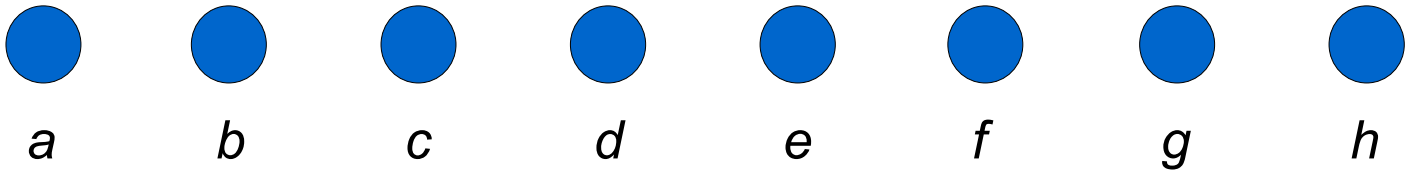
Planning

- Blind Search for Planning
- Plan Skeletons
- Pruning Techniques
- Planning with Preferences

Example: Coin Game



- Jump with one coin over two coins onto one coin
- After 4 moves, obtain 4 stacks with 2 coins each

Game Description: Initial Situation

$$Holds(f, S_0) \equiv f = Cell(a, 1) \vee f = Cell(b, 1) \vee f = Cell(c, 1) \vee f = Cell(d, 1) \vee$$
$$f = Cell(e, 1) \vee f = Cell(f, 1) \vee f = Cell(g, 1) \vee f = Cell(h, 1)$$

Game Description: Preconditions

$$Poss(Jump(x, y), s, t) \equiv t = Do(Jump(x, y), s) \wedge$$
$$Holds(Cell(x, 1), s) \wedge Holds(Cell(y, 1), s) \wedge$$
$$CoinsBetween(x, y, s) = 2$$

Game Description: Effects

$$Poss(Jump(x, y), s, t) \supset$$
$$(\forall f)[f = Cell(x, 0) \vee f = Cell(y, 2) \vee (Holds(f, s) \wedge f \neq Cell(x, 1) \wedge f \neq Cell(y, 1))$$
$$\supset Holds(f, t)]$$
$$\wedge$$
$$(\forall f)[f = Cell(x, 1) \vee f = Cell(y, 1) \vee (\neg Holds(f, s) \wedge f \neq Cell(x, 0) \wedge f \neq Cell(y, 2))$$
$$\supset \neg Holds(f, t)]$$

Game Description: Goal

$$Goal(s) \stackrel{\text{def}}{=} (\forall x, v)\,(Holds(Cell(x, v), s) \supset v = 0 \vee v = 2)$$

