Overview: Representation Techniques

Week 6

- Representations for classical planning problems
  - deterministic environment; complete information

Week 7

- Logic programs for problem representations
  - including planning problems, games

Week 8

- First-order logic to describe dynamic environments
  - deterministic environment; (in-)complete information

Week 9

- State transition systems to describe dynamic environments
  - nondeterministic environment; (in-)complete information

Answer Set Programming: Overview

- Foundations: Stable models of a logic program
- How to represent problems as Answer Set Programs (ASP)
- ASP solver technology
- Solving planning problems and general single-player games with ASPs

Background reading

Answer Set Programming at a Glance by Gerhard Brewka, Thomas Eiter, Miroslaw Truszczynski, Communications of the ACM 54(12):93-103, 2011

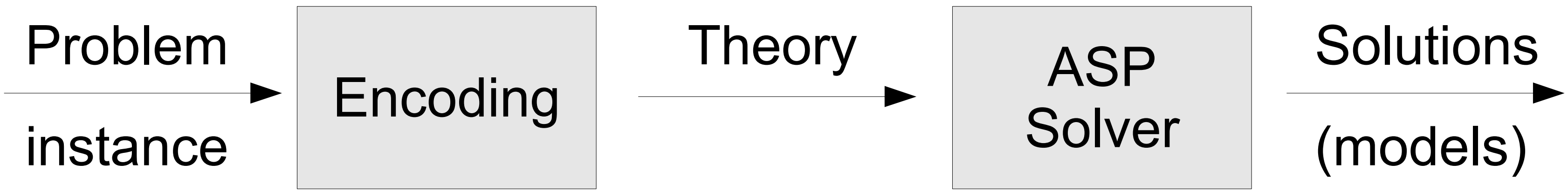
Potassco User Guide
http://sourceforge.net/projects/potassco/files/potassco_guide/2010-10-04/guide.pdf/download

Handbook of Knowledge Representation by Bruce Porter, Vladimir Lifschitz, Frank Van Harmelen, Elsevier 2007. Chapter 7

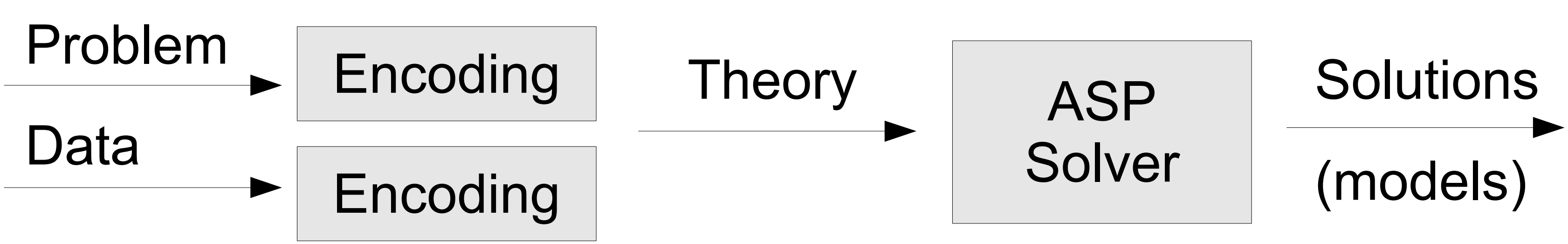
History

- Roots: logic programming, nonmonotonic reasoning
- Based on some formal system with semantics that assigns a theory a collection of answer sets (models)
- An **ASP solver**: computes answer sets for a theory
- Solving a problem in ASP:
    - Encode the problem as a theory such that **solutions** to the problem are given by **answer sets** of the theory

Solving a Problem with an ASP



- Uniform encoding:
  separate problem specification and data
- Compact, easily maintainable representation
- Integrating KR, DB (i.e. database), and search techniques
- Handling dynamic, knowledge intensive applications



Example: k-Colouring Problem

- Given a graph **(V,E)** find an assignment of one of **k** colours to each vertex such that no two adjacent vertices share a colour

```
% Problem encoding
1 { coloured(V,C) : colour(C) } 1 :- vtx(V).
:- edge(V,U), colour(C), coloured(U,C), coloured(V,C).
```

```
% Data
vtx(a). …
edge(a,b). …
colour(r). colour(b). …
```

- Legal colourings of the graph given as data and stable models of the problem encoding and data correspond:
  - a vertex *v* coloured with a colour *c* iff `coloured(v,c)` holds in a stable model

Semantics: Stable Models

Stable Models of a Logic Program

- Consider normal logic program rules

  $A \leftarrow B1, \ldots, Bm, \text{not } C1, \ldots, \text{not } Cn$

- Seen as "constraints" on an answer set (stable model):
  - if B1, . . . ,Bm are in the set and
  - none of C1, . . . ,Cn is included,

  then A must be included in the set

- A **stable model** is a set of atoms

  (i) which satisfies the rules and

  (ii) where each atom is **justified** by the rules

  ⇨ negation by default; closed world assumption (CWA)

Stable Models (cont'd)

- Program:

  $b \leftarrow$

  $f \leftarrow b, \text{not } eb$

  $eb \leftarrow e$

Stable model:

{ b , f }

- Another candidate model: {b, eb} satisfies the rules but is not a proper stable model:
  - eb is included for no reason.
- Justifiability of stable models is captured by the notion of a **reduct** of a program

Definite Programs

- For the reduct we first need to consider negation-free programs, i.e. programs without `not`
- Such a program P has a unique least model **Least-Model(P)** satisfying the rules
- **Least-Model(**P**)** can be constructed by forward chaining

Examples.

