Planning

—

KRR for Agents in Dynamic Environments

Overview

- Last week discussed **Decision Making** in some very general settings: Markov process, MDP, HMM, POMDP.
- This week look at a practical application of these ideas in a more restricted setting.
- **Planning** (or **AI Planning**) is about agents that execute actions to reach goals (e.g., a robot delivering an item).
- Note: ties closely to **Reasoning about Actions** (Week 9).

Some Dictionary Definitions of “Plan”

plan *n.*

1. A scheme, program, or method worked out beforehand for the accomplishment of an objective: *a plan of attack.*
2. A proposed or tentative project or course of action: *had no plans for the evening.*

[a representation] of future behaviour … usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents.

– Austin Tate, *MIT Encyclopaedia of the Cognitive Sciences*, 1999

Classical Planning

- Deterministic environment; complete information
- Representations for classical planning
- Solving planning problems
  - Modern heuristics for state-space planning
  - Answer Set Programming and Graphplan

Background reading

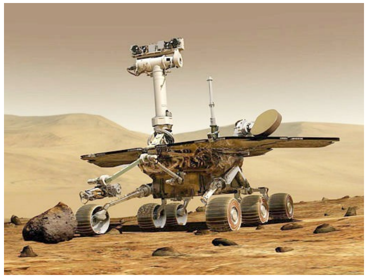
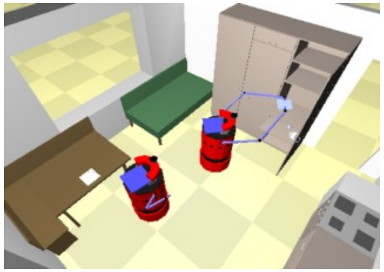
Automated Planning, Malik Ghallab, Dana Nau, Paolo Traverso, Morgan Kaufmann 2004. Chapters 1, 2, 4 & 6

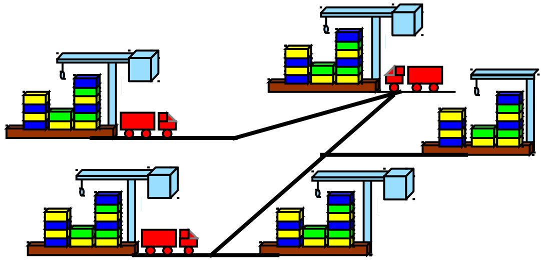
Artificial Intelligence: A Modern Approach, Stuart Russell, Peter Norvig, Prentice Hall 2003 (2nd Edition). Chapter 11.

Note: I think Chapter 10 for 3rd Edition of Russell and Norvig.

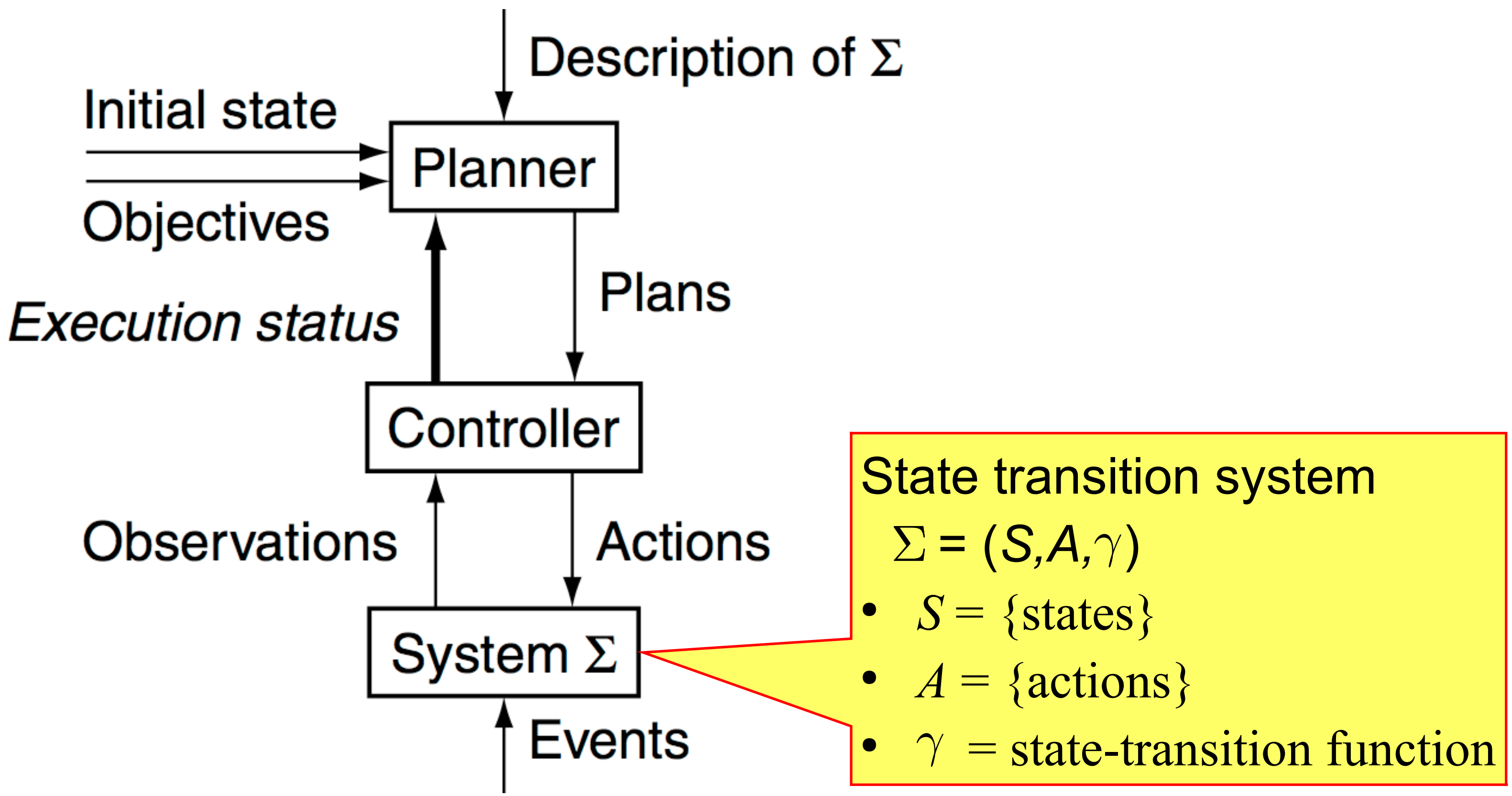
Planning Overview

- Dynamic environment.
- One or more agents: (virtual) agents, robots.
- Agents take actions that change the environment.
- Agents have goals that they want to achieve.
- What sequence of actions will allow the agent to achieve its goals?
- Blocksworld is a prototypical example of a classical planning problem.



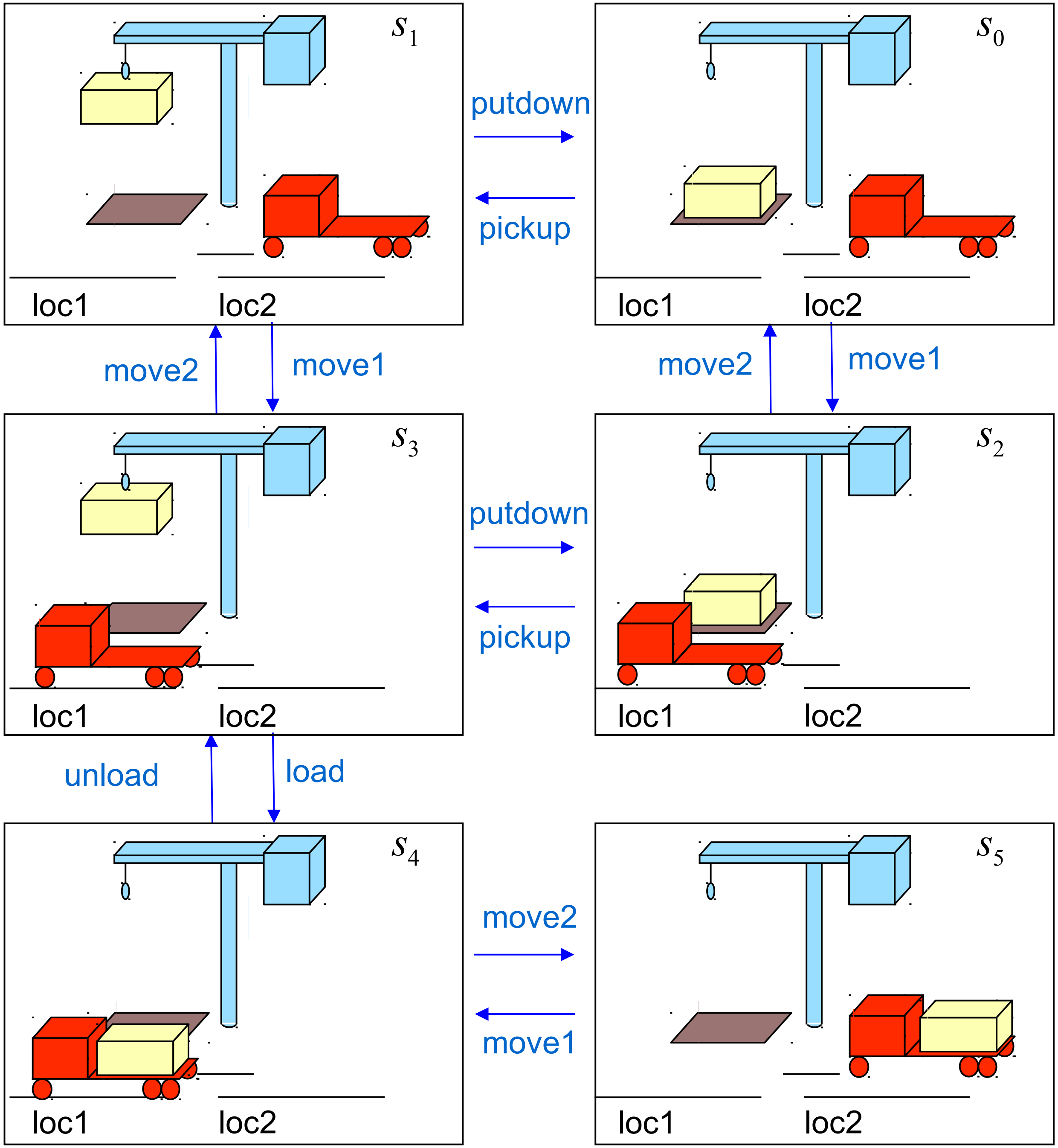
Planning for an Agent/Robot in a Dynamic World