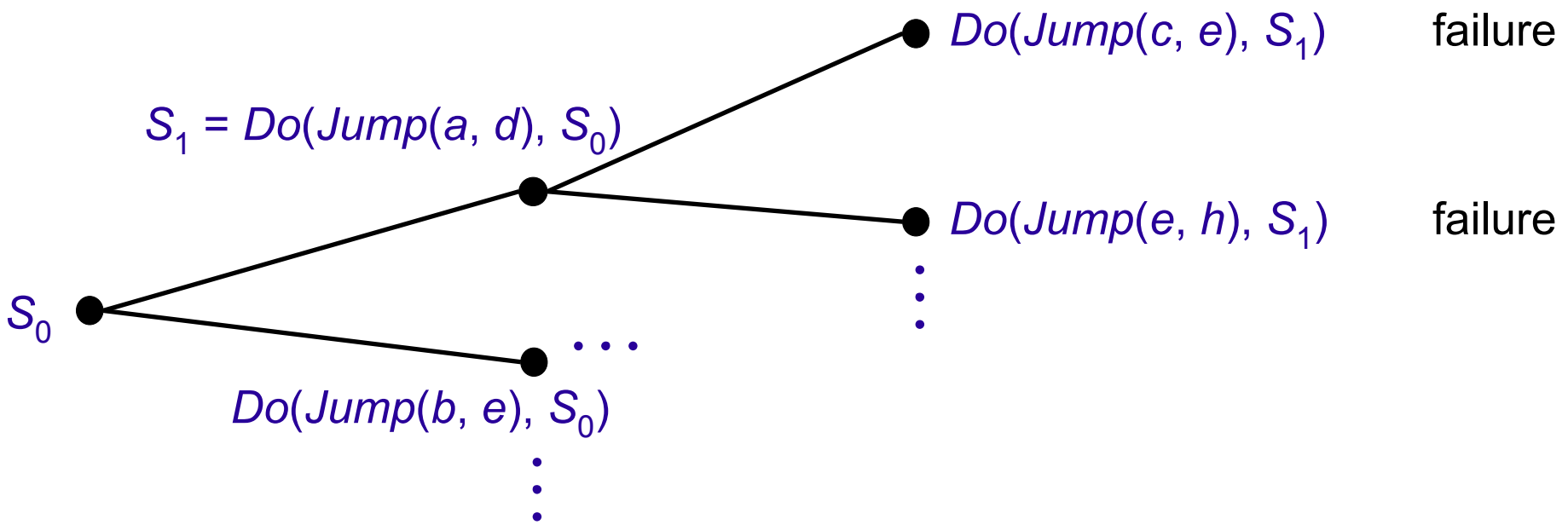
A Simple GOLOG Planning Program

while ¬Goal **do**

$\pi a.\ a$

endWhile

If executed offline, this program defines the whole search space:



Efficient Blind Search

with

Answer Set Programs

The Game Description Language GDL

Game-independent keywords

```
init(fluent)
true(fluent)
does(player,action)
next(fluent)
legal(player,action)
goal(player,value)
terminal
```

“Coin” Planning Problem in GDL (1)

```
succ(a,b). succ(b,c). ... succ(g,h).

init(cell(a,1)).
init(cell(Y,1)) :- succ(X,Y).

next(cell(X,0)) :- does(player,jump(X,Y)).
next(cell(Y,2)) :- does(player(jump(X,Y)).
next(cell(X,C)) :- does(player,jump(Y,Z)), true(cell(X,C)),
                   distinct(X,Y), distinct(X,Z).

terminal :- not continuable.
continuable :- legal(player,M).
goal :- not lonelycoin.
lonelycoin :- true(cell(X,1)).
```

“Coin” Planning Problem in GDL (2)

```
legal(P,jump(X,Y),T) :- holds(cell(X,1),T),
                        holds(cell(Y,1),T), twobetween(X,Y).
legal(P,jump(X,Y),T) :- holds(cell(X,1),T),
                        holds(cell(Y,1),T), twobetween(Y,X).

zerobetween(X,Y) :- succ(X,Y).
zerobetween(X,Y) :- succ(X,Z), true(cell(Z,0)),
                    zerobetween(Z,Y).
onebetween(X,Y) :- succ(X,Z), true(cell(Z,0)),
                   onebetween(Z,Y).
onebetween(X,Y) :- succ(X,Z), true(cell(Z,1)),
                   zerobetween(Z,Y).
twobetween(X,Y) :- succ(X,Z), true(cell(Z,0)),
                   twobetween(Z,Y).
twobetween(X,Y) :- succ(X,Z), true(cell(Z,1)),
                   onebetween(Z,Y).
twobetween(X,Y) :- succ(X,Z), true(cell(Z,2)),
                   zerobetween(Z,Y).
```

Adding Linear Time to GDL

1. Each occurrence of `init(f)` is replaced by `holds(f,1)`

2. Each occurrence of `true(f)` is replaced by `holds(f,T)` and each occurrence of `next(f)` by `holds(f,T+1)`

3. Each occurrence of an atom $p(\ t_1, ..., t_n)$ is replaced by $p(\ t_1, ..., t_n,$ `T`$)$ provided that $P \notin$ {`init`, `true`, `next`, `distinct`}

