Research Findings - CS 151 Final Project

Optimizing Hex: The Relative Efficiency of Various Implementation Paradigms in Logic Programming

Hunter Guru*, Anthony Weng*

Stanford University

**Both authors contributed equally to the writing and research in this study. Their names are listed in alphabetical order.*

June 10, 2019

Introduction

0.1 Hex

Hex is a two-player game, independently developed by Mathematicians John Nash and Piet Hein. Traditionally played on an 11x11 rhombus, in Hex, both players take turns placing pieces on open spaces, in order to create a chain connecting their sides of the board.

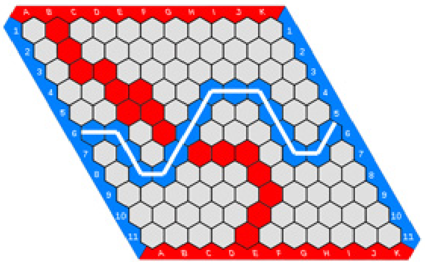

Figure 1: 11x11 Hex Board in which Blue has won. *Image by Jean-Luc W [CC BY-SA 3.0]*

Developing an optimal strategy for Hex is exceptionally difficult due to the volume of the search space of the game; utilizing statistical branching methods, it is estimated that the game contains approximately 2.4×10^{56} valid states [1]. Although Nash proved that the first player always has a winning strategy, no such strategy has been found on boards larger than 7x7 [2].

0.2 Logic Programming & GDL

General Game Playing (*GGP*) is a sub-field of artificial intelligence, in which intelligent systems must legally play and handle various game types [3]. In order to interpret the rules of various legal games through a robust framework, many general game playing systems operate by accepting legally encoded descriptions of arbitrary finite synchronous games via Game Description Language (*GDL*) [4].

Logic Programming is a practical paradigm for this class of problems, as it allows for the inquiry of dynamic states in a finite information ontology. This research paper explores various paradigmatic implementations of Hex in Epilog, a variant of Dynamic Logic Programming; the rules and methodologies explored in this paper, can easily be translated to accommodate GDL [5].

1 Board State Representation

1.1 Cell Representation

As previously mentioned, a game of Hex concludes when a player connects a sequence of tiles that span the entirety of the width of the board (or height). In order to tell when cells are adjacent to one another, it is useful to define a relation that expresses adjacency conditions. Commonly, cells of the Hex board are indexed via their column number $\{1, 2, ..., n\}$ and their row letter $\{A_1, A_2, ..., A_n\}$; however, defining cell adjacency rules with such a scheme 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 via modular numbering, labeling each hexagon with an integer which wraps around from row to row.

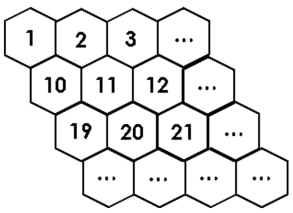

Figure 2: A 9x9 Hex Board, representing modular numbering

Observation: Let c be an arbitrary cell on an $n \times n$ Hex board. Since c is a regular hexagon, we can conclude that it has precisely three pairs of parallel edges. If we draw perpendicular lines between each pair of parallel edges, we notice that these lines each have an absolute slope of $|n-1|, |n|, |1|$. Since adjacency can be traversed in either direction on these "lines", we notice that the six adjacent cells are given by $c \pm |n-1|, c \pm |n|, c \pm |1|$. However, if c is a cell in the first column ($1 \bmod n = c \bmod n$), then certain rules do not apply, e.g. $|n-1|$; this truth is also extended to when c is in the first/last column and row. Which can be accomplished by a rule such as the following:

$$isAdjacent(X, Y) :- cell(X), cell(Y), Y \text{ is } X + 1, \sim col_n(X)$$

2 Rule Paradigm Reformulation

2.1 Naive Sequence Generation

Although logic programming may be utilized to both define and verify the existence of a winning game state, abstracting the definition process away from logic programming could greatly speed in-game run time. Essentially, another easy-to-use programming language (e.g. Python), can be used to generate a set of winning sets, $\mathcal{W}$, delegating the task of verifying to the logic program. Let $\mathcal{M}_p$, be the set of all moves played by p. The following can be utilized to determine if p has won:

$$\exists w_m \in \mathcal{W} : w_m \subseteq \mathcal{M}_p \implies hasWon(p)$$

In such paradigm, the Epilog compiler is extremely fast at determining whether the player has won, as it simply makes efficient queries in a grounded data-set, while abstracting away the complexity of the search to the programmer.

