Game Description Language Compiler Construction

Jakub Kowalski, Marek Szykuła

29 listopada 2013

What is General Game Playing?

General game playing is the design of artificial intelligence programs to be able to play more than one game successfully.

Wikipedia, The Free Encyclopedia

A General Game Playing System is one that can accept a formal description of a game and play the game effectively without human intervention.

Michael Genesereth, Nathaniel Love; Stanford University

General Game Playing is about playing games that you've never seen before.

Sam Schreiber; ggp.org

GDL

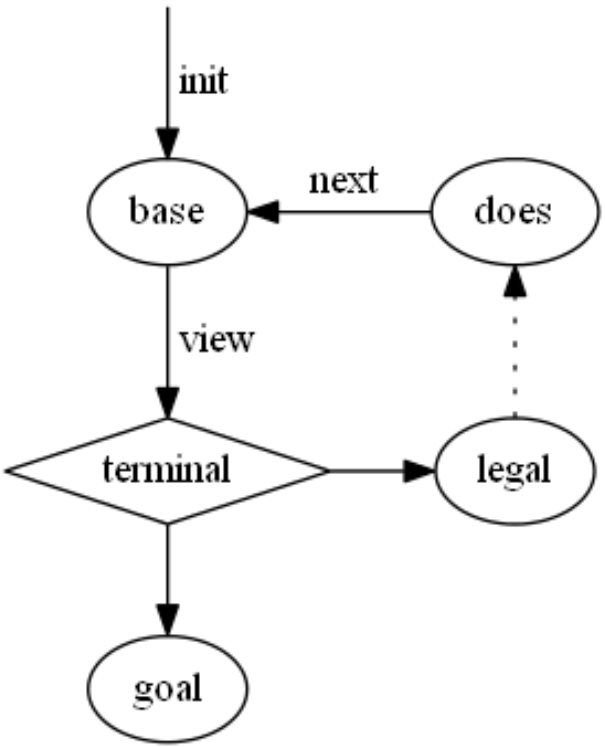
Basics

- Variant of Datalog
- Prefix notation rules with ? preceded variables
- Only few keywords
- Pure first-order logic: no built-in assumptions, no arithmetic, no physics, no common sense
- Wide range of games: finite, deterministic, with full information and simultaneous moves

Formal description of n-players game

S - set of states
$A_1, \ldots, A_n$ – sets of possible actins (for each player)
$l_1, \ldots, l_n$ – legal actions ($l_i \subseteq A_i \times S$)
$u : S \times A_i \times \ldots \times A_n \rightarrow S$ – update function
$s_1 \in S$ – initial state
$T \subseteq S$ – set of terminal states
$g_1, \ldots, g_n$ – goal relations ($g_i \subseteq S \times \mathbb{N}$)

GDL Example: Tic Tac Toe




```
(role xplayer)
(init (cell 1 1 b))
(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))
(<= (next (cell ?m ?n x))
    (does xplayer (mark ?m ?n))
    (true (cell ?m ?n b)))
(<= terminal
    (line x))
(<= (goal xplayer 100)
    (line x))
```

Reasoning about the state

Rule-ordered sequence of queries and insertions to predicates database.

Base

control
xplayer

cell		
1	1	o
1	2	b
1	3	x
2	1	x
2	2	o
2	3	x
3	1	b
3	2	b
3	3	b

View

row

legal

```
(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
(<= (legal oplayer noop)
    (true (control xplayer)))
(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))
(<= (next (cell ?m ?n x))
    (does xplayer
          (mark ?m ?n))
    (true (cell ?m ?n b)))
```

Reasoning about the state

Rule-ordered sequence of queries and insertions to predicates database.

Base

control
xplayer

cell
1 1 o
1 2 b
1 3 x
2 1 x
2 2 o
2 3 x
3 1 b
3 2 b
3 3 b

View

row
3 b

legal

```
(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
(<= (legal oplayer noop)
    (true (control xplayer)))
(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))
(<= (next (cell ?m ?n x))
    (does xplayer
          (mark ?m ?n))
    (true (cell ?m ?n b)))
```

Reasoning about the state

Rule-ordered sequence of queries and insertions to predicates database.