- Σ is an abstraction that deals only with the aspects that the planner needs to reason about

Example

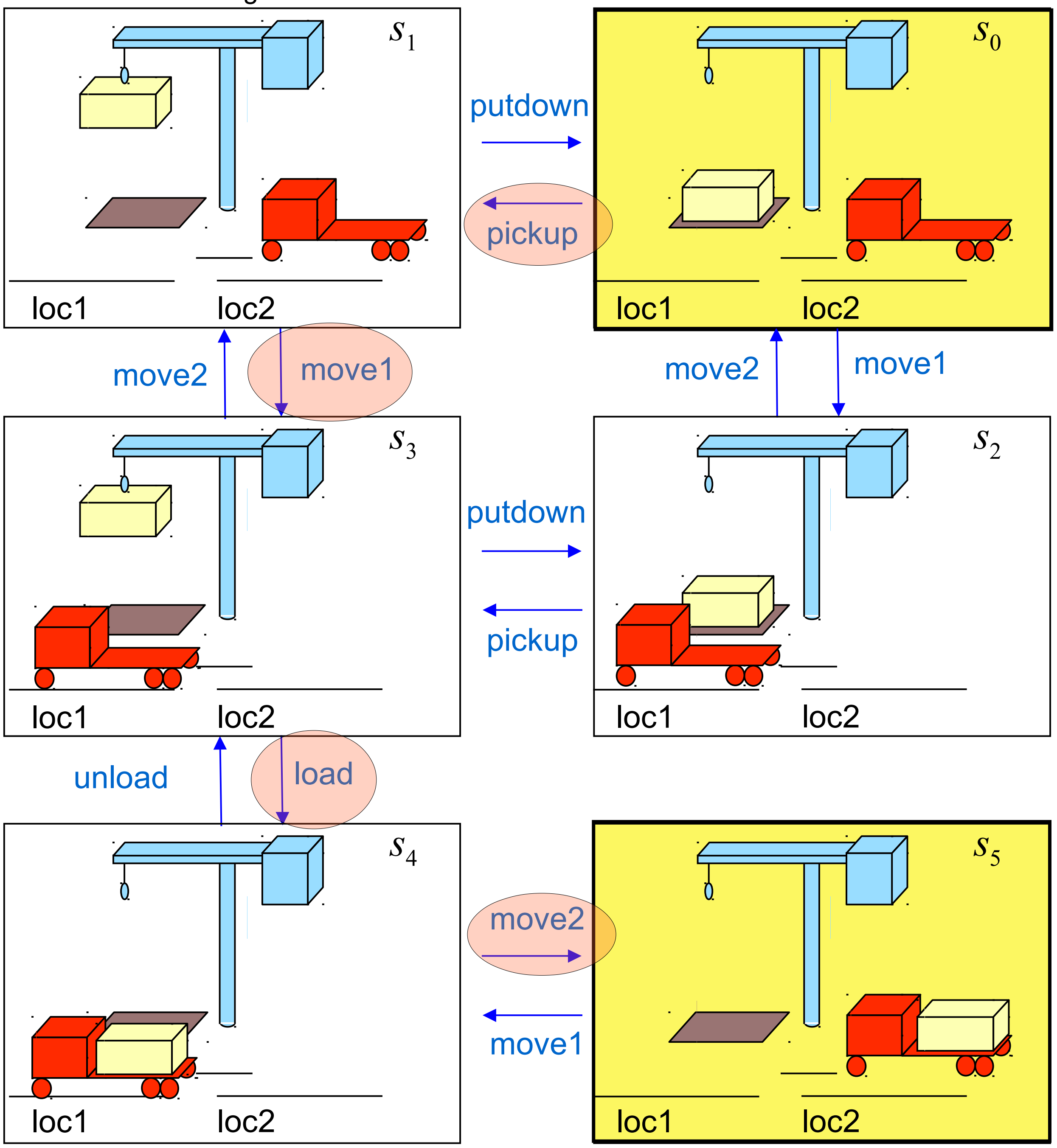
- Example $\Sigma = (S, A, \gamma)$:
  - $S = \{s_0, \ldots, s_5\}$
  - A = {move1, move2, putdown, pickup, load, unload}
  - γ : see the arrows



Dock Worker Robots (DWR) example

Example

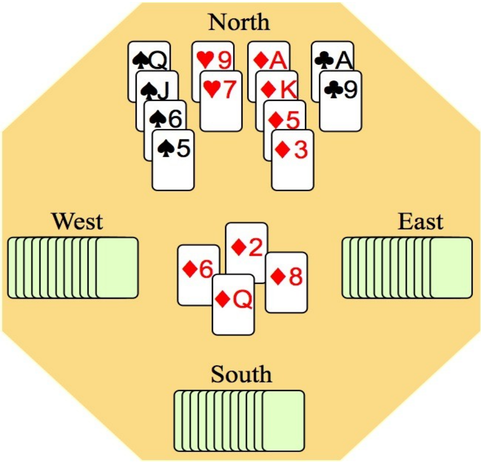
- **Classical plan**: a sequence of actions
  ⟨pickup, move1, load, move2⟩

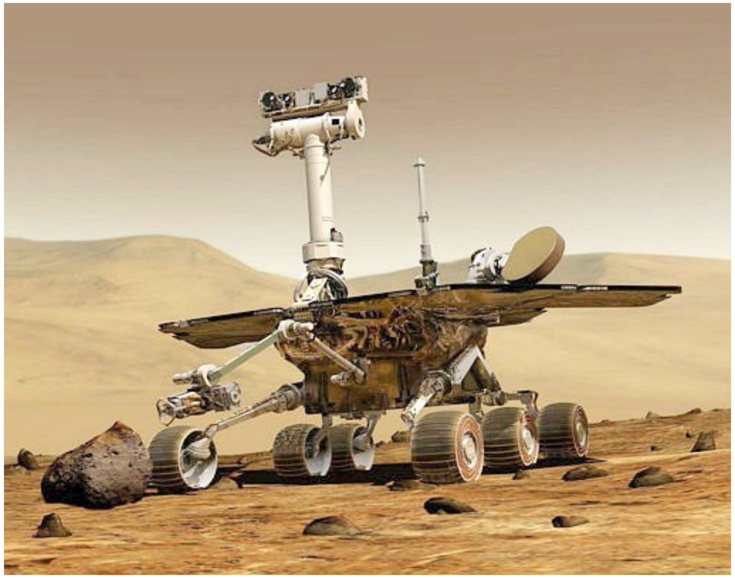


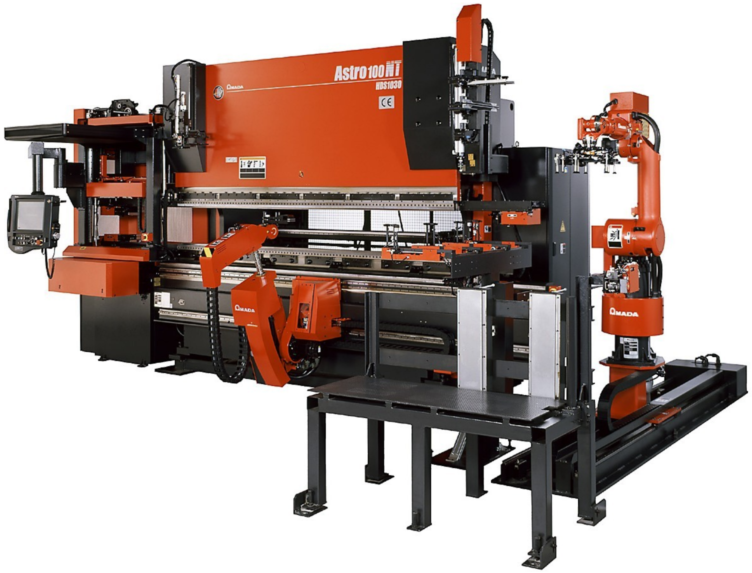
Dock Worker Robots (DWR) example

Domain-Specific Planners

- Many successful real-world planning systems work this way
  - Mars exploration, sheet-metal bending, playing bridge, etc.
- Often use problem-specific techniques that are difficult to generalise to other planning domains
- For example, encodes the knowledge of domain experts (e.g., computer poker player)

