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

Downloads – General Game Playing 2/05/11 12:25 PM

- Home
- Activities
- Research
- Literature
- Getting Started
- Downloads
- Links

Downloads

We provide programs that might help you to implement your own General Game Playing system. All programs contain source code and are distributed under GPL.

GameController

GameController is a standalone game master clone written entirely in Java and developed as part of the GGPServer project. It is particularly useful for testing your own general game playing system. GameController comes with a simple GUI and a command line interface. Send bug reports and suggestions to Stephan Schiffel.

Download the most recent version from the sourceforge project page.

System requirements:

- Java 1.6 runtime environment

Usage:

```
java -jar gamecontroller-XYZ.jar
```

Basic Prolog Player

Prolog Player

A basic player implemented in ECLiPSe Prolog based on code from FLUXPLAYER.

Download current version (1.1)

System requirements:

- ECLiPSe Prolog version 5.10 or higher

Changes since version 1.0

- the port should be free now after stopping the player

(last update: 12 March 2009)

Basic 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.

Java Player

Basic C++ Player

A basic player implemented in C++ with the reasoner of the prolog player above.

Download current version (1.6)

C++ Player

System requirements:

- Linux/Unix (or any system which provides sockets)

http://www.general-game-playing.de/downloads.html Page 1 of 2

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 σ to an expression φ is the expression $\varphi\sigma$ obtained from φ 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 σ is a unifier for an expression φ and an expression ψ if and only if $\varphi\sigma=\psi\sigma$.

```
move(X,Y){X/a,Y/b,V/b} = move(a,b)
move(a,V){X/a,Y/b,V/b} = 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 σ is more general than a substitution θ if and only if there is a substitution τ such that $\sigma \circ \tau = \theta$.
- A substitution σ 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.

```
move(X,Y){X/a,Y/V} = move(a,V)
move(a,V){X/a,Y/V} = move(a,V)

move(X,Y){X/a,V/Y} = move(a,Y)
move(a,V){X/a,V/Y} = move(a,Y)
```

Resolution

Given:

Query $L_1 \wedge L_2 \wedge \ldots \wedge L_m$ (without negation)

Clauses (without negation)

Let:

$A \Leftarrow B_1 \wedge \ldots \wedge B_n$ "fresh" variant of a clause

σ mgu of L_1 and A

Then $L_1 \wedge L_2 \wedge \ldots \wedge L_m \rightarrow (B_1 \wedge \ldots \wedge B_n \wedge L_2 \wedge \ldots \wedge L_m)\sigma$

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 $\sigma_1 \circ \ldots \circ \sigma_n$ 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)) ∧ true(control(P))
legal(xplayer,noop) <= true(control(oplayer))
legal(oplayer,noop) <= true(control(xplayer))
```

Query `?- 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 $L_1 \wedge L_2 \wedge \ldots \wedge L_m$

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
    $L_1 \wedge L_2 \wedge \ldots \wedge L_m \rightarrow L_2 \wedge \ldots \wedge L_m$

  - if there is a successful derivation for A then
    $L_1 \wedge L_2 \wedge \ldots \wedge L_m \rightarrow$ 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

$$A \Leftarrow B \wedge (C_1 \vee C_2) \wedge D$$

is logically equivalent to the conjunction of the clauses

$$A \Leftarrow B \wedge C_1 \wedge D$$

$$A \Leftarrow B \wedge C_2 \wedge D$$

Some Rules You Don't Want to Allow

```
role(player(X))
```

```
next(control(white)) <= p
next(control(black)) <= r
p <= ¬r
r <= ¬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) <= p(X,Y) ∧ q(Y,Z) ∧ ¬r(X,Z)`

- Unsafe Rule:
  `r(X,Z) <= p(X,Y) ∧ q(Y,X)`

- Unsafe Rule:
  `r(X,Y) <= p(X,Y) ∧ ¬q(Y,Z)`

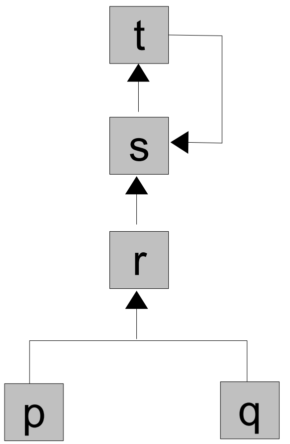
In GDL, all rules are required to be safe.
(Note that this implies all facts to be variable-free.)

Dependency Graph

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.