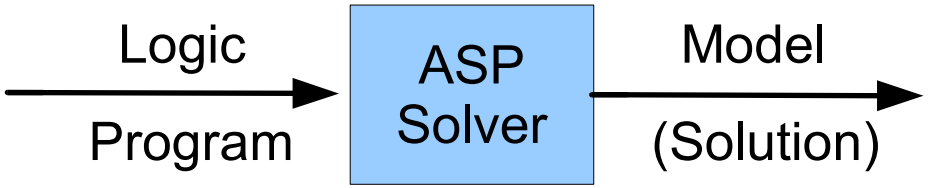
Example (Part 1)

```
succ(a,b,T). succ(b,c,T). ... succ(g,h,T).

holds(cell(a,1),1).
holds(cell(Y,1),1) :- succ(X,Y,1).

holds(cell(X,0),T+1) :- does(player,jump(X,Y),T).
holds(cell(Y,2),T+1) :- does(player(jump(X,Y),T).
holds(cell(X,C),T+1) :- does(player,jump(Y,Z),T),
                        holds(cell(X,C),T),
                        distinct(X,Y), distinct(X,Z).

terminal(T) :- not continuable(T).
continuable(T) :- legal(player,M,T).
goal(T) :- not lonelycoin(T).
lonelycoin(T) :- holds(cell(X,1),T).
```

Example (Part 2)

```
legal(P,jump(X,Y),T) :- holds(cell(X,1),T),
                        holds(cell(Y,1),T), twobetween(X,Y,T).
legal(P,jump(X,Y),T) :- holds(cell(X,1),T),
                        holds(cell(Y,1),T), twobetween(Y,X,T).

zerobetween(X,Y,T) :- succ(X,Y,T).
zerobetween(X,Y,T) :- succ(X,Z,T), holds(cell(Z,0),T),
                      zerobetween(Z,Y,T).
onebetween(X,Y,T) :- succ(X,Z,T), holds(cell(Z,0),T),
                     onebetween(Z,Y,T).
onebetween(X,Y,T) :- succ(X,Z,T), holds(cell(Z,1),T),
                     zerobetween(Z,Y,T).
twobetween(X,Y,T) :- succ(X,Z,T), holds(cell(Z,0),T),
                     twobetween(Z,Y,T).
twobetween(X,Y,T) :- succ(X,Z,T), holds(cell(Z,1),T),
                     onebetween(Z,Y,T).
twobetween(X,Y,T) :- succ(X,Z,T), holds(cell(Z,2),T),
                     zerobetween(Z,Y,T).
```

Answer Set Programming (ASP)



- Very efficient problem solving technique (without heuristics)
- Roots: logic programming
- Solutions = Answer sets to logic program

ASP vs. Prolog

- Prolog not directly suitable for ASP

  - Models vs. proofs + answer substitiutions

  - Prolog not entirely declarative

- Stable model semantics: alternative semantics for negation-as-failure

- Existing ASP Systems: CLINGO, SMODELS, DLV and others

Stable Models: Example

```
does(player,a,1) :- not does(player,b,1).
does(player,b,1) :- not does(player,a,1).
holds(bad,2)     :- does(player,a,1).
goal(2)          :- not holds(bad,2).
f                :- not goal(2), not f.
```

The only stable model is {`does(player,b,1)`, `goal(2)`}

- {`does(player,a,1)`, `goal(2)`} is not a stable model (it's not a model)
- {`does(player,a,1)`, `holds(bad,2)`, `goal(2)`} is not a stable model (`goal(2)` included for no reason)

Stable Models: Definition

Consider a program P of ground clauses

$$A \text{ :- } B_1, \ldots, B_m, \texttt{not } C_1, \ldots, \texttt{not } C_n \qquad (m \geq 0, n \geq 0)$$

Let S be a set of ground atoms.