Unfortunately, copious time is taken for the external program to compute $\mathcal{W}$, and does not scale well as the dimension, n, increases; it also leaves a heavy memory footprint, breaking general game players due to the massive size of the "pre-grounded" library.

2.2 Power-set Constraints

Instead of generating *all* winning sets prior to game play, we can consider using logic programming to also analyze whether a sequence of pieces form a chain across the board. Let $c_1, c_2, ..., c_n$ be the columns of an $n \times n$ hex board. Notice that in order to win Hex, there must exist at least one played piece $pp \in c_1, c_2, ..., c_n$. As such, we can narrow our search space to:

$$\mathcal{S} = \{x \in \mathscr{P}(\mathcal{M}_p) \mid |x| = n\}$$

Then we repeat the following algorithm:

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.
4. End after finding the existence of just one chain (which may be implemented via Prolog's cut feature).

This is logically equivalent to:

$$\exists x \in \mathcal{S} : isAdjacent(x_1, x_2) \ \wedge \ ... \ \wedge \ isAdjacent(x_{n-1}, x_n)$$

Such a process will find all winning chains of length n: therefore, it doesn't detect longer chains which may include loops, or going backwards. We must also search over all sets sized i, where $n \leq i \leq \frac{n^2}{2}$, as it is possible to have a winning string up to half of the board.

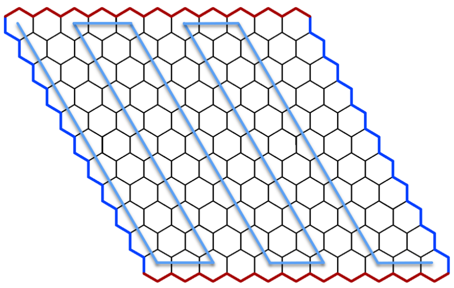

Figure 3: An example of a long chain formable on a Hex Board

As compared to naïve sequence generation, this paradigm is more efficient early on, e.g. if a player builds a winning chain that is shorter; this is because it only computes and verifies the possible winning sequences given a certain board state with a finite bound (instead of all winning sequences). It also eases the implementation of cycles, since we essentially fix the length of the subset to a finite amount. Additionally, the program can materialize the set of all tested sets and avoid testing them again. For example, assume p played cell c in column i. Rather than computing all sequences with c added in the database, we can simply iterate over all materialized datum, and replace the i^{th} column with the new cell, and then apply adjacency analysis.

Unfortunately, once the number of cells in a chain expands beyond 9, the restriction of one element corresponding to each column becomes less valuable; you end up brute-force checking all powersets of size i, such that there exists at least one piece in each column.

2.3 Follow the Line

Considering the principal failure of the power-set constraints paradigm was its inability to efficiently trace winning sets 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:

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. Unfortunately, while caching all sequences, the program develops an exponentially large memory footprint, and becomes extremely slow. It essentially checks every path using breadth first search, which becomes very expensive as the number of tiles on the board increases; this is essentially an expected issue when iteratively verifying paths in a PSPACE-complete problem [6]. It becomes evident that the time required to find a sequence essentially converges to the extended implementation of power-set constraints, and the preference between *2.2* and *2.3* becomes one of personal implementation preference, rather than performance, although *2.2* avoids the issue of cycles in a much more elegant way.

2.4 Minimal Spanning Trees

In this final reformulation, we wish to address the issue of repeated computation of non-winning sequences present in the previous paradigms. Notice that all "groups" of adjacent cells, which we classify *clusters*, are disjoint (these clusters form equivalence classes). As a result, all clusters adjacent to a played cell c merge into a larger set. Let $c_1, c_2, ..., c_n$ represent the clusters of cells which are adjacent to a piece k, played by p. We can form a larger cluster $c_s = \bigcup_{i=1}^{n} c_n$, which encompasses all the smaller clusters.

Throughout the latter paradigms, we did not opt to utilize our own data structures to alter the data-set from turn to turn; ideally, we would like to update a data structure throughout the course of the game, that allows us to calculate adjacency more efficiently, rather than expensively calling a verifier to conduct analysis from the ground up, *every* turn.

