**Hex Optimizations**

Anthony Weng, Hunter Guru

**I. Introduction**

Independently devised by mathematicians John Nash and Piet Hein, Hex is a two-player, position-based game. Two players take turns placing hexagonal tiles on a *n x n* board (commonly *n* = 9). Both players aim to place their tiles such that there is an uninterrupted sequence of their tiles from one side of the board to the other.



*Figure 1.* An 11 *x* 11 Hex board in which Blue has won

Though the game follows simple rules, devising optimal strategies for Hex gameplay is exceptionally difficult due to the sheer number of legal states of the game (approximately $2.4\* 10^{56}$ for an 11 *x* 11 board). In this projects’ context of game-playing engines (that construct optimal strategies by analyzing which future move(s) most probably lead to a victory), Hex presents a common computational challenge: how can one most efficiently detect a “winning” or “losing” position given the extreme number of legal game states? Using logic programming, we seek to reformulate and encode the rules of Hex such that a maximal efficiency is achieved. More formally, this project utilizes logic programming to investigate the following questions:

**(1)** *What are the different paradigms in which we can encode the rules of Hex (to detect the lack/presence of a winning position)?*

**(2)** *In terms of algorithm efficiency, how does each paradigm perform (relatively)?*

**Note:** Following discussions on optimal rule reformulations assume a 9 *x* 9 Hex Board but may be extrapolated to any dimension. The same is true regarding project inputs (example logic programs of rule reformulations) and outputs (algorithm analysis and runtimes).

**II. Board State Representation**

As previously defined, the winning condition for Hex

is achieved when one player connects a sequence of tiles that spans the entirety of the board’s width (or height). As any winning sequence of tiles would be composed of multiple pairs of adjacent cells, evaluating cell adjacency is a necessary condition for detecting a winning state.

Commonly, cells of the Hex board are indexed via their column number (‘1’ – ‘9’) and their row letter (‘A’ – ‘I’); however, defining cell adjacency rules in such a manner usually requires a two-step check (verifying that two cells exist in adjacent columns, and then in adjacent rows). As an alternative, we can assign each cell of the board a numerical index as follows:



*Figure 2.* (Partial) numerical indexing of 9 *x* 9 Hex Board

By indexing the cells in this way, we note the following:

**(1)** *Adjacency rules can be codified arithmetically[[1]](#footnote-1):*



*Figure 3.* Two cells (11, 38) and all their adjacent cells

E.g. for cell 11 (denoted cell *c*), adjacent cells are those indexed at 2, 3, 10, 12, 19, and 20. For cell 38, adjacent cells are those indexed at 29, 30, 37, 39, 46, and 47. Using the proposed numbering scheme, adjacent cells can be defined for**both**cells as those cells at indexes *c –* 8, *c –* 9, *c* – 1, *c* + 1, *c* + 8, and *c* + 9[[2]](#footnote-2).

**(2)** The column any given tile is in can be easily computed via modulo 9 conversion (e.g. $10 mod 9=column 1, 67 mod 9=column 4)$

**III. Rule Paradigm Reformulation**

*i. Naïve Sequence Generation*

Though logic programming can be used to both define and then verify the existence of a winning game state, abstracting the definition process away from logic programming could greatly speed in-game runtime. Essentially, another programming language (e.g. Python) can be used to generate “winning sets[[3]](#footnote-3)” while logic programming is used as a “verifier.” That is, after each move, the logic program could simply check if the cells controlled by a given player is a superset of a winning set.

Under this paradigm, in-game win verification runs in—at worst—linear time and often faster than that. However, an enormous amount of time is necessary for the external program to compute all of these winning sets. Moreover, this pre-game computation time scales incredibly poorly as the size of Hex board increases. For these reasons, this approach is best reserved for very small Hex boards.

*ii. Power-set Constraints*

Rather than generating all possible winning sets prior to gameplay, we can consider using logic programming to both generate and verify possible winning sets. Noting the condition that a player must control at least one cell in each column (or analogously, row) in order to have won, we devise the following approach:



*Figure 4.* Schematic of power-set constraint conceptualization

**(1)** *Maintain a collection of all cells controlled by a player.*

**(2)** *After each turn, generate all 9-length subsets of all cells controlled by a player such that each element exists in a unique column.*

**(3)** *For each subset, check if each element (omitting the first) obeys a numerical adjacency rule with regards to the prior element.*

As compared to naïve sequence generation, this paradigm is more efficient as it only computes and verifies the possible winning sequences given a certain board state (instead of all winning sequences). However, this paradigm fails to detect any winning subset that consists of more than 9 elements. The immediately apparent solution is to remove the 9-length restriction and to simply generate and verify all 9 to *m*-length subsets (where *m* is the number of cells a given player controls). But, once the number of cells in a subset expands beyond 9, the restriction of one element corresponding to each column becomes invalidated as well. With these edge cases and contradictions arising when we try to expand the base paradigm of power-set constraints, its value remains mostly as a step toward a more efficient paradigm.

*iii. Stepwise Search (“Following the Line”)*

Considering the principal failure of the power-set constraints paradigm was its inability to consider winning sets consist g of more than 9 cells, we wish to devise an alternative paradigm that is restricted in its search for a winning sequence only by the cells a player controls, and not an arbitrary length. Such an approach could be of the following form:



*Figure 5.* Schematic of stepwise search conceptualization

**(1)** *For a given player, consider each cell that they control in column 1.*

**(2)** *Using the adjacency rules, compute the indices of all cells adjacent to this “origin” cell.*

**(3)** *If the player controls any of these adjacent cells, repeat the adjacency check with the previously adjacent cell serving as the new origin cell.*

Ultimately, if the logic program can follow a path of adjacent cells from column 1 of the board to column 9, a win will have been detected. Preventing cycles (i.e. traversing back to a previous cell) in the sequential adjacency checks is possible as well via maintaining a list of previously “visited” cells and stipulating new cells cannot be in this prior list.

With cycle prevention, this paradigm presents itself as a universal (i.e. not restricted by board dimension) rule reformulation; however, it still suffers from the repeated computation of non-winning sequences. Our following reformulation addresses this issue.

*iv. Minimum Spanning Trees*

In this final reformulation, we wish to address the issue of repeated computation of non-winning sequences present in the previous paradigms. Ideally, we would like to maintain the partial sequences that a player controls between turns. Moreover, we would like adjacent cells to belong to the same partial sequence. If two of a player’s partial sequences are then connected by a new cell, we merge these sequences into a new, single sequence. Ultimately, if one of a player’s partial sequences contains a cell in both columns (or rows) 1 and 9, they have won.

The approach discussed above is a form of determining if a minimum spanning tree exists between the two sides of the board (analogous to graph vertices) given a player’s cells (analogous to graph edges). In terms of both universality and efficient computation, this paradigm is the best. However, it is not perfect—maintaining all of both player’s trees is difficult as the board size scales. Further reformulations should seek to address this issue.

**V. Discussion: The Value of Logic Programming**

Through this process of continuously reformulating the rules for Hex, both the universality and extrapolation of our encoding of Hex rules has progressed considerably. Further optimization may be pursued through general optimization techniques (grounding, sub-goal reordering, etc.) or more abstract reformulation (Hex as a Maker-Breaker game). Crucially though, logic programming was essential and will continue to be in this process. By design, logic programming excels at the condition testing present in this project (“Given a set of cells, has a player won?”). However, utilizing logic programming allows us to easily extend the ideas developed in this project to the design of AI game players and strategy formulation for non-identical but similar games. Ultimately, logic programming was not necessary to our project, but its flexibility in design and implementation made it an invaluable asset in this effort.

1. One set of adjacency rules? Like a bee’s honey, that’s pretty sweet! [↑](#footnote-ref-1)
2. For cells where certain adjacent cells do not exist (e.g. cell 1 does not have adjacent cell at index *n* – 8), we can simply stipulate that the index of the adjacent cell must be bounded between 1 and $n^{2}$. [↑](#footnote-ref-2)
3. A winning set is defined as a sequence of cells controlled by one player such that said player has won. Such winning sets are *minimal* – i.e. they possess no extraneous cells that are non-essential for connecting one side of the board to the other. [↑](#footnote-ref-3)