Base

control
xplayer

cell
1 1 o
1 2 b
1 3 x
2 1 x
2 2 o
2 3 x
3 1 b
3 2 b
3 3 b

View

row
3 b

legal
oplayer noop

```
(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
(<= (legal oplayer noop)
    (true (control xplayer)))
(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))
(<= (next (cell ?m ?n x))
    (does xplayer
          (mark ?m ?n))
    (true (cell ?m ?n b)))
```

Reasoning about the state

Rule-ordered sequence of queries and insertions to predicates database.

Base

control
xplayer

cell
1 1 o
1 2 b
1 3 x
2 1 x
2 2 o
2 3 x
3 1 b
3 2 b
3 3 b

View

row
3 b

legal
oplayer noop
xplayer (mark 1 2)
xplayer (mark 3 1)
xplayer (mark 3 2)
xplayer (mark 3 3)

```
(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
(<= (legal oplayer noop)
    (true (control xplayer)))
(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))
(<= (next (cell ?m ?n x))
    (does xplayer
          (mark ?m ?n))
    (true (cell ?m ?n b)))
```

Reasoning about the state

Rule-ordered sequence of queries and insertions to predicates database.

Base

control
xplayer

cell
1 1 o
1 2 b
1 3 x
2 1 x
2 2 o
2 3 x
3 1 b
3 2 b
3 3 x

View

row
3 b

legal
oplayer noop
xplayer (mark 1 2)
xplayer (mark 3 1)
xplayer (mark 3 2)
xplayer (mark 3 3)

does
oplayer noop
xplayer (mark 3 3)

```
(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
(<= (legal oplayer noop)
    (true (control xplayer)))
(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))
(<= (next (cell ?m ?n x))
    (does xplayer
          (mark ?m ?n))
    (true (cell ?m ?n b)))
```

Implementations

Approaches

- Prolog engine
- Compilation (top-down/bottom-up)
- Propositional Networks
- Instantiation, binary decision diagrams

Efficiency importance

- Number of Monte Carlo simulations
- Min-Max frontier depth
- Heuristic functions fitness checking

Implementations

Approaches

- Prolog engine
- Compilation (top-down/bottom-up)
- Propositional Networks
- Instantiation, binary decision diagrams

Efficiency importance

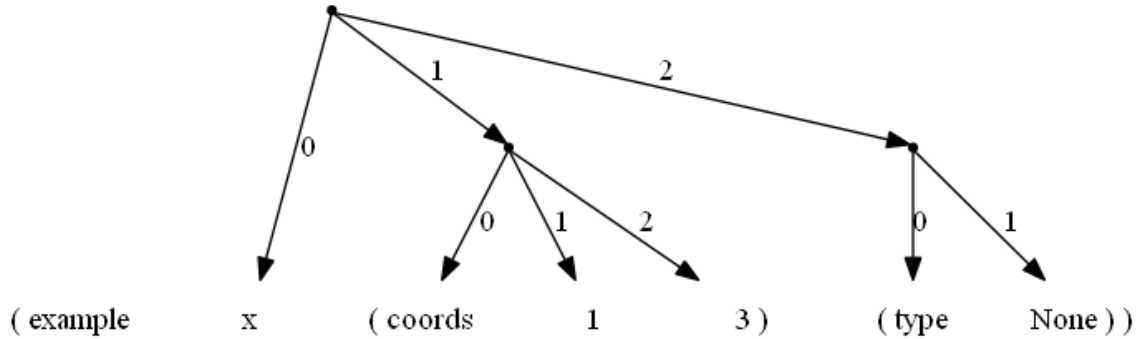
- Number of Monte Carlo simulations
- Min-Max frontier depth
- Heuristic functions fitness checking

Compiler construction

Optimizations

- Speed of queries
  - Calculating predicates' domains
  - Flattening of predicates occurences
  - Assignment of data structures
- Order of computations
  - Calculating order of predicates
  - Division into game phases
  - Calculating order of rules
  - Calculating order of queries and insertions

Calculating Domains



$\triangleright_R$ domain dependance relation

$(P, \tilde{p}) \triangleright_R (Q, \tilde{q})$ if in rule R the same variable occurs in head predicate P at position $\tilde{p}$ and in body predicate Q at position $\tilde{q}$.

Nesting dependance property

If $(P, \tilde{p}) \triangleright_R (Q, \tilde{q})$ then for every valid paths $\tilde{q}'' = \tilde{q} + \tilde{q}'$ also $(P, \tilde{p} + \tilde{q}') \triangleright_R (Q, \tilde{q}'')$.

