TDDC17:
Introduction to Automated Planning

Jonas Kvarnström
Automated Planning Group
Department of Computer and Information Science
Linköping University

jonas.kvarnstrom@liu.se – 2016

Introduction to Planning

One way of defining planning:

*Using **knowledge** about the world,*
including possible actions and their results,
*to **decide** what to do and when*
*in order to achieve an **objective**,*
***before** you actually start doing it*

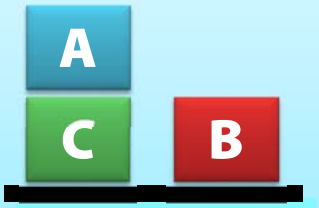
Domains 1: Blocks World

jonkv@ida

- Classical example: The **Blocks World**

You

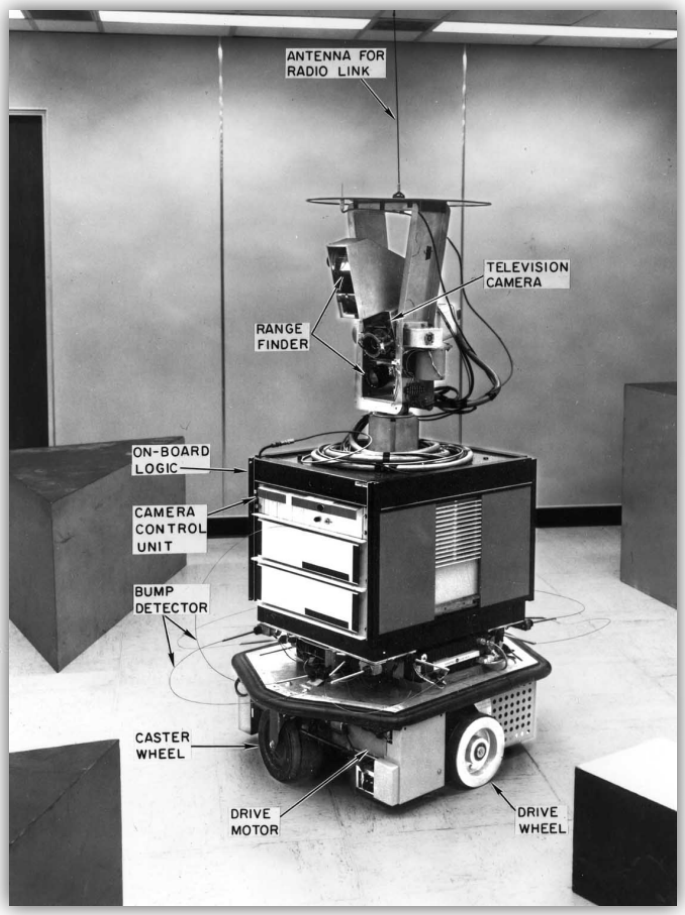
Current state of the world



Your greatest desire

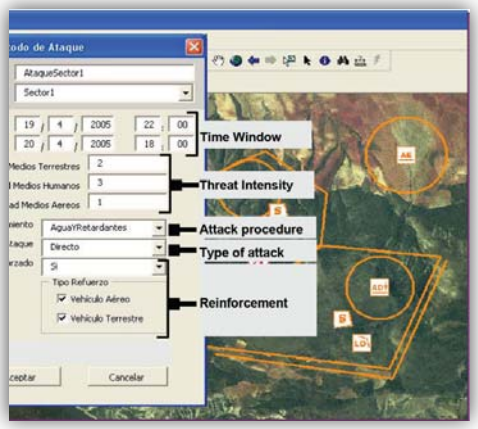
Domains 2: Shakey

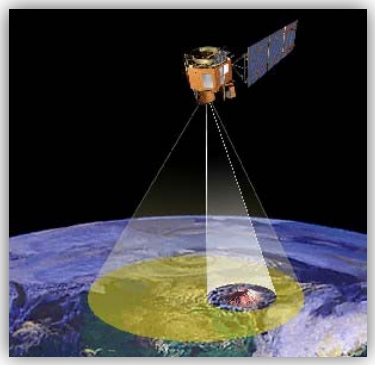
- Classical robot example: **Shakey** (1969)
  - Available **actions**:
    - *Moving* to another location
    - *Turning* light switches on and off
    - *Opening* and *closing* doors
    - *Pushing* movable objects around
    - …
  - **Goals**:
    - *Be in room 4 with objects A,B,C*
    - http://www.youtube.com/watch?v=qXdn6ynwpiI



Domains 3

Logistics:
Use a fleet of trucks to efficiently deliver packages





On-board planning
to view interesting natural events:
http://ase.jpl.nasa.gov/

SIADEX –
plan for firefighting
Limited resources
Plan execution is dangerous!

Domains 4: Dock Worker Robots (DWR)

Containers shipped in and out of a harbor

Cranes move containers between "piles" and robotic trucks

Problem description

Solver: Planning Algorithm

Solution

Could be a **customized** solver

```
plan            = [ ];
s               = current state of the world;
while (exists b1,b2 [ s.isOn(b1,b2) ]):
    plan        +=  "unstack(b1,b2)"
    s           =   apply(unstack(b1,b2), s)
    plan        +=  "putdown(b1)"
    s           =   apply(putdown(b1), s)
while (exists b1,b2 [ goal.isOn(b1,b2) & !s.isOn(b1,b2) &
                      s.isClear(b1) & s.isClear(b2) ]):
    plan        +=  "pickup(b1)"
    s           =   apply(pickup(b1), s)
    plan        +=  "stack(b1,b2)"
    s           =   apply(stack(b1,b2), s)
return plan
```

Tear down all towers

Rebuild in the right order

Efficient plans → more complex solvers
Complex domains → more complex solvers
Programming is time-consuming
Problem changes → more programming required

We want general + efficient algorithms!

AI Planning: A Simplified View

Knowledge about the world and how we can affect it

Description of objectives to achieve

General Planning Algorithm

Solution:
Plan with *actions* to perform, *ordering* constraints,
…

Execution System

Sensors

Actuators

Simplified, because in reality:

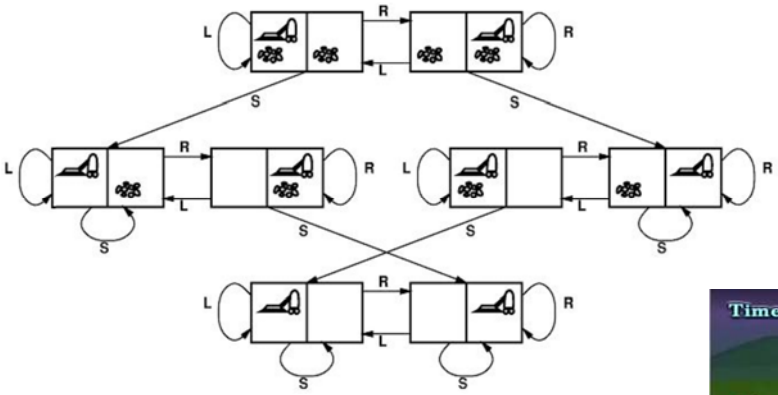
Execution will *fail* ➔ feedback, monitoring, replanning, plan repair

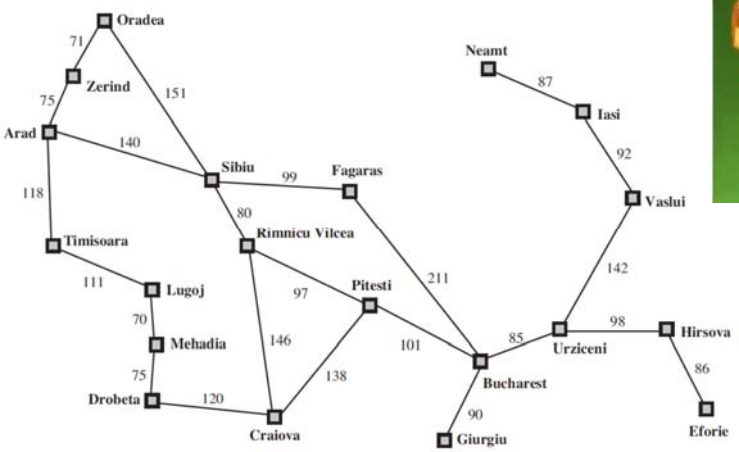
Planning occurs in a *context* ➔ Where are objectives generated?

Multiple agents may plan, interact ➔ negotiation, delegation of tasks, collaboration, competition,

And so on…

You have already planned – using general search algorithms!

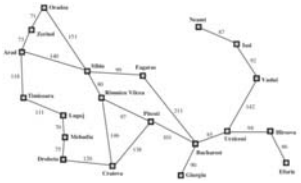




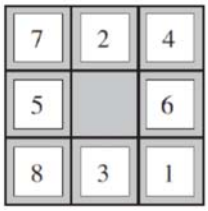
But the **representation** was problem-specific…

States: triple (x,y,z) with $0 \leq x,y,z \leq 3$, where x, y, and z represent the number of missionaries, cannibals and boats currently on the original bank.

And so was the search **guidance**!



Straight line distance from city n to goal city n'



$h_2(n)$: The sum of the manhatten distances for each tile that is out of place.
(3+1+2+2+2+3+3+2=18) . The manhatten distance is an under-estimate because there are tiles in the way.

How to **generalize**?

Classical Planning

First requirement: A **formal language** for planning problems!

- We want a **general** language:
  - Allows a wide variety of domains to be modeled

- We want good **algorithms**
  - Generate plans quickly
  - Generate high quality plans
- Easier with more **limited** languages

Conflicting desires!

- Many early planners made **similar tradeoffs**
  - Later called "classical planning"
    Restricted, but a good place to start

Modeling Classical Planning Problems

Planning Domain, Problem Instance

jonkv@ida

Split knowledge into two parts

Planning Domain

- General properties of DWR problems
  - There are *containers*, *cranes*, …
  - Each object has a *location*
  - Possible actions:
    Pick up container, put down container, drive to location, …

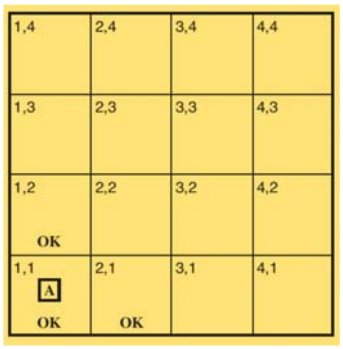
Problem Instance

- Specific problem to solve
  - *Which* containers and cranes exist?
  - Where is everything?
  - Where *should* everything be?
    (More general:
    What should we achieve?)

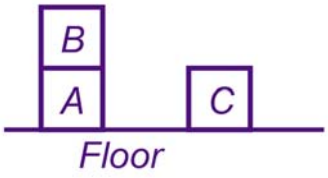
Objects 1: First-Order → Objects Exist

- Many planning languages have their roots in **logic**
  - Great – you've already seen this!

- Propositional syntax
  - Plain **propositional facts**:
    ~B1,1 , ~S1,1 , OK1,1 , OK2,1 , OK1
  - Used in *some* planners to simplify parsing, …



- First-order syntax
  - **Objects** and **predicates** (relations):
    On(B,A), On(C,Floor), Clear(B), Clear(C)
  - Used in most implementations



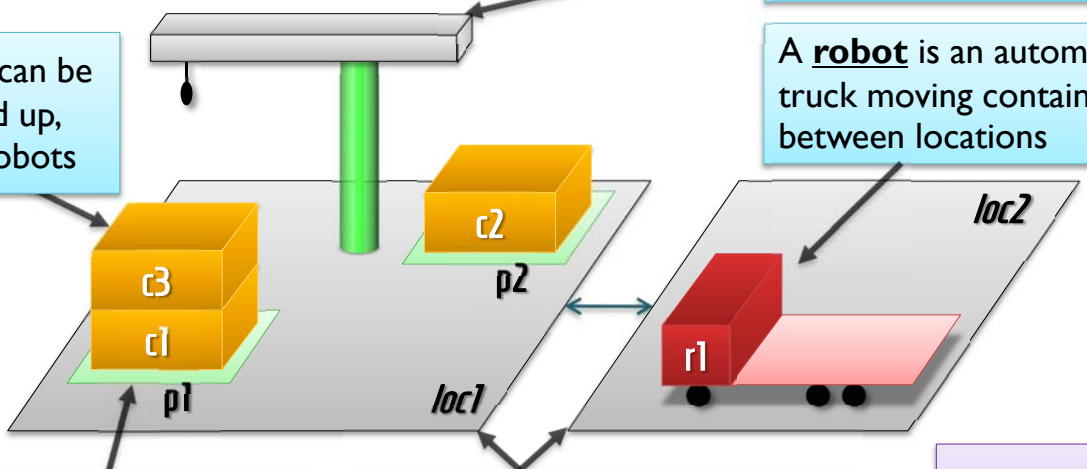
Objects 2: Object Types

- The domain often specifies **object types**

A **crane** moves containers between piles / robots

A **container** can be stacked, picked up, loaded onto robots

A **robot** is an automated truck moving containers between locations



A **pile** is a stack of containers – at the bottom, there is a **pallet**

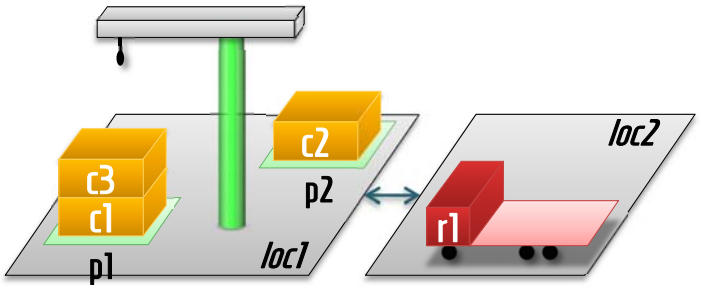
A **location** is an area that can be reached by a single crane. Can contain several piles, at most one robot.

We can skip:
Drivers
Wheels
…

Essential: Determine what is **relevant** for the **problem** and **objective**!

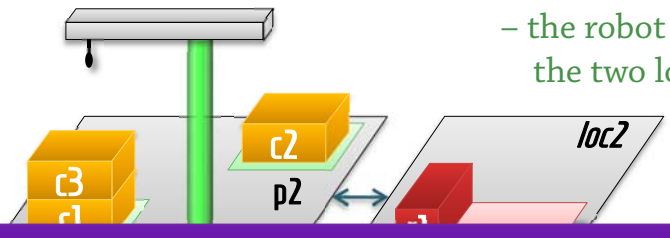
Objects 3: In the problem instance

- **Objects** are generally specified in the **problem instance**
  - robot: { r1 }
  - location: { loc1, loc2 }
  - crane: { k1 }
  - pile: { p1, p2 }
  - container: { c1, c2, c3, pallet }



Facts

- In **first-order representations** of planning problems:
  - Any **fact** is represented as a logical **atom**: Predicate symbol + arguments
  - **Properties** of the **world**
    - **raining** – it is raining [not a DWR predicate!]
  - **Properties** of **single objects**…
    - **empty**(crane) – the crane is not holding anything
  - **Relations** between objects
    - **attached**(pile, location) – the pile is in the given location
  - **Relations** between >2 objects
    - **can-move**(robot, location, location) – the robot can move between the two locations [Not a DWR predicate]



Essential: Determine what is **relevant** for the **problem** and **objective**!

Facts / Predicates in DWR

- All predicates for DWR, and their intended meaning:

"Fixed/Rigid" (can't change)

adjacent	(*loc1*, *loc2*)	; can move from *loc1* directly to *loc2*
attached	(*p*, *loc*)	; pile *p* attached to *loc*
belong	(*k*, *loc*)	; crane *k* belongs to *loc*

"Dynamic" (modified by actions)

at	(*r*, *loc*)	; robot *r* is at *loc*
occupied	(*loc*)	; there is a robot at *loc*
loaded	(*r*, *c*)	; robot *r* is loaded with container *c*
unloaded	(*r*)	; robot *r* is empty
holding	(*k*, *c*)	; crane *k* is holding container *c*
empty	(*k*)	; crane *k* is not holding anything
in	(*c*, *p*)	; container *c* is somewhere in pile *p*
top	(*c*, *p*)	; container *c* is on top of pile *p*
on	(*c1*, *c2*)	; container *c1* is on container *c2*

States 1: State of the World

- A **state (of the world)** should specify exactly *which facts* (***ground atoms***) *are true/false in the world* at a given time

We know all **predicates** that exist:
adjacent(location, location), …

We know which **objects** exist for each type

→ We can calculate all *ground atoms*

adjacent(loc1,loc1)
adjacent(loc1,loc2)
…
attached(pile1,loc1)
…

These are the *facts* to keep track of!

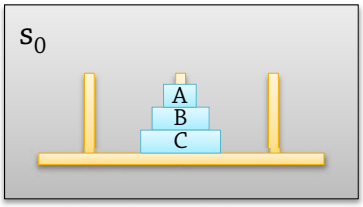
We can find all possible states!

Every assignment of true/false to the ground atoms is a distinct state

Number of states: $2^{\text{number of atoms}}$ – enormous, but finite (for classical planning!)

States 2: Efficient Representation

- **Efficient specification and storage** for a **single state**:
  - Specify **which atoms are true**
    - All other atoms have to be false – what else would they be?
  - ➔ A **state of the world** is specified as a **set** containing all **variable-free atoms** that [are, were, will be] true in the world
    - s_0 = { on(A,B), on(B,C), in(A,2), in(B,2), in(C,2), top(A), bot(C) }



s_0	top(A)	in(A,2)
	on(A,B)	in(B,2)
	on(B,C)	in(C,2)
	bot(C)	

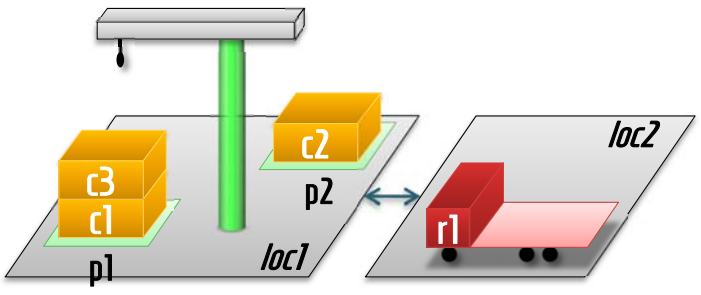
top(A) ∈ s_0 ➔ top(A) is true in s_0
top(B) ∉ s_0 ➔ top(B) is false in s_0

States 3: Initial State

- Initial states *in classical STRIPS planning*:
  - We assume *complete information* about the **initial state** (before any action)

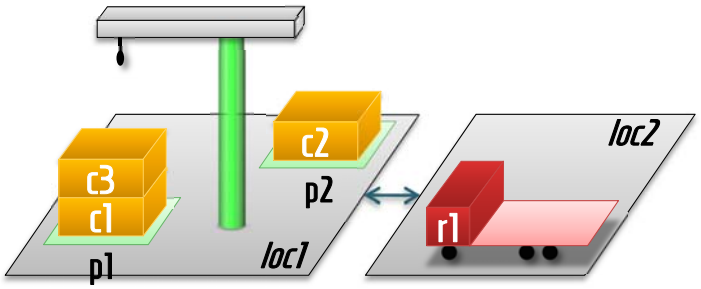
Complete *relative to the model*:
We must know everything
about those predicates and objects
we have specified...
But not whether it's raining!