In our programming paradigm, a player can mark a cell by using the predicate *cell(X, ROLE)*, where *X* is the valid numerical ID of a cell, and *ROLE* is one of the roles in the game (either blue or red). We then divide the event space into two cases:

1. A player plays a piece adjacent to ≥ 1 clusters.
2. A player plays a piece adjacent to 0 clusters.

In the case of (1), we need to merge all the adjacent clusters together, and add the cell to the cluster as well. In (2), we simply need to create a single cluster, containing the cell *ROLE* played.

We can begin implementing this by defining a custom predicate:

$$cluster(id, cell, role).$$

A cluster contains a unique ID (arbitrarily determined by the logic program to avoid redundancies), a single cell that is an element of the cluster, and the role to which the cluster belongs. For example, if p has a cluster with id 4, containing cells $1, 3, 5$, the following facts would appear true in the cluster view:

- $cluster(4, 1, p)$
- $cluster(4, 3, p)$
- $cluster(4, 5, p)$

Now, after defining the sufficient data structures to represent each cluster, we begin to define transition rules for each disjoint situation.

When a cell has ≥ 1 adjacent cluster:

$$cell(X, C), isAdjacentCluster(X, cluster(A, B, C)), cluster(A, D, C) \text{ ==>\textasciitilde } cluster(A, D, C)$$

$$cell(X, C), isAdjacentCluster(X, cluster(A, B, C)), cluster(A, D, C) \text{ ==> } cluster(X, D, C)$$

When a cell has ≥ 0 adjacent clusters:

$$cell(X, C) \text{ ==> } cluster(X, X, C)$$

These transition rules are applied every time p marks a new cell, and are used in conjunction with the following verifier.

$$hasWon(p) :- cluster(X, C, p), column1(C), cluster(X, D, p), columnN(D)$$

This verifier works almost instantly, and when performing general efficiency techniques (described in the next section), they perform almost instantaneously. 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.

3 Hex as a Maker Breaker Game

Hex is defined as a Maker-Breaker game. Maker-Breaker games are positional games with winning sets $\mathcal{W}$. In Maker-Breaker games, the Maker wins if they are able to hold the elements of a winning set, i.e. $\exists w_s \in \mathcal{W} : w_s \subseteq \mathcal{M}_p$, whereas the breaker wins if they manage to prevent this from happening. In these games, draws are not possible [7].

Hex was proven to be a Maker-Breaker game by John Nash as, if there exists no possible winning chain for p, player q must have already connected a chain, and won the game. In other words, rather than than both players aiming to be the first to draw a chain across the rhombus, Hex can be thought of as a game in which the second player defends against the first player, and wins if they hold at least one element in every reachable winning-set.

Formally, the following logic holds when there exist players p, q such that $p \neq q$:

$$hasWon(p) \iff hasLost(q)$$

$$\forall w_s \in \mathcal{W}.\ w_s \cap \mathcal{M}_p \neq \emptyset \implies hasWon(p)$$

The natural question then becomes, "How can we use these facts to aid our logic programming implementation?". Rather than sequentially checking to see if player p or q have won, the properties of Maker-Breaker games allow us to instead verify if player p has won or lost. It turns out this property can help make our game even more efficient when we choose to utilize Minimal Spanning Trees.

For the sake of simplicity assume that in every step s, player p and q play pieces on the Hex board such that they do not play on the same cell. Let i be the number of steps that have taken place throughout the game, so $0 \leq i \leq \frac{n^2}{2}$ (Note: in a two player game of hex on an $n \times n$ board, there are n^2 cells).

Notice, If we let p_c, q_c be the sets of cells played by p and q, since the board can be broken in terms of equivalence classes, with p, q and blank pieces, the blank pieces are simply the set difference of the entire board; $\therefore\ b_c = validcell \setminus (p_c \cap q_c)$ where b_c is the set of all blank cells. We know that $|b_c| = n^2 - 2i$ because after i steps, the rest of the board is composed of blank cells. When $|b_c| \leq |q_c|$, it no longer makes sense for us to carry a game tree for q_c because the memory footprint of blank cells becomes increasingly smaller in comparison, and we can use the blank cells to conclude the same amount of information. That is, we can tweak our verifier to consider blank pieces as our own "temporary clusters"; as such, if a win "exists" for p when considering blank pieces as p's pieces, then it is still possible for player p to win. By extension, since Hex is a Maker-Breaker game, we can equivalently conclude that p has not lost.