- Reduct $P^S \stackrel{\text{def}}{=}$
  - delete each clause with some $\texttt{not } C_i$ s.th. $C_i \in S$
  - delete each $\texttt{not } C_i$ s.th. $C_i \notin S$
- S stable model $\stackrel{\text{def}}{=}$ S = least-model(P^S)

Properties

- Programs can have multiple stable models.

```
p1 :- not q1.    q1 :- not p1.
 ⋮                ⋮
pn :- not qn.    qn :- not pn.
```

  This program has 2^n stable models

- Programs can have no stable models.

```
p :- not q.
q :- p.
```

Programs with Variables and Functions

- Clause seen as shorthand for all its ground instances

```
holds(cell(X,0),T+1) :- does(player,jump(X,Y),T).
```

stands for

```
holds(cell(a,0),2) :- does(player,jump(a,d),1).
holds(cell(b,0),3) :- does(player,jump(b,d),2).
...
```

- Constraint

$:- B_1, ..., B_m,$ `not` $C_1, ..., $ `not` C_n

shorthand for $f :- B_1, ..., B_m,$ `not` $C_1, ..., $ `not` $C_n,$ `not` f

- Weight atom

`m {` $p : d$ `} n`

means $\geq$ `m` and $\leq$ n instances of p under condition d are in the answer

Solving Planning Problems with ASP

1. Exactly one action at a time
2. Each action is possible when executed
3. Planning goal achieved in the end

```
1{does(player,M,T) : move_domain(M)}1 :- not terminal(T).
:- does(player,M,T), not legal(player,M,T).
goal :- goal(T).
:- not goal.
```

Example

```
move_domain(jump(X,Y)) :- co(X), co(Y).

co(a). ... co(h).
```

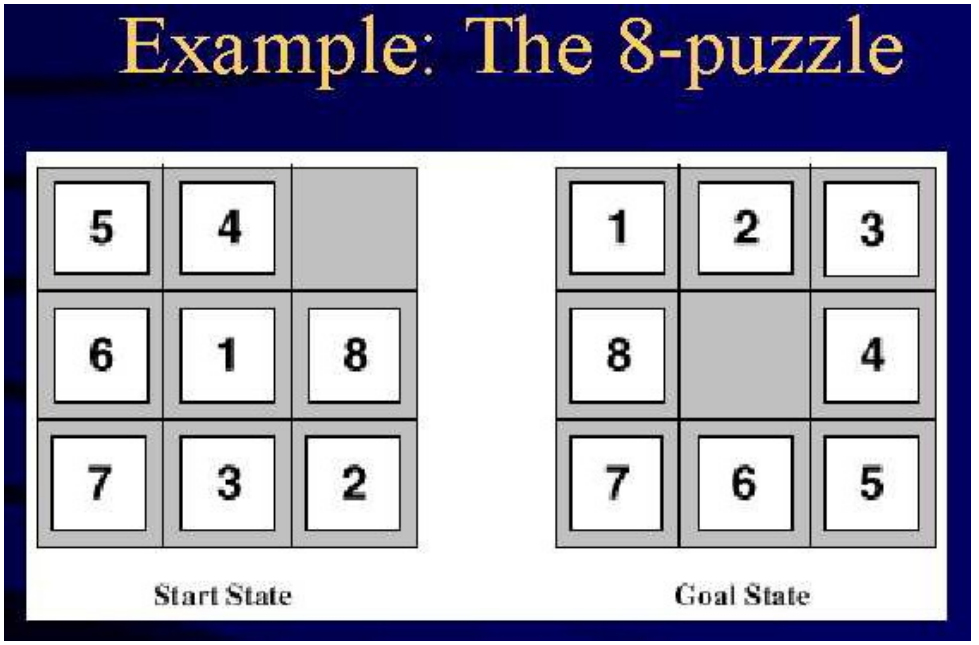
Add this to the program on slides 13 and 14 along with the search clauses from the previous slide.

```
does(player,jump(d,g),1)  does(player,jump(f,b),2)

does(player,jump(c,a),3)  does(player,jump(e,h),4)

terminal(5)               goal(5)

...
```

ASP in Practice

ASP is an efficient method for blind search to solve planning problems.