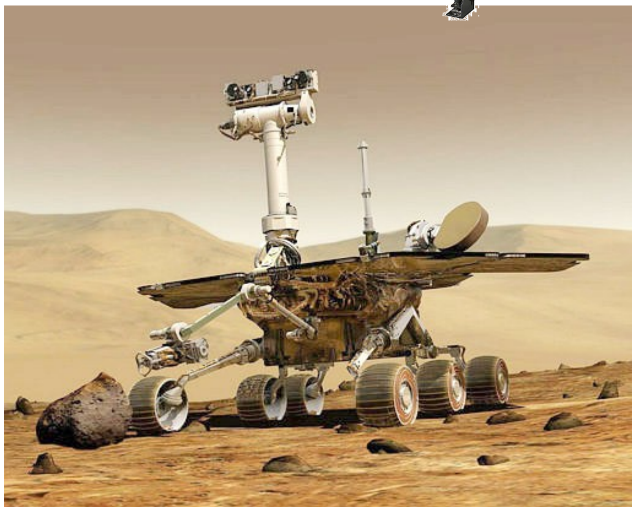






Domain-Independent Planners

- No domain-specific knowledge except the description of the system Σ
- In practice,
  - Not feasible to make domain-independent planners work well in all possible planning domains



- Make simplifying assumptions to restrict the set of domains
  - **Classical planning**

    ⇨ Historical focus of most research on automated planning

Classical Planning

- Reduces to the following problem:

  Given Σ, initial state s_0, and goal states S_g,

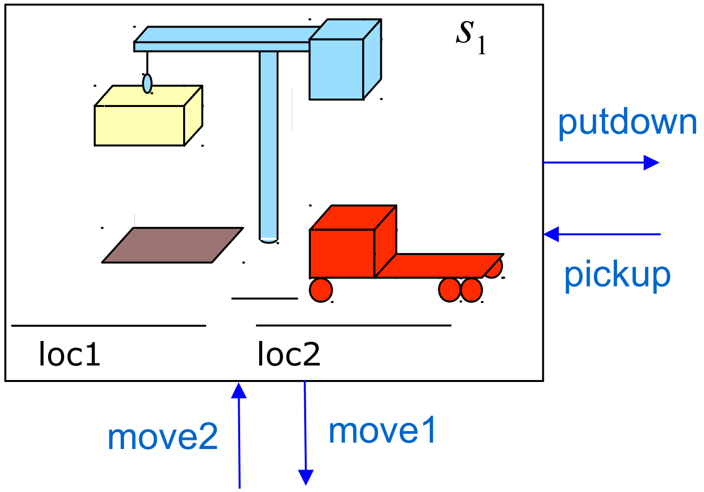
  find a sequence of actions $(a_1, a_2, \ldots a_n)$ that produces

  a sequence of state transitions $(s_0, s_1, s_2, \ldots, s_n)$ such that $s_n \in S_g$

Is this trivial?

- Generalise the earlier example:
  - Five locations, three robot carts, 100 containers, three piles

    ⇨ 10^{277} states



- Automated-planning research has been heavily dominated by classical planning. There are dozens of different algorithms.

Representations for Classical Planning

Classical Representations: Motivation

- In most problems, far too many states to try to represent all of them explicitly as s_0, s_1, s_2, …

  ⇨ represent each state as a set of **atomic features**
  Example feature, *light(on)* or *light(off);* the light can be on or off**.**

- Define a set of **operators** that can be used to compute state-transitions

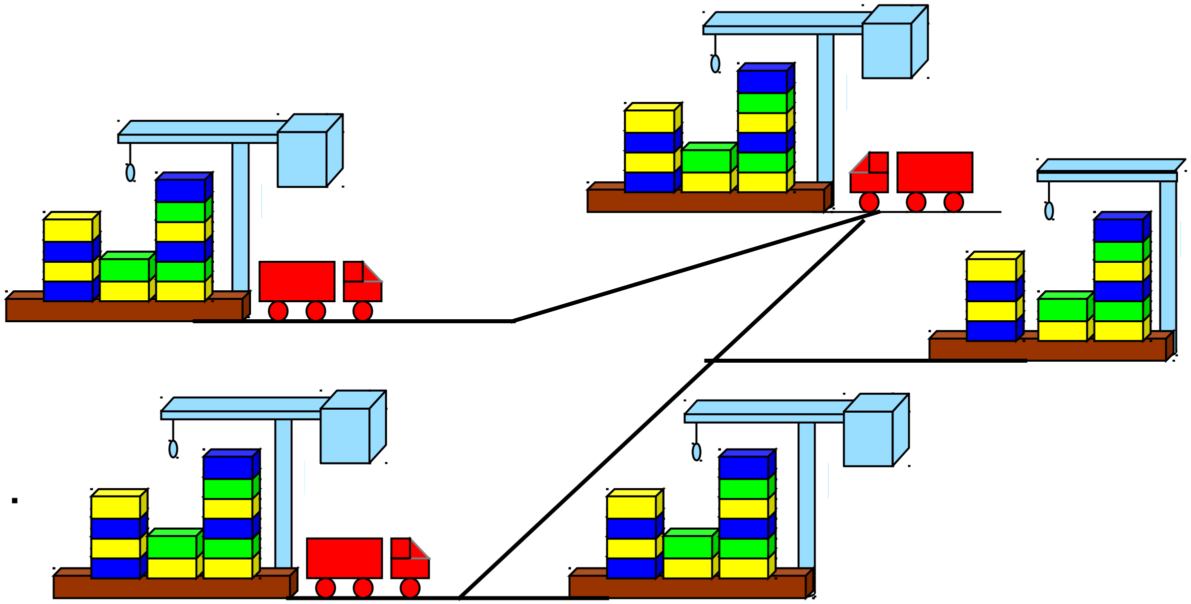
  Example operator, *switch(on)*; turn the switch on.

- Don't give all of the states explicitly
  - Just give the initial state
  - Use the operators to generate the other states as needed

Classical Representation

- Language of first-order logic but without function symbols

  ⇨ finitely many predicate symbols and constant symbols
- Classical planning problems often described using the STRIPS action language (developed in 1970s), or PDDL (a more modern language).
- We use STRIPS syntax, but for our purposes can think of STRIPS and PDDL as being used to represent the same sorts of problems.
- Example: the DWR domain
  - Locations: l1, l2, …
  - Containers: c1, c2, …
  - Piles: p1, p2, …
  - Robot carts: r1, r2, …
  - Cranes: crane1, crane2, …



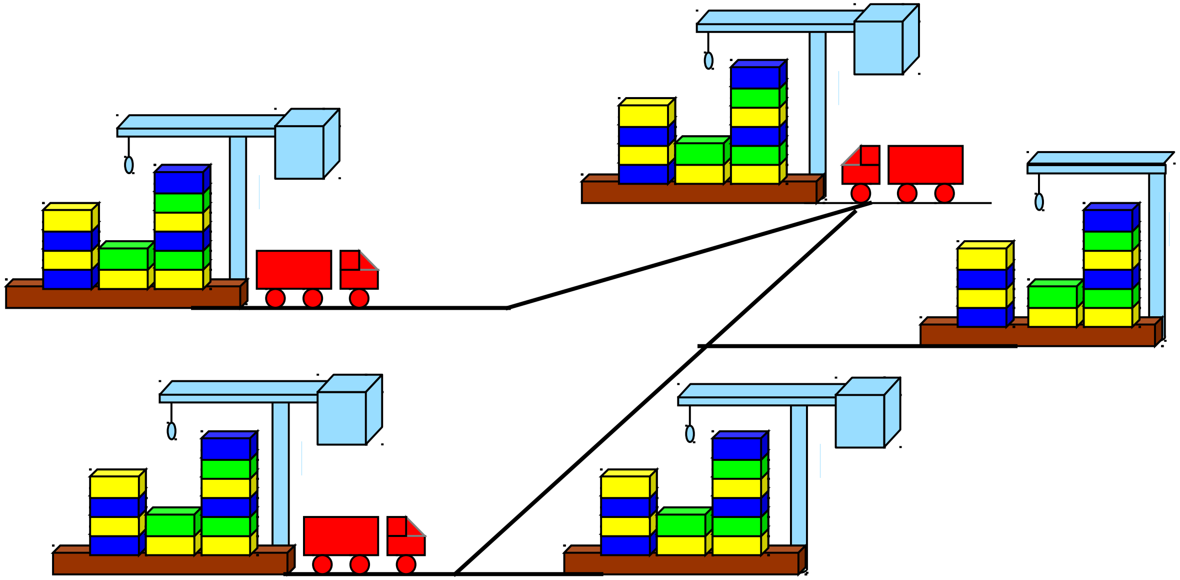
Example (cont'd)

