens performance when used alone, but improves performance when combined with the other features.

10 Verification

We have no independent method to verify the correctness of our algorithm; however, we have executed our algorithm in different environments without detecting any errors. Our algorithm correctly solves all openings for all regular and irregular (e.g. 7×8) boards smaller than 8×8, and confirms principal variations for the two Mishima et al. [2006] 8×8 solutions. We embedded our solver in automated Hex players, and observed that an augmented player always weakly-dominates its unaugmented version. We also used our solver for post-game analysis and endgame problem generation. Over the
course of several months and several hundred games, no
error has been detected in any of these uses.

Fig. 15 shows the (abbreviated) principal variation for each
opening move’s search, namely the first win found where the
losing player prolongs as much as possible. The short prin-
cipal variations for openings c5,e3,g2 are due to database hits.
The $8 \times 8$ database, which stores all solved states with at most
five stones, is available for download.

11 Conclusion

We have developed several efficient features that preserve the
theoretical value of a Hex state while reducing the search
space. In the future we hope to solve some $9 \times 9$ openings,
thereby surpassing human-designed solving efforts (which
work particularly well on near-centre openings). Solving $8 \times 8$
Hex openings took about 1300 times as many nodes as solv-
ing $7 \times 7$ Hex openings, even though (as suggested by Fig. 1)
the $8 \times 8$ search space is about a million times as big as the
$7 \times 7$ search space. After two weeks of computation time, our
solver made no significant progress on either of e5 or h2, the
two presumably easiest $9 \times 9$ openings.

References

[Anshelevich, 2000] Vadim Anshelevich. A hierarchical ap-
proach to computer Hex. Artificial Intelligence, 134:101–
120, 2000.

[Björnsson et al., 2007] Yngvi Björnsson, Ryan Hayward,
Michael Johanson, and Jack van Rijswijck. Dead Cell
Analysis in Hex and the Shannon Game. In Adrian Bondy,
Jean Fonlupt, Jean-Luc Fouquet, Jean-Claude Fournier,
and Jorge L. Ramírez Alfonsín, editors, Graph Theory in
Paris: Proceedings of a Conference in Memory of Claude

[Browne, 2000] Cameron Browne. Hex Strategy: Mak-
ing the Right Connections. A. K. Peters, Natick, Mas-

[Hayward et al., 2005] Ryan B. Hayward, Yngvi Björnsson,
Michael Johanson, Morgan Kan, Nathan Po, and Jack van
Rijswijck. Solving $7 \times 7$ Hex with domination, fill-in,
and virtual connections. Theoretical Computer Science,

[Hein, 1942] Piet Hein. Vil de laere Polygon? Article in
Politiken newspaper, 26 December 1942.

[Henderson and Hayward, 2008] Philip Henderson and
Ryan B. Hayward. Probing the 4-3-2 edge template in
Hex. In H. Jaap van den Herik, Xinhe Xu, Zongmin
Ma, and Mark H. M. Winands, editors, Computers and
Games (6th International Conference, CG 2008, Beijing,
China, September 29 - October 1, 2008. Proceedings),
volume 5131 of Lecture Notes in Computer Science, pages

www.drking.plus.com/hexagons/hex/templates.html,


[Mishima et al., 2006] Ken Mishima, Hidetoshi Sakurai, and
Kohei Noshita. New proof techniques and their applica-
tions to winning strategies in Hex. Proceedings of 11th
Game Programming Workshop in Japan, pages 136–142,
2006.

[Nash, 1952] John Nash. Some Games and Machines for
1952.

[Noshita, 2005] Kohei Noshita. Union-Connections and
Straightforward Winning Strategies in Hex. International
2005.

[Rasmussen et al., 2007] Rune K. Rasmussen, Frederic D.
Maire, and Ross F. Hayward. A template matching ta-
ble for speeding-up game-tree searches for Hex. In AI
2007: Advances in Artificial Intelligence, Lecture
Notes in Computer Science, pages 283–292. Springer,

[Rasmussen, 2008] Rune Rasmussen. Algorithmic Ap-
proaches for Playing and Solving Shannon Games. PhD
thesis, Queensland University of Technology, Brisbane,
Australia, 2008.

[Reisch, 1981] Stefan Reisch. Hex ist PSPACE-vollstän-

Uiterwijk, and J. van Rijswijck. Games solved: Now and
in the future. Artificial Intelligence, 134(1-2):277–311,
2002.

[van Rijswijck, 2003] Jack van Rijswijck. Search and eval-
uation in Hex. Technical report, University of Alberta,


www.ee.umanitoba.ca/˜jingyang.