P_1:

$p \leftarrow$

$r \leftarrow p$

$LM(P_1) = \{p, r\}$

P_2:

$p \leftarrow r$

$r \leftarrow p$

$LM(P_2) = \{\}$

P_3:

$p \leftarrow r$

$r \leftarrow p$

$p \leftarrow$

$LM(P_3) = \{p, r\}$

Stable Models (Formal Definition)

- Consider the propositional (variable free) case:

  P – ground program

  S – set of ground atoms

- Reduct P^S
  - delete each rule having a body literal `not` C with $C \in S$
  - remove all negative body literals from the remaining rules
- P^S is a definite program (and has unique Least-Model**(**P^S**)**)
- S is a stable model of P iff S = Least-Model(P^S)

Examples: Stable Models

S	P	P^S	$LM(P^S)$
{ b, f }	$b \leftarrow$ $f \leftarrow b, \texttt{not}\ eb$ $eb \leftarrow e$	$b \leftarrow$ $f \leftarrow b$ $eb \leftarrow e$	{ b, f }
{ b, eb }	$b \leftarrow$ $f \leftarrow b, \texttt{not}\ eb$ $eb \leftarrow e$	$b \leftarrow$ $eb \leftarrow e$	{ b }

- The set {b, eb} is not a stable model of P
- {b, f} is the (unique) stable model of P

More Examples

- A program can have none, one, or multiple stable models
- Program: **Two** stable models:

$$\boxed{\begin{array}{l} p \leftarrow \texttt{not}\ r \\ r \leftarrow \texttt{not}\ p \end{array}}$$

{ p }

{ r }

- Program: **No** stable models

$$\boxed{p \leftarrow \texttt{not}\ p}$$

Exercise

Determine the Stable Models

- Program P

Candidate S	P^S	is stable model?

$p \leftarrow p$

$r \leftarrow \text{not } p$

- Program P

Candidate S	P^S	is stable model?

$p \leftarrow \text{not } r$

$r \leftarrow \text{not } s$

Programs with Variables

- Variables are needed for uniform encodings
- Semantics: Herbrand models
- A rule is seen as a shorthand for the set of its ground instantiations over the **Herbrand universe** of the program
- Recall: The Herbrand universe is the set of terms built from the constants and functions in the program

Example. For the program P:

```
edge(1,2).
edge(1,3).
edge(2,4).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

the Herbrand universe is `{1,2,3,4}`

Programs with Variables

- Hence, the rule `path(X,Y) :- edge(X,Y)` in P represents:

```
path(1,1) :- edge(1,1).
path(1,2) :- edge(1,2).
path(2,1) :- edge(2,1).
path(2,2) :- edge(2,2).
path(1,3) :- edge(1,3).
...
```

- The **Herbrand base** of a program is the set ground atoms built from the predicates and the Herbrand universe of the program
- For P the Herbrand base is

  {`path(1,1), edge(1,1), path(1,2),` …}

- A Herbrand model is a subset of the Herbrand base

Programs with Variables

- The grounding of a program P yields:
  - a propositional logic program
  - built from atoms in the Herbrand base of P, denoted HB**(**P**)**
- Grounding is denoted grnd**(**P**)**
- M ⊆ HB**(**P**)** is a stable model of P if M is a stable model of grnd**(**P**)**

Example: Nonmonotonic Reasoning

- Consider the program

```
flies(X) :- bird(X), not exceptionalBird(X).
bird(tweety).
bird(sam).
```

- It has a single stable model:

`{ bird(sam), bird(tweety), flies(sam), flies(tweety) }`

- If we add an exception:

```
bird(X) :- emu(X).
exceptionalBird(X) :- emu(X).
emu(sam).
```

- then the extended program has a new unique stable model:

```
{ bird(teweety), flies(tweety),
  emu(sam), bird(sam), exceptionalBird(sam) }
```


Stable Models: Complexity

- It requires **linear time to check** whether a set of atoms is a stable model of a ground program
- It is **NP-complete to decide** whether a ground program has a stable model

Extensions to Normal Programs

- An **integrity constraint** (or simply: **constraint**) is a rule without a head:

  $\leftarrow$ B1, . . . ,Bm, not C1, . . . , not Cn

  - It can be seen as a shorthand for

    F $\leftarrow$ not F, B1, . . . ,Bm, not C1, . . . , not Cn

  - and it eliminates stable models where the body B1, . . . ,Bm, not C1, . . . , not Cn is satisfied

- **Built-in arithmetics**