- **Fixed (static) relations**: same in all states

  adjacent(*l*,*l'*) attached(*p*,*l*) belong(*k*,*l*)

- **Dynamic relations (fluents)**: differ between states

  occupied(*l*) at(*r*,*l*)
  loaded(*r*,*c*) unloaded(*r*)
  holding(*k*,*c*) empty(*k*)
  in(*c*,*p*) on(*c*,*c'*)
  top(*c*,*p*) top(pallet,*p*)



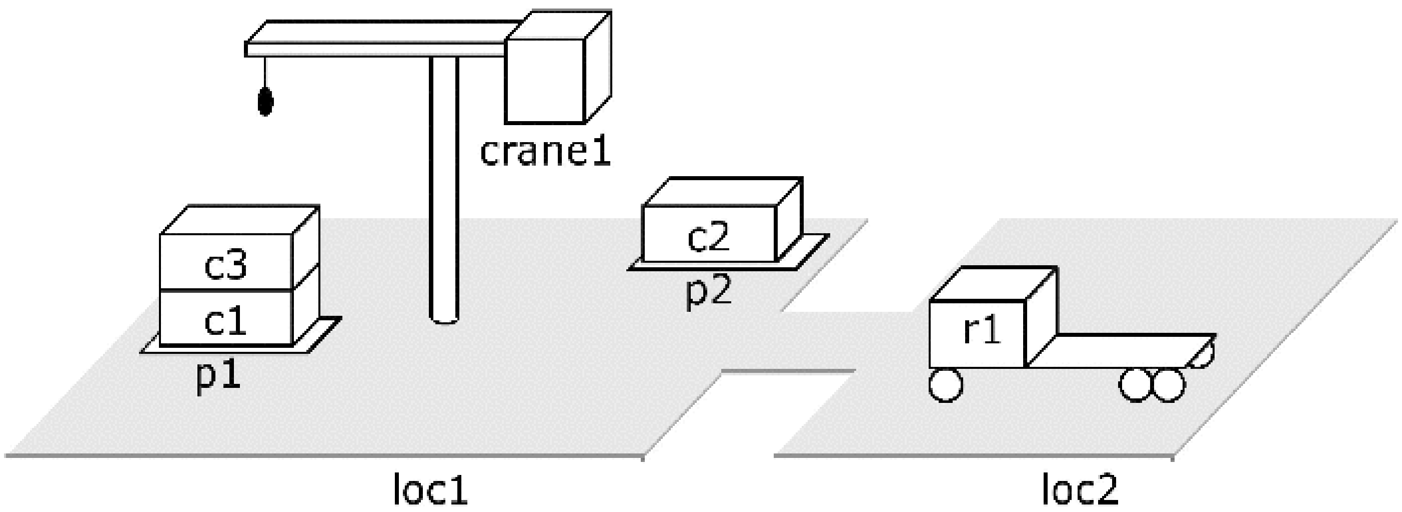
- **Actions**:

  pickup(*c*,*k*,*p*) putdown(*c*,*k*,*p*)
  load(*r*,*c*,*k*) unload(*r*) move(*r*,*l*,*l'*)

States

A **state** is a set *s* of ground atoms

- The atoms represent the things that can be true in some states
- Only finitely many ground atoms, so only finitely many possible states



s_1 = {attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}

Operators

An **operator** is a triple o = (name(o), precond(o), effects(o))

- name(o): a syntactic expression of the form $n(x_1,\ldots,x_k)$
  - $(x_1,\ldots,x_k)$ is a list of every variable symbol (parameter) that appears in o
- precond(o): **preconditions**
  - literals that must be true in order to use the operator
- effects(o): **effects**
  - literals the operator will make true

Example

```
pickup(k,l,c,d,p)
  ;; crane k at location l takes c off of d in pile p
  precond: belong(k,l), attached(p,l),empty(k), top(c,p),
           on(c,d)
  effects: holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p),
           ¬on(c,d), top(d,p)
```

Actions

An **action** is a ground instance (via a substitution) of an operator

```
pickup(k,l,c,d,p)
  ;; crane k at location l takes c off of d in pile p
  precond: belong(k,l), attached(p,l),empty(k), top(c,p),
           on(c,d)
  effects: holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p),
           ¬on(c,d), top(d,p)
```

- Let σ = {*k* / crane1, *l* / loc1, *c* / c3, *d* / c1, *p* / p1}
- Then pickup(*k,l,c,d,p*)σ is the following action:

pickup(crane1,loc1,c3,c1,p1)

precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)

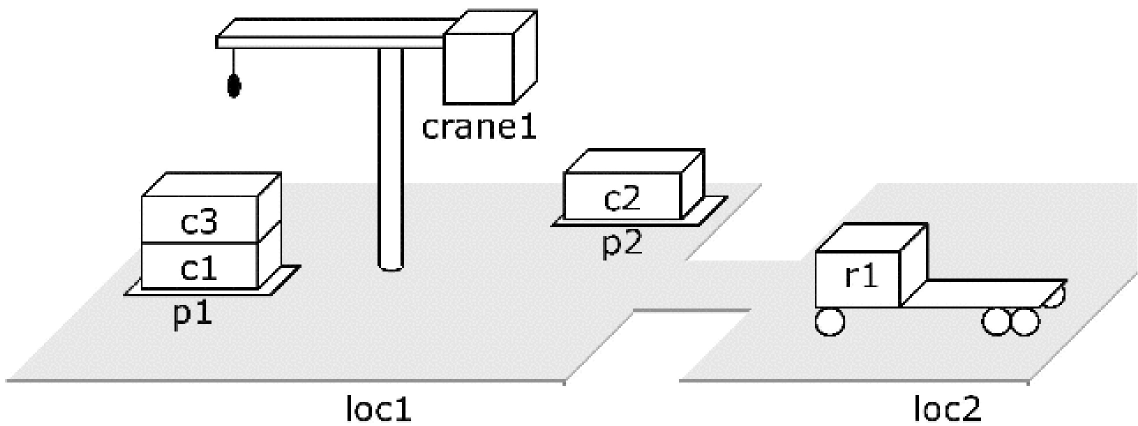
effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

Applicability and Result of Actions

- Let S be a set of literals. Then
  S^+ = {atoms that appear positively in S}
  S^- = {atoms that appear negatively in S}
- Let a be an operator or action. Then
  $\text{precond}^+(a)$ = {atoms that appear positively in a's preconditions}
  $\text{precond}^-(a)$ = {atoms that appear negatively in a's preconditions}
  $\text{effects}^+(a)$ = {atoms that appear positively in a's effects}
  $\text{effects}^-(a)$ = {atoms that appear negatively in a's effects}

- Action a is **applicable** to (or **executable** in) S if
  - $\text{precond}^+(a) \subseteq s$
  - $\text{precond}^-(a) \cap s = \emptyset$
- The **result** of applying action a to state S is
  - $\gamma(s,a) = (s \setminus \text{effects}^-(a)) \cup \text{effects}^+(a)$

Example: Applicability



- An action:

  pickup(crane1,loc1,c3,c1,p1)