Domain update

$\texttt{Dom}((P, \tilde{p})) \texttt{ := Dom}((P, \tilde{p})) \cup \bigcap\{(Q, \tilde{q}) : (P, \tilde{p}) \triangleright_R (Q, \tilde{q})\}$

Flattening

Flattened predicate

Predicate is *flattened* if all its occurences have the same arity and nesting level 1.

Example

```
(Item (Book (LordOfTheRings BookII)) 15gp)
(Item ?b 15gp)
(Item (Potion SmallHealthPotion ?t 25p) 25gp)
(Item (Potion BigManaPotion (Type Mana) 25p) ?gp)
```

Lemma

For every occurrence of a valid predicate with a fixed domain, there exist its unique flattened form and an algorithm that converts these forms.

Lemma

Arity of flattened predicate can be exponential.

Flattening

Flattened predicate

Predicate is *flattened* if all its occurences have the same arity and nesting level 1.

Example

```
(Item Book LordOfTheRings BookII #nil #nil #nil 15gp)
(Item ?b0 ?b1 ?b2 ?b3 ?b4 ?b5 15gp)
(Item Potion SmallHealthPotion #nil ?t0 ?t1 25p 25gp)
(Item Potion BigManaPotion #nil Type Mana 25p ?gp0)
```

Lemma

For every occurrence of a valid predicate with a fixed domain, there exist its unique flattened form and an algorithm that converts these forms.

Lemma

Arity of flattened predicate can be exponential.

Basic structures

Types of query shapes

`cell 3 3 b`, `cell ?x ?y ?z`, `cell ?x 1 ?z`, `cell ?x ?x b`

d	- predicate arity
n	- number of inserted facts
$c_1, \ldots, c_d$	- sizes of arguments domains
s	- query answer size

Vector

- Insertion in $O(1)$ (without checking duplicates)
- All queries take $O(n)$ in worst case.

Array

- Insertion in $O(1)$
- Queries about the facts in $O(1)$ time.
- Querying for larges subsets $O(\prod_{i=1}^{d} c_i)$ in worst case.

Composed Structures

Trie

- Insertion worst case time is $O(\sum_{i=1}^{d} c_i)$
- Queries about the facts in $O(1)$ time.
- Querying for larges subsets from $O(s)$ to $O(n)$ depends on arguments order.

Composed structures consist of a set of other structures made especially for efficient maintaining of different query shapes.

Lemma

The *trie-composed* structure takes $O(s)$ time for a query and $O(c_1 + \ldots + c_d)$ time for an insertion.

Lemma

The *tree-composed* structure takes $O(s + \log n)$ time for a query and $O(\log n)$ time for an insertion.

Predicates Ordering

Predicate Dependency Graph

Directed graph with predicates as vertices, where edge (P, Q) exists iff. there exist rule with P in head and Q in body.