```
next_state(T+1) :- current_state(T).
```

More Extensions: Encoding of Choices

- A key point in ASP
- Choices can be encoded using normal rules

  $a \leftarrow \texttt{not}\ a', b, \texttt{not}\ c$
  $a' \leftarrow \texttt{not}\ a$

  ⇨ if b holds and c doesn't, can choose whether or not to include a

- **Choice rules**, however, provide a much more intuitive encoding:

  $\{a\} \leftarrow b, \texttt{not}\ c$

  Meaning is the same as above

- Choice rules with cardinality constraints

```
2 { a1; a2; a3; a4; a5; a6 } 4
```

- Choice rules with conditional literals

```
1 { coloured(V,C):colour(C)} 1 :- vtx(V).
```

Representing Problems as Answer Set Programs

Example: Graph Colouring

```
% Problem encoding

% Generator rule
1 { coloured(V,C) : colour(C) } 1 :- vtx(V).

% Tester rule
:- edge(V,U), colour(C), coloured(V,C), coloured(U,C).

% Data
vtx(a). …
edge(a,b). …
colour(r). colour(g). …
```

Generator Rules

- The idea is to define the potential answer sets
- Typically encoded using choice rules
- Example. Choice on a subset of `{a1,...,an}` given `b`:

```
{a1;...;an} :- b.
```

The program with the fact `b` and this rule alone has 2^n stable models:

`{b}, {b, a1},…,{b, a1, …, an}`

- Example. Choice on a cardinality limited subset of `{a1,...,an}` given `b`:

```
2 {a1;...;an} 3 :- b.
```

- Typically rules with variables used

```
1 {coloured(V,C):colour(C)} 1 :- vtx(V).
```

Given a vertex `v`, choose exactly one ground atom `coloured(v,c)` such that `colour(c)` holds

Tester Rules

- Integrity constraints

```
:- a1,...,an, not b1,...,not bm.
```

- Eliminate stable models but cannot introduce new ones:
  - Let P be a program and IC a set of integrity constraints

    Then S is a stable model of P ∪ IC iff:
    - S is a stable model of P, and
    - S satisfies IC

- Example.

```
:- edge(V,U), colour(C), coloured(V,C), coloured(U,C).
```

Definitory Rules

- Often the tester and generator rules need auxiliary conditions
- This part of the encoding usually looks similar to a Prolog program
- As ASP has Prolog style rules with a similar semantics, Prolog style programming techniques can be used here for handling, e.g., data base operations (unions, joins, projections)

- Example. Join two relations: `p(X,Y) :- q(X,Z), r(Z,Y).`
- Example. The largest score `S` from a relation `score(P,S)`:

```
has_larger(S) :- score(P,S), score(P1,S1), S<S1.
max_score(S)  :- score(P,S), not has_larger(S).
```

Another Example: Reviewer Assignment

```
% Data
reviewer(r1). …
proposal(p1). …
expert(r1,p2). …
coi(r1,p1). …            % Conflict of interest

% Problem encoding

% Generator rule
% Each project proposal assigned to exactly 3 reviewers
3 { assigned(P,R) : reviewer(R) } 3 :- proposal(P).
```


Example: Reviewer Assignment (cont'd)

```
% Tester rules

% No assignments with a conflict of interest
    :- assigned(P,R), coi(P,R).
% No reviewer gets a proposal outside their expertise
    :- proposal(P), reviewer(R), assign(P,R),
       not expert(R,P).
% No reviewer has more than 8 proposals
    :- 9 { assigned(P,R) : proposal(P) }, reviewer(R).
% Each reviewer has at least 6 proposals
    :- { assigned(P,R) : proposal(P) } 5, reviewer(R).
```

Last Example: Hamiltonian Cycles

- A Hamiltonian cycle: a closed path that visits all vertices of a graph exactly once
- Input: a directed graph
  - `vtx(a).` …
  - `edge(a,b).` …
  - `initialvtx(a0).` for some vertex `a0`

Hamiltonian Cycles (cont'd)

- Candidate answer sets: subsets of edges
- Generator:

Allows zero or more instances

```
{ hc(X,Y) } :- edge(X,Y).
```

- Stable models of the generator given a graph:
  - input graph and
  - a subset of the ground facts `hc(a0,a1)` for which there is an input fact `edge(a0,a1)`

- Tester (1): each vertex has at most one chosen incoming/outgoing edge

```
:- hc(X,Y), hc(X,Z), Y!=Z.
:- hc(Y,X), hc(Z,X), Y!=Z.
```

  - Only subsets of chosen edges `hc(a,b)` that form paths (possibly closed) pass this test

Hamiltonian Cycles (cont'd)

- Tester (2): each vertex is reachable from a given initial vertex through chosen `hc(a,b)` edges:

```
:- vtx(X), not r(X).
r(Y) :- hc(X,Y), initialvtx(X).
r(Y) :- hc(X,Y), r(X).
```

  - Only Hamiltonian cycles pass tests (1) and (2)

- Given:
  - the graph, the generator rule, and the tester rules (1),(2)

- Hamiltonian cycles and stable models correspond
- A Hamiltonian cycle: atoms `hc(a,b)` in a stable model

Exercise

Exercise: Find 3-Cliques in Graphs

- Given an undirected graph G=(V,E), decide whether there is a set of 3 vertices that are pairwise adjacent
- Generate a subset of V that forms such a clique