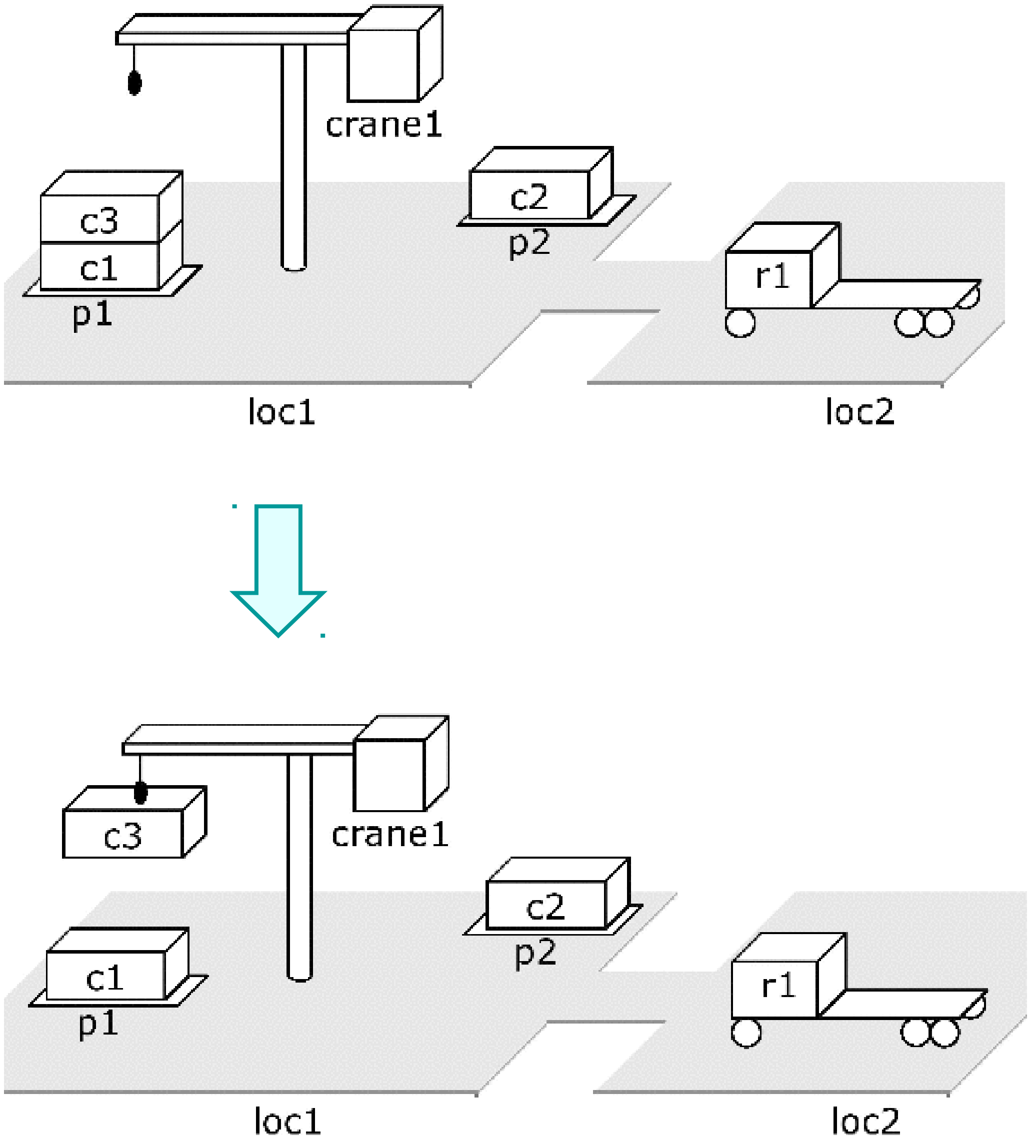
  precond: belong(crane,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)

  effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

- A state it's applicable to

  s_1 = {**attached(p1,loc1)**, in(c1,p1), in(c3,p1), **top(c3,p1)**, **on(c3,c1)**, on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), **belong(crane1,loc1)**, **empty(crane1)**, adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2, unloaded(r1)}

Example: Result



pickup(crane1,loc1,c3,c1,p1)

precond: belong(crane,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)

effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

s_2 = {attached(p1,loc1), in(c1,p1), ~~in(c3,p1)~~, ~~top(c3,p1)~~, ~~on(c3,c1)~~, on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), ~~empty(crane1)~~, adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2, unloaded(r1), **holding(crane1,c3)**, **top(c1,p1)**}

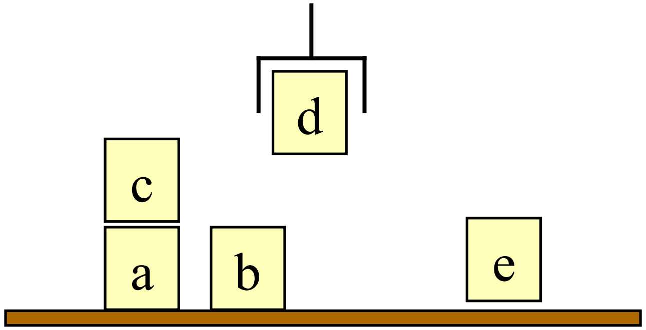
Exercise

Exercise: The Blocks World

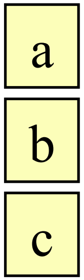
- Infinitely wide table, finite number of children's blocks
- Ignore where a block is located on the table
- A block can sit on the table or on another block
- There's a robot gripper that can hold at most one block

- Want to move blocks from one configuration to another
  - e.g.,

initial state

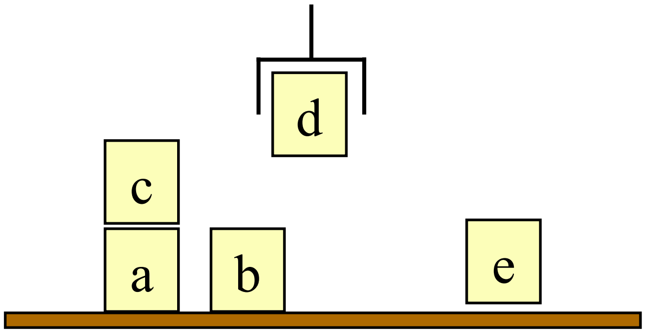


goal



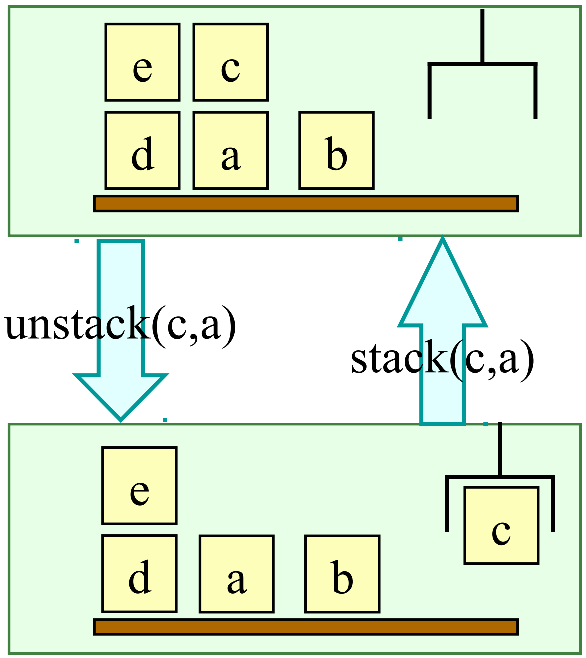
Exercise: Classical Representation – Symbols

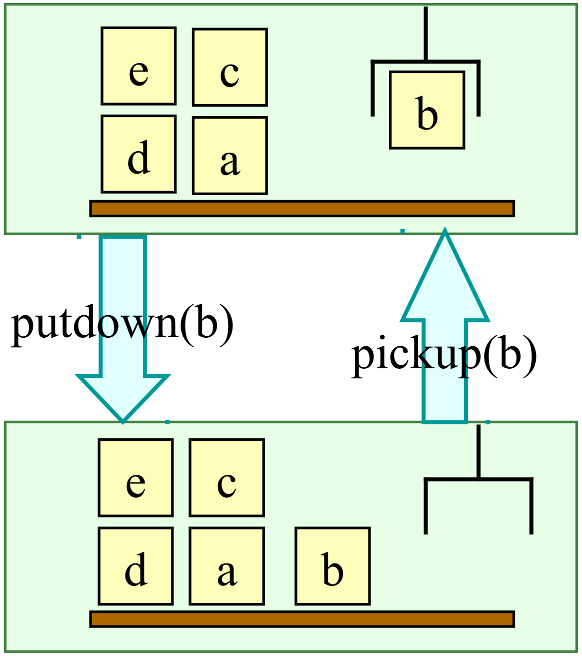
- Constant symbols:
  - The blocks: a, b, c, d, e
- Dynamic relations?



Exercise: Classical Operators

- Preconditions and effects?





Summary: Planning Problems

Given a planning domain (language *L*, operators *O*)

- **Representation** of a planning problem: a triple $P = (O, s_0, g)$
  - *O* is the collection of operators
  - s_0 is a state (the initial state)
  - *g* is a set of literals (the goal formula)

Plans and Solutions

Let $P = (O, s_0, g)$ be a planning problem

- **Plan**: any sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ such that each a_i is an instance of an operator in O
- Plan π is a **solution** for $P = (O, s_0, g)$ if it is executable and achieves g
  - i.e., if there are states $s_0, s_1, \ldots, s_n$ such that

$$\gamma(s_0, a_1) = s_1$$

$$\gamma(s_1, a_2) = s_2$$

$$\vdots$$

$$\gamma(s_{n-1}, a_n) = s_n$$

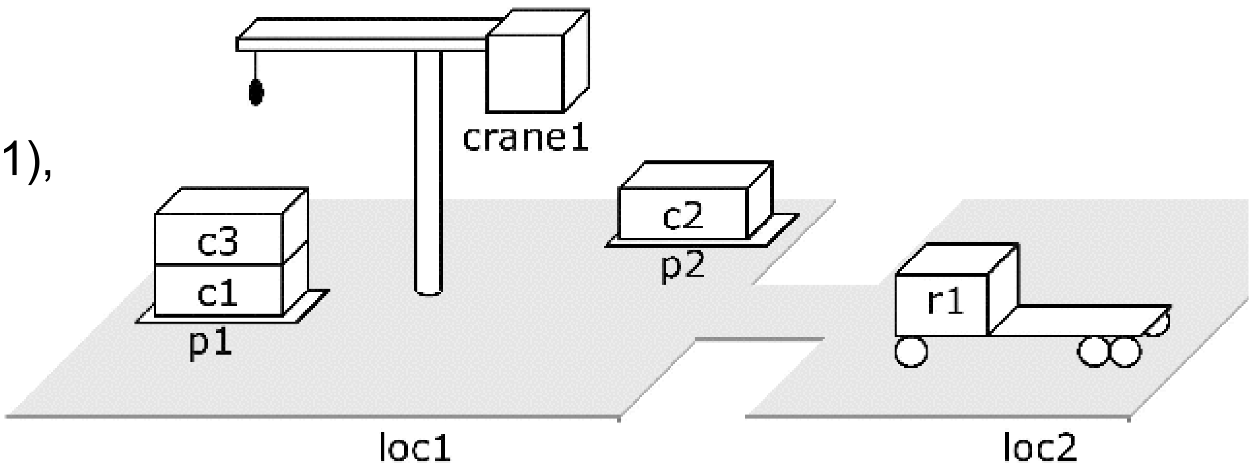
s_n satisfies g

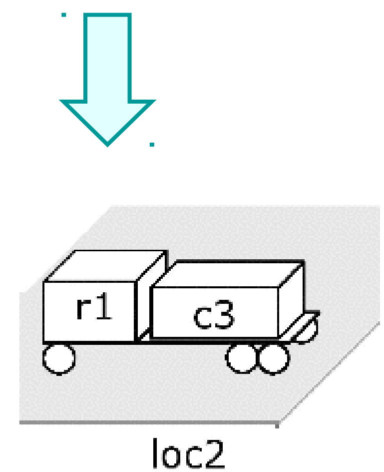
Example: The 5 DWR Operators

```
move(r,l,m)
  ;; robot r moves from location l to location m
  precond: adjacent(l,m), at(r,l), ¬occupied(m)
  effects: at(r,m), occupied(m), ¬occupied(l), ¬at(r,l)

load(k,l,c,r)
  ;; crane k at location l loads container c onto robot r
  precond: belong(k,l), holding(k,c), at(r,l), unloaded(r)
  effects: empty(k), ¬holding(k,c), loaded(r,c), ¬unloaded(r)

unload(k,l,c,r)
  ;; crane k at location l takes container c onto robot r
  precond: belong(k,l), at(k,l), loaded(r,c), empty(k)
  effects: ¬empty(k), holding(k,c), unloaded(r), ¬loaded(r,c)

putdown(k,l,c,d,p)
  ;; crane k at location l puts c onto d in pile p
  precond: belong(k,l), attached(p,l), holding(k,c), top(d,p)
  effects: ¬holding(k,c), empty(k), in(c,p), top(c,p), on(c,d), ¬top(d,p)

pickup(k,l,c,d,p)
  ;; crane k at location l takes c off of d in pile p
  precond: belong(k,l), attached(p,l),empty(k), top(c,p), on(c,d)
  effects: holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p), ¬on(c,d), top(d,p)
```

Example

- Let $P = (O, s_0, g)$, where
  - O = {the 5 DWR operators}
  - s_0 = {attached(p1,loc1), in(c1,p1),
    in(c3,p1), top(c3,p1),
    on(c3,c1), on(c1,pallet),
    attached(p2,loc1),
    in(c2,p2), top(c2,p2),
    on(c2,pallet),
    belong(crane1,loc1), empty(crane1),
    adjacent(loc1,loc2), adjacent(loc2,loc1),
    at(r1,loc2), occupied(loc2), unloaded(r1)}
  - g = {loaded(r1,c3), at(r1,loc2)}

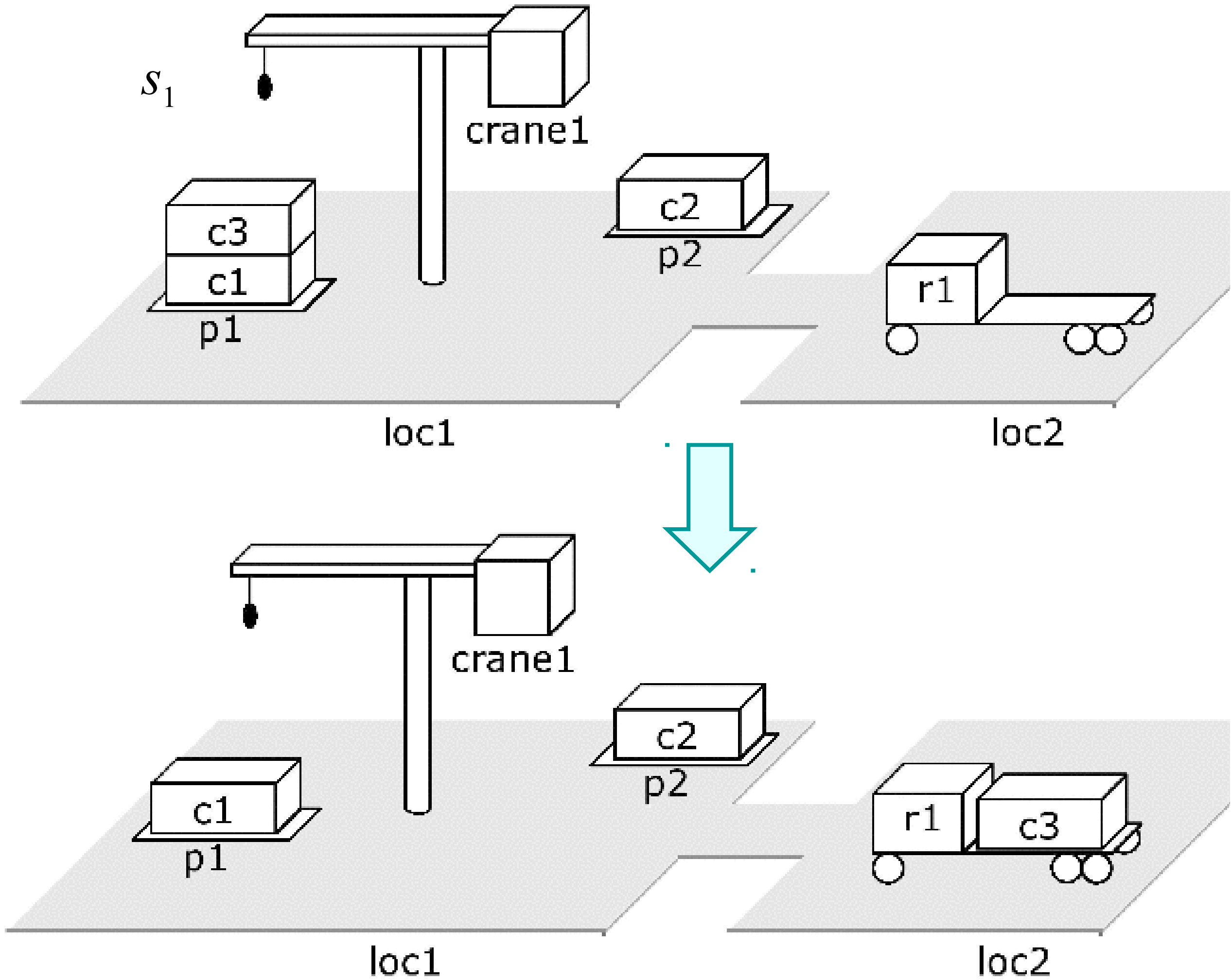




- Two *redundant* solutions (can remove actions and still have a solution):

  (move(r1,loc2,loc1),
  pickup(crane1,loc1,c3,c1,p1),
  ~~move(r1,loc1,loc2)~~,
  ~~move(r1,loc2,loc1)~~,
  load(crane1,loc1,c3,r1),
  move(r1,loc1,loc2))

  (pickup(crane1,loc1,c3,c1,p1),
  ~~putdown(crane1,loc1,c3,c2,p2)~~,
  move(r1,loc2,loc1),
  ~~pickup(crane1, loc1,c3,c2,p2)~~,
  load(crane1,loc1,c3,r1),
  move(r1,loc1,loc2))