```
r(X,Y) <= p(X,Y) ∧ q(X,Y)
s(X,Y) <= r(X,Y)
s(X,Z) <= r(X,Y) ∧ t(Y,Z)
t(X,Z) <= s(X,Y) ∧ s(Y,X)
```



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:

```
t(X,Y) <= q(X,Y) ∧ ¬r(X,Y)
r(X,Z) <= p(X,Y)
r(X,Z) <= r(X,Y) ∧ r(Y,Z)
```

- Not stratified:

```
r(X,Z) <= p(X,Y)
r(X,Z) <= p(X,Y) ∧ ¬r(Y,Z)
```

In GDL, all game descriptions are required to be stratified.

Metagaming: Rule Optimisation

Example:

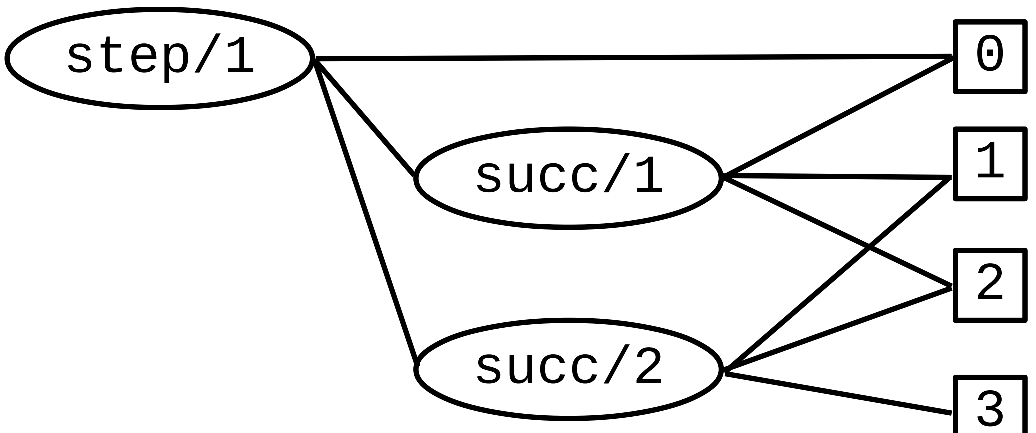
```
goal(X,Z) <= p(X,Y) ∧ q(Y,Z) ∧ distinct(Y,b)
```

Better:

```
goal(X,Z) <= p(X,Y) ∧ distinct(Y,b) ∧ 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)) <=
  true(step(Y)) ∧
  succ(Y,X)
```



Rule Optimisation Based on Domains

Example:

```
wins(P) <= true(cell(X,Y,P)) ∧ corner(X,Y) ∧ king(P)
```

Solution Set Sizes:

```
|true(cell(X,Y,P))| = 768
|corner(X,Y)|       =   4
|queen(P)|          =   2
```

Better Version:

```
wins(P) <= king(P) ∧ corner(X,Y) ∧ true(cell(X,Y,P))
```

Pre-Computing Answers

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