Stratification

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

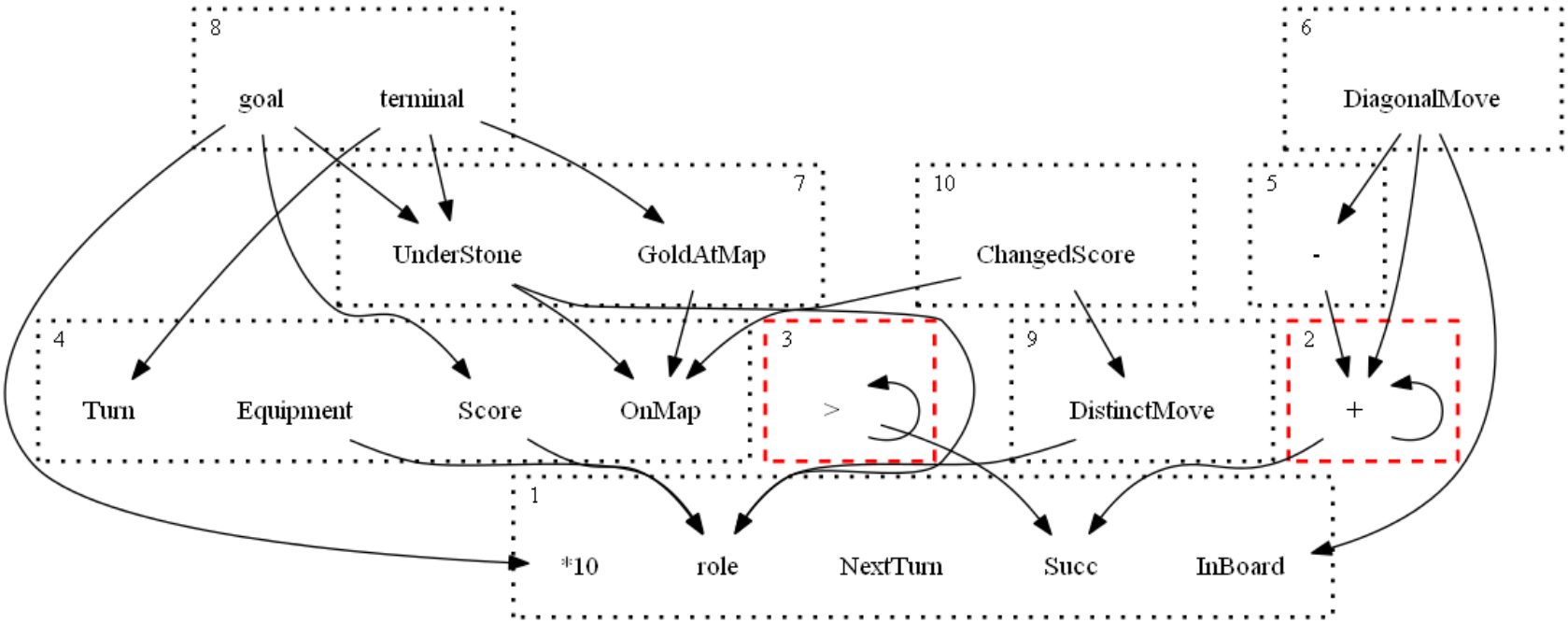
Layering

Layer is set of predicates which should be computed together to minimize number of overall computations. Predicates are devided into layers of two types:

- *Acyclic* – there is no path in dependency graph between any two elements of layer (maximal size; computed once).
- *Cyclic* – elements of layer form a strongy connected component (minimal size; computed until fixpoint is reached).