```
% Data
```

```
% Generator rule
```

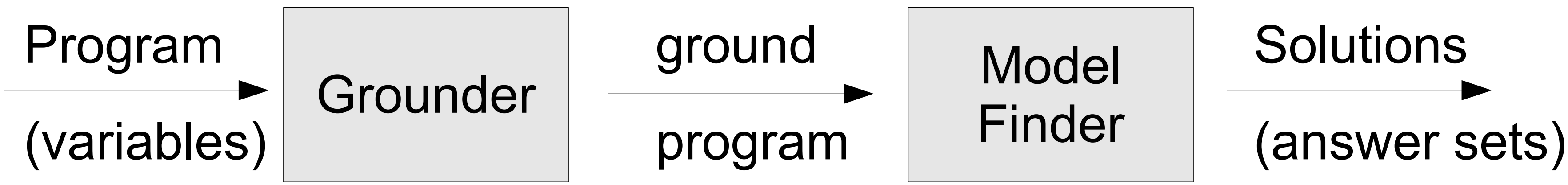
```
% Tester rule
```

ASP Solver Technology

ASP Solvers

- ASP solvers need to handle two challenging tasks
  - complex data
  - search
- Separation of concerns ⇒ two level architecture
  - **Grounding** step handles complex data:
    - Given program P with variables, generate a set of ground instances of the rules which preserves the models
    - Logic Programming and deductive database techniques
  - **Model search** for ground programs:
    - Special-purpose search procedures
    - Exploiting SAT (i.e., satisfiability) solver technology

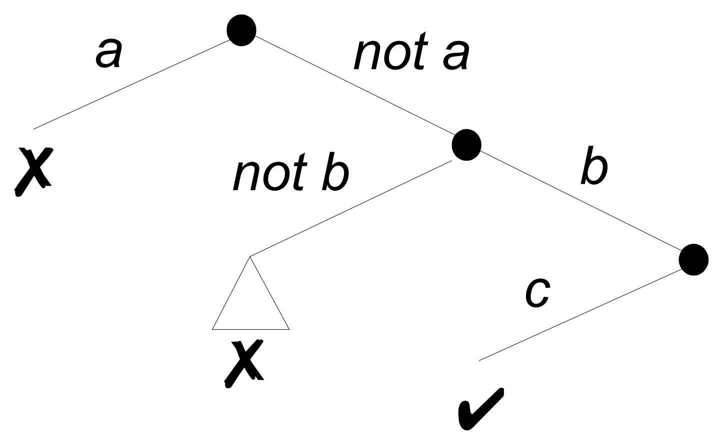
Typical ASP System Tool Chain



- Grounder
  - (deductive) DB techniques
  - built-in predicates/functions (e.g. arithmetic)
  - function symbols
- Model finder
  - SAT technology (propagation, conflict driven clause learning)
  - Special propagation rules for recursive rules
  - Built-in support for cardinality and weight constraints, optimisation

Search

- Backtracking over truth-values for atoms



- Each node consists of a model candidate (set of literals)
- **Propagation rules** are applied after each choice
  - A propagation rule extends a model candidate by one or more new literals
  - Example. Given $\boxed{r \leftarrow p,\ \mathtt{not}\ q}$ and candidate {p,`not` r}, derive q
  - Propagation rules need to be **correct**: If *L* is derived from model candidate *A* then *L* holds in every stable model compatible with *A*

Example Propagation Rules

(R_1) All literals in the body of a clause are true ⇨ head must be true

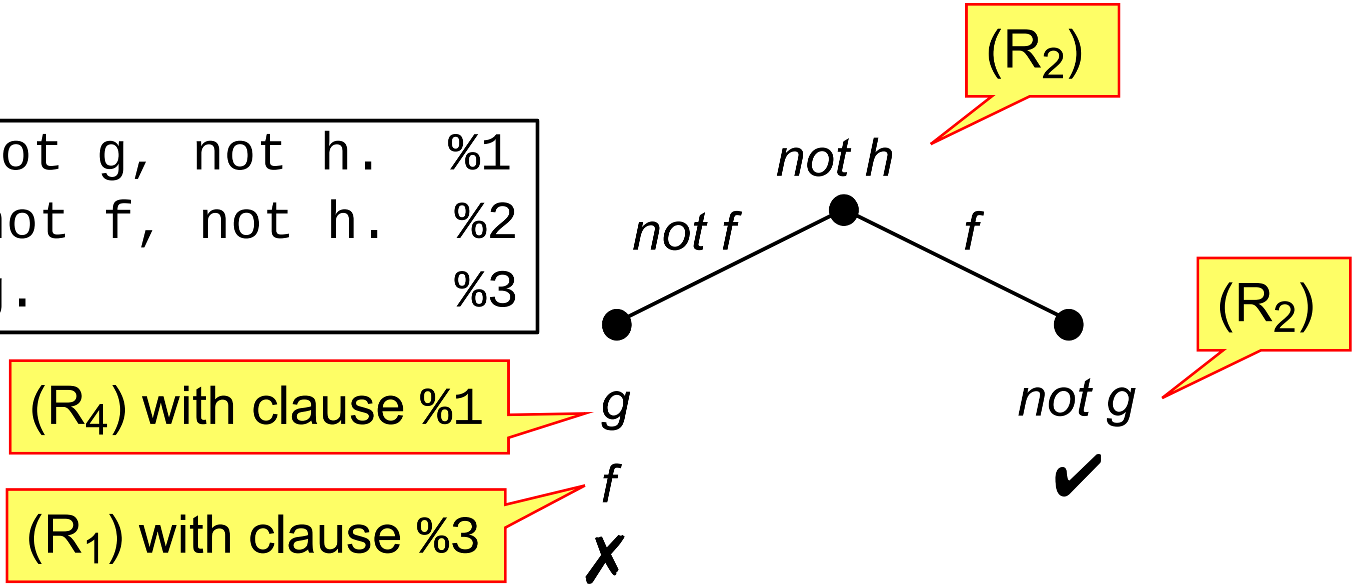
(R_2) In all clauses for p at least one body literal is false ⇨ p must be false

(R_3) p is true and there is only one clause for p ⇨ all body literals must be true

(R_4) p is false and there is a clause for p in which all body literals are true except L ⇨ L must be false

Example.