  - So we can use a set of true atoms
    - {
      attached(p1, loc1), in(c1, p1), on(c1, pallet), in(c3, p1), on(c3, c1), top(c3, p1),
      attached(p2, loc1), in(c2, p2), on(c2, pallet), top(c2, p2),
      belong(crane1, loc1), empty(crane1),
      at(r1, loc2), unloaded(r1), occupied(loc2) ,
      adjacent(loc1, loc2), adjacent(loc2, loc1),
      }



States 4: Goal States

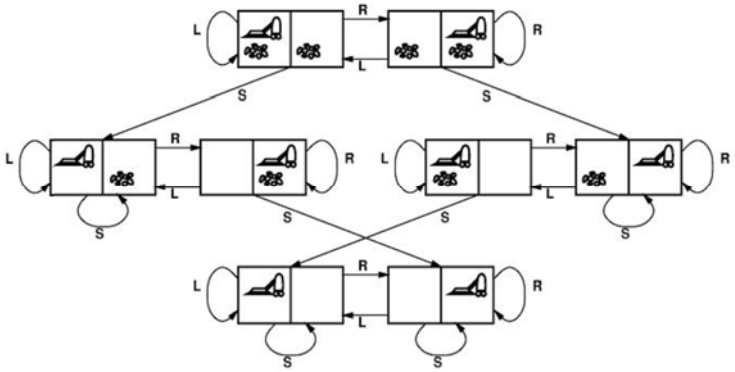
- Classical STRIPS planning: Reach one of possibly many **goal states**
  - Can be specified as a **set of literals** that must hold
  - Example: **Containers 1 and 3 should be in pile 2**
    - We don't care about their order, or any other fact
    - { in(c1,p2), in(c3,p2) }



Actions 1: Intro

- Actions in **plain search** (lectures 2-3):
  - Assumed a *transition / successor function*

Result(*State*, *Action*) - A description of what each action does (Transition function)



  - But how to **specify** it **succinctly**?

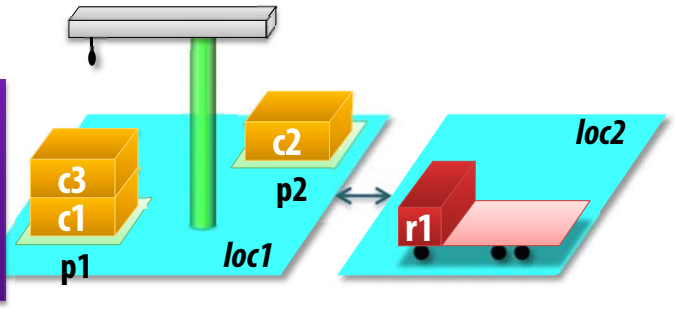
Actions 2: Operators

- Define **operators** or **action schemas**:
  - **move**(robot, location1, location2)
    - Precondition: **at**(robot, location1) ∧ **adjacent**(location1, location2) ∧ ¬ **occupied**(location2)
    - Effects: ¬**at**(robot, location1), **at**(robot, location2), ¬**occupied**(location1), **occupied**(location2)

The action is **applicable** in a state *s* if its precond is true in *s*

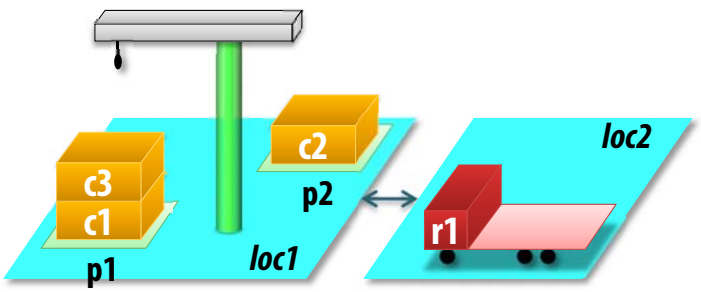
The result of applying the action in state *s*: *s* – negated effects + positive effects

Classical planning:
Known initial state, known state update function ➔ **deterministic**, can completely predict the state of the world after a sequence of actions!



Actions 3: Instances

- The planner **instantiates** these schemas
  - Applies them to any combination of parameters of the correct type
  - **Example: move**(r1, loc1, loc2)
    - Precondition: **at**(r1, loc1) ∧ **adjacent**(loc1, loc2) ∧ ¬ **occupied**(loc2)
    - Effects: ¬**at**(r1, loc1), **at**(r1, loc2), ¬**occupied**(loc1), **occupied**(loc2)



Actions 4: Step by Step

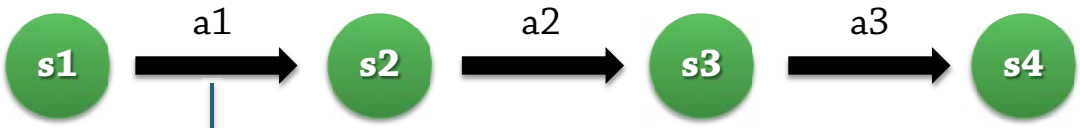
jonkv@ida

- In **classical** planning (the basic, limited form):

We know the initial state

Each action corresponds to **one** state update

Time is not modeled,
and never multiple state updates for one action



We know how states are changed by actions

➔ **Deterministic**, can completely predict the state of the world after a sequence of actions!

The **solution** to the problem will be a **sequence** of actions

Plan Generation

With or Without Search

jonkv@ida

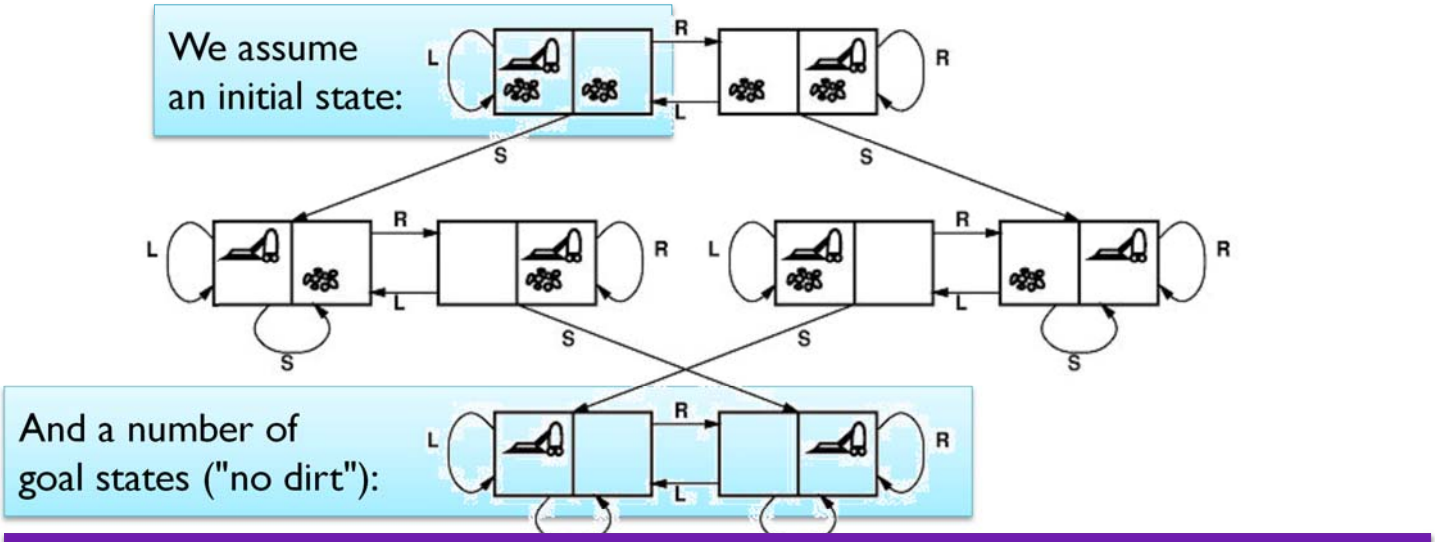
- Important distinction:
  - **Planning** means **deciding in advance** which **actions** to perform in order to **achieve a goal**
  - **Search** will often be a **useful tool**, but you can get by without it

```
plan                    = [ ];
s                       = current state of the world;
while (exists b1,b2 [ s.isOn(b1,b2) ]):
    plan                +=      "unstack(b1,b2)"
    s                   =       apply(unstack(b1,b2), s)
    plan                +=      "putdown(b1)"
    s                   =       apply(putdown(b1), s)
while (exists b1,b2 [ goal.isOn(b1,b2) & !s.isOn(b1,b2) &
                                  s.isClear(b1) & s.isClear(b2) ]):
    plan                +=      "pickup(b1)"
    s                   =       apply(pickup(b1), s)
    plan                +=      "stack(b1,b2)"
    s                   =       apply(stack(b1,b2), s)
return plan
```

If we use search:
What search space?

Plan Generation Method 1: Forward State Space Search

- Forward state space search: As explored in the **Vacuum World**
  - A **search node** is simply a **world state**
  - Successor / transition function:
    - One outgoing edge for every executable (applicable) action
    - The action specifies where the edge will lead



Find a path (not necessarily shortest) – SRS, RSLS, LRLRLSSSRLRS, …

How does a state space **look**?

State Spaces 1: Towers of Hanoi

Toy Problem 1: Towers of Hanoi has a very *regular* state space…

General Knowledge: Problem Domain

- There are *pegs* and *disks*
- A disk can be *on* a peg
- One disk can be *above* another
- Actions:
  Move topmost disk from *x* to *y*, without placing larger disks on smaller disks

Specific Knowledge and Objective: Problem Instance

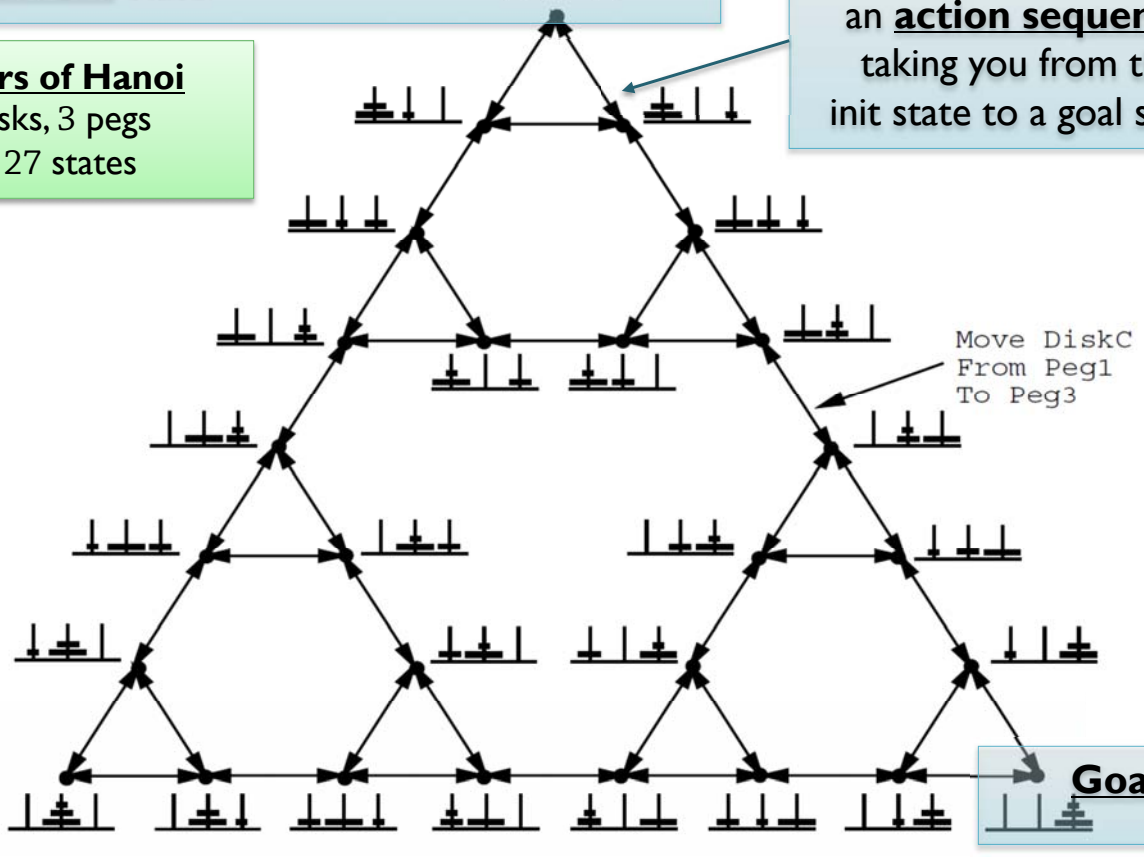
- 3 pegs, 7 disks
- All disks currently on peg 2, in order of increasing size
- We want: All disks on the *third* peg, in order of increasing size

State Spaces 2: ToH, Actions

Initial/current state

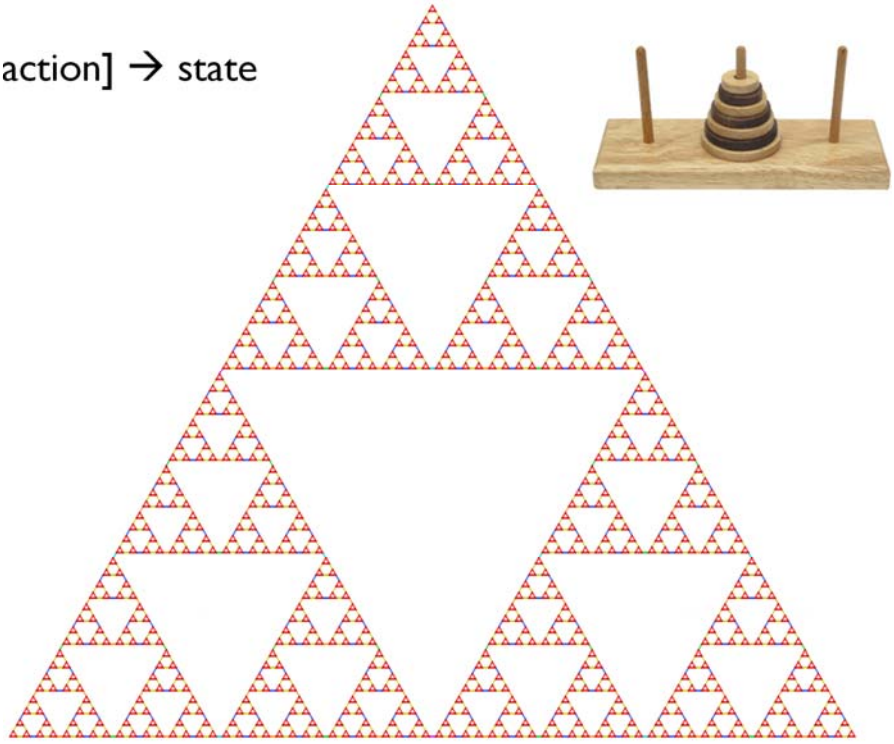
Towers of Hanoi
3 disks, 3 pegs
➔ 27 states

A classical solution plan is an **action sequence** taking you from the init state to a goal state



State Spaces 3: Larger Example

- Larger state space – interesting symmetry
  - 7 disks
  - 2187 states
  - 6558 transitions, [state, action] → state



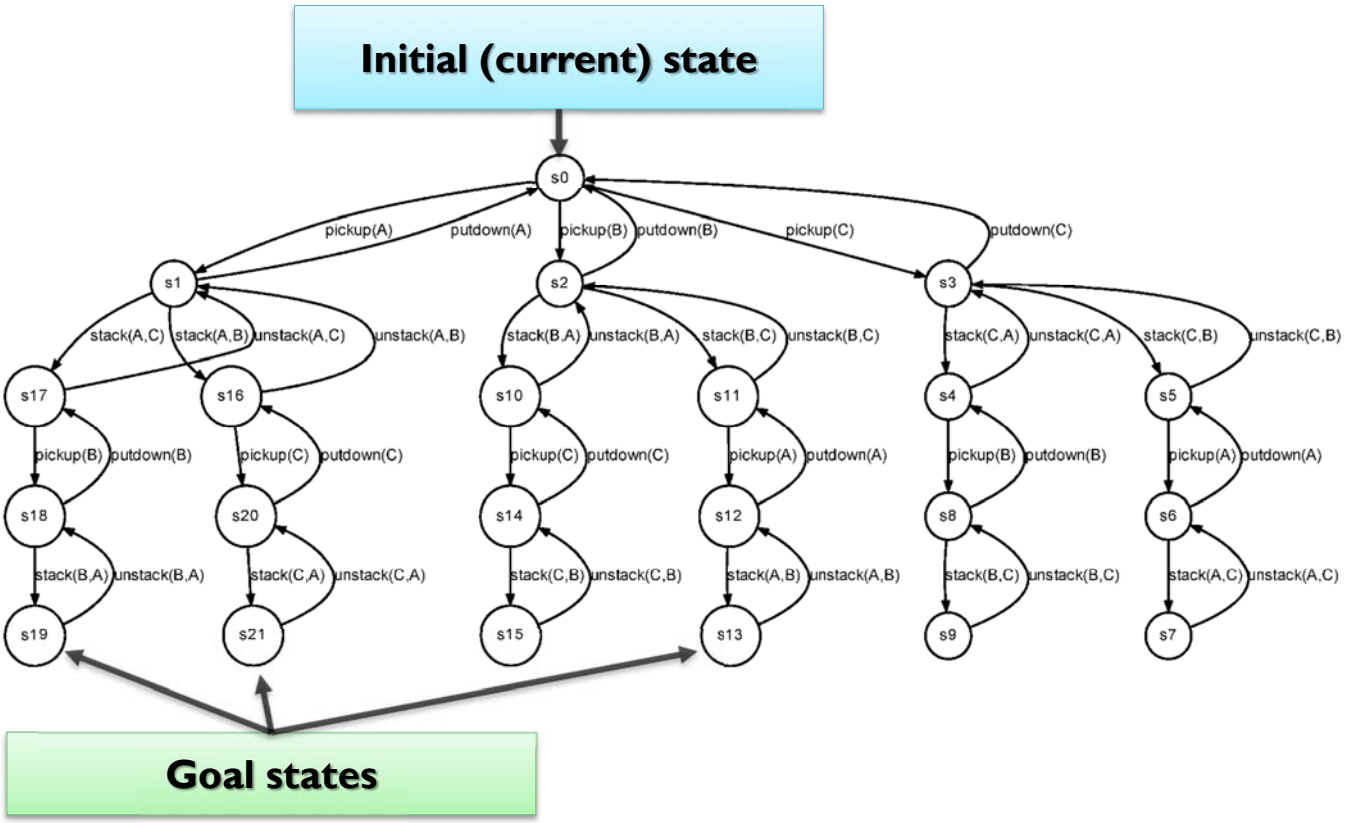
State Spaces 4: Blocks World

- A common blocks world version, with **4 operators**
  - **pickup**(x) – takes x from the table
  - **putdown**(x) – puts x on the table
  - **unstack**(x, y) – takes x from on top of ?y
  - **stack**(x, y) – puts x on top of y

- Predicates (relations) used:
  - **on**(x, y) – block x is on block y
  - **ontable**(x) – x is on the table
  - **clear**(x) – we can place a block on top of x
  - **holding**(x) – the robot is holding block x
  - **handempty** – the robot is not holding any block