Layers and Rule Ordering

Example of dependency graph with layers



Rule computations order

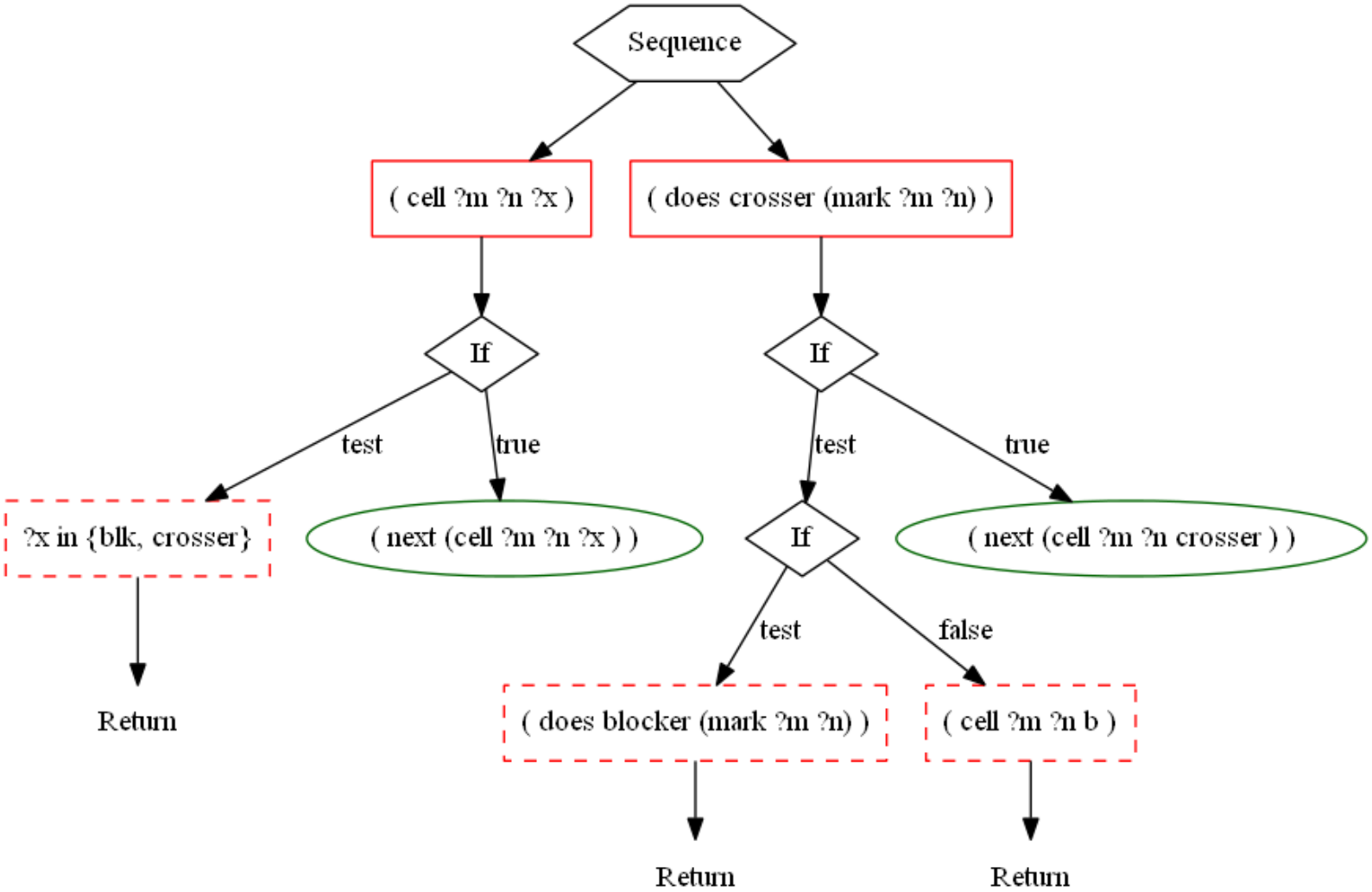
Let L_h^R be number of rule R head's layer and let $L_{b\ \max}^R$ be the highest layer of predicates in body of R. If $L_h^R \neq L_{b\ \max}^R$ then rule can be computed in any layer in range $(L_{b\ \max}^R, L_h^R)$.

Filter Tree Structure

Definition

Filter tree is a structure appointing exact order of computations. Reasoning about the state indicates traversing filter tree from the root and performing actions assigned to nodes types. During this processa set of local variables and set of containers are maintained.
The nodes have the following types:

- *Sequence* – executes children trees in order.
- *Query* – queries specified container for subset of facts matching the query, can filter domains.
- *Accept* – inserts fact with bounded variables into the conatiner. Sets *repeat* flag.
- *Repeat* – repeats subtree until *repeat* flag is unset.
- *If* – execute *test* child, then if *Return* node is reached executes the *true* childe, otherwise *false*.




```
(<= (next (cell ?m ?n ?x))
    (true (cell ?m ?n ?x))
    (distinct ?x b))
```

```
(<= (next (cell ?m ?n crosser))
    (does crosser (mark ?m ?n))
    (not (does blocker (mark ?m ?n)))
    (true (cell ?m ?n b)))
```

Implementation

- Java program producing a reasoner code in C++
- Compiled reasoner is a module allowing to maintain game states.
- Reasoner is part of GGP player Dumalion (J. Kowalski, M. Szykuła).
- Every node of filter tree is directly inlined in the reasoner code.

Comparison with Prolog engine

Game	The reasoner of Dumalion		Prolog	Ratio
	Compilation	Simulations	Simulations	
Tic-Tac-Toe	0.736 s	331,647	2,076	159.75
Blocker	0.645 s	194,628	1,020	190.81
Connect Four	0.857 s	15,092	287	52.59
Breakthrough	1.074 s	3,086	55	56.11
Checkers	10.482 s	211	12	17.58
Skirmish	7.810 s	71	5	14.2

(simulations over 1 second)