```
f :- not g, not h.   %1
g :- not f, not h.   %2
f :- g.              %3
```



- The right branch determines the only stable model, S={f}

Some Common Search Heuristics

Heuristics to select the next atom for splitting the search tree:

- an atom with the maximal number of occurrences in clauses of minimal size
- an atom with the maximal number of propagations after the split
- an atom with the smallest remaining search space after split + propagation

Exercise

Compute All Stable Models

```
h(0).                           %1
e(0) :- not b(0).               %2
b(0) :- not e(0).               %3
:- e(0), not h(0).              %4
:- b(0), h(0).                  %5
h(1) :- b(0).                   %6
h(1) :- h(0), not e(0).         %7
n(1) :- e(0).                   %8
n(1) :- n(0).                   %9
:- not n(1).                    %10
```

Using ASP for Planning and Game Playing

Planning/Single-Player Games with ASP: Overview

- Devise a logic program such that **stable models correspond to plans**:
  - of length at most n
  - that are valid
  - and that reach the goal

- Generator:
  execution sequences of operator instances from time 0 to n

- Tester: eliminate those sequences that
  - contain actions whose preconditions are not satisfied
  - do not reach the goal

Preliminaries

Cake-Example revisited

- Two state properties: `have`, `eaten`
- Action `eat`, which is possible if `have` is true; effect: `eaten`
- Action `bake`, which is possible if `have` is false; effect: `have`
- Initially, `have` is true
- The goal is to make `eaten` true

- Add to each state feature and action a **time argument**
  - `p(T)` – p is true at time T
  - `a(T)` – action a is taken at time T

- Initial state:

```
have(0).
```

Planning with ASP – Preconditions & Effects

- Plan length (τ = search depth):

```
time(0..τ).
```

Stands for: `time(0).`
`time(1).`
`...`
`time(τ).`
where τ a number ≥ 0

- Generator: one action at a time

```
1 { bake(T); eat(T) } 1 :- time(T).
```

- Tester (1): Action preconditions

```
:- eat(T), not have(T).
:- bake(T), have(T).
```

`eat` possible if `have` **true**
`bake` possible if `have` **false**

- Auxiliary rules: Action effects

```
have(T+1)  :- bake(T), time(T).
have(T+1)  :- have(T), not eat(T), time(T).
eaten(T+1) :- eat(T), time(T).
eaten(T+1) :- eaten(T), time(T).
```

Condition under which `have` remains **unchanged**

Goal Conditions

- Tester (2):
  exclude models where the goal has not been reached at time τ+1

```
% Goal
:- not eaten(τ+1).
```

Remember:
the goal is to make eaten **true**

Plans

