Outline

- Implementing a basic general game player
- Foundations of logic programming
- Metagaming: rule optimisation
Uses of Logic

Use logical reasoning for game play
- Computing the legality of moves
- Computing consequences of actions
- Computing goal achievement
- Computing termination

Easy to convert from logic to other representations:
many orders of magnitude speedup on simulations

Things that may better be done in Logic
- Game Reformulation
- Game Analysis
Available Basic Players

**Prolog Player**
- A basic player implemented in ECLiPSe Prolog based on code from PLXPLAYER.
- Download: [current version](#).

**Java Player**
- A basic player implemented in Java which comes with a framework for implementing your strategies, analyzing the game, etc. It can be found on the Palamedes-IDE website.

**C++ Player**
- A basic player implemented in C++ with the same library as the Prolog player above.
All Players Use Some Form of Logic Programming

true(cell(1,1,b))
...
true(cell(3,3,b))
true(control(xplayer))

legal(P,mark(M,N)) <= true(cell(M,N,b)) ∧ true(control(P))
legal(xplayer,noop) <= true(control(oplayer))
legal(oplayer,noop) <= true(control(xplayer))

Given this logic program, answer the query

?- legal(P,M)
Substitutions

A substitution is a finite set of replacements of variables by terms

\{X/a, Y/f(b), V/W\}

The result of applying a substitution \(\sigma\) to an expression \(\varphi\) is the expression \(\varphi\sigma\) obtained from \(\varphi\) by replacing every occurrence of every variable in the substitution by its replacement.

\(p(X, X, Y, Z) \{X/a, Y/f(b), V/W\} = p(a, a, f(b), Z)\)
Unification

A substitution $\sigma$ is a unifier for an expression $\varphi$ and an expression $\psi$ if and only if $\varphi\sigma = \psi\sigma$.

\[
\text{move}(X, Y) \{X/a, Y/b, V/b\} = \text{move}(a, b) \\
\text{move}(a, V) \{X/a, Y/b, V/b\} = \text{move}(a, b)
\]

If two expressions have a unifier, they are said to be unifiable.

move ($X, X$) and move ($a, b$) not unifiable
Most General Unifiers

- A substitution $\sigma$ is more general than a substitution $\theta$ if and only if there is a substitution $\tau$ such that $\sigma \circ \tau = \theta$.

- A substitution $\sigma$ is a most general unifier (mgu) of two expressions if and only if it is more general than any other unifier.

**Theorem**: If two expressions are unifiable, then they have an mgu that is unique up to variable permutation.

\[
\begin{align*}
\text{move}(X, Y) \{X/a, Y/V\} & = \text{move}(a, V) \\
\text{move}(a, V) \{X/a, Y/V\} & = \text{move}(a, V) \\
\text{move}(X, Y) \{X/a, V/Y\} & = \text{move}(a, Y) \\
\text{move}(a, V) \{X/a, V/Y\} & = \text{move}(a, Y)
\end{align*}
\]
Resolution

Given:

Query $L_1 \land L_2 \land ... \land L_m$ (without negation)
Clauses (without negation)

Let:

$A \leq B_1 \land ... \land B_n$ “fresh” variant of a clause
$\sigma$ mgu of $L_1$ and $A$

Then $L_1 \land L_2 \land ... \land L_m \rightarrow (B_1 \land ... \land B_n \land L_2 \land ... \land L_m)\sigma$

is a resolution step.
Query Answering

- A sequence of resolution steps is called a **derivation**.
- A **successful** derivation ends with the empty query.
- The **answer substitution** (computed by a successful derivation) is obtained by composing the mgu's $\sigma_1 \circ ... \circ \sigma_n$ of each step (and restricting the result to the variables in the original query).
- A **failed** derivation ends with a query to which no clause applies.
Example

true(cell(1,1,b))
...
true(cell(3,3,b))
true(control(xplayer))

legal(P,mark(M,N)) <= true(cell(M,N,b)) \land true(control(P))
legal(xplayer,noop) <= true(control(oplayer))
legal(oplayer,noop) <= true(control(xplayer))

Query \texttt{?- legal(P,M)} has the following answers:

\{P/xplayer, M/mark(1,1)}}, ..., \{P/xplayer, M/mark(3,3)}
\{P/oplayer, M/noop\}
Query Answering with Negation

Given:

Query $L_1 \land L_2 \land \ldots \land L_m$
Clauses

- If $L_1$ is an atom, proceed as before
- If $L_1$ is of the form $\neg A$:
  
  - if all derivations for $A$ fail then
    $L_1 \land L_2 \land \ldots \land L_m \rightarrow L_2 \land \ldots \land L_m$
  
  - if there is a successful derivation for $A$ then
    $L_1 \land L_2 \land \ldots \land L_m \rightarrow \text{fail}$
Example

role(red)
role(blue)
role(green)
true(freecell(blue))
trapped(P) <= role(P) ∧ ¬true(freecell(P))
goal(P,100) <= role(P) ∧ ¬trapped(P)

Query ?- goal(P,100) has the only answer {P/blue}
Query Answering with Disjunction

A clause with a disjunction

\[ A \leq B \land (C_1 \lor C_2) \land D \]

is logically equivalent to the conjunction of the clauses

\[ A \leq B \land C_1 \land D \]
\[ A \leq B \land C_2 \land D \]
Some Rules You Don't Want to Allow

\[
\text{role(player}(X)\text{)}
\]

\[
\text{next(control}(\text{white})\text{)} \leq p \\
\text{next(control}(\text{black})\text{)} \leq r \\
p \leq \neg r \\
r \leq \neg p
\]
How to Guarantee Finiteness (Part 1)

A clause is safe if and only if every variable in the clause appears in some positive subgoal in the body.

- **Safe Rule:**
  \[ r(X, Y) \leftarrow p(X, Y) \land q(Y, Z) \land \neg r(X, Z) \]

- **Unsafe Rule:**
  \[ r(X, Z) \leftarrow p(X, Y) \land q(Y, X) \]

- **Unsafe Rule:**
  \[ r(X, Y) \leftarrow p(X, Y) \land \neg q(Y, Z) \]

In GDL, all rules are required to be safe.
(Note that this implies all facts to be variable-free.)
The dependency graph for a set of clauses is a directed graph in which
- the nodes are the relations mentioned in the head and bodies of the clauses
- there is an arc from a node $p$ to a node $q$ whenever $p$ occurs in the body of a clause in which $q$ is in the head.

\[
\begin{align*}
    r(X,Y) & \leq p(X,Y) \land q(X,Y) \\
    s(X,Y) & \leq r(X,Y) \\
    s(X,Z) & \leq r(X,Y) \land t(Y,Z) \\
    t(X,Z) & \leq s(X,Y) \land s(Y,X)
\end{align*}
\]

A set of clauses is recursive if its dependency graph contains a cycle.
How to Guarantee Finiteness (Part 2)

A set of rules is said to be **stratified** if there is no recursive cycle in the dependency graph involving a negation.

- **Stratified:**
  \[
  \begin{align*}
  t(X,Y) & \Leftarrow q(X,Y) \land \neg r(X,Y) \\
  r(X,Z) & \Leftarrow p(X,Y) \\
  r(X,Z) & \Leftarrow r(X,Y) \land r(Y,Z)
  \end{align*}
  \]

- **Not stratified:**
  \[
  \begin{align*}
  r(X,Z) & \Leftarrow p(X,Y) \\
  r(X,Z) & \Leftarrow p(X,Y) \land \neg r(Y,Z)
  \end{align*}
  \]

In GDL, all game descriptions are required to be stratified.
Metagaming: Rule Optimisation

Example:
\[
\text{goal}(X,Z) \Leftarrow p(X,Y) \land q(Y,Z) \land \text{distinct}(Y,b)
\]

Better:
\[
\text{goal}(X,Z) \Leftarrow p(X,Y) \land \text{distinct}(Y,b) \land q(Y,Z)
\]

The argument domains can be determined from the rules of the game with the help of the dependency graph.

```
succ(0,1)
succ(1,2)
succ(2,3)
init(step(0))
next(step(X)) \Leftarrow 
  true(step(Y)) \land succ(Y,X)
```
Rule Optimisation Based on Domains

Example:

\[ \text{wins}(P) \leq \text{true(cell}(X,Y,P)) \land \text{corner}(X,Y) \land \text{king}(P) \]

Solution Set Sizes:

\[ |\text{true(cell}(X,Y,P))| = 768 \]
\[ |\text{corner}(X,Y)| = 4 \]
\[ |\text{queen}(P)| = 2 \]

Better Version:

\[ \text{wins}(P) \leq \text{king}(P) \land \text{corner}(X,Y) \land \text{true(cell}(X,Y,P)) \]
Pre-Computing Answers

The ancestor relation is the transitive closure of the parent relation:

\[
\text{ancestor}(X, Y) \leq \text{parent}(X, Y) \\
\text{ancestor}(X, Z) \leq \text{ancestor}(X, Y) \land \text{ancestor}(Y, Z)
\]

The “samefamily” relation is true of all pairs of people that are relatives, i.e., that have a common ancestor:

\[
\text{sf}(Y, Z) \leq \text{ancestor}(X, Y) \land \text{ancestor}(X, Z)
\]

If we pre-compute \text{ancestor} then we increase the computational efficiency of answering the query \text{sf}.

Hitch: database storage space
Not a Good Idea to Pre-Compute $sf$
Better: Pre-Compute \textit{ancestor}
Even Better: Pre-Compute a New Relation
Outlook: Building a Good General Game Player

- Playing Single-Player Games (a.k.a. Planning)
- Stochastic Search
- Automatic Heuristics Generation