Logic Programming

Datalog

Michael Genesereth
Computer Science Department
Stanford University
Logic Programming

Datalog

Michael Genesereth
Computer Science Department
Stanford University
Logic Programming

Database

Michael Cox
Computer Science Department
Stanford University
Logic Programming

DataLog

Optimizing Hex

Hunter Guru and Anthony Weng
Michael Genesereth
Computer Science Department
Stanford University
Hex is a two-player game invented by John Nash and Piet Hein (independently).

Players take turns placing tiles on any cell of their choosing.

Players win by connecting a chain of tiles, such that they form a line spanning from one edge of the board to the opposite edge.

Hex is a game commonly studied by Mathematicians in Computer Science in order to shed light on topics including: graph theory, combinatorics, game theory, and AI.

In 11×11 Hex, there are approximately $2.4 \times 10^{56}$ possible legal positions! (Approximated using an exponential function and branching factor analysis)
What are we Investigating?

- What are the different paradigms in which we can encode the rules of Hex?
- How does each paradigm perform (relatively)?
Why Logic Programming? (GDL)

- **Testing Games in a Generalizable Fashion:** Logic Programming is the methodology of describing games in the field General Gameplaying. GDL is widely accepted as the language of General Gameplaying!

- **Condition Testing:** We are really just solving a condition problem, namely: given this set of data, is X true? Logic Programming is very good for that!

- **Avoiding the “background implementations”:** In a traditional imperative programming language, we would have to focus on building the “back-end” framework from game-to-game; logic programming avoids that!
General Observations:

- We can assign each cell in the Hex board a numerical index.
- In this way, we can codify mathematical rules defining adjacency:
  - E.g. Cells X & Y are adjacent if $Y = X + 1$
  - One tile must be in each column (or row) in order for a player to have won (as a necessary, but not sufficient condition)
What if you abstracted half of the problem away from logic programming?

Use logic programming as a “verifier” and another language (Python) to generate the “Winning Sets”.

E.g. \(\{1,2,3,4,5,6,7,8,9\} \in W\)

After every move, check if the cells “controlled” by a given player is a superset of the winning set.

\[ \exists w \in W. \ w \subseteq M, \ \text{where } M \text{ is the set of cells } p \text{ has played.} \]
Approach #2: Power-set Constraints

1. Maintain set of all cells controlled by player p

2. Generate all* 9-length subsets s.t.
   each $E_i$ (element in the ith position)
   is a member of the ith column

3. For each element in a set, check if the subsequent element obeys an "adjacency" rule.

*To avoid repeatedly checking non-winning sets, one can preserve all previous non-winning sets and check all new sets generated by replacing the corresponding column entry in the previously generated sets

E.g. If you play in Column 6 ...

$$\{C_{Old} \rightarrow C_{New}\}$$ in all sets
Power-set Constraints: Worst-Case Analysis

\{28, 2, 12, 13, 42, 16, 8, 45\}

\{55, 11, 3, 31, 21, 42, 8, 27\}

2. Generate all 9-length subsets s.t. each \(E_i\) (element in the \(i\)th position) is a member of the \(i\)th column

Note:
What if we generated all 9-length subsets without our unique column restriction?

Suppose player \(p\) controls \(n\) cells \((n_{\text{max}} = 81)\):

\[\binom{81}{9} \quad \text{vs.} \quad 9^9 \quad \rightarrow \quad \frac{260887834350}{387420489}\]

\[\approx \times 674 \text{ more computations!}\]
But wait! It isn’t that simple!

This winning sequence is 61 tiles long!
Approach #2: Power-set Constraints

1. Maintain set of all cells controlled by player $p$

2. Generate all* $9$-length subsets $n$-length subsets $\{9, 61\}$, s.t. each $E_i$ (element in the $i$th position) is a member of the $i$th column

3. For each element in a set, check if the subsequent element obeys an “adjacency” rule.

\{28, 2, 12, 13, 42, 16, 8, 45\}
\{55, 11, 3, 31, 21, 42, 8, 27\}

$n - 9$?

$n - 8 \rightarrow 65 \in S ?$

$n + 1 \rightarrow 74 \in S ?$
Approach #3: Following the Line

1. For a given player $p$, consider each cell they control in column 1 (Indices: 1, 10, 19, … , 73)

2. Using the adjacency rules, compute all in the next column (column $i + 1$) that would be adjacent to the current cell (in column $i$).

3. If player $p$ controls any of the adjacent cells, repeat the adjacency check. If you can “follow the line” all the way to the end column, the player has won!
1. Define a “connected” relation: connected(TREE_NUM, ROW, COL)

2. After each turn, update the connected relations in the dataset:

3. The game is won if there is some set of connected relations s.t. there exists some “connected(N\text{WIN}, R_1, C)” and “connected(N\text{WIN}, R_8, C)” (with analogous reasoning extending to spanning a column). This set of connected relations defines the eponymous minimal spanning tree.
Beyond the Paradigm: General Optimization Techniques (GOT)

Grounding:

Sub-goal Reordering:

Sub-goal Pruning:

```
Lambda:
p(X) :- index(X)
index(1)
index(2)
index(3)
p(1) :- index(1)
p(2) :- index(2)
p(3) :- index(3)
index(1)
index(2)
index(3)

s(X,Y) :- p(X) & r(X,Y) & q(X)

r(X,Y) :- p(X,Y) & q(Y) & q(Z)

s(X,Y) :- p(X) & q(X) & r(X,Y)

r(X,Y) :- p(X,Y) & q(Y)
```
**GOT Efficiency Analysis?**:

- **Conjecture**: The majority of the time is spent in verifying whether a victory exists or not.

- **Technique**: Devise a particularly difficult example, and see if the verifier can (not) detect a victory.
  - Analysis was conducted on a board with 34 tiles filled, and no victory determined.
Without grounding and sub-goal reordering

Javascript:
```
grindem(compfinds(read('winner(X)'), read('winner(X)'), repository, library))
```

Eval

Output:
127448 milliseconds
winner(red)

With grounding and sub-goal reordering

Javascript:
```
grindem(compfinds(read('winner(X)'), read('winner(X)'), repository, library))
```

Eval

Output:
16 milliseconds
winner(red)

With grounding and sub-goal reordering
Hex as a Maker-Breaker Game

- A “Maker-Breaker” game can be thought of a game with two distinct players:
  - Maker: wins by taking elements from a finite set until they have a winning set
  - Breaker: wins by stopping the Maker

- Framing Hex as a Maker-Breaker game:
  - **Don’t** think: “Has Red won? Has Blue won?”
  - **Think:** “Has Red won? Has Red lost? (Can Red still win?)”

- Hex implementation:
  - After each play, populate all blank cells with red tiles
  - On Blue’s turn, if a red path still exists, then Red hasn’t lost
  - On Red’s turn, if a red path still exists, then Red can still win!

- Maker-Breaker general strategy: populating available moves with Maker’s moves