```
time(0..0).           % equivalent to just "time(0)."
have(0).
1 { eat(T); bake(T) } 1 :- time(T).
:- eat(T), not have(T).
:- bake(T), have(T).
have(T+1)   :- bake(T), time(T).
have(T+1)   :- have(T), not eat(T), time(T).
eaten(T+1) :- eat(T), time(T).
eaten(T+1) :- eaten(T), time(T).
:- not eaten(1).
```

cf Exercise Slide 41

- Plans correspond to answer sets:
  - there is a stable model iff there is a valid sequence of n moves that leads to the goal
- A valid plan:
  - all action instances in the stable model. Here: `eat(0)`

General Game Playing

General Game Players are systems

- able to understand formal descriptions of arbitrary games
- able to learn to play these games effectively

Translation: They don't know the rules until the game starts

Unlike specialised game players (e.g. Deep Blue), they do not use algorithms designed in advance for specific games

International Activities

Websites – games.stanford.edu ggp.org general-game-playing.de

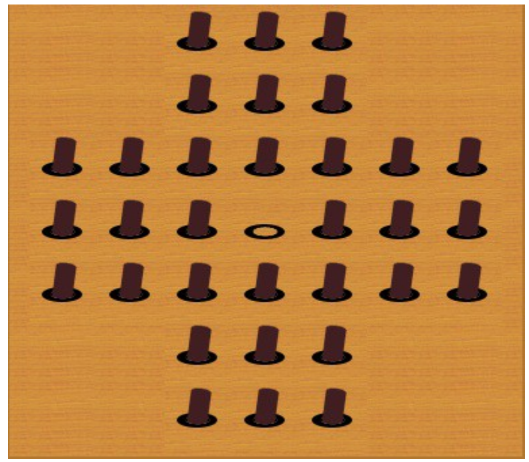
- Games
- Game Manager
- Reference Players
- Development Tools
- Literature

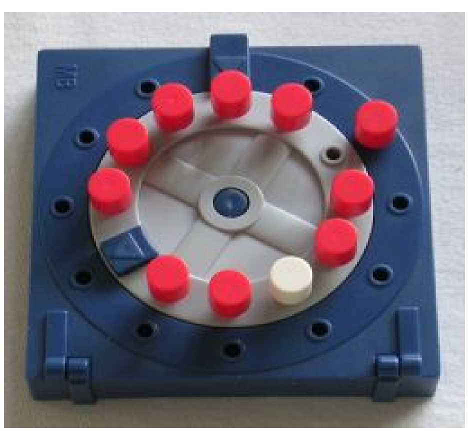
World Cup, administered by Stanford University

- 2005 – Cluneplayer (USA)
- 2006 – Fluxplayer (Germany)
- 2007, 2008, 2012 – Cadiaplayer (Iceland)
- 2009, 2010 – Ary (France)
- 2011, 2013 – TurboTurtle (USA)
- 2014 – Sancho (USA)
- 2015 – Galvanise (USA)

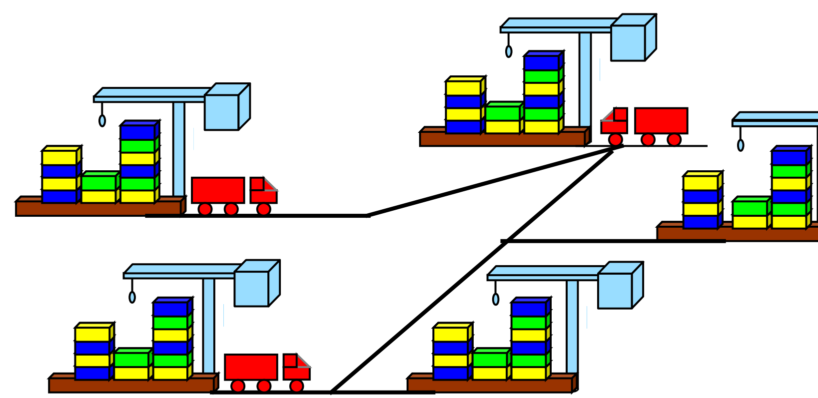
Single-Player General Game Playing

- In single-player games, the player has no direct opponent

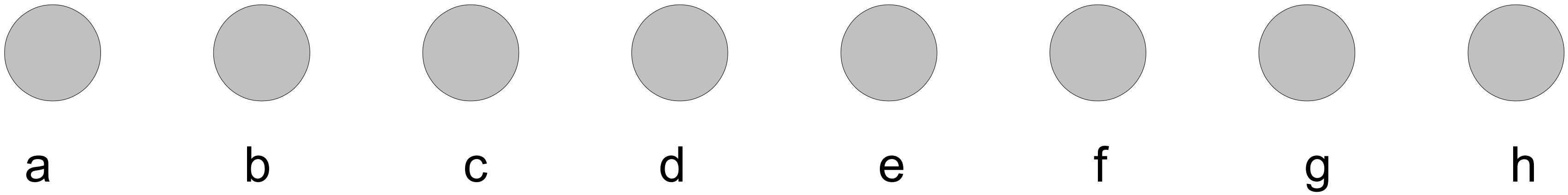




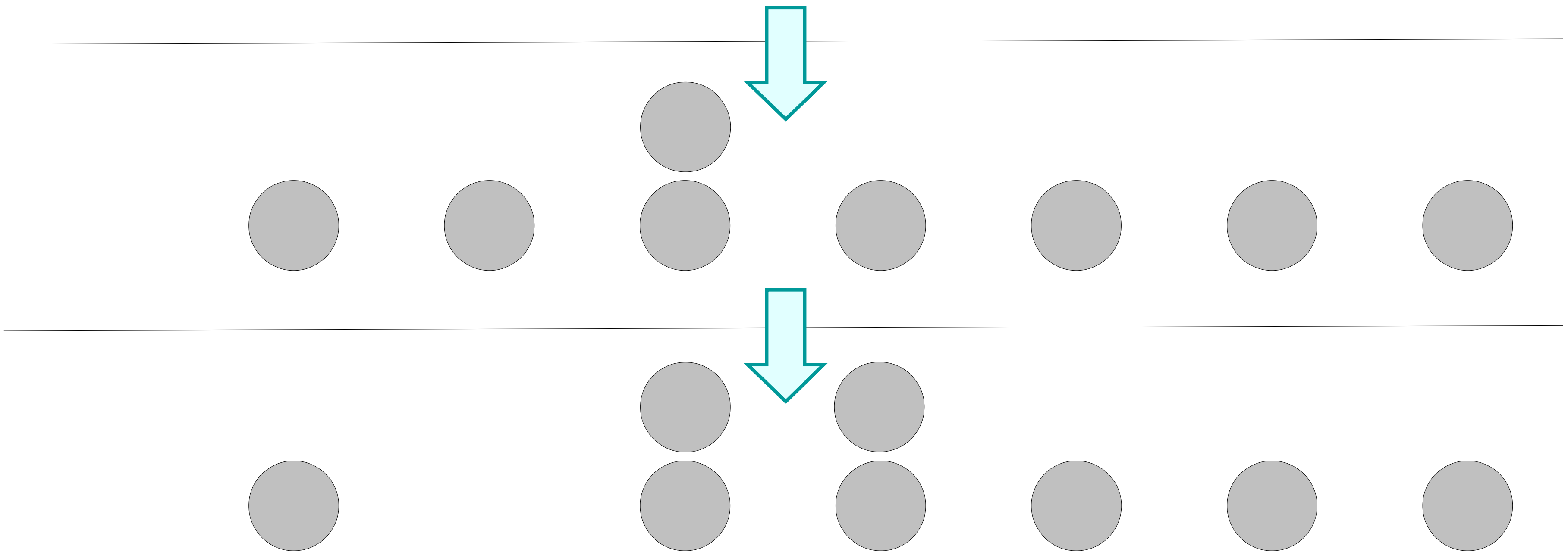
32	35	30	25	08	05	50	55
29	24	33	36	51	56	07	04
34	31	26	09	06	49	54	57
23	28	37	12	01	52	03	48
38	13	22	27	10	47	58	53
19	16	11		61	02	43	46
14	39	18	21	44	41	62	59
17	20	15	40	63	60	45	42



Example: Coin Game



- Jump with singleton coin over two coins onto singleton coin
- End up with 4 stacks of two coins each



dead end

A KR Language for Games: Game Description Language (GDL)

Some Facts