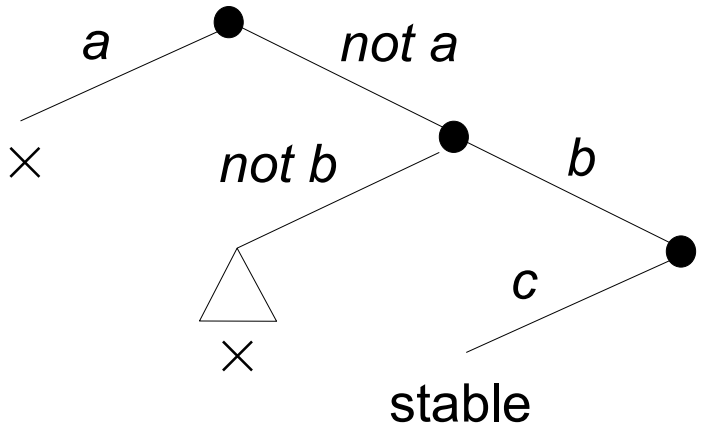
For example, 8-puzzle solved in ~1 minute.



This is due to an efficient combination of search and propagation.

Search

- Backtracking over truth-values for atoms



- Each node consists of a model candidate (set of literals)
- Propagation rules are applied after each choice

Propagation Rules

- A propagation rule extends a model candidate by one or more new literals.
- Propagation rules need to be correct: If *L* is derived from model candidate *A* then *L* holds in every stable model compatible with *A*.

Propagation Rule “Upper Bound”

Consider program *P* and candidate model *A*

Let *P′* be all clauses in *P*

- whose body is not false under *A*
- without negative body literals

If $p \notin$ least-model (*P′*) derive not *p*

P : `p2 :- p1, not q1.` *A* : {`q2`} *P′* : `p2 :- p1.`

`p1 :- p2, not q1.` `p1 :- p2.`

`p2 :- not q2.`

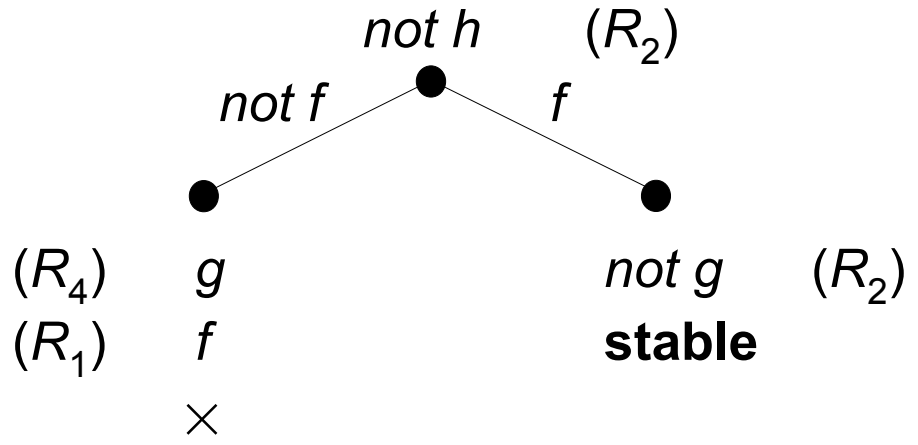
Derive: `not p1, not p2, not q1, not q2`

Local Propagation Rules

	Only clauses for q	A	Derive
(R_1)	$q \leftarrow p_1, \mathit{not}\ p_2$	$p_1, \mathit{not}\ p_2$	q
(R_2)	$q \leftarrow p_1, \mathit{not}\ p_2$ $q \leftarrow p_3, \mathit{not}\ p_4$	$p_2, \mathit{not}\ p_3$	$\mathit{not}\ q$
(R_3)	$q \leftarrow p_1, \mathit{not}\ p_2$	q	$p_1, \mathit{not}\ p_2$
(R_4)	$q \leftarrow p_1, \mathit{not}\ p_2$	$\mathit{not}\ q, p_1$	p_2

Example

```
f :- not g, not h
g :- not f, not h
f :- g
```



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