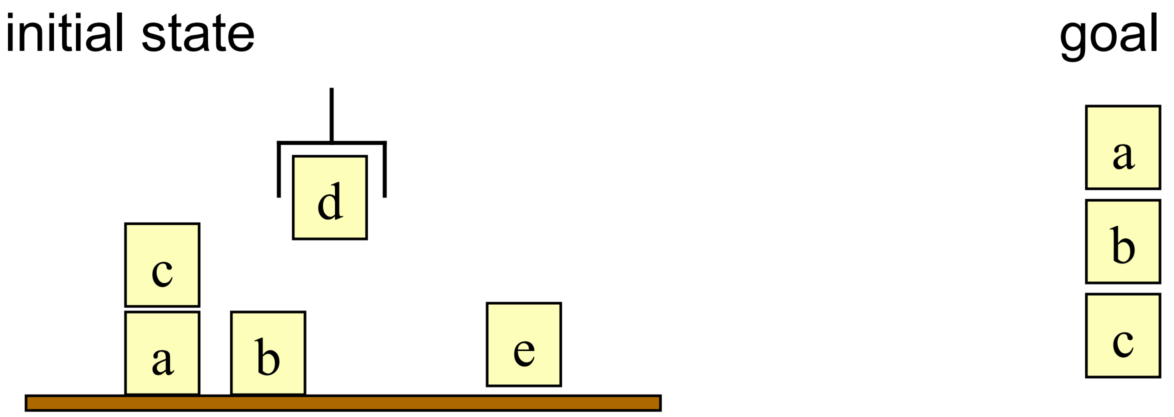
- A solution that is both *irredundant* and *shortest*:

  (move(r1,loc2,loc1), pickup(crane1, loc1,c3,c1,p1),
  load(crane1,loc1,c3,r1), move(r1,loc1,loc2))

- Are there any other shortest solutions? Are irredundant solutions always the shortest?

Exercise

Exercise: Plans

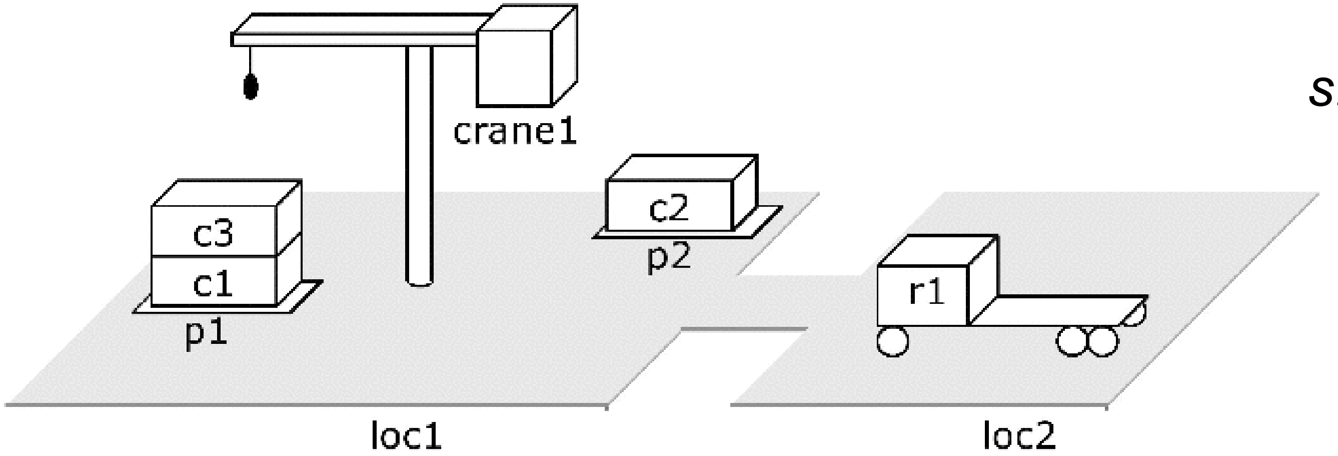


- Solution?

State-Variable Representation

- Alternative to the classical representation
- Use ground atoms for properties that do not change, e.g., adjacent(loc1,loc2)
- For properties that can change, assign values to **state variables**
  - Like fields in a record structure

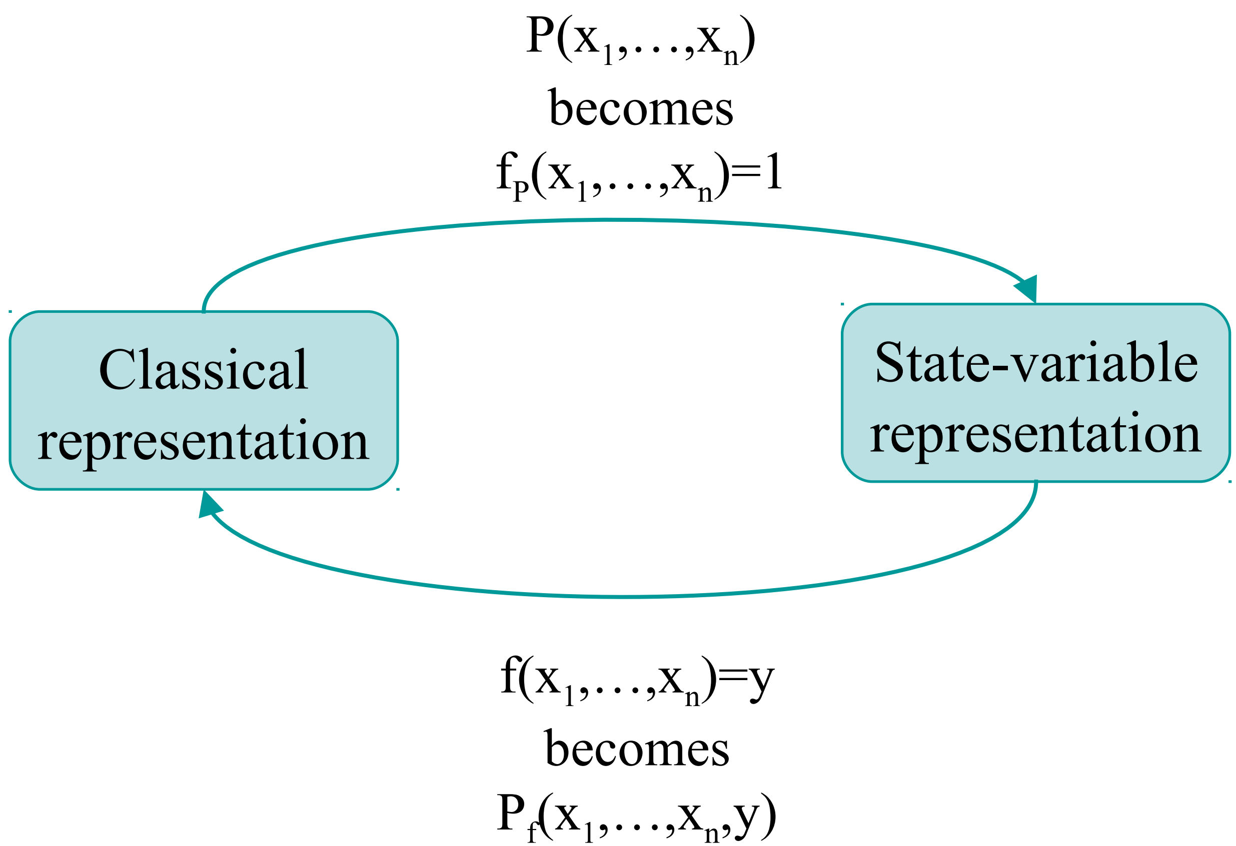
move(r, l, m)
;; robot r at location l moves to an adjacent location m
precond: $\text{rloc}(r) = l, \text{adjacent}(l, m)$
effects: $\text{rloc}(r) \leftarrow m$



s_1 = {top(p1)=c3,
cpos(c3)=c1,
cpos(c1)=pallet,
holding(crane1)=nil,
rloc(r1)=loc2,
loaded(r1)=nil, …}

Expressive Power

- Any problem that can be represented in one representation can also be represented in the other
- Can convert in linear time and space



Comparison

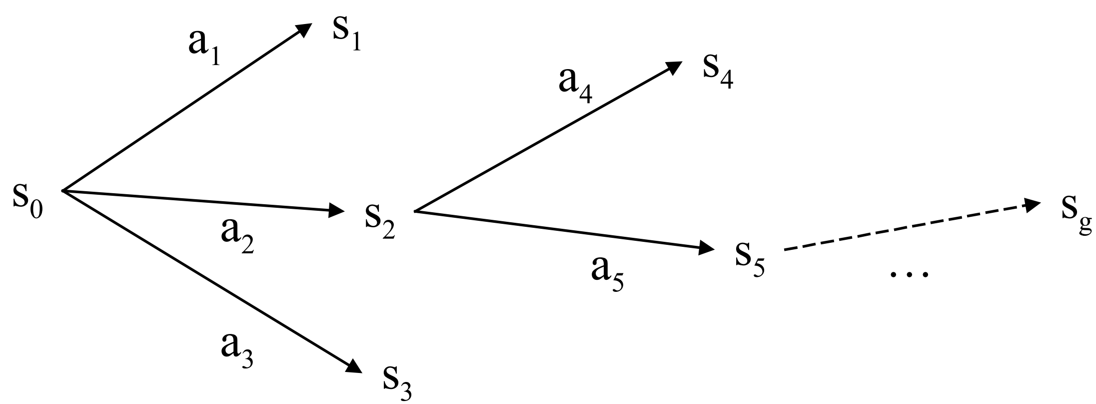
- Classical representation
  - The most popular for classical planning, partly for historical reasons
- State-variable representation
  - Equivalent to classical representation in expressive power
  - Less natural for logicians, more natural for engineers and most computer scientists
  - Useful in non-classical planning problems as a way to handle numbers, functions, time

State-Space Search

Search Algorithms

Search tree

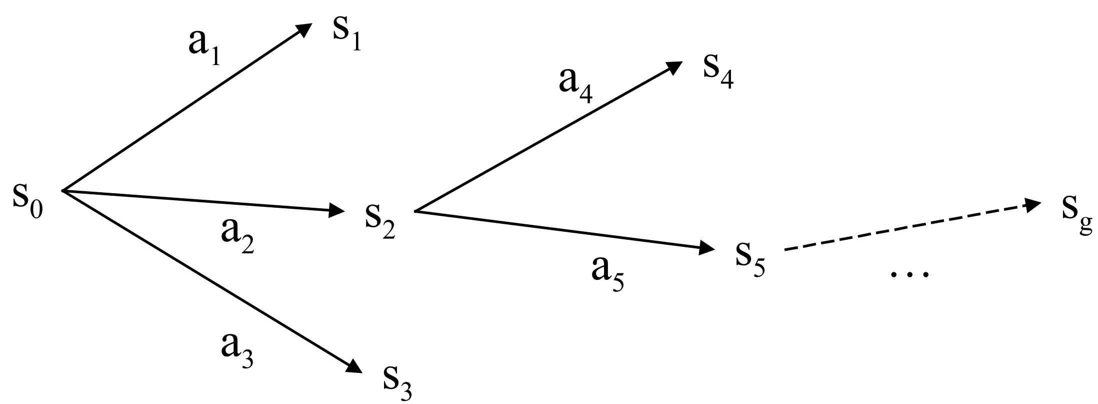
- nodes = states
- edges = actions



Search Algorithms

Search tree

- nodes = states
- edges = actions



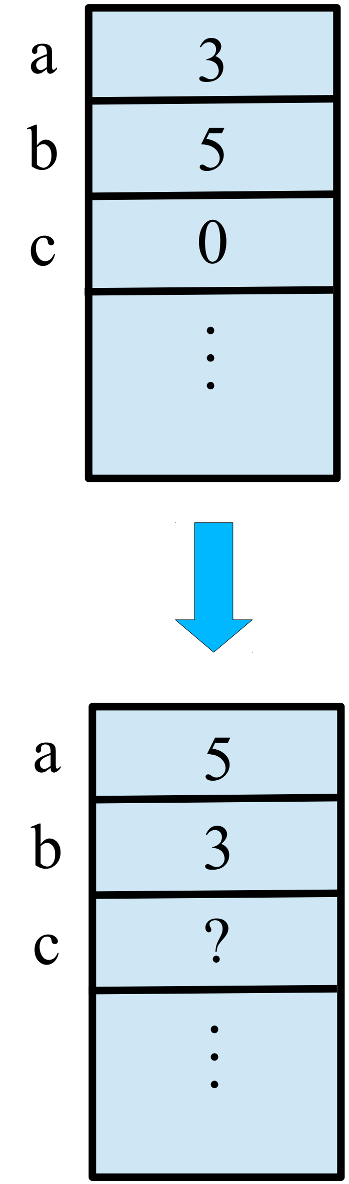
- Most common search method: **depth-first** search
  - In general, sound but not complete
    - But classical planning has only finitely many states
    - ⇨ can make depth-first search complete by doing loop-checking

Exercise

Exercise: Interchange Values of Variables

- Operator `assign(v,w,x,y)`

```
;; assign the value of v (which is currently x)
;; to the value of w (which is currently y)
precond:  value(v,x), value(w,y)
effects:  ¬value(v,x), value(v,y)
```



- Initial state s_0 = { value(a,3), value(b,5), value(c,0) }
- Goal g = { value(a,5), value(b,3) }

- In the search tree for this planning problem,
  - what is the length of the shortest path to a solution?
  - what is the length of the longest path in the tree?

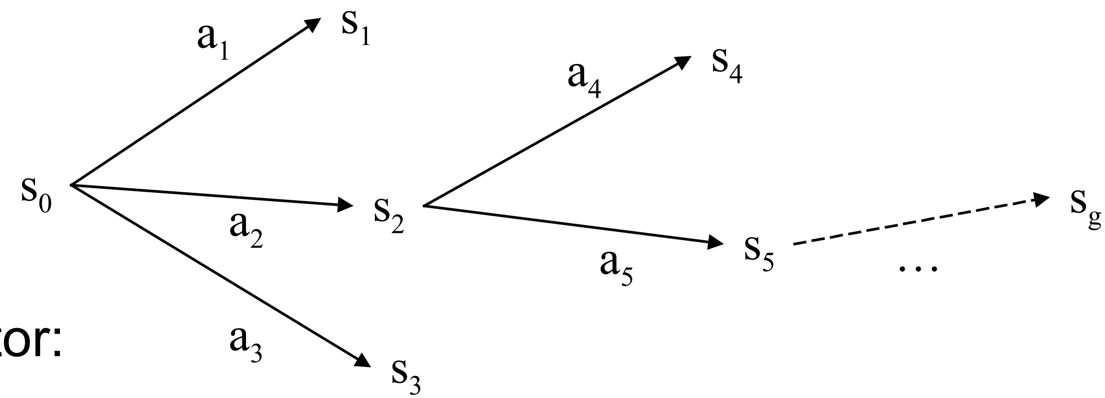
Algorithms/Technologies for Solving Planning Problems

How to Solve Classical Planning Problems

- Heuristic search
  - Heuristics aren't guaranteed to work - but work well as a guide
  - Many different heuristics
  - Is it possible to generate heuristics automatically?
- Planning with Answer Set Programming
  - Requires setting a maximum length to the path
- Graphplan (won't cover here)
  - Algorithm developed in 1995
  - At the time it set a new benchmark for planning!

Motivation

- A standard tree search may try lots of actions that are unrelated to the goal

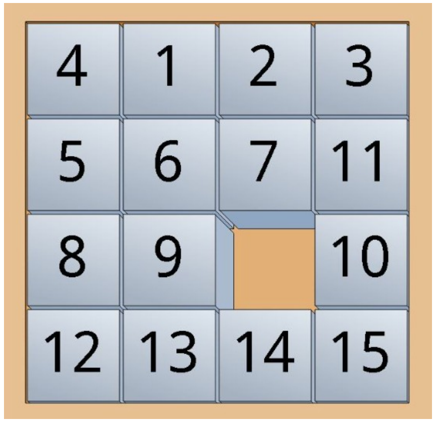


- One way to reduce branching factor:
- First create a **relaxed problem**
  - Remove some restrictions of the original problem

    ⇨ Want the relaxed problem to be easy to solve (polynomial time)
  - The solutions to the relaxed problem will include all solutions to the original problem
- Then do a modified version of the original search
  - Restrict its search space to include only those actions that occur in solutions to the relaxed problem

Planning with Heuristic Search

- Explicitly search with heuristic h(s) that estimates cost from s to goal
- General idea:
  heuristic function = length of optimal plan for a **relaxed problem**
- Example:
  - Manhattan distance in 15-puzzle
    (sum of distances to correct positions)
  - Manhattan distance is an *admissible* heuristic
    (it never overestimates the cost).



- How to get such heuristics automatically?

Relaxation Heuristic - General-Purpose Heuristics for Classical Planning

- Automatic extraction of informative heuristic function **from the problem P itself**

- Most common relaxation in planning: **ignore all negative effects** of the operators.

> Let P^+ be obtained from planning problem P by dropping the negative effects.
> If $c^*(P^+,s)$ is optimal cost of P^+ with initial state s, then the heuristic is set to
>
> $$h(s) = c^*(P^+,s)$$

- This heuristic is intractable in general, but easy to approximate

Example.

- Operator `assign(v,w,x,y)`

```
precond:  value(v,x), value(w,y)
effects:  ~~¬value(v,x),~~ value(v,y)
```

- s_0 = { value(a,3), value(b,5), value(c,0) }, g = { value(a,5), value(b,3) }
- Optimal relaxed plan: assign(a,b,3,5), assign(b,a,5,3), hence $h(s_0) = 2$

Planning with ASP – Simple Example

Cake-Example

- Two state properties: `have`, `eaten`
- Action `eat`, which is possible if `have` is true; effects: `eaten, not have`
- 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**

Planning with ASP - 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**

Planning with ASP - 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).
```

- 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)`

Summary

- Representations for classical planning
  - Classical representation
  - State-variable representation
- State-space search
  - with heuristics
- Solving planning problems
  - With heuristics
  - ASP
  - Graphplan