```
(role robot)

(init (cell a 1))
(init (cell b 1))
...
(init (cell h 1))
(init (step 1))
```

Some Rules

```
(<= (legal robot (jump ?x ?y))
    (true (cell ?x 1))
    (true (cell ?y 1))
    (twobetween ?x ?y))

(<= (next (cell ?x 0))
    (does robot (jump ?x ?y)))
(<= (next (cell ?y 2))
    (does robot (jump ?x ?y)))
```

Variable

prefix syntax

All highlighted expressions are pre-defined keywords in GDL

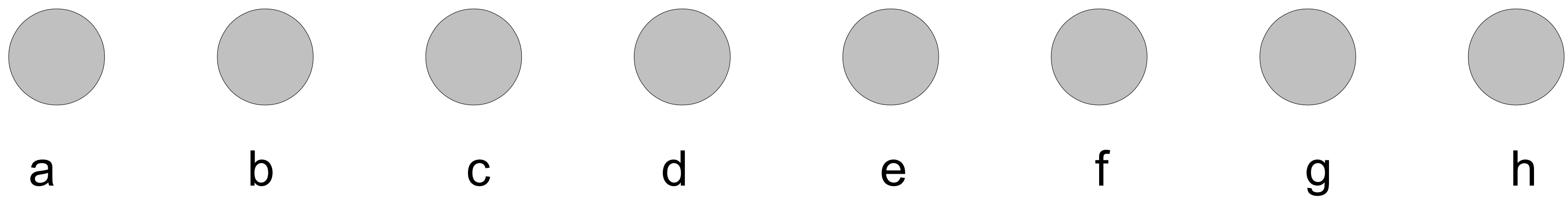
GDL Keywords

In GDL, a game is described by a logic program written in prefix notation

GDL uses the following predicates as keywords

- `role(r)` means that `r` is a role (i.e. a player) in the game
- `init(f)` means that `f` is true in the initial position (state)
- `true(f)` means that `f` is true in the current state
- `does(r,a)` means that role `r` does action `a` in the current state
- `next(f)` means that `f` is true in the next state
- `legal(r,a)` means that it is legal for `r` to play `a` in the current state
- `goal(r,v)` means that `r` gets goal value `v` in the current state
- `terminal` means that the current state is a terminal state
- `distinct(s,t)` means that terms `s` and `t` are syntactically different

The Coin Game in GDL

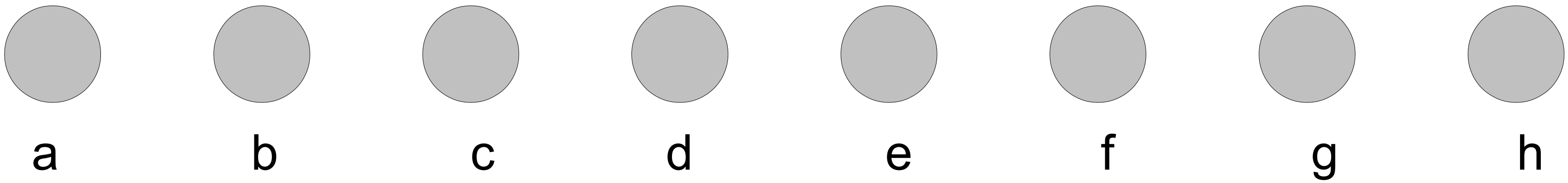