```
ancestor(X,Y) <= parent(X,Y)
ancestor(X,Z) <= ancestor(X,Y) ∧ ancestor(Y,Z)
```

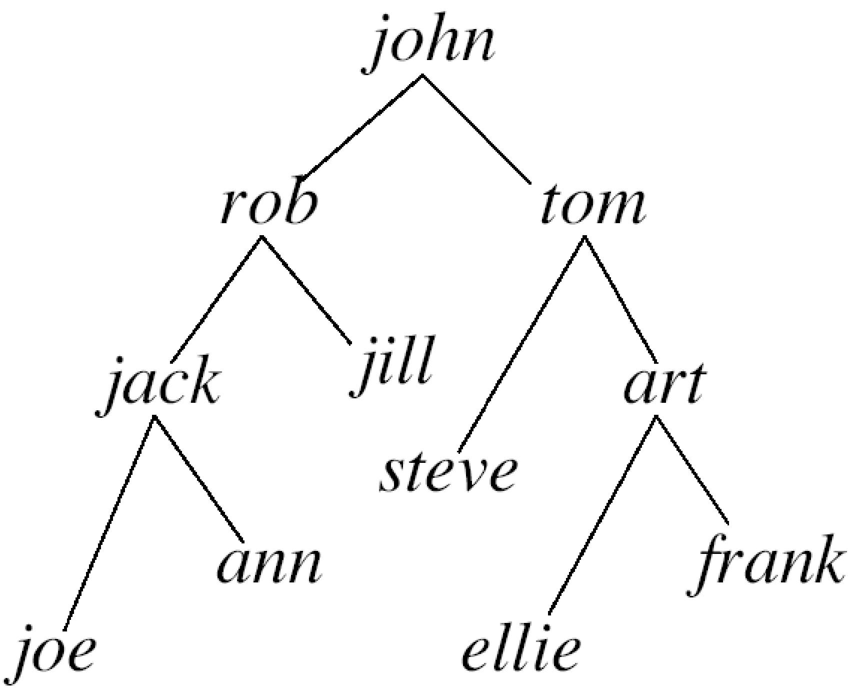
The “samefamily” relation is true of all pairs of people that are relatives, i.e., that have a common ancestor:

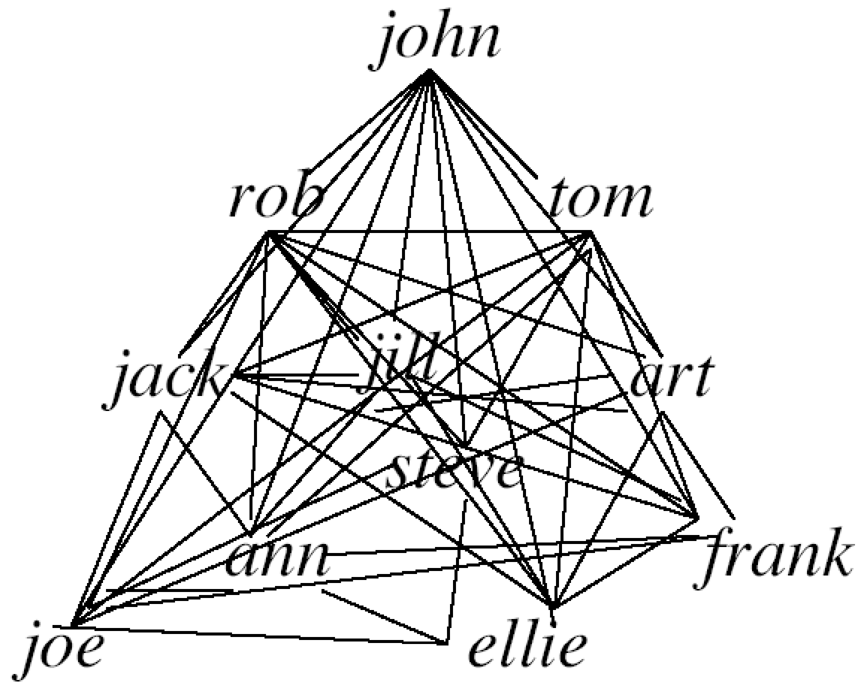
```
sf(Y,Z) <= ancestor(X,Y) ∧ ancestor(X,Z)
```