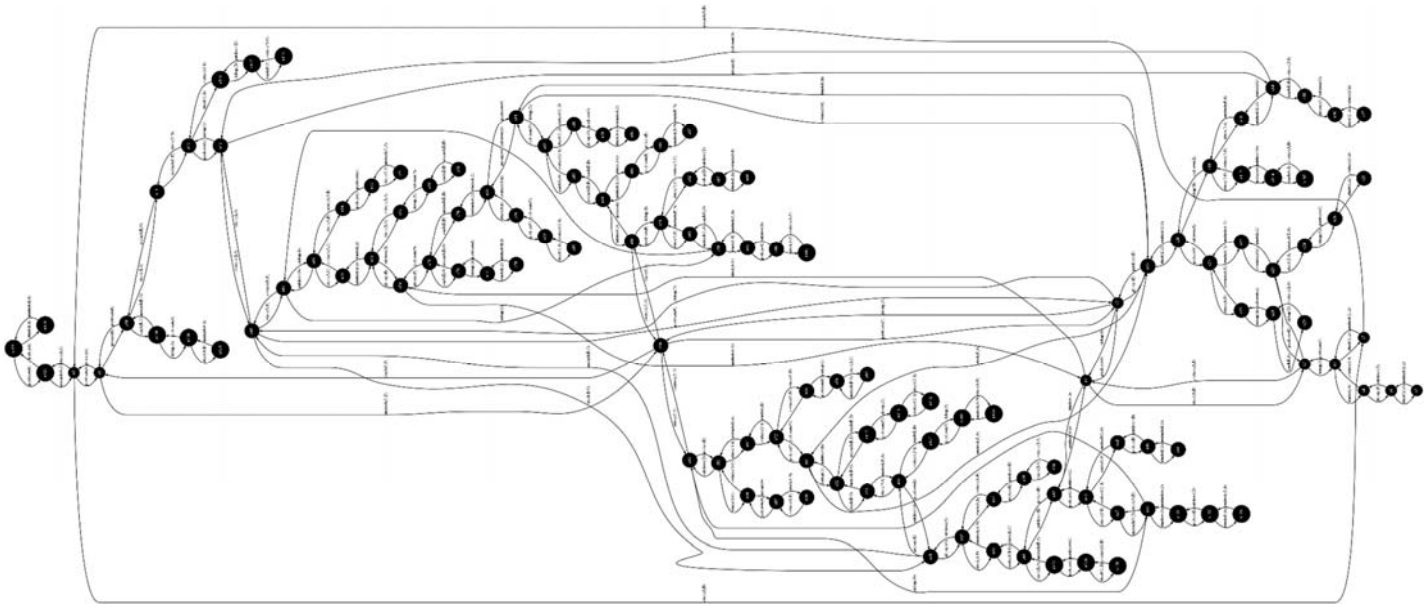
clear(A)
on(A, C)
ontable(C)
clear(B) ontable(B)
clear(D) ontable(D)
handempty

State Spaces 5: Blocks World, 3 blocks

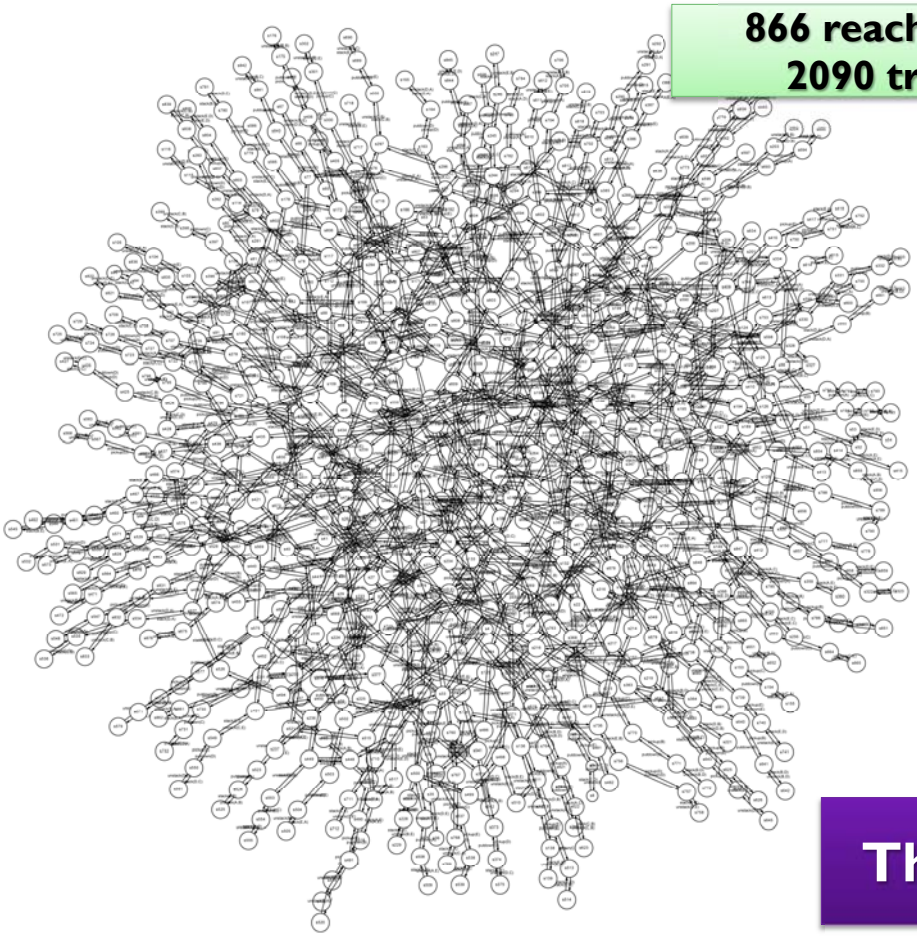


State Spaces 6: Blocks World, 4 blocks

125 reachable states
272 transitions



State Spaces 7: Blocks World, 5 blocks



866 reachable states
2090 transitions

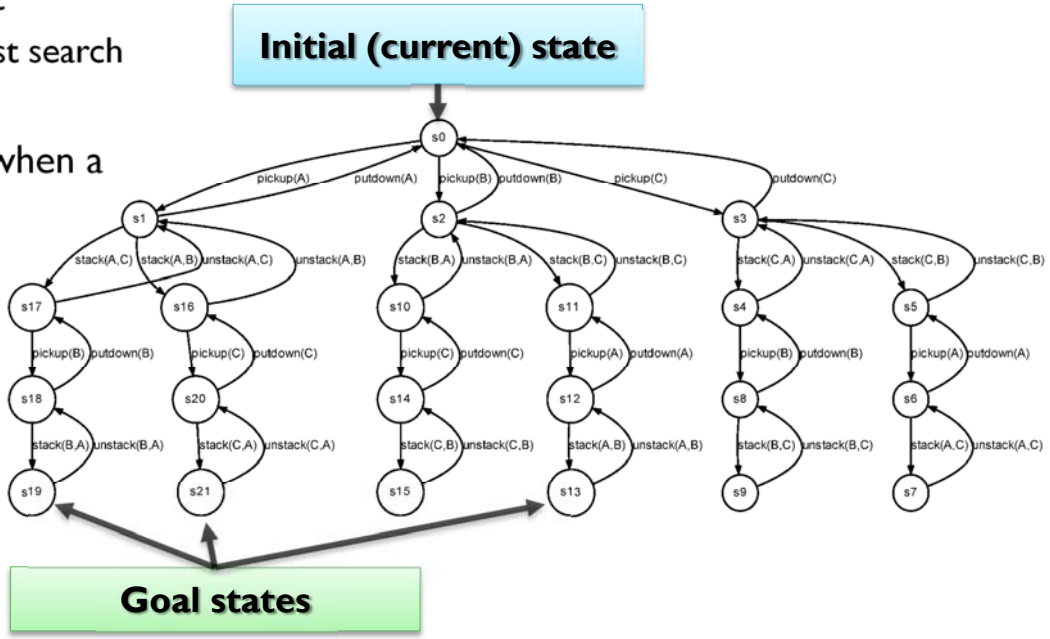
This is tiny!

State Spaces 8: Reachable State Space

Blocks	States reachable from "all on table"	Transitions (edges) in reachable part
0	1	0
1	2	2
2	5	8
3	22	42
4	125	272
5	866	2090
6	7057	18552
7	65990	186578
8	695417	2094752
9	8145730	25951122
10	…	…
…30	>197987401295571718915006598239796851	

Forward Search 1

- Forward search:
  - **Start** in the initial state
  - Apply a **search** algorithm
    - Depth first
    - Breadth first
    - Uniform-cost search
    - …
  - **Terminate** when a goal state is found



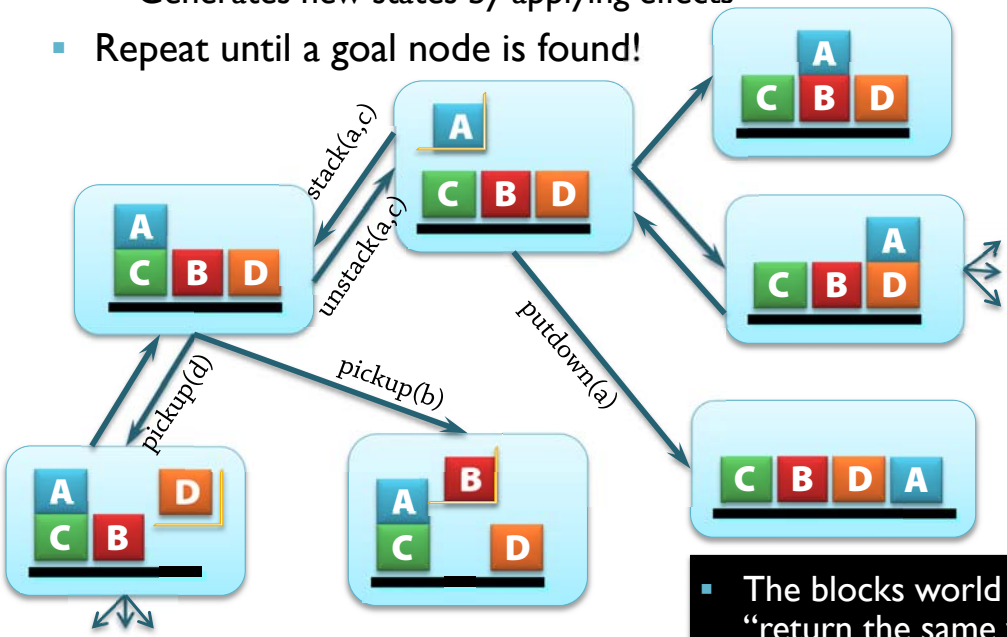
Forward Search 2: Step by step

Extremely many states – must generate the graph as we go

- **Generate** the initial node == initial state
  - From the initial state *description* in the input

Forward Search 3: Step by step

- **Incremental expansion**: **Choose** a node (using **search strategy**)
- Expand all possible successors
  - *"What actions are applicable in the current state, and where will they take me?"*
  - Generates new states by applying effects
- Repeat until a goal node is found!



- The blocks world is *symmetric*: Can always "return the same way"
- Not true for all domains!

Dijkstra

- Which search strategy to use?
  - Breadth first, depth first, iterative deepening depth first, …
- One possible strategy: **Dijkstra's algorithm**
  - Uniform-cost search, but terminate when a goal state is found
  - **Efficient**: $O(|E| + |V| \log |V|)$
    - $|V|$ = the number of nodes
    - $|E|$ = the number of edges
  - And generates **optimal** (cheapest) paths/plans!

Would this algorithm be a solution?

Dijkstra's Algorithm: Analysis

- Blocks world, 400 blocks initially on the table, goal is a 400-block tower
  - Given that all actions have the same cost, Dijkstra will first consider **all** plans that stack **less than 400 blocks**!
    - Stacking 1 block: = 400*399 plans, …
    - Stacking 2 blocks: > 400*399 * 399*398 plans, …
  - More than

163056983907893105864579679373347287756459484163478267225862419762304263994207997664258213955766581163654137118
163119220488226383169161648320459490283410635798745232698971132939284479800304096674354974038722588873480963719
240642724363629154726632939764177236010315694148636819334217252836414001487277618002966608761037018087769490614
847887418744402606226134803936935233568418055950371185351837140548515949431309313875210827888943337113613660928
318086299617953892953722006734158933276576470475640607391701026030959040303548174221274052329579637773658722452
549738459404452586503693169340[redacted]0912754853265795909113444084441755664211796
274320256992992317773749830375[redacted]882657444844563187930907779661572990289194
810585217819146476629300233604[redacted]372350568748665249021991849760646988031691
394386551194171193331440315[redacted]302649432305620215568850657684229678385177
725358933986112127352452988031[redacted]872017424323607291625273875080732255786 30
777685901637435541458440833878[redacted]488351917210773338752306956814809908671 09
051332104820413607822206465352727110739066118003761944104289000710136954383590946416822538563947433356785458 24
320932106973317498515711006719985304982604755110167254854766188619128917053933547098435020659778689499606904157
077005797632287669764145095581565056589811721520434612770594950613701730879307727141093526534328671360002096924
483494302424649061451726645947585860104976845534507479605408903828320206131072217782156434204572434616042404375
211052324038225805405713157329159846351931265562731096039371882295044 00

1.63 * 10^1735

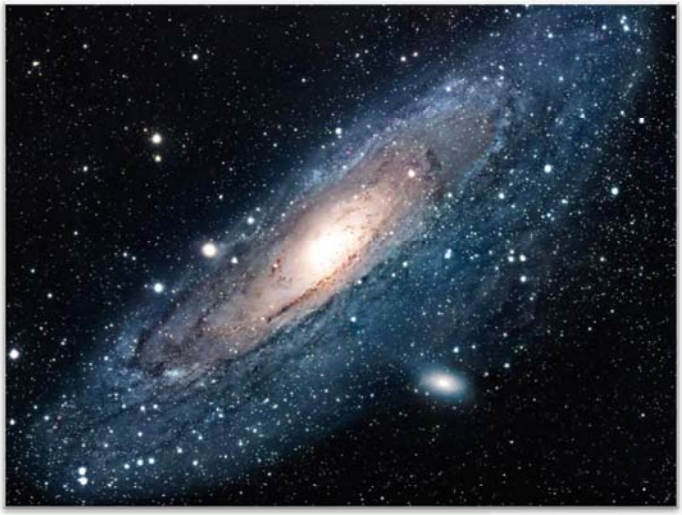
Efficient in terms of the **search space size**: O(|E| + |V| log |V|)

The search space is **exponential** in the size of the input description…

Fast Computers, Many Cores

- But computers are getting **very fast**!
  - Suppose we can check 10^20 states per second
    - >10 billion states *per clock cycle* for today's computers, each state involving complex operations
  - Then it will only take 10^1735 / 10^20 = 10^1715 seconds…

- But we have **multiple cores**!
  - The universe has at most 10^87 particles, including electrons, …
  - Let's suppose every one is a CPU core
  - ➔ only 10^1628 seconds > 10^1620 years
  - The universe is around 10^10 years old



Hopeless? No: We need informed search!

Heuristic Forward State Space Search

Domain-Independent Heuristics

jonkv@ida

- **We need:**
  - A **heuristic function** h(*n*) estimating the cost or difficulty of reaching the goal from node *n*
    - Sometimes, cost = number of actions
    - Sometimes, real costs specified per action
  - A **search strategy** selecting nodes depending on h(n)
- What's the difference from heuristics in the search lectures?
  - Previously we **adapted** heuristics to the problem at hand
    - Romania Travel ➔ straight line distance
    - 8-puzzle ➔ pieces out of place, sum of Manhattan distances
  - Now we want to define a **general** heuristic function
    - Without knowing what planning problem is going to be solved!

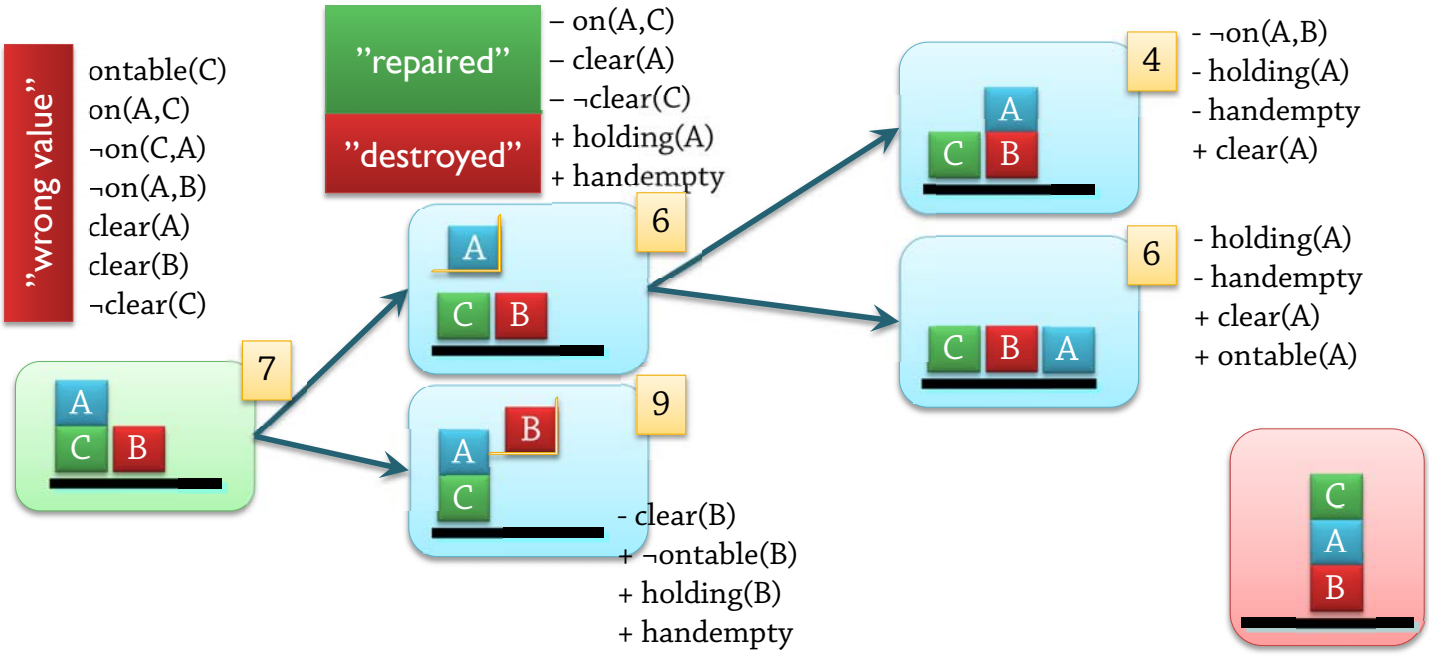
Optimal and Satisficing Planning

- What about admissibility?
  - If we must **guarantee** optimal plans:
    - **Need** optimal search strategies such as A*
    - **Must** use admissible heuristics – or we may get suboptimal plans
  - If we are satisfied with a **high-quality plan**:
    - **Satisficing** planning
      - Find a plan that is sufficiently *good*, sufficiently *quickly* (where "sufficient" is usually not well-defined)
    - Can use **non-admissible** heuristics, **non-optimal** strategies
      - …which can often be more **informative** (give us more help in finding *reasonably good* alternatives)
    - Generally much faster
      - ➔ Handles more complex domains, larger problem instances

Domain-Independent Heuristics (1)

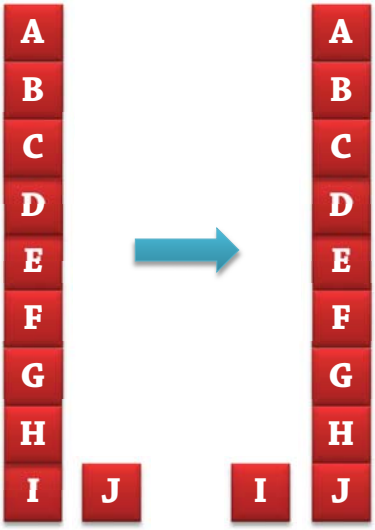
- A very simple **domain-independent** heuristic:
  - **Count** the number of facts that are "wrong" in the current state (similar to "number of pieces out of place")

Optimal:
unstack(A,C)
stack(A,B)
pickup(C)
stack(C,A)



Domain-Independent Heuristics (2)

- Helpful, but not very…



Optimal solution:

unstack(A,B)	pickup(G)
putdown(B)	stack(G,H)
unstack(B,C)	pickup(F)
putdown(C)	stack(F,G)
unstack(C,D)	pickup(E)
putdown(D)	stack(E,F)
…	…
unstack(H,I)	
stack(H,J)	

"Wrong facts": on(H,I), on(H,J), clear(I), clear(J)

Must remove many "good facts" to reach the goal

Creating Admissible Heuristic Functions: The General Relaxation Principle

How does relaxation apply to **planning**?

How can we make the definition more **precise**?

Fundamental ideas

- **We want**:
  - A way to find an **admissible heuristic h(s)** for planning problem P

- One general method:
  1. Find a problem P' that we can **solve quickly**, such that:
     - cost(optimal-solution(P')) <= cost(optimal-solution(P))
  2. **Solve** problem P' **optimally** starting in s, resulting in solution $\pi(s)$
  3. Let $h(s) = \text{cost}(\pi(s))$

Or:
Find the *cost* of an optimal solution directly from P', without actually generating that solution

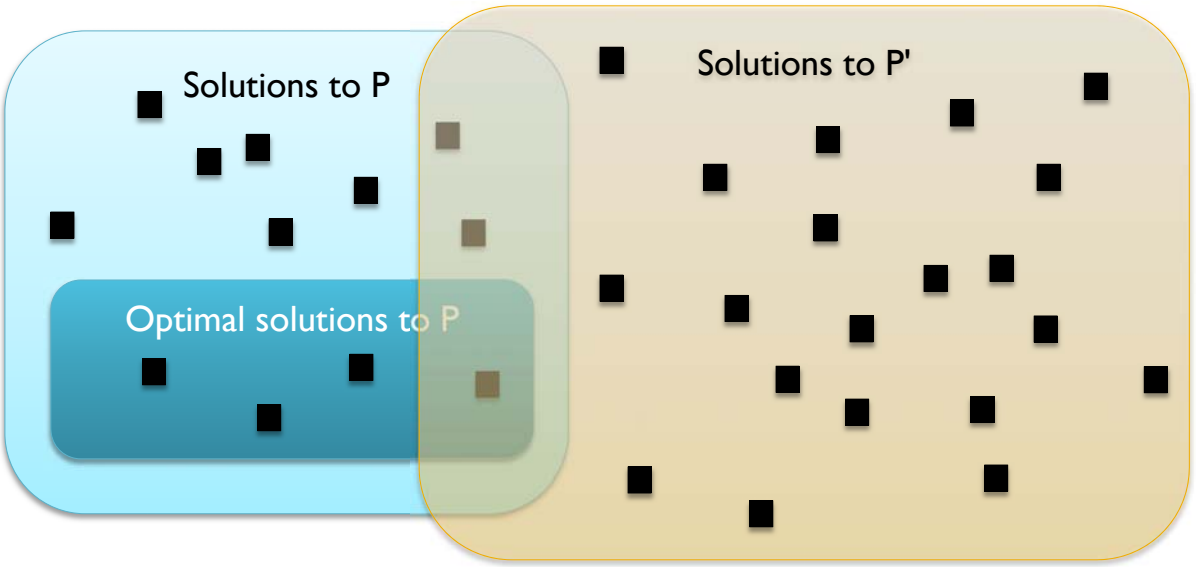
("sum of Manhattan distances")

- We then know:
  - h(s) = cost($\pi(s)$) = cost(optimal-solution(P')) ≤ cost(optimal-solution(P))
  - h(s) is admissible

Fundamental ideas (2)

- How to find a *different* problem with *equal or lower* solution cost?
  - Sufficient criterion: **One optimal solution** to P **remains** a solution for P'
    - cost(optimal-solution(P')) = min { cost(π) | π is any solution to P' } <= cost(optimal-solution(P))

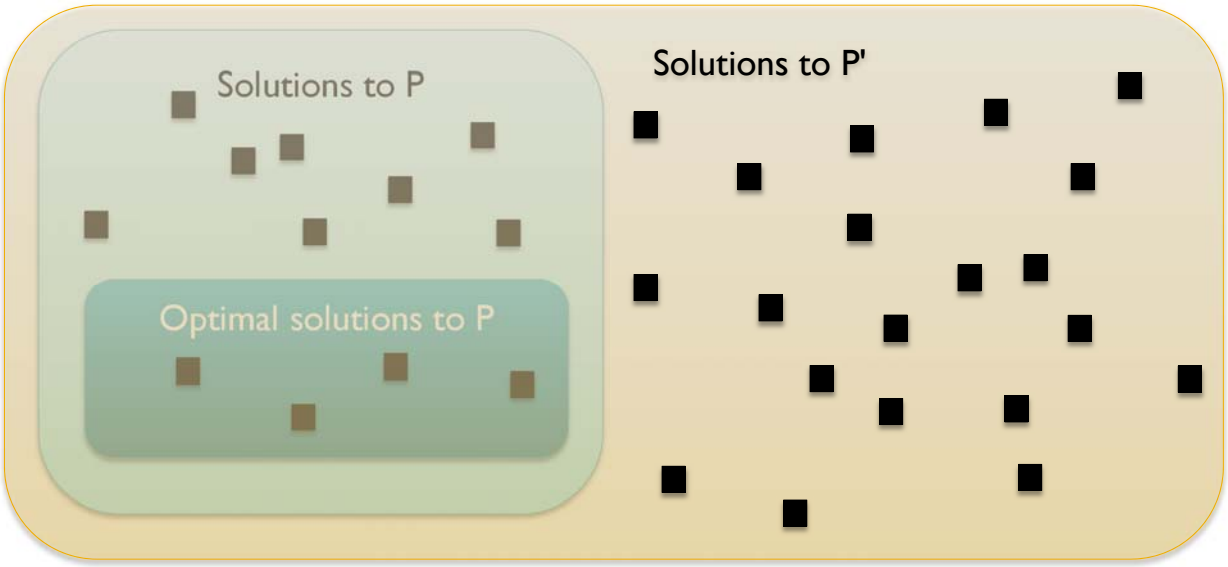
Includes the optimal solutions to P, so min {…} cannot be greater



Fundamental ideas (3)

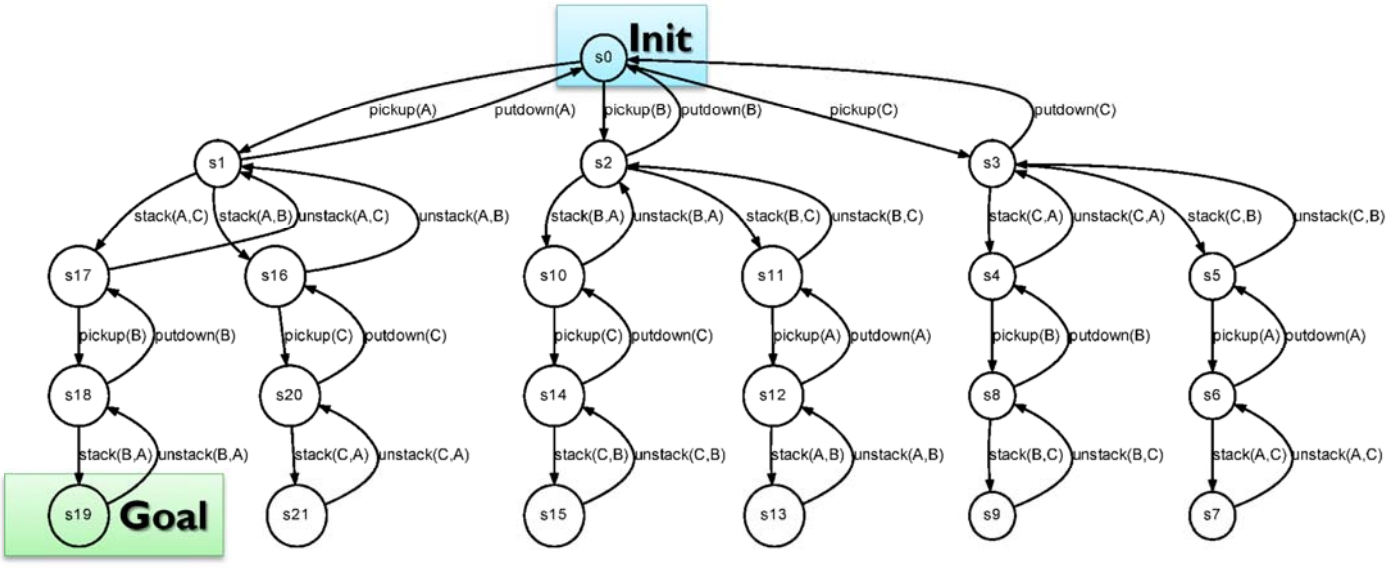
- Another sufficient criterion: **All solutions** to P **remain** solutions for P'
  - Stronger, but often **easier to prove**

- **Relaxes** the constraint on what is accepted as a solution:
  The *is-solution(plan)* test is "expanded, relaxed" to cover additional plans
  - Called **relaxation**



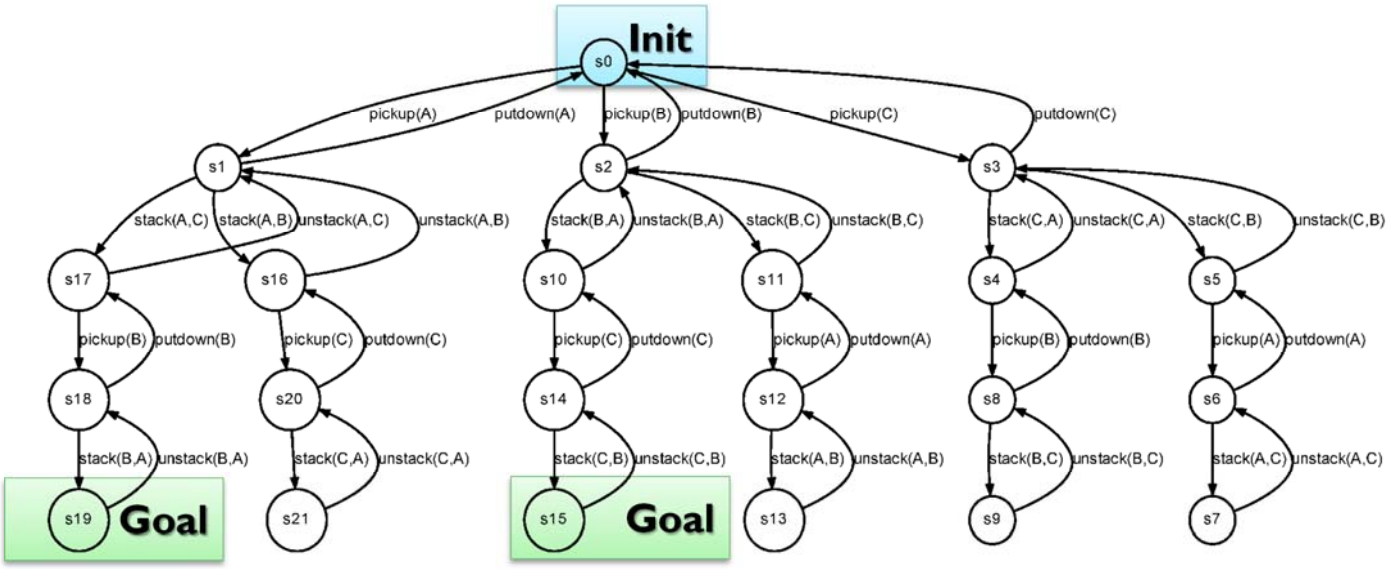
Relaxation Example: Basis

- A simple planning problem (domain + instance)
  - Blocks world, 3 blocks
  - Initially all blocks on the table
  - Goal: (and (on B A) (on A C)) (only satisfied in s19)
  - Solutions: **All** paths from init to goal (infinitely many – can have cycles)



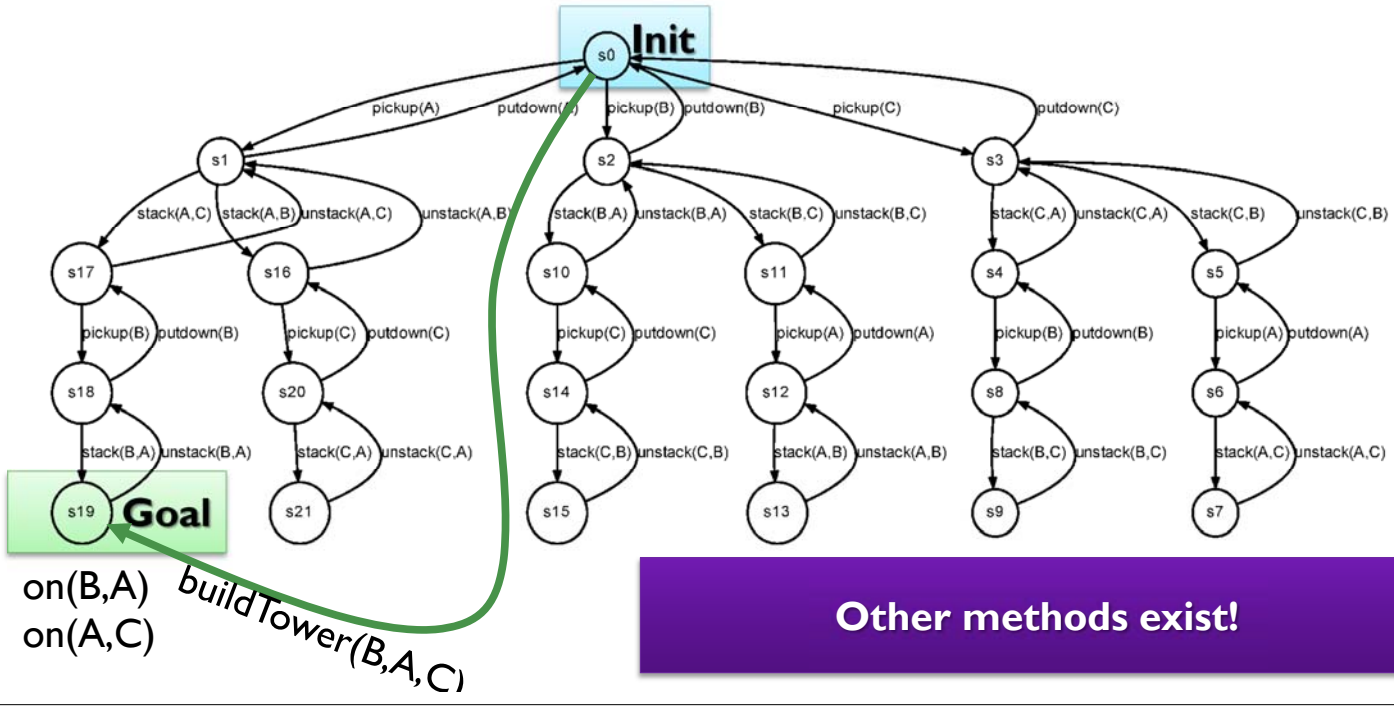
Relaxed Problem 1

- Relaxation "method 1": **Adding goal states**
  - New goal formula: (and (on B A) **(or (on A C) (on C B))**)
    - All old solutions still valid
    - Some non-solution paths become solutions



Relaxed Problem 2

- Relaxation "method 2": **Adding new actions to the domain**
  - **Modifies** the state transition system (states/actions)
    - This particular example: a *shorter* solution becomes possible



Understanding Relaxation

- Important:

Relaxation is not simply "removing preconditions"
There are **many other relaxations**
Key: Preserving old solutions!

You **cannot** "use a relaxed problem as a heuristic".
What would that mean? The problem is not a number or function…
You use the **cost** of an **optimal solution** to the relaxed problem as a heuristic.

If you just take the cost of **any** solution to the relaxed problem,
it can be inadmissible!

You have to solve it **optimally** to get the admissibility guarantee.

You don't just solve the relaxed problem once.
Every time you reach a new state and want to calculate a heuristic value,
you have to solve the relaxed problem
of getting from **that** state to the goal.

Understanding Relaxation (2)

You've seen relaxations adapted to the 8-puzzle

What can you do automatically,
independently of the planning domain used?

Example:
Pattern Database Heuristics

PDB 1: State Variables

- Let's use a **non-Boolean state variable** representation
  - Example for 4 blocks:
    - **aboveA** ∈ { clear, B, C, D, holding }
    - **aboveB** ∈ { clear, A, C, D, holding }
    - **aboveC** ∈ { clear, A, B, D, holding }
    - **aboveD** ∈ { clear, A, B, C, holding }
    - **posA** ∈ { on-table, other }
    - **posB** ∈ { on-table, other }
    - **posC** ∈ { on-table, other }
    - **posD** ∈ { on-table, other }
    - **hand** ∈ { empty, full }

PDB 2: Ignoring Variables

- Create an **abstraction** of the planning problem
  - **Ignoring** some of the state variables
  - Everywhere: In preconditions, effects, states, and goals

- Example: Care about **{ aboveB, aboveD, posB, posD }**
  - **Rewrite** the goal
    - Original: { aboveA = B, aboveB = C, aboveC = D, aboveD = clear, hand = empty }
    - Abstract: { **aboveB = C,** **aboveD = clear**}

  - **Rewrite** actions, removing some preconditions / effects
    - (unstack D C) still requires aboveD = clear
    - But not necessarily aboveC = D

  - …

PDB 3: Patterns

- The **set of included variables** is called a **pattern**
  - States containing only these variables are called **abstract states**:
    Many states **match** a single abstract state

Example: Pattern { aboveB, aboveD, posB, posD }

aboveA=clear, aboveB=clear, aboveC=clear, aboveD=clear, posA=on-table, posB=on-table, posC=on-table, posD=on-table, hand=empty

≈

aboveA=C, aboveB=clear, aboveC=A, aboveD=clear, posA=on-table, posB=on-table, posC=on-table, posD=on-table, hand=full

≈

aboveA=B, aboveB=clear, aboveC=D, aboveD=clear, posA=on-table, posB=on-table posC=on-table, posD=on-table, hand=empty

represented by a single abstract state

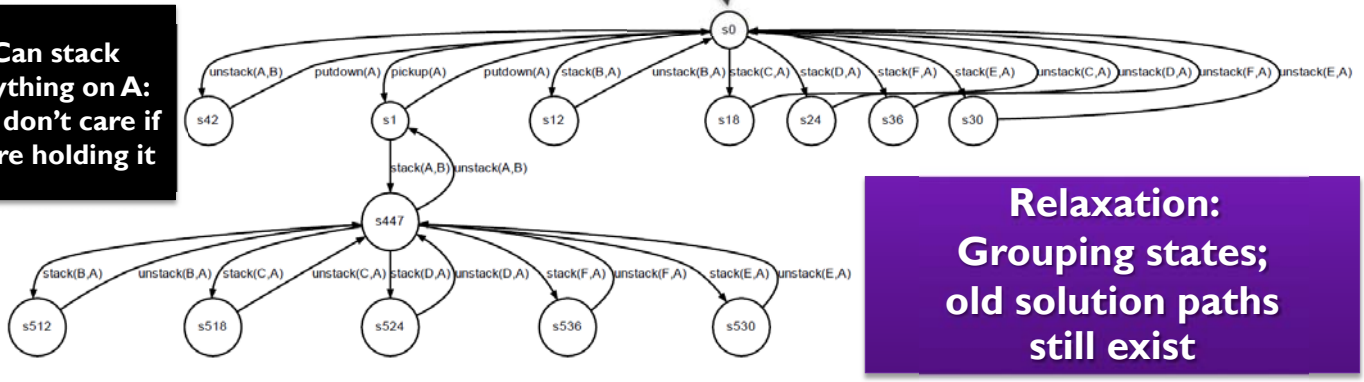
aboveB=clear, aboveD=clear, posB=on-table, posD=on-table

PDB 4: State Space Size

- **Abstract** states reachable from "all on table", **by pattern**…

Blocks	All variables	Pattern={aboveA}	{aboveA,aboveB}
4	125	10	96
5	866	12	140
6	7057	14	192
7	65990	16	252
8	695417	18	320
9	8145730	20	396

Can stack anything on A: We don't care if we're holding it



Relaxation: Grouping states; old solution paths still exist

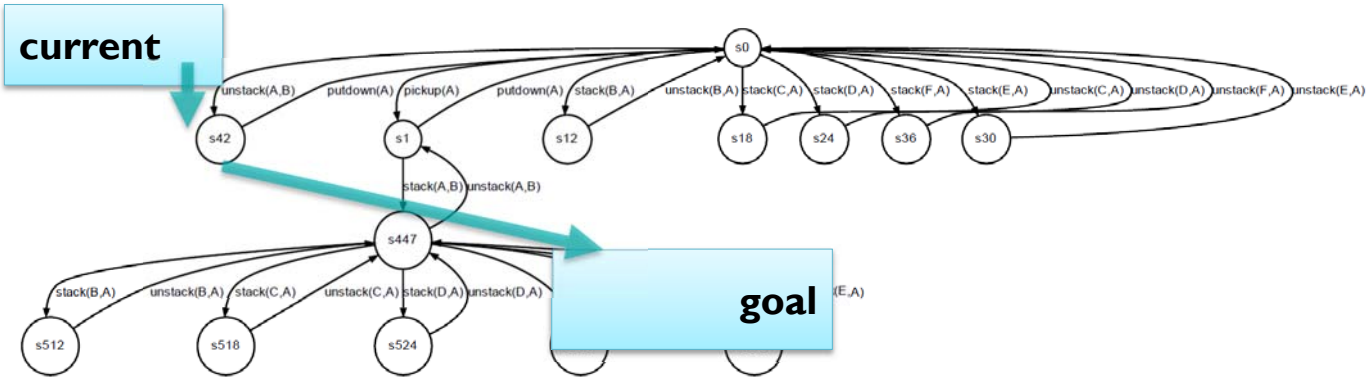
PDB 5: Main Idea

- To calculate $h(s)$ for a newly encountered state s:
  - Convert to abstract state

aboveA=B,
aboveB=clear,
aboveC=D,
…

→ aboveA=B

  - Find **optimal** path to abstract goal state – in a much smaller search space!
    - Relaxation ➔ its **cost** is an admissible heuristic

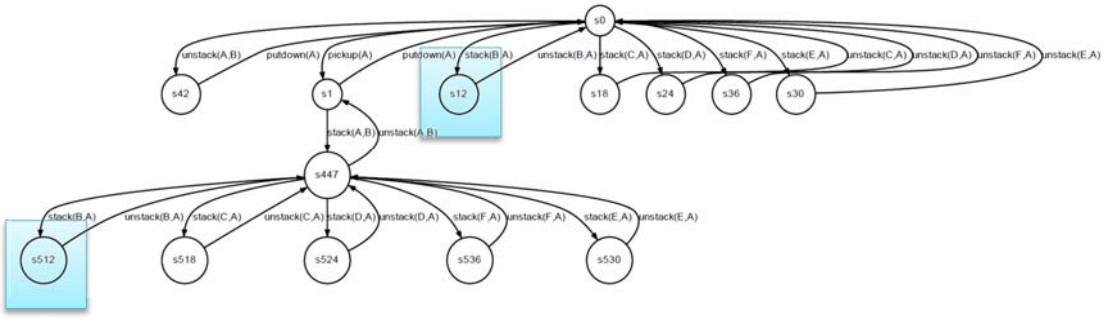


PDB 6: Main Idea

- To make PDB heuristics **more informative**:
  - Calculate costs for **several** patterns
    - {aboveA,aboveB} and {aboveC,aboveD} and …
  - Take the **maximum** of the computed heuristic values
- For even more information:
  - Use patterns satisfying specific **additivity constraints**, measuring *different aspects* of the total cost
  - The **sum** of the computed heuristic values is admissible
- Real difficulty:
  - Choosing which patterns to use…

PDB 7: Databases

- Where did the databases go?
  - Planning visits many **abstract** states over and over again
  - For each pattern, we **precompute** all reachable abstract states



  - Determine which ones satisfy the **abstract goal**
    - Abstract: { aboveA = B }
  - Calculate shortest paths from ***every*** abstract state to ***any*** abstract goal state
    - Dijkstra's algorithm backwards – possible in a small search space
    - Store in a fast look-up table, a *database*

Example:
Landmark Heuristics

Intention:
Get a **rough idea** about how planners might find guidance!

Landmark Heuristics (1)

Landmark:
”a **geographic feature** used by explorers and others
to **find their way** back or through an area”

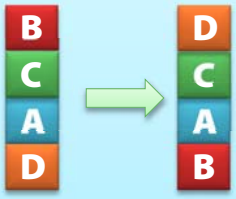
Landmark Heuristics (2)

Landmarks in planning:
Something you must *pass by/through* in *every solution* to a specific planning problem

Assume we are currently in state *s*…

Fact Landmark for *s*:

A **fact** that is **not true** in *s*,
but must be true at some point
in *every* solution starting in *s*

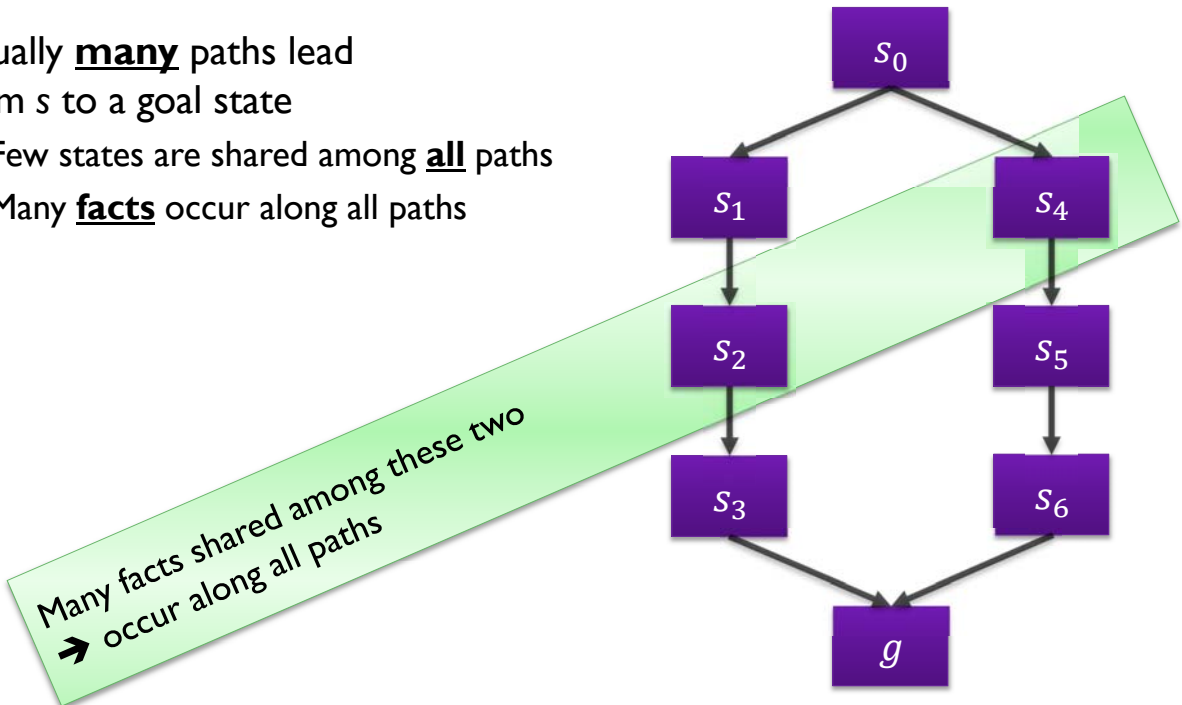


clear(A)
holding(C)
…

Landmark Heuristics (3)

Facts and formulas, not complete states!

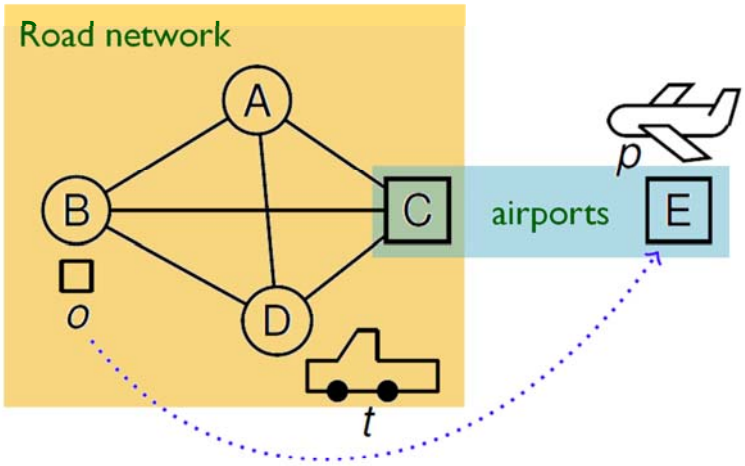
- Usually **many** paths lead from *s* to a goal state
  - Few states are shared among **all** paths
  - Many **facts** occur along all paths

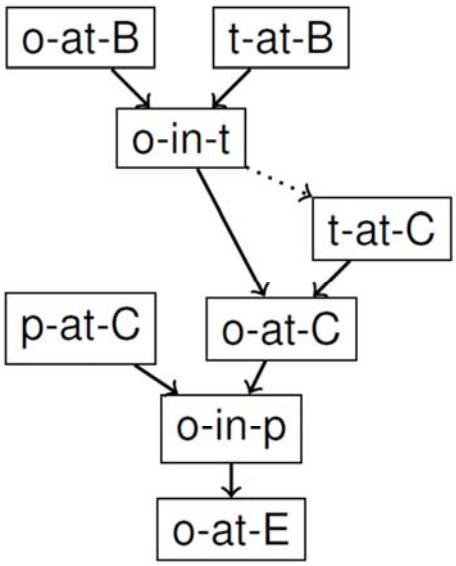


Finding landmarks: Complicated; not covered here

Landmarks as Subgoals

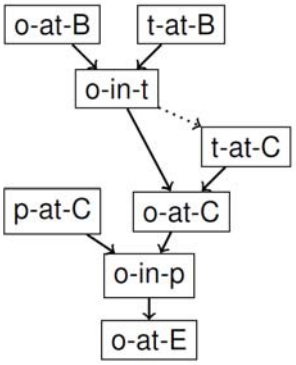
- Use of landmarks:
  - As **subgoals**: Infer necessary **orderings** between landmarks, then try to achieve each landmark **in succession**
    - Example from *Karpas & Richter: Landmarks – Definitions, Discovery Methods and Uses*





Landmark Counts and Costs

- Use of landmarks:
  - As a basis for **heuristic estimates**
    - Used by LAMA, the winner of the *sequential satisficing* track of the International Planning Competition in 2008, 2011
  - LAMA **counts** landmarks:
    - Identifies a set of landmarks that still need to be achieved after reaching state *s* through path (action sequence) π
    - **L(s,π) =** **(L \ Accepted(s,π))** ∪ **ReqAgain(s,π)**



All discovered landmarks, minus those that are *accepted* as achieved (has become true *after* predecessors are achieved!)

Plus those we can show will have to be re-achieved

Search Algorithms for Non-Admissible Heuristics

- Which search algorithm to use?
  - A* **expands many nodes** (expensive),
    and for non-admissible heuristics, *there is still no optimality guarantee*!
  - Some planners use variations on **Hill Climbing**

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current ← MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor ← a highest-valued successor of current
        if neighbor.VALUE ≤ current.VALUE then return current.STATE
        current ← neighbor
```

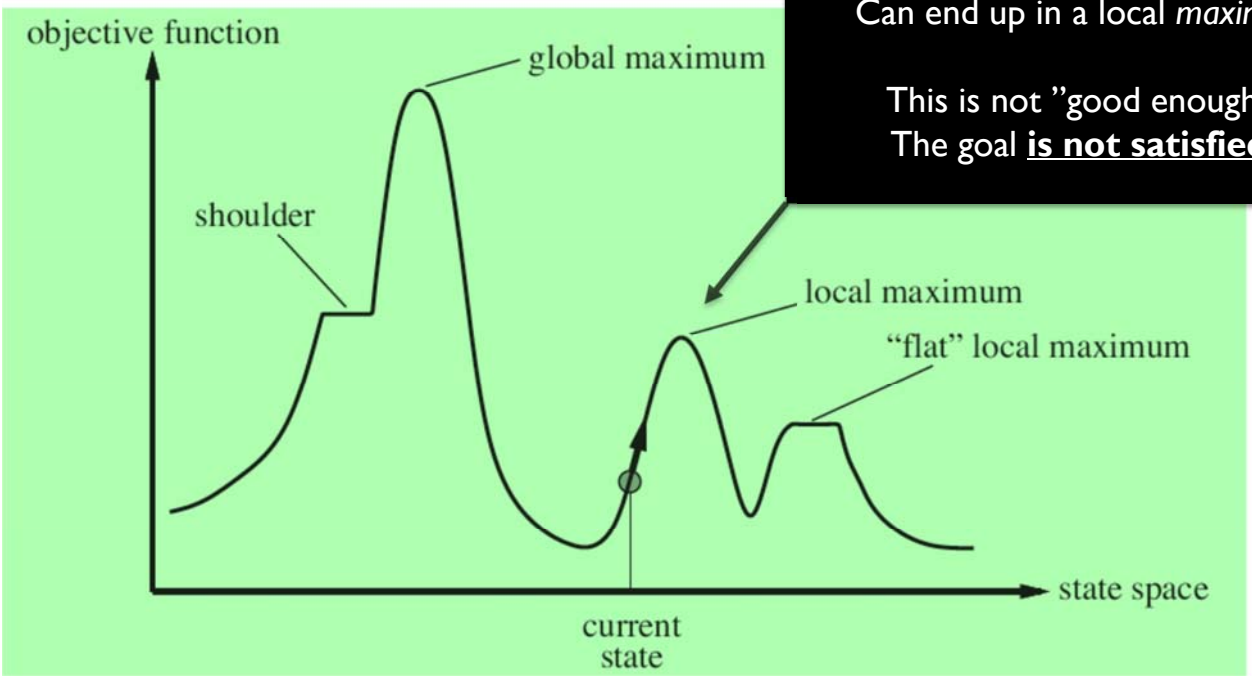
Successors: States created by applying an action to the current state

Value: Calculated as –h(s)
(maximize –h(s) ➔ minimize h(s))

Only considers children of the current node
➔ tries to move quickly towards low heuristic values
➔ doesn't examine "all nodes" with cost(n)+f(n) <= some value

Local Optima in Planning

- Problem:

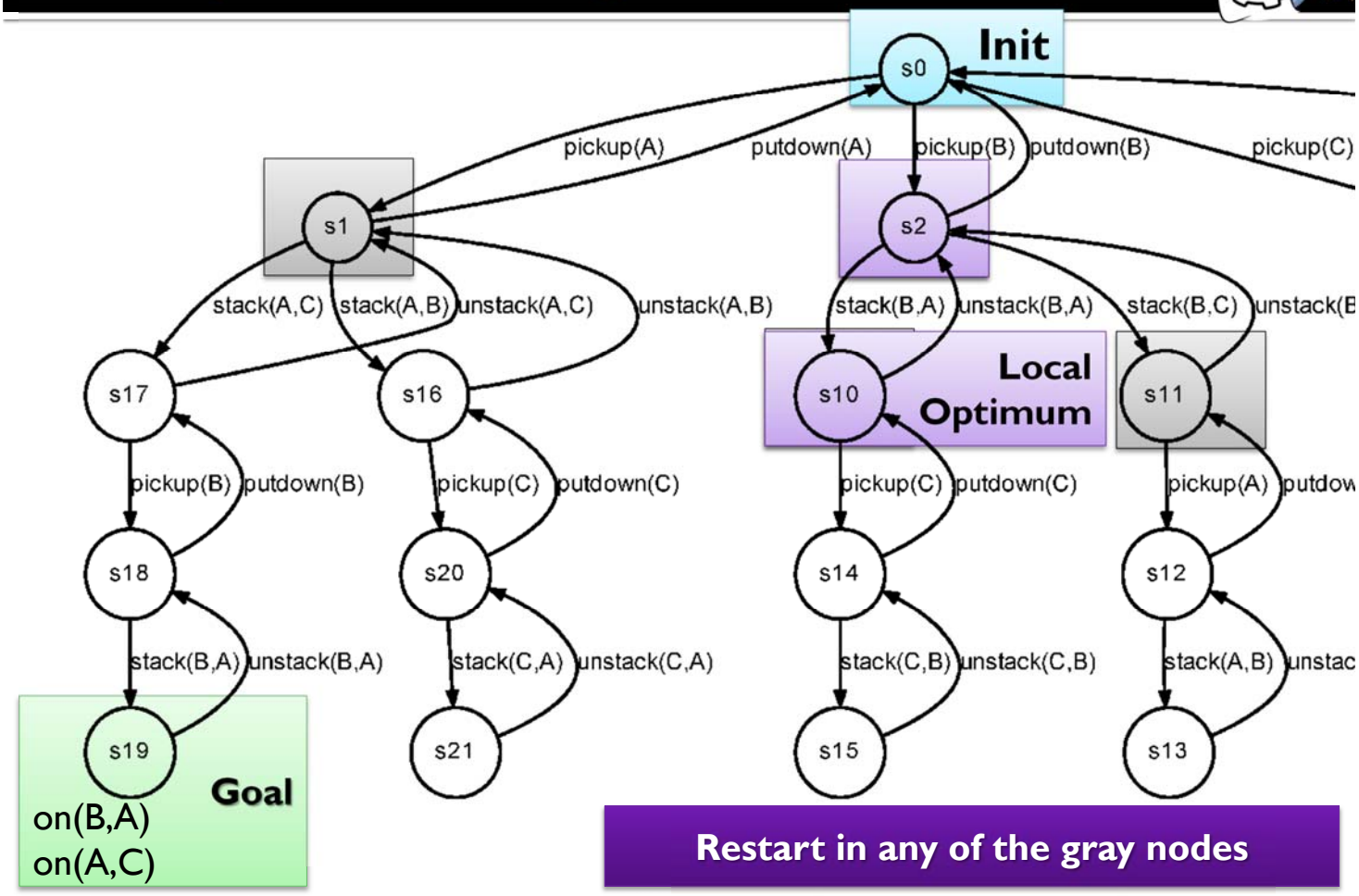


Can end up in a local *maximum*

This is not "good enough":
The goal **is not satisfied**!

Possible solution:
Restart search from another expanded state

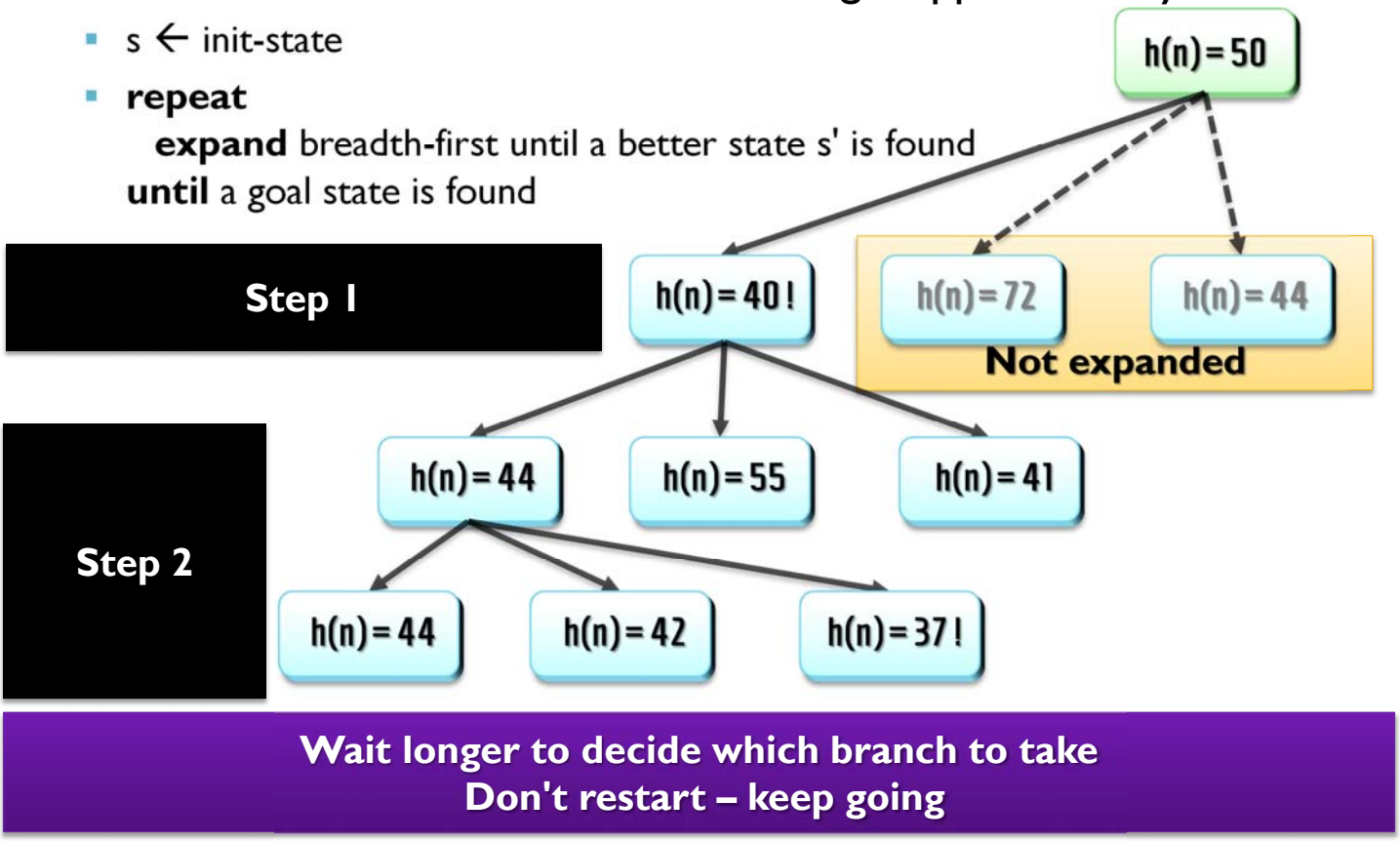
Local Optima: Restarting



Restart in any of the gray nodes

Enforced Hill Climbing

- FastForward uses **enforced** hill climbing – approximately:
  - s ← init-state
  - **repeat**
    expand breadth-first until a better state s' is found
    until a goal state is found



Wait longer to decide which branch to take
Don't restart – keep going

Summary of Forward Search

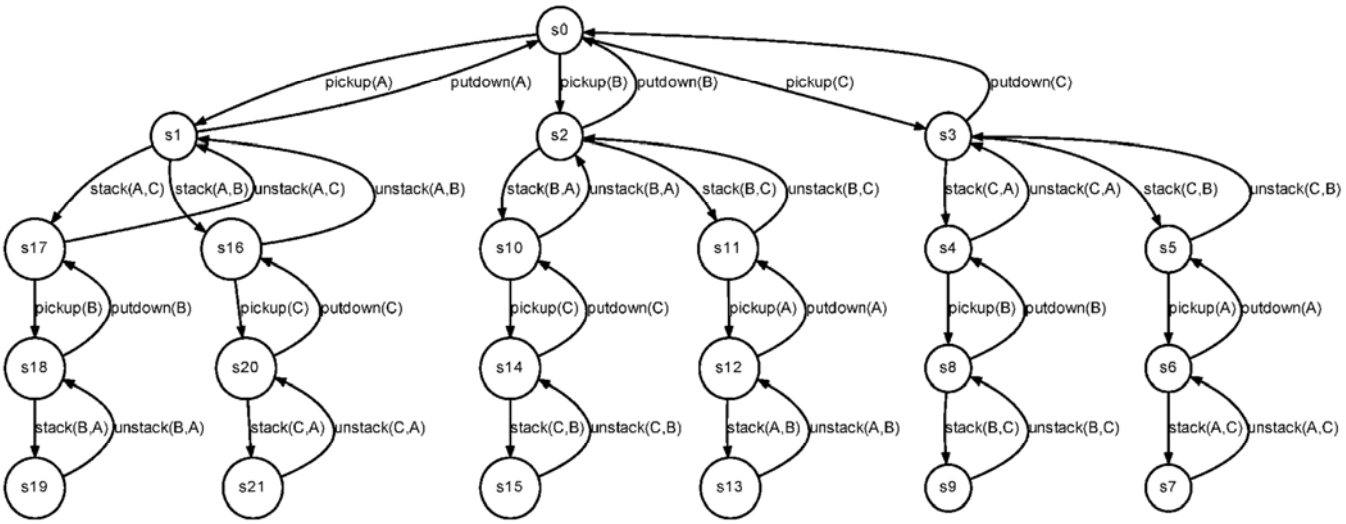
- Forward State Space Search:
  - Very common planning framework
  - **Many** heuristic functions defined
    - TDDC17: Focus on understanding the basics,
      and general principles for defining general heuristics
      without knowing the problem to be solved (blocks, helicopters, …)
    - TDDD48: Many additional heuristics (and principles for creating them)
  - **Many** search strategies defined
    - TDDC17: Focus on applying standard strategies from general search
    - TDDD48: Additional planning-inspired strategies
      and their relations to specific heuristic functions

Plan-Space Search: An overview

Plan-Space Search

jonkv@ida

- Just because we are looking for a path in this graph…

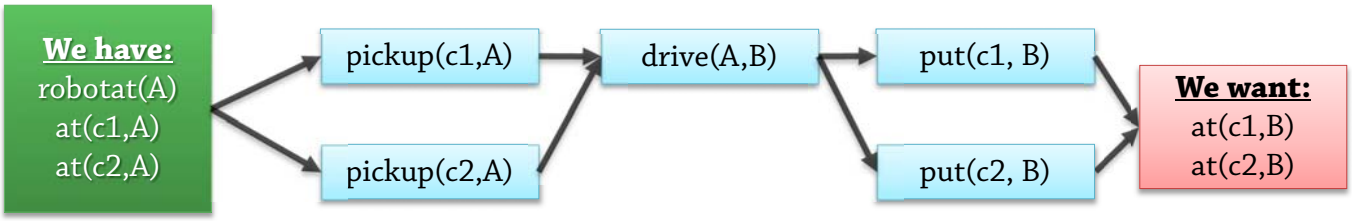


- …doesn't mean we have to use it as a search space!

What if a search step could insert actions <u>anywhere</u> in a plan?

POCL 1: Intuitions

- Idea: Partial Order Causal Link (POCL) planning
  - Means-ends analysis:
    - Find a condition (goal or precondition) that remains to be achieved
    - Add an action that achieves this condition
  - But use a **partial order** for actions!
    - Insert actions "at any point" in a plan
    - **Least/late commitment** to ordering – "don't decide until you have to"

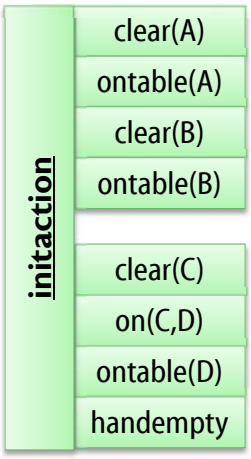


More sophisticated "bookkeeping" required!

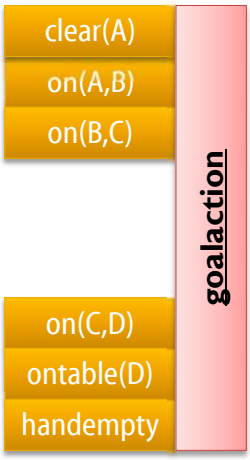
Partial-Order Planning

- How to keep track of initial state and goal?

A "fake" initial action, whose effects are the facts of the initial state



A "fake" goal action, whose preconditions are the conjunctive goal facts



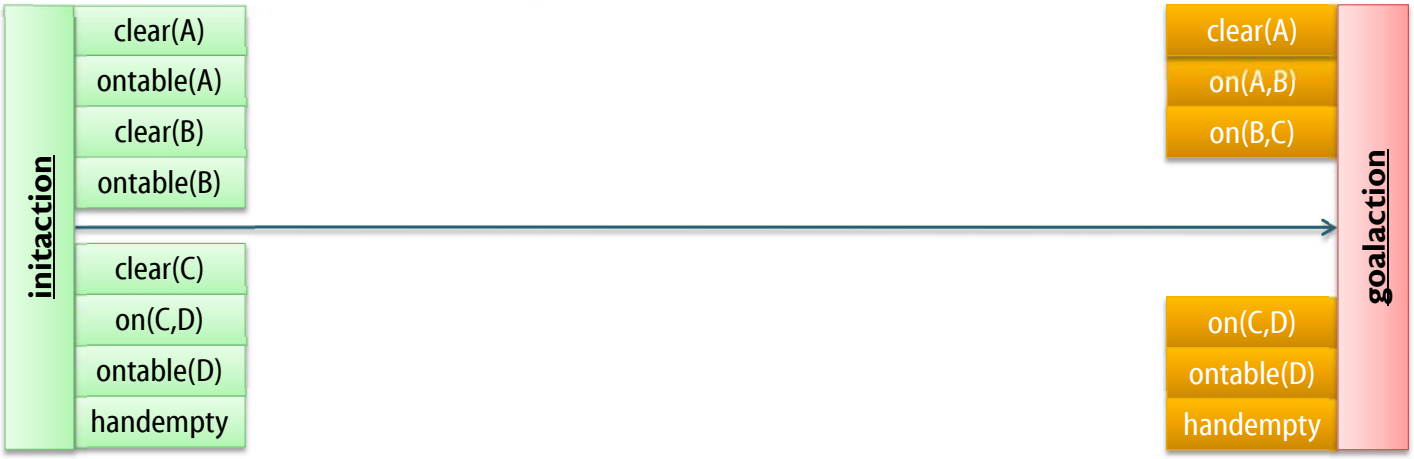
Streamlines the planner: *Initial state* and *action effects* are handled equivalently
goal and *action preconditions* are handled equivalently.

Partial-Order Planning

- No sequence ➔ must have explicit **precedence constraints**

A "fake" initial action, whose effects are the facts of the initial state

A "fake" goal action, whose preconditions are the conjunctive goal facts



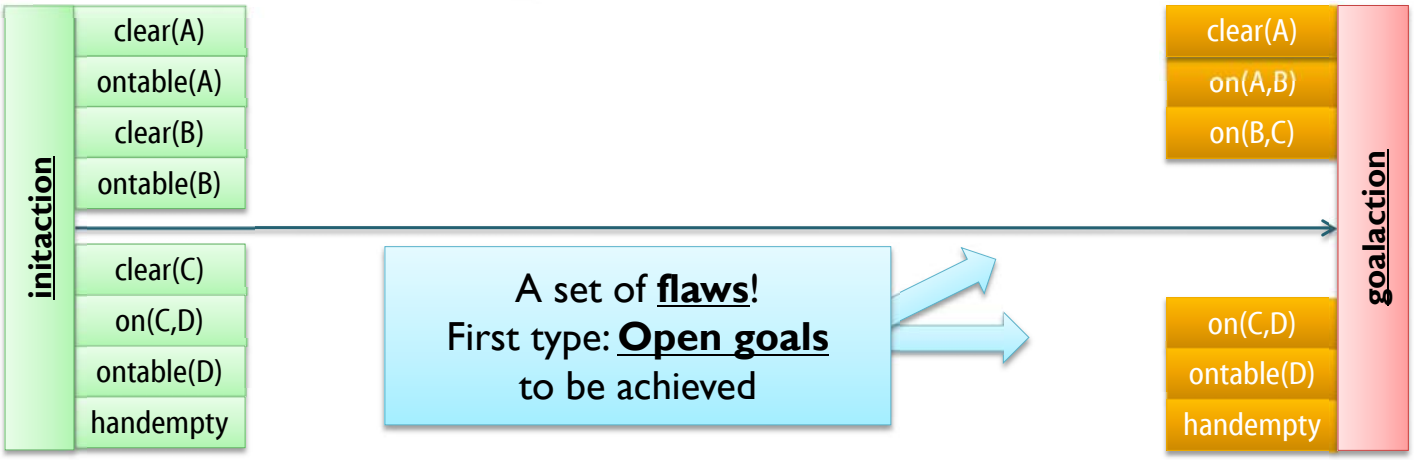
initaction happens *before* goalaction

Partial-Order Planning

- No current state – how to keep track of what remains to do?

A "fake" initial action, whose effects are the facts of the initial state

A "fake" goal action, whose preconditions are the conjunctive goal facts



A set of **flaws**!
First type: **Open goals**
to be achieved

Resolving Open Goals 1

- To resolve an open goal:

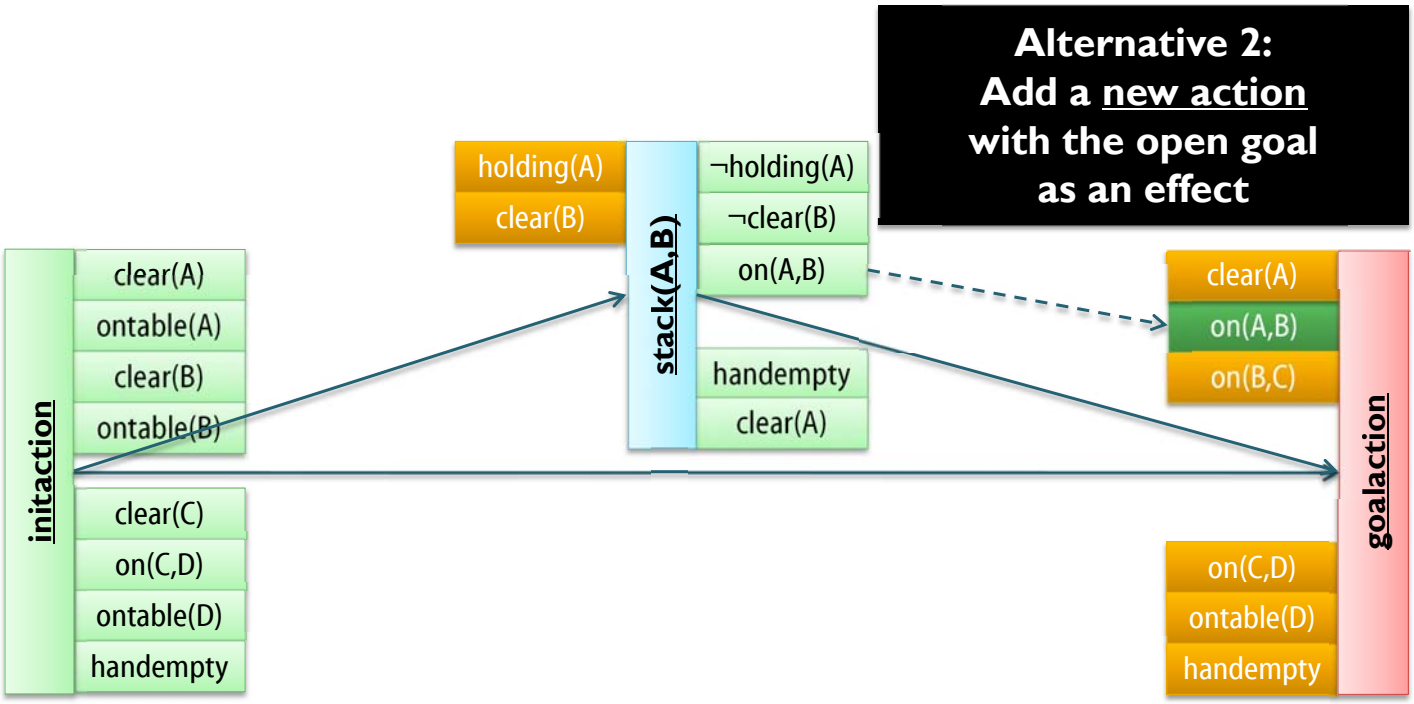
Alternative 1:
Add a causal link from an action that already achieves it

Causal link:
1) "initaction causes (achieves) clear(A) *for* goalaction"
2) "From the start of the link until its end, the property [clear(A)] will be true!"

Resolving Open Goals 2

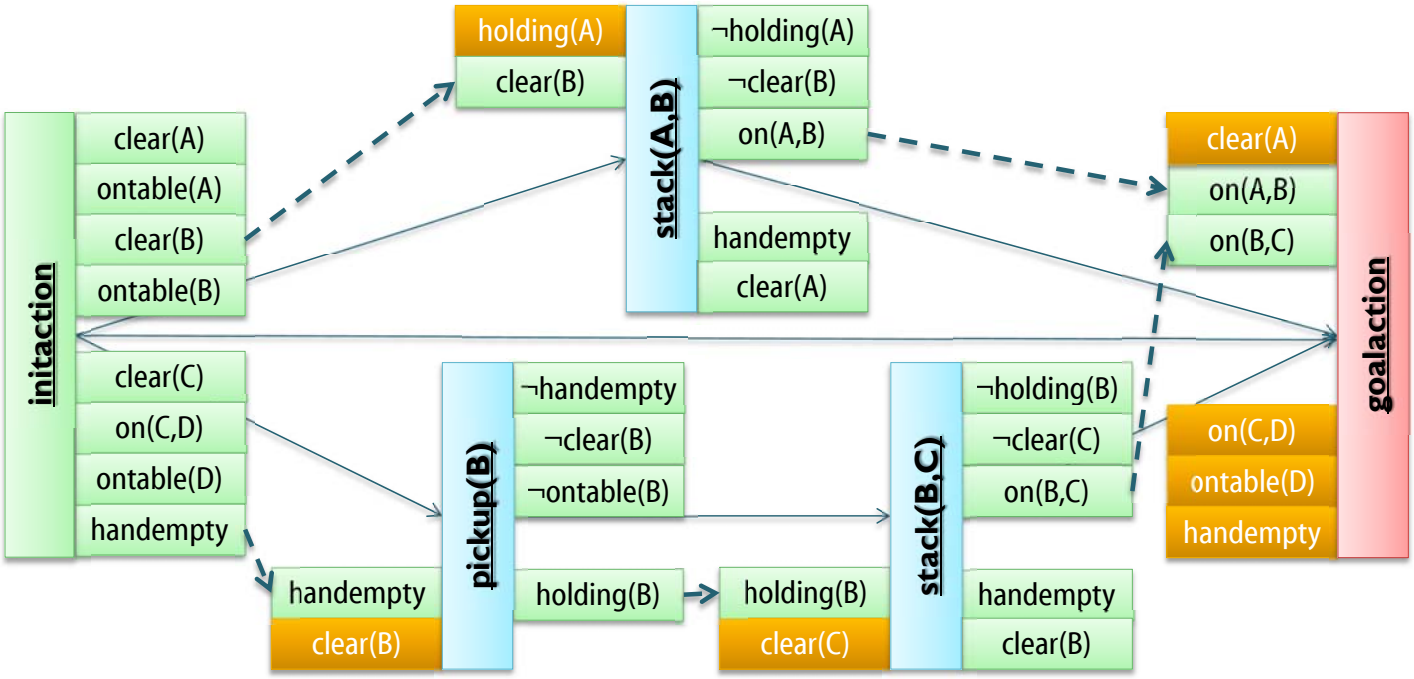
- To resolve an open goal:

Alternative 2:
Add a new action with the open goal as an effect



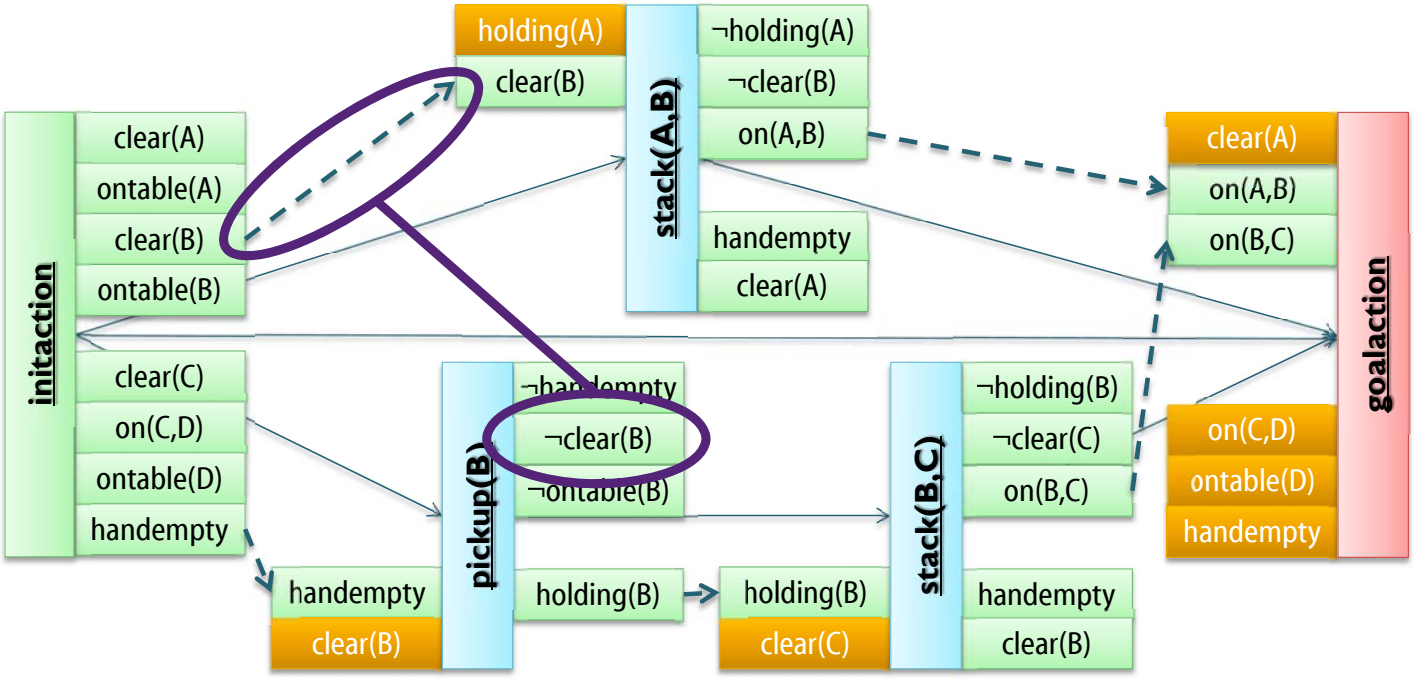
Partial Orders

- Some pairs of actions can remain **unordered**



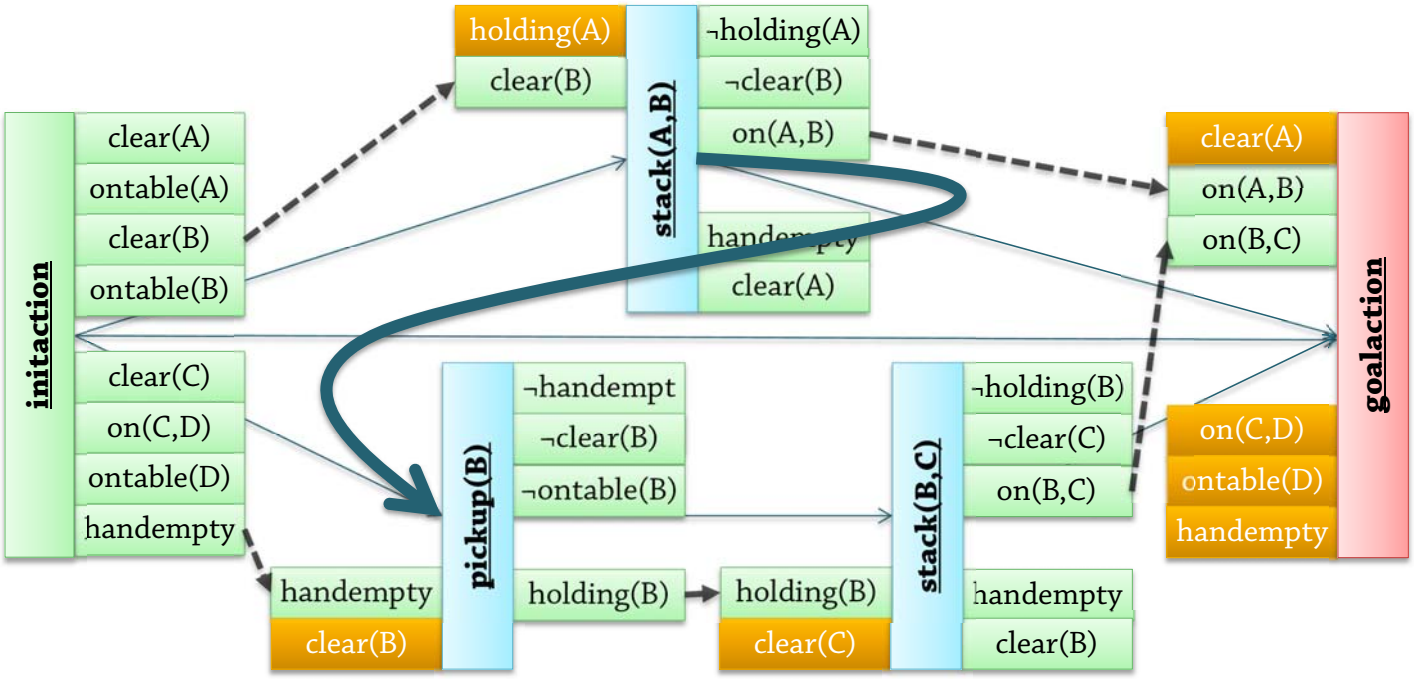
Threats

- Second flaw type: A **threat** to be resolved
  - initaction **supports** clear(B) for stack(A,B) – there's a **causal link**
  - pickup(B) **deletes** clear(B), and **may** occur **"during" the link**
  - We can't be certain that clear(B) still holds when stack(A,B) starts!



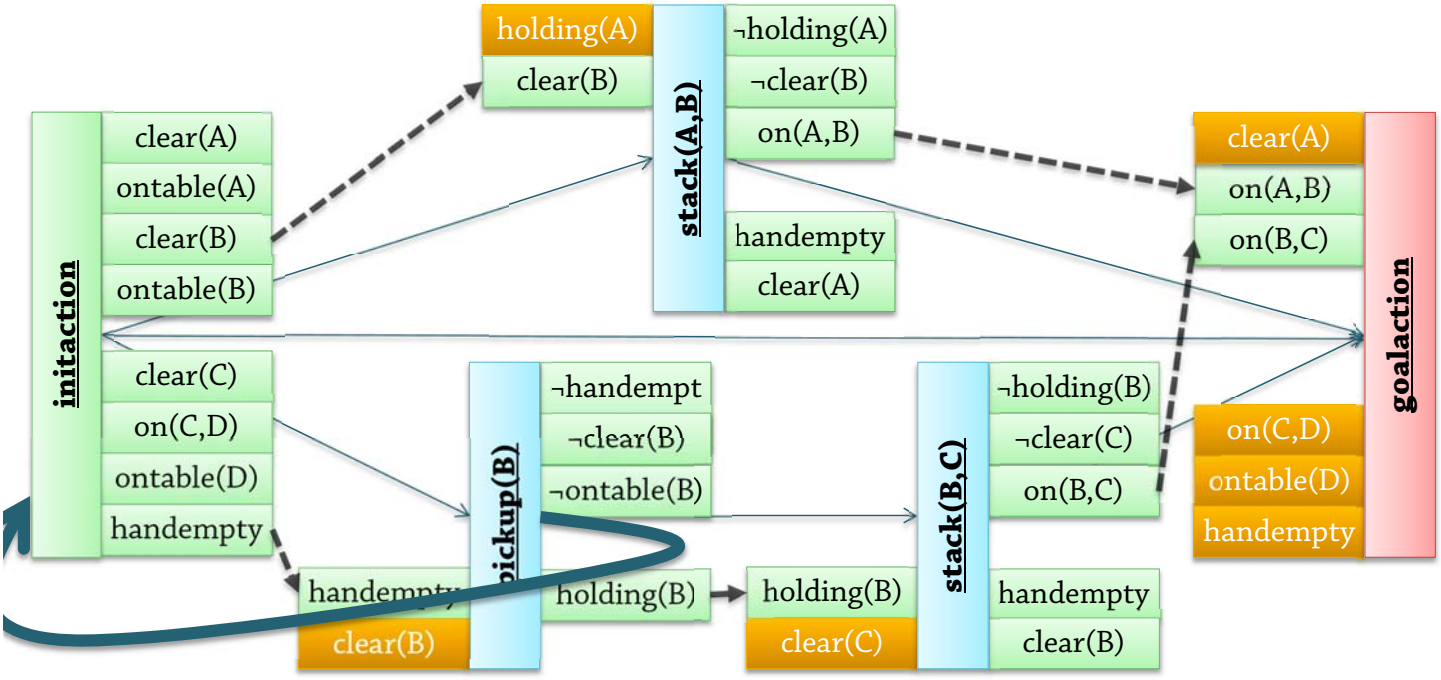
Resolving Threats 1

- How to **make sure** that clear(B) holds when stack(A,B) starts?
  - Alternative 1: The action that **disturbs** the precondition is placed **after** the action that **has** the precondition
    - Only possible if the resulting partial order is consistent (acyclic)!



Resolving Threats 2

- Alternative 2:
  - The action that **disturbs** the precondition is placed **before** the action that **supports** the precondition
  - Only possible if the resulting partial order is consistent – not in this case!



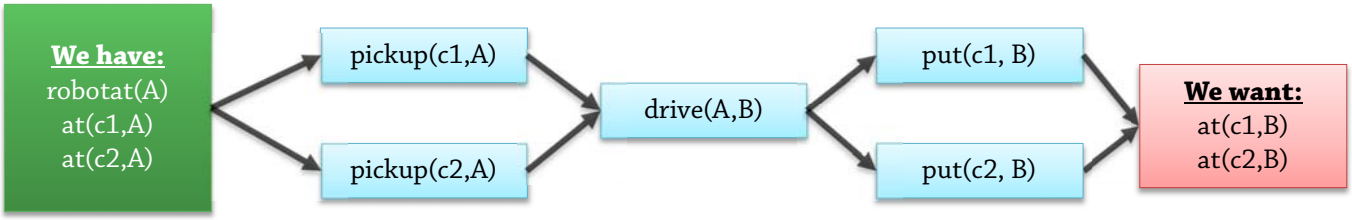
Heuristics

- Again, this is not sufficient in itself
  - Heuristics are required
- But it has advantages
  - No need to commit to order

Remember what a flaw is!
- **Not something "wrong" in a final solution**
- **Not a mistake**
- **Simply bookkeeping for "something we haven't taken care of yet"**
  - **Open goal: Must decide how to achieve a precondition**
  - **Threat: Must decide how to order actions**

Partial-Order Solutions

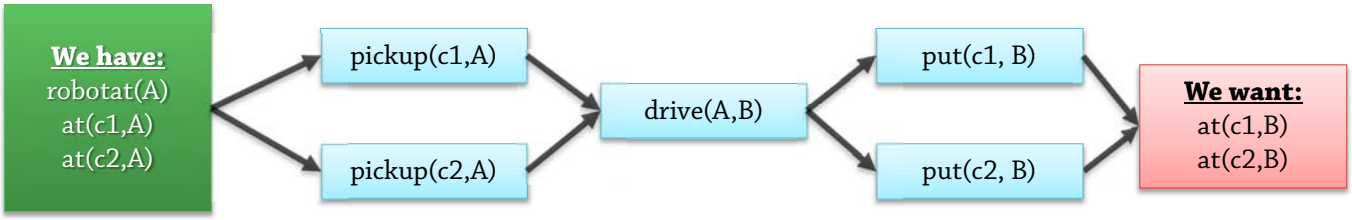
- Original motivation: **performance**
  - Therefore, a partial-order plan is a **solution** iff **all sequential** plans satisfying the ordering are solutions
    - Similarly, **executable** iff corresponding sequential plans are executable
    - <pickup(c1,A), pickup(c2,A), drive(A,B), put(c1,B), put(c2,B)>
    - <pickup(c2,A), pickup(c1,A), drive(A,B), put(c1,B), put(c2,B)>
    - <pickup(c1,A), pickup(c2,A), drive(A,B), put(c2,B), put(c1,B)>
    - <pickup(c2,A), pickup(c1,A), drive(A,B), put(c2,B), put(c1,B)>



Partial Orders and Concurrency

jonkv@ida

- Can be **extended** to allow **concurrent execution**
  - Requires an extended action model!
    - Our model *does not define* what happens if c1 and c2 are picked up **simultaneously** (or even if we are **allowed** to do this)!



Beyond State Space Planning

Many alternative search spaces exist;
some examples here!

SAT Planning

- SAT: Boolean satisfiability, first problem proven NP-complete
  - Translate planning to a SAT problem – propositions + complex formula
    - Apply existing SAT solver
    - Each **SAT solution** corresponds to a **solution plan**

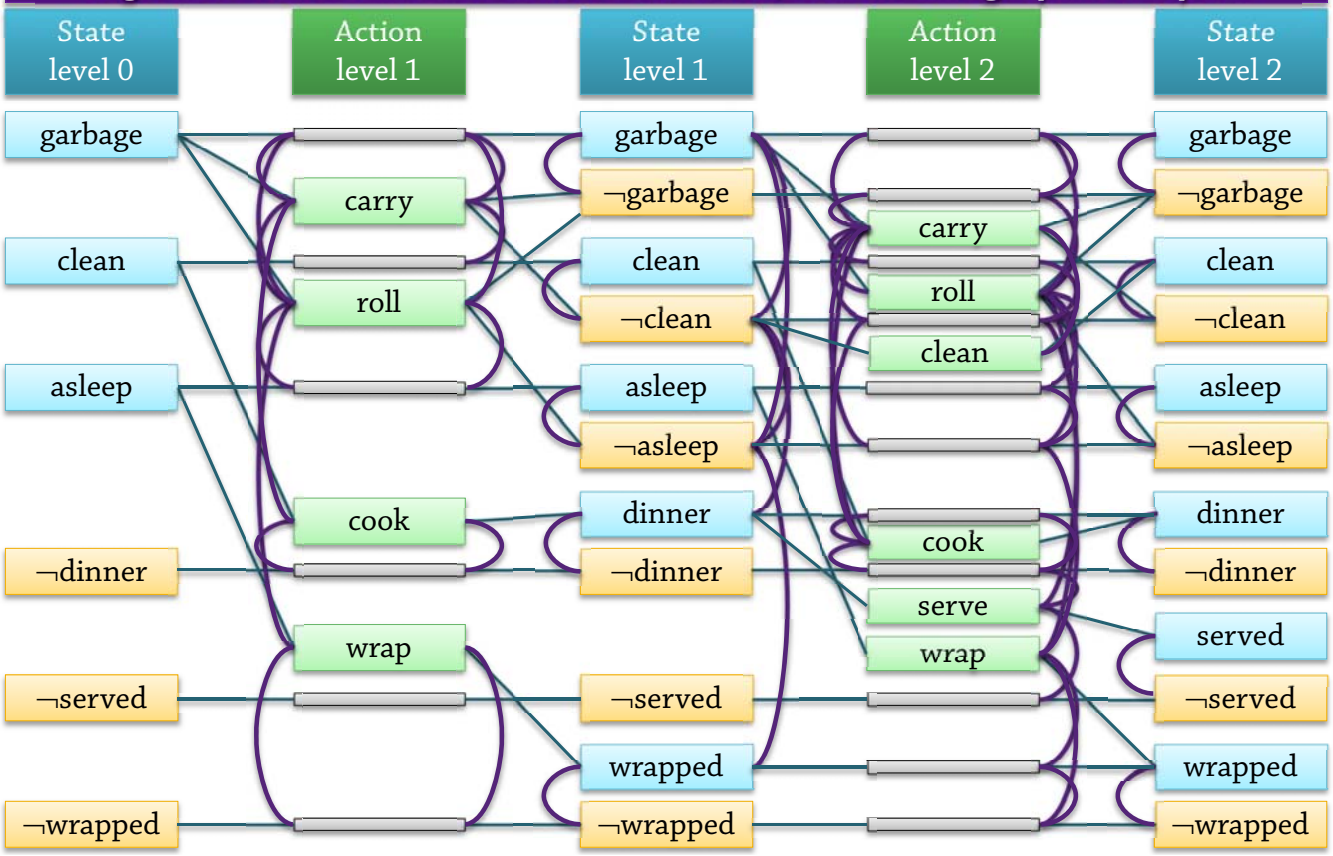
"Action propositions": Which actions are executed, and when?			"Fact propositions": Which facts are true, and when?				
…	…	…	…	…	…	φ	Solutions
–	–	–	–	–	–	–	
–	–	true	true	–	–	–	
–	true	–	true	–	true	–	
–	true	true	–	–	true	true	←
true	–	–	true	true	–	–	
true	–	true	true	true	–	–	
true	true	–	–	true	true	true	←
true	true	true	true	true	true	–	

Seq. plan!

Seq. plan!

Planning Graphs

Compute a graph representing what might be achievable per time step, together with mutual exclusion relations; search this graph for a plan



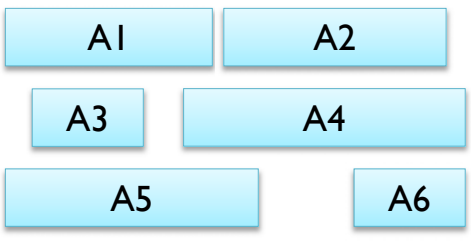
Beyond Classical Planning: Time, Concurrency, Resources

Time, Concurrency and Resources

- Many planners support **time** and **concurrency**
  - Actions have **durations**
  - Actions can be **scheduled** in parallel, suitable for multiple agents
    - **Example: Duration determined by current state**

```
(:durative-action heat-water
        :parameters (?p - pan)
        :duration (= ?duration (/ (- 100 (temperature ?p)) (heat-rate)))
        :condition (and (at start (full ?p))
                        (at start (onHeatSource ?p))
                        (at start (byPan))
                        (over all (full ?p))
                        (over all (onHeatSource ?p))
                        (over all (heating ?p))
                        (at end (byPan)))
        :effect (and (at start (heating ?p))
                     (at end (not (heating ?p)))
                     (at end (assign (temperature ?p) 100)))
)
```

Conditions / effects at the *start* or *end* of the action



Implemented by quite a few planners – affects algorithms and heuristics!

Time, Concurrency and Resources (2)

- **Example: Duration selected by planner**

(:**durative-action** heat-water
:**parameters** (?p - pan)
:**duration** (at end (<= ?duration (/ (- 100 (temperature ?p)) (heat-rate))))
:**condition** (and (at start (**full** ?p))
(at start (**onHeatSource** ?p))
(at start (**byPan**))
(over all (**full** ?p))
(over all (**onHeatSource** ?p))
(over all (**heating** ?p))
(at end (**byPan**)))
:**effect** (and (at start (**heating** ?p))
(at end (not (**heating** ?p)))
(at end (**increase** (temperature ?p) **(* ?duration (heat-rate))**))))

General increase and decrease effects
for numeric values
(handles concurrent production and consumption of resources)

Time, Concurrency and Resources (3)

- Some planners support **continuous linear effects**
  - Occurring throughout the duration of the action
    - (:**durative-action** fly
      :**parameters** (?p - airplane ?a ?b - airport)
      :**duration** (= ?duration (flight-time ?a ?b))
      :**condition** (and (at start (at ?p ?a))
      (over all (inflight ?p))
      (over all (>= (fuel-level ?p) 0)))
      :**effect** (and (at start (not (at ?p ?a)))
      (at start (inflight ?p))
      (at end (not (inflight ?p)))
      (at end (at ?p ?b))
      (decrease (fuel-level ?p) **(* #t (fuel-consumption-rate ?p))**)))
    - (:**action** midair-refuel
      :**parameters** (?p)
      :**precondition** (inflight ?p)
      :**effect** (assign (fuel-level ?p) (fuel-capacity ?p)))

Time, Concurrency and Resources (4)

- **Hybrid discrete/continuous** planning
  - (:durative-action navigate

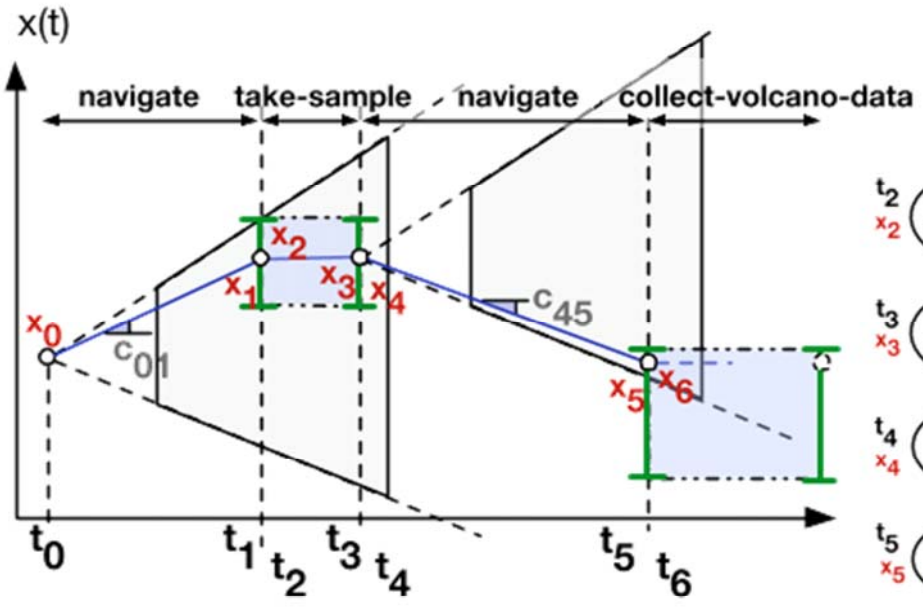
```
:control-variables ((velX) (velY))
:duration (and (<= ?duration 5000))
:condition (and
        (over all (>= (velX) -4)) (over all (<= (velX) 4))
        (over all (>= (velY) -4)) (over all (<= (velY) 4))
        (over all (<= (x) 700)) (over all (>= (x) 0))
        (over all (<= (y) 700)) (over all (>= (y) 0))
        (at start (AUV-ready)))
:effect (and
        (at start (not (AUV-ready)))
        (at end (AUV-ready))
        (increase (x) (* 1.0 (velX) #t))
        (increase (y) (* 1.0 (velY) #t)))
```

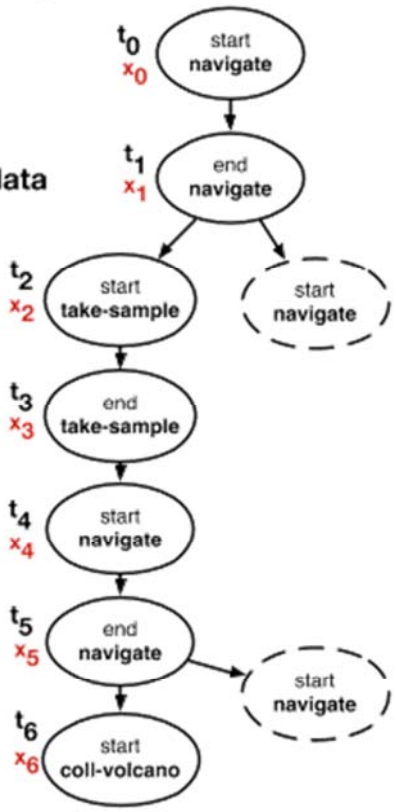
Can vary continuously during the action

Continuous effects depend on the **instantaneous** values of control variables

Time, Concurrency and Resources (5)

- Flow tubes result in **reachable regions** (shaded)
  - Rectangles are conditions for **take-sample, collect**





Beyond Classical Planning: Incomplete Information

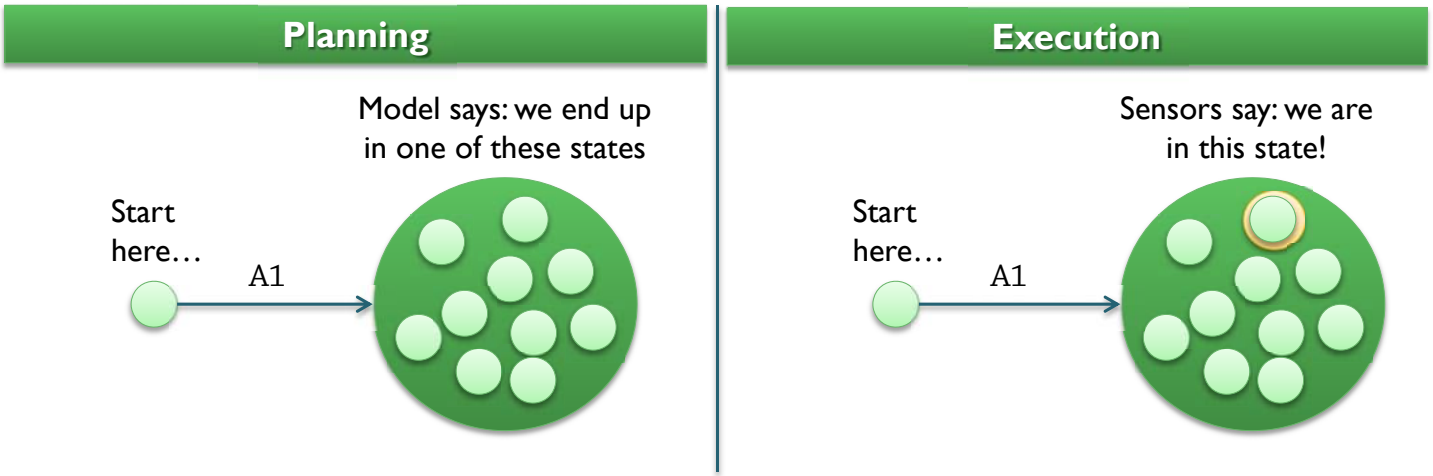
Multiple Outcomes

- In reality, actions may have **multiple outcomes**
  - Some outcomes can indicate **faulty / imperfect execution**
    - **pick-up(object)**
      Intended outcome: carrying(object) is true
      Unintended outcome: carrying(object) is false
    - **move(100,100)**
      Intended outcome: xpos(robot)=100
      Unintended outcome: xpos(robot) != 100
    - **jump-with-parachute**
      Intended outcome: alive is true
      Unintended outcome: alive is false
  - Some outcomes are more **random**, but clearly **desirable / undesirable**
    - Pick a present at random – do I get the one I longed for?
    - Toss a coin – do I win?
  - Sometimes we have **no clear idea** what is desirable
    - Outcome will affect how we can continue, but in less predictable ways

To a *planner*, there is generally no difference between these cases!

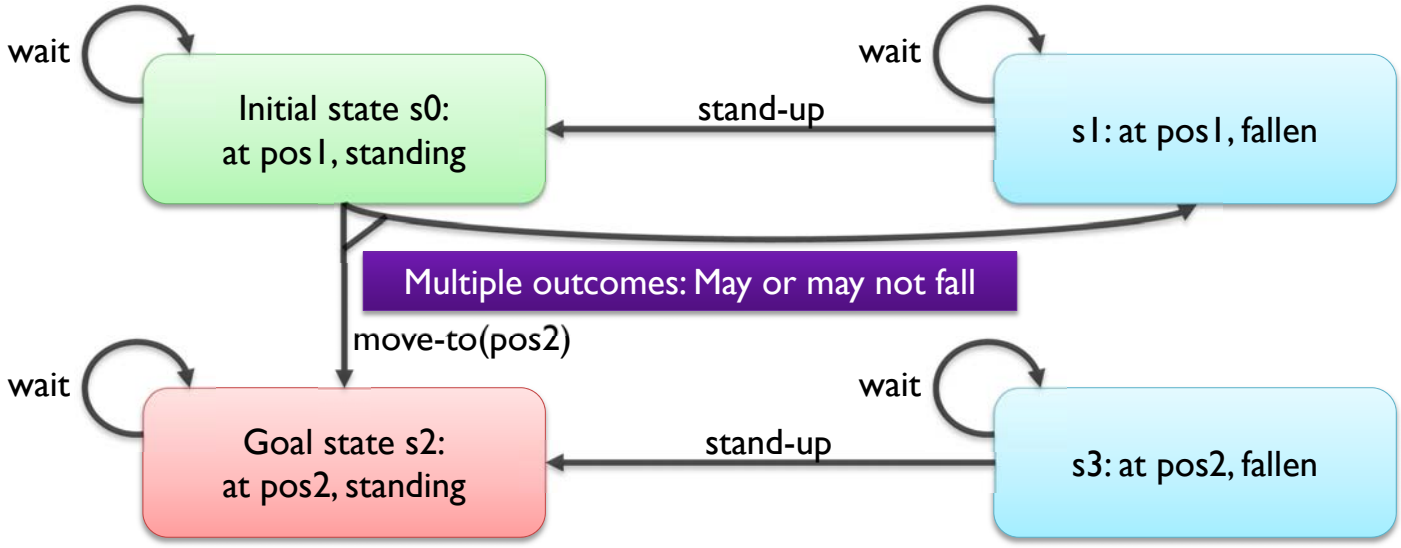
FOND Planning

- **Nondeterministic** planning:
  - Many possible outcomes for each action
- FOND: **Fully Observable** Non-Deterministic
  - After executing an action, sensors determine exactly which state we are in



FOND Planning: Plan Structure (1)

- Example state transition system:



- **Intuitive strategy**:
  - **while** (not in s2) {
    move-to(*pos2*);
    if (fallen) stand-up;
    }

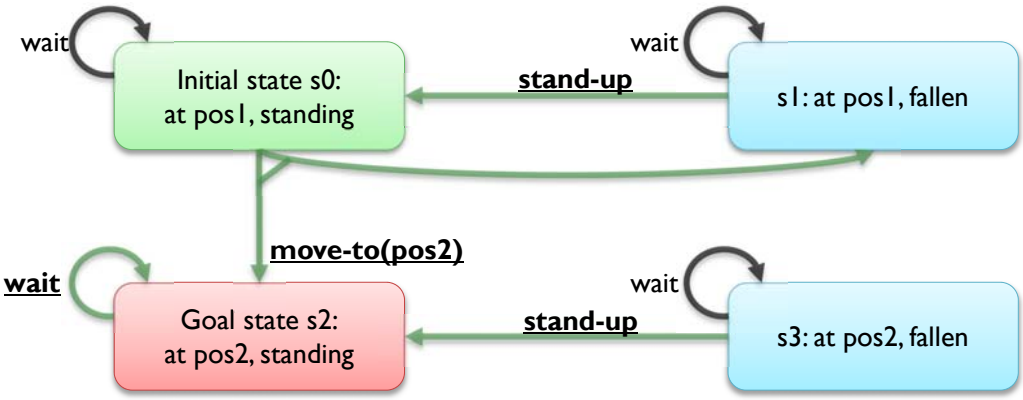
In FOND planning, action choice should depend on actual outcomes (through the current state)

There may be no upper bound on how many actions we may have to execute!

FOND Planning: Plan Structure (2)

- Examples of formal plan structures:
  - **Conditional plans** (with if/then/else statements)
  - **Policies** $\pi : S \rightarrow A$
    - Defining, **for each state**, which action to execute **whenever** we end up there
    - $\pi(s0)$ = move-to(pos2)
    - $\pi(s1)$ = stand-up
    - $\pi(s2)$ = wait
    - $\pi(s3)$ = stand-up

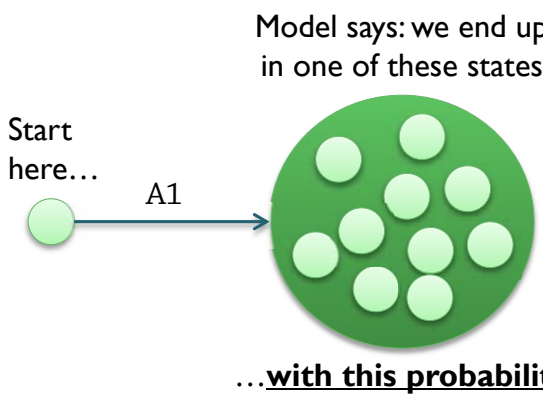
Or at least, for every state
that is *reachable* from the given initial state
(A policy can be a *partial function*)



Stochastic Systems

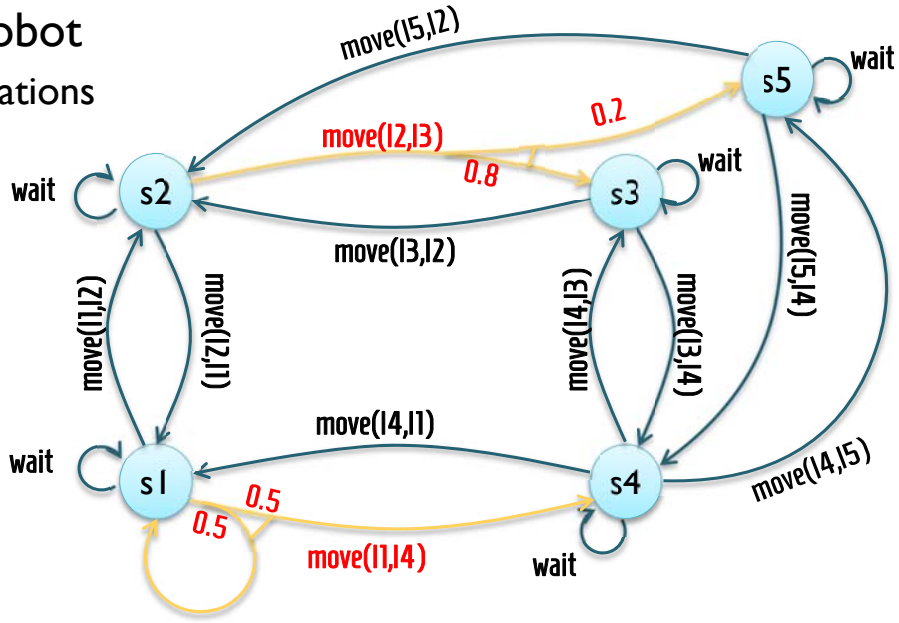
- **Probabilistic planning** ➔ a **stochastic system**
  - *P(s, a, s'):* Given that we are in *s* and execute *a*, the **probability** of ending up in *s'*
  - For every state *s* and action *a*, we have $\sum_{s' \in S} P(s, a, s') = 1$:
    The world gives us 100% probability of ending up in *some* state

Planning



Stochastic System Example

- **Example**: A single robot
  - Moving between 5 locations
  - For simplicity, states correspond directly to locations
    - s1: at(r1, l1)
    - s2: at(r1, l2)
    - s3: at(r1, l3)
    - s4: at(r1, l4)
    - s5: at(r1, l5)



  - Some transitions are **deterministic**, some are **stochastic**
    - Trying to move from l2 to l3: You may end up at l5 instead (20% risk)
    - Trying to move from l1 to l4: You may stay where you are instead (50% risk)
  - (Can't always move in both directions, e.g. due to terrain gradient)

Representation Example: RDDL

- **Relational Dynamic Influence Diagram Language – a different view**
  - Based on Dynamic Bayesian Networks
  - **domain** prop_dbn {

```
requirements = { reward - deterministic };
// Define the state and action variables ( not parameterized here )
pvariables {
        p : { state - fluent , bool , default = false };
        q : { state - fluent , bool , default = false };
        r : { state - fluent , bool , default = false };
        a : { action - fluent , bool , default = false };
};
// Define the conditional probability function for each next
// state variable in terms of previous state and action
cpfs {
        p' = if (p ^ r) then Bernoulli (.9) else Bernoulli (.3);
        q' = if (q ^ r) then Bernoulli (.9)
        else if (a) then Bernoulli (.3) else Bernoulli (.8);
        r' = if (~q) then KronDelta (r) else KronDelta (r <=> q);
};
// Define the reward function ; note that boolean functions are
// treated as 0/1 integers in arithmetic expressions
reward = p + q - r;
}
```

Overview

	Non-Observable: No information gained after action	**Fully Observable:** Exact outcome known after action
Deterministic: Exact outcome known in advance	Classical planning (possibly with extensions) (Information dimension is irrelevant; we know everything *before* execution)	
Non-deterministic: Multiple outcomes, no probabilities	NOND: Conformant Planning	FOND: Conditional / Contingent Planning
Probabilistic: Multiple outcomes with probabilities	Probabilistic Conformant Planning (Special case of POMDPs, Partially Observable MDPs)	Probabilistic Conditional Planning Stochastic Shortest Path Problems Markov Decision Processes (MDPs)

Conclusions

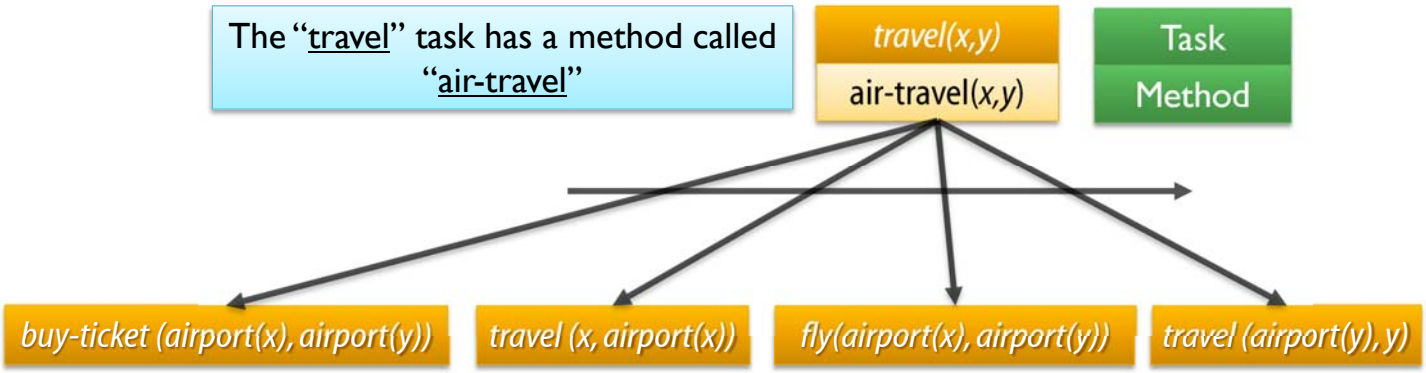
Example Questions

- Example questions:
  - Describe and explain:
    - Relaxation, and how it can be applied to create an admissible heuristic
    - PDB heuristics
    - Landmarks
    - The main advantage of partial-order planning over forward state space planning
    - The different kinds of flaws that can occur in a partially ordered plan
    - The ways in which can you resolve an open goal
    - Why planners don't use Dijkstra's algorithm to search the state space
  - Is the number of open goals an *admissible* heuristic? Why / why not?
  - Given a domain, problem and initial state:
    - Expand the forward state space by two levels
  - How can you apply hill climbing to planning?
    - How do you have to modify standard hill climbing, and why?

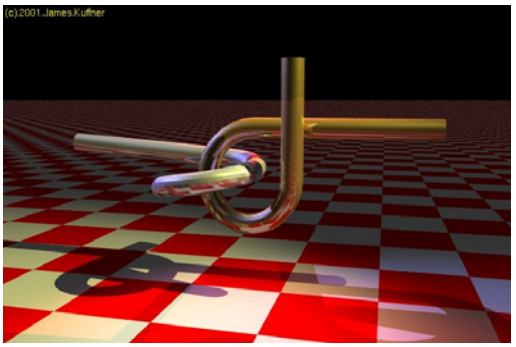
TDDD48 Automated Planning (1)

- **Deeper discussions** about all of these topics
- **Formal basis** for planning
  - Alternative representations of planning problems
  - Simple and complex state transition systems
- Completely different **principles** for **heuristics**
  - Landmarks
  - Pattern databases
- Alternative **search spaces**
  - Planning by conversion to boolean satisfiability problems
- Extended **expressivity**
  - Planning with time and concurrency
  - Planning with probabilistic actions (Markov Decision Processes)
  - Planning with non-classical goals

- Planning with **domain knowledge**
  - Using what you know: Temporal control rules
  - Breaking down a task into smaller parts: Hierarchical Task Networks



- Alternative **types** of planning
  - Path planning



- And so on…