The relative performance can be mapped by the following equation, with "steps" as the domain.

$$f(i) = \frac{n^2 - 2i}{i} = \frac{n^2}{i} - 2 \tag{1}$$

Notice that as i increases, the search space becomes significantly smaller, given by a factor of $f(i)$; this matches our intuition, and as the game goes on, our search space can represented as a concave down quadratic function.

4 General Efficiency Techniques

As both an alternative and supplement to reformulating the rules of Hex, a set of logic programming-specific general optimization techniques (GOTs) can be utilized to optimize runtime costs. Like any optimization, these techniques aim to reduce unnecessary computations. However, unlike rule reformulations which alter both the paradigm and all resulting computations of a program, GOTs focus on removing computations in a specific goal or task of an already-established paradigm. Logic programming GOTs can be characterized as one of three classes:

1. **Grounding:** Grounding is the explicit computation of facts that are true given a ruleset and a dataset. These computed facts are then stored in the dataset, and can be accessed in the same time as any other fact in the original dataset. Though the technique does require an upfront computational cost, grounding can save considerably more time than it costs by not repeatedly computing common facts.

2. **Sub-goal reordering:** Sub-goal ordering is simply the reordering of sub-goals within a rule to optimize worst case performance. For instance, the relation "s(X,Y) :- p(X) & r(X,Y) & q(X)" may be more efficiently be written as "s(X,Y) :- p(X) & q(X) & r(X,Y)" as the former case has a worst cost of evaluating proportional to n^4 while the latter's cost is proportional to only n^3.

3. **Sub-goal pruning:** Sub-goal pruning involves removing redundant sub-goals within rules. Consider the relation "r(X,Y) :- p(X,Y) & q(Y) & q(Z)." We note that the sub-goal q(Z) is redundant as if there is some Y such that q(Y) is true, that q(Z) is guaranteed to be true as well. Thus, a more efficient version of the rule can be written as "r(X,Y) :- p(X,Y) & q(Y)," omitting q(Z).

Within the context of Hex reformulations, these GOTs can be applied such that reformulations with poor efficiency in one respect can effectively be "patched up" to achieve similar levels of efficiency to other reformulations. Alternatively, these GOTs can also be used to further improve the efficiencies of more optimal rule reformulations. To investigate the runtime gains of these GOTs in both of the previously described applications, we apply every combination of grounding and sub-goal reordering (i.e. none, only grounding, only sub-goal reordering, or both) to both the 9-length power-set and minimum spanning tree rule reformulations in a devised "difficult" context.

Specifically, we define the "difficult" context to be a set of 17 cells on a 9×9 Hex board where there is no winning path and use the "grindem" query available in EpilogJS and use the output time of the query as a measure of algorithmic runtime.

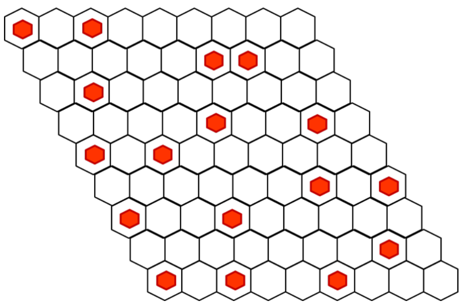

Figure 4: A 9×9 Hex Board with 17 cells and no winning path.

The specific application of GOTS to each reformulation follows:

Application of GOTS in Power-set reformulation:

1. **Grounding:** Grounding was applied to the ruleset which computed the column each cell resided in. This column-index ruleset was an apt ruleset to ground as part of the verification process for a "winning" power-set is checking that each cell is located in a different column. The post-grounding dataset includes facts that encode what column each cell at a given index is located in.

2. **Sub-goal reordering:** Sub-goal reordering was applied to the "potential winning sets" rule (i.e. the rule which generated those sets of 9 cells where each cell resides in a unique column). Without sub-goal reordering, the rule first selected any arbitrary set of 9 cells among the 17 the example player controlled and then the verification of the unique column restriction took place. Sub-goal reordering was applied by performing the selection of the 9 cells and verification of the unique column restriction in a step-wise manner; that is, the first cell was selected and verified to exist in the first column before the second cell was selected, the second cell was selected and verified to exist in the second column before the third cell was selected, etc.

Application of GOTS in Minimum Spanning Tree reformulation:

1. **Grounding:** (Same application as in the power-set rule reformulation).

2. **Sub-goal reordering:** Sub-goal reordering was applied to the "isAdjacentCell" rule. Without sub-goal reordering, the rule first found two adjacent cell indices (via the numeric adjacency rules) and then checked if they were valid cell indices (i.e. within the cell indices bounded by 1 and n^2 where n is the dimension of the Hex Board). With sub-goal reordering, the rule first finds two adjacent cell indices and then checks if they are adjacent.

To analyze the efficiecncy gains, we define the "Runtime Savings Factor" (RSF) to be the following proportion:

$$RSF = \frac{\text{Maximum time required}}{\text{Time with GOT}}$$

After applying the appropriate combinations of GOTs to each reformulation, the following runtimes were observed (a maximum tolerable runtime was set at 15 minutes = 900,000 milliseconds):

GOTs Applied	**Post-GOTs Runtime (ms)**	**Runtime Savings Factor (RSF)**
None	*900,000 (Does not run)*	*1*
Grounding only	*900,000 (Does not run)*	*1*
Sub-goal reordering only	*368*	*~2500*
Both	*51*	*~20,000*

Table 1: GOTs Analysis for Power-set Constraint Reformulation

GOTs Applied	**Post-GOTs Runtime (ms)**	**Runtime Savings Factor (RSF)**
None	*41*	*1*
Grounding only	*6*	*~10*
Sub-goal reordering only	*5*	*~10*
Both	*1*	*~50*

Table 2: GOTs Analysis for Minimal Spanning Tree Reformulation

From these query tests, we see that appropriate application of GOTs can significantly improve the performance of runtime-poor reformulations to a level where runtimes are comparable to better reformulations (e.g. 51 ms for the power-set reformulation with both GOTs applied $\approx$ 41 ms of the minimum spanning tree reformulation with no GOTs). Moreover, GOTs can be necessary to algorithms even compiling (e.g. power-set reformulation not running without appropriate sub-goal reordering) and can still improve the performance of optimized reformulations. Ultimately, the universal applicability of GOTs makes them an invaluable tool for Hex reformulation and logic programming as a whole.

Conclusions

4.1 Application to Other Connectivity Games

In general, if the board of a game can be broken into disjoint sets and equivalence classes, our formalization of Minimal Spanning Trees can largely be carried over. For example, in the game of three dimensional tic-tac-toe, which is another connectivity game in which equivalence classes are formed between Player X, Player O, and the blank cells, the same methodology proposed in this paper can be applied, with slight adjustments regarding explicit win conditions. Additionally, this paper also outlines generalized optimization techniques, which may be utilized by General Game Players in order to build a faster library from which they can query.

4.2 The Paradigm of Logic Programming Beyond Relation to Hex

Through this process of continuously reformulating the rules for Hex, both the universality and extrapolation of our encoding of Hex rules has progressed considerably. While further optimization may be pursued through general optimization techniques or more abstract reformulations (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. In the future, we hope to test our implementations on general game players to see how significant of an impact the library of a game can have on the performance of a player.

References

[1] C. Browne, “Hex strategy,” *AK Peters, Wellesley MA*, 2000.

[2] J. Beck, *Combinatorial Games: Tic-Tac-Toe Theory*, ser. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2011. [Online]. Available: https://books.google.com/books?id=A4lBYgEACAAJ

[3] M. Genesereth, N. Love, and B. Pell, “General game playing: Overview of the aaai competition,” *AI magazine*, vol. 26, no. 2, pp. 62–62, 2005.

[4] M. Genesereth, “Game definition language,” 2013. [Online]. Available: http://games.stanford.edu/games/gdl.html

[5] ——, “Epilog.” [Online]. Available: http://epilog.stanford.edu/homepage/index.php

[6] S. Reisch, “Hex is pspace-complete,” *Acta Informatica*, vol. 15, no. 2, pp. 167–191, 1981.

[7] D. Gale, “The game of hex and the brouwer fixed-point theorem,” *The American Mathematical Monthly*, vol. 86, no. 10, pp. 818–827, 1979.