If we pre-compute `ancestor` then we increase the computational efficiency of answering the query `sf`.

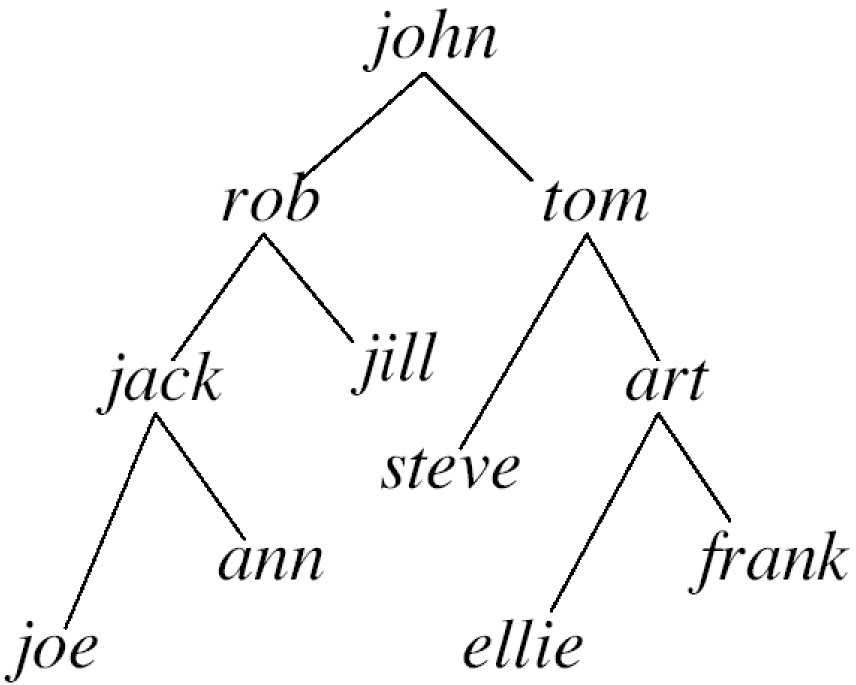
Hitch: database storage space

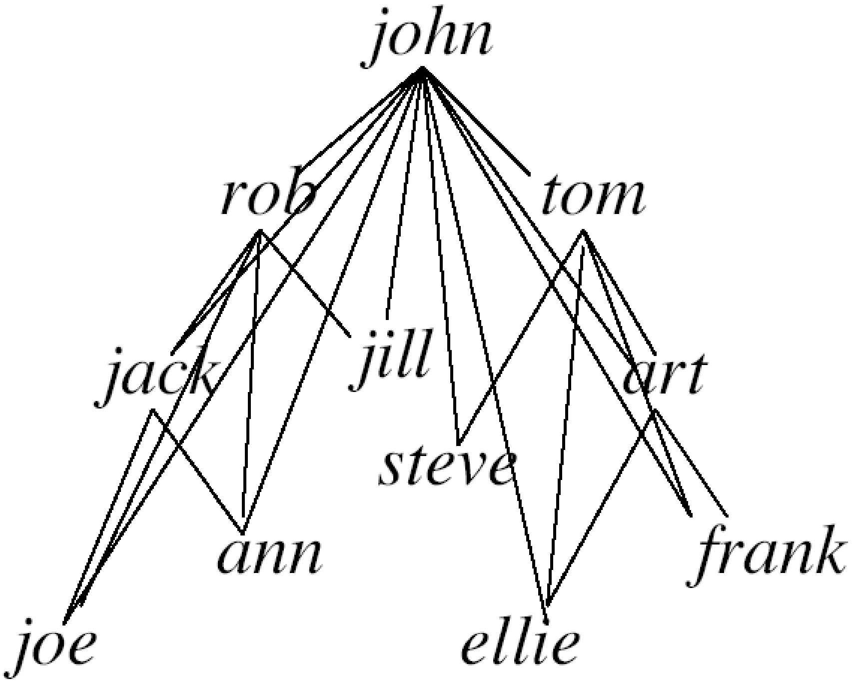
Not a Good Idea to Pre-Compute `sf`



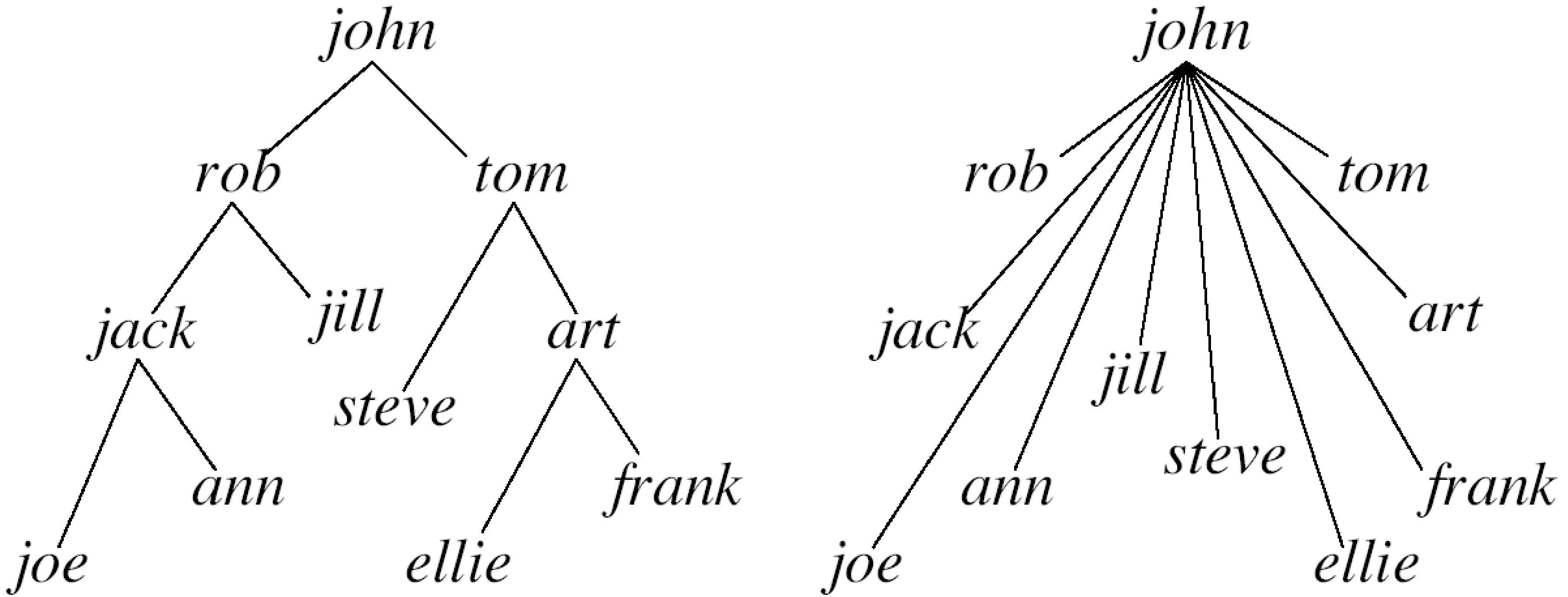


Better: Pre-Compute `ancestor`





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