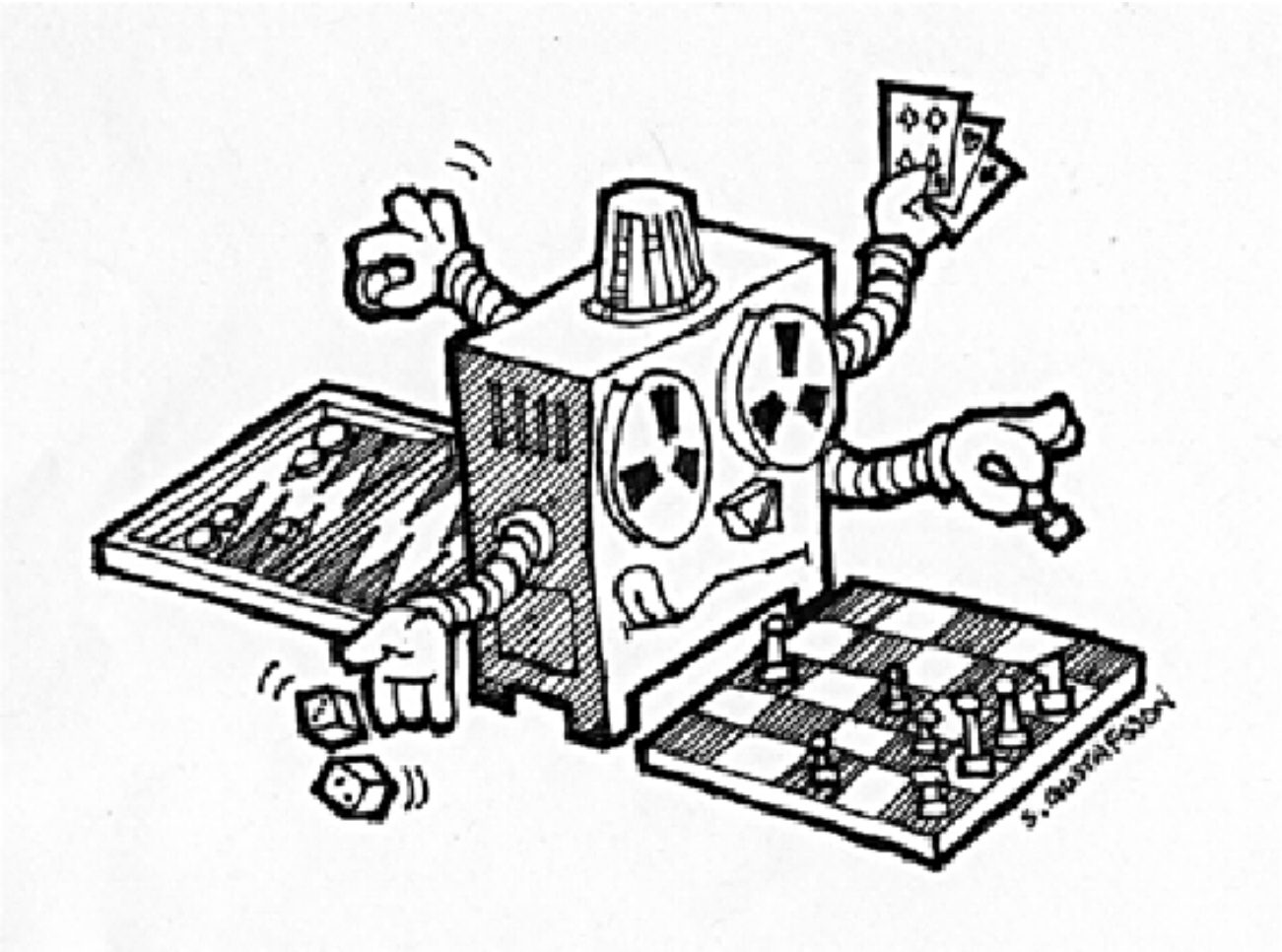
Rough comparison with other reasoners

	Dumalion	Forward Chaining	Fluxplayer
	Intel Core i7 3.3 GHz 8GB RAM	Intel Core i7 3.4 GHz 8GB RAM	Intel Xeon 2.5 GHz 4 GB RAM
Tic-Tac-Toe	2,860,888	898,000	14,471
Connect Four	353,283	111,000	1,780
Breakthrough	200,901	168,000	3,783
Skirmish	7,114	-	3,278

(states examined over 1 second)

Schofield, M. and Saffidine, A.: High Speed Forward Chaining for General Game Playing, GIGA'13, 2013

Björnsson, Y. and Schiffel, S.: Comparison of GDL Reasoners, GIGA'13, 2013



Thank you