```
(role robot)

(init (cell a 1))
(init (cell b 1))
(init (cell c 1))
(init (cell d 1))
(init (cell e 1))
(init (cell f 1))
(init (cell g 1))
(init (cell h 1))
(init (step 1))
```

```
(succ a b)
(succ b c)
(succ c d)
(succ d e)
(succ e f)
(succ f g)
(succ g h)

(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
```

```
(<= terminal
    (not anylegalmove))

(<= anylegalmove
    (legal robot ?m))

(<= (goal robot 100)
    (true (step 5)))
(<= (goal robot 0)
    (true (cell ?x 1)))
```


The Coin Game in GDL (cont'd)




```
(<= (legal robot (jump ?x ?y))
    (true (cell ?x 1))
    (true (cell ?y 1))
    (twobetween ?x ?y)
(<= (legal robot (jump ?y ?x))
    (legal robot (jump ?x ?y)))

(<= (next (cell ?x 0))
    (does robot (jump ?x ?y)))
(<= (next (cell ?y 2))
    (does robot (jump ?x ?y)))
(<= (next (cell ?x ?c))
    (true (cell ?x ?c))
    (does robot (jump ?y ?z))
    (distinct ?x ?y)
    (distinct ?x ?z))

(<= (next (step ?y))
    (true (step ?x))
    (succ ?x ?y))
```

```
(<= (twobetween ?x ?y))
    (succ ?x ?z)   (true (cell ?z 2))
    (zerobetween ?z ?y))
(<= (twobetween ?x ?y))
    (succ ?x ?z)   (true (cell ?z 1))
    (onebetween ?z ?y))
(<= (twobetween ?x ?y))
    (succ ?x ?z)   (true (cell ?z 0))
    (twobetween ?z ?y)))
(<= onebetween ?x ?y))
    (succ ?x ?z)   (true (cell ?z 1))
    (zerobetween ?z ?y))
(<= (onebetween ?x ?y))
    (succ ?x ?z)   (true (cell ?z 0))
    (onebetween ?z ?y))
(<= (zerobetween ?x ?y)
    (succ ?x ?y))
(<= (zerobetween ?x ?y)
    (succ ?x ?z)   (true (cell ?z 0))
    (zerobetween ?z ?y))
```

Valid GDL Descriptions

Conditions that GDL game descriptions must satisfy:

- `role(r)` only appears in facts
- `init(f)` only appears in the head of clauses and does not depend on `true`, `legal`, `does`, `next`, `terminal`, or `goal`
- `true(f)` only appears in the body of clauses
- `does(r,a)` only appears in the body of clauses, and none of `legal`, `terminal` or `goal` depends on `does`
- `next(f)` only appears in the head of clauses
- `distinct(s,t)` only appears in the body of clauses

Solving Single-Player Games: From GDL to ASP

1. Replace `(init φ)` by `holds(φ,0)`
2. Replace `(true φ)` by `holds(φ,T)`
3. Replace `(next φ)` by `holds(φ,T+1)`
4. Replace all other $(p\ \ t_1\ \dots\ t_n)$ by $p(t_1,\dots,t_n,T)$

Result for the Coin Game, written in ASP notation:

```
holds(cell(a,1),0).
...
holds(cell(h,1),0).
holds(step(1),0).

legal(robot,jump(X,Y),T) :- holds(cell(X,1),T),
                             holds(cell(Y,1),T),
                             twobetween(X,Y,T).
holds(cell(X,0),T+1) :- does(robot,jump(X,Y),T).
holds(cell(Y,2),T+1) :- does(robot,jump(X,Y),T).
...
```

From GDL to ASP (cont'd)

1. One move made in every time step
2. Don't make a move that is not legal
3. Terminal position eventually reached
4. Goal value achieved in terminal position

ranges over all moves in the game

```
1 { does(role,M,T) : move_domain(M) } 1 :- time(T),
                                            not terminal(T).

:- does(role,M,T), not legal(role,M,T).

:- 0 { terminal(T) : time(T) } 0.

:- terminal(T), not terminal(T-1),
   not goal(role,100,T).
```

Every answer set corresponds to a game solution and vice versa, e.g.

`jump(d,g),0   jump(f,b),1   jump(c,a),2   jump(e,h),3`

Summary

- Stable models = answer sets
- How to write ASPs
- ASP solver technology
- Planning and solving general single-player games with ASPs

Available ASP solvers

- Grounder
  - Gringo `http://potassco.sourceforge.net/`
  - Lparse `http://www.tcs.hut.fi/Software/smodels/`
- Model finder
  - Clasp `http://potassco.sourceforge.net/`
  - Assat `http://assat.cs.ust.hk/`
- `Clingo = Gringo + Clasp`