

TDDC17: Introduction to Automated Planning

Jonas Kvarnström

Automated Planning Group

Department of Computer and Information Science

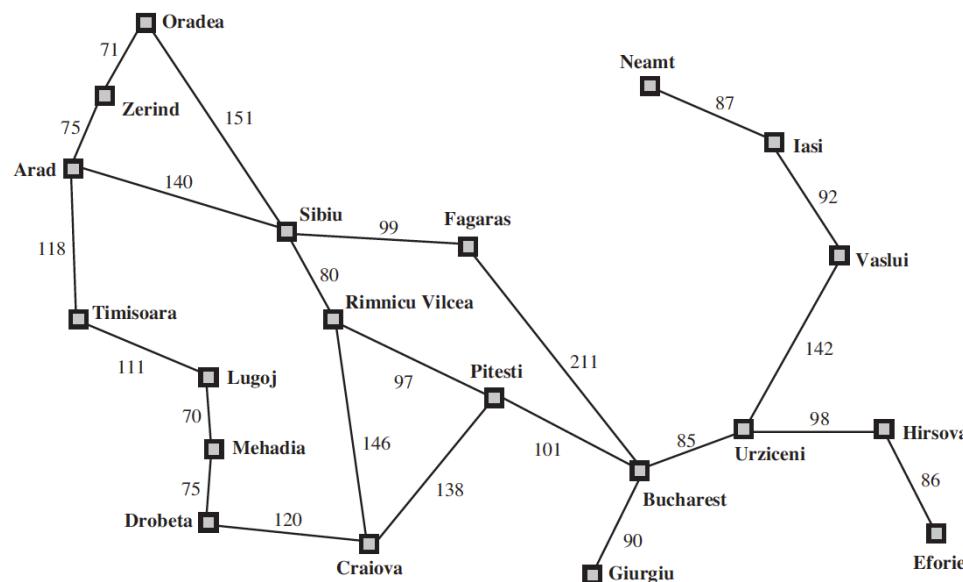
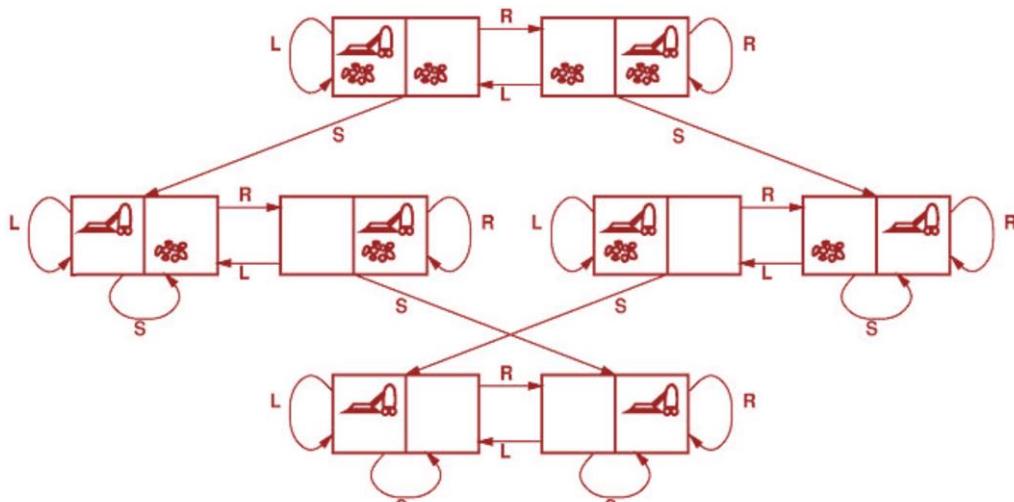
Linköping University

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*

You have done this before!

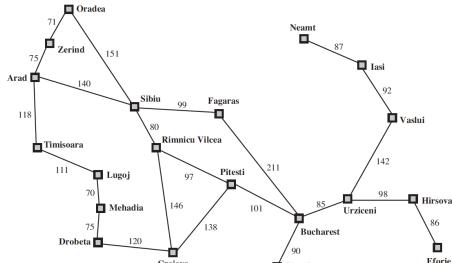


Using **knowledge** about the world, including possible actions and their results, to **decide** what to do and when in order to achieve an **objective**, **before** you actually start doing it

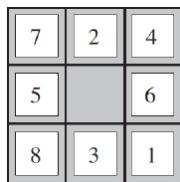


Are we done?

Domain-specific search guidance – too much (human) work!



Straight line distance from city n to goal city n'



$h_2(n)$: The sum of the manhattan distances for each tile that is out of place.

($3+1+2+2+3+3+2=18$) . The manhattan distance is an under-estimate because there are tiles in the way.

More complex



???

Planning:
General heuristics

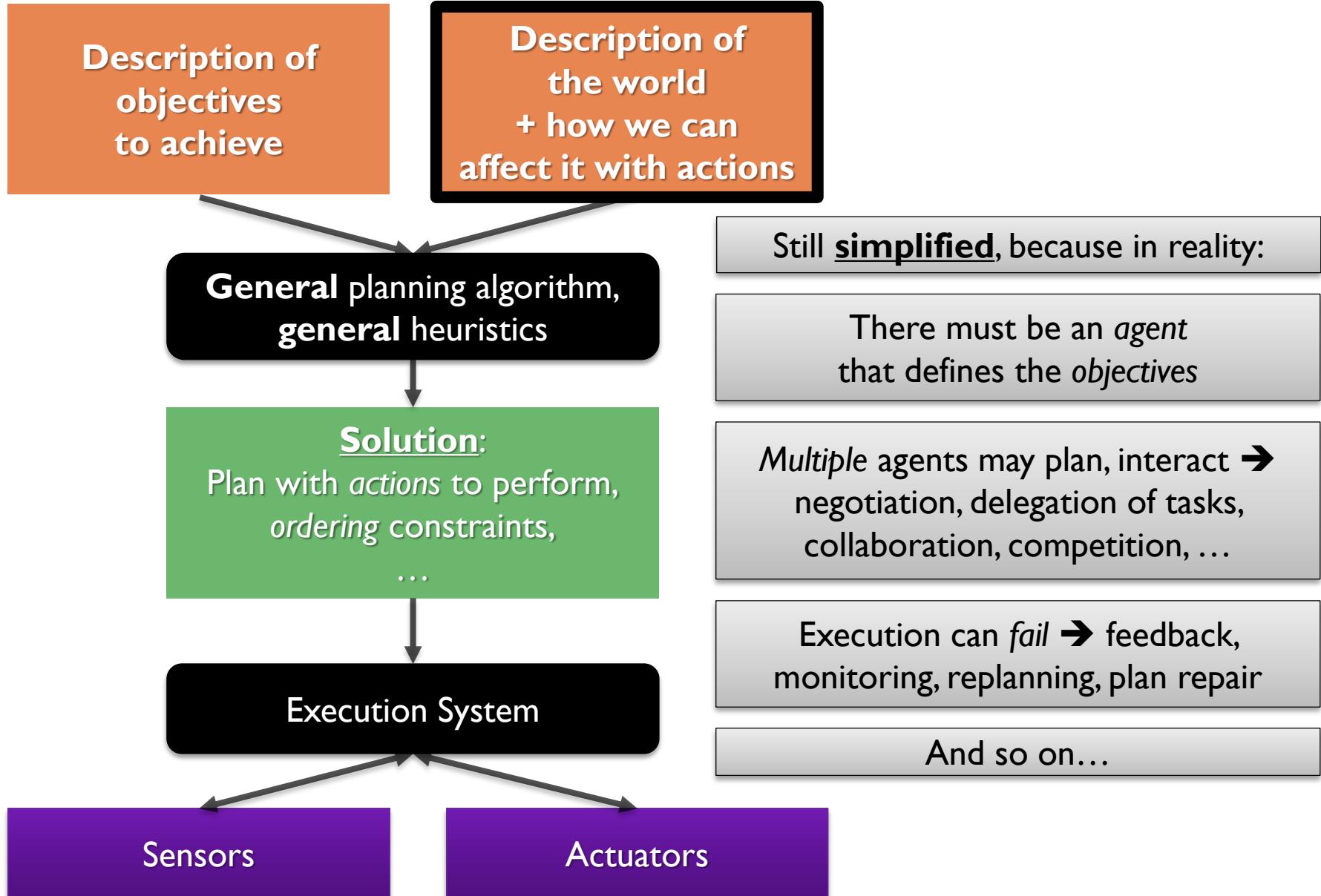
Pattern Databases
Landmarks
 FF , h^m , merge-and-shrink, ...

Planning:
Entirely different search spaces

Partial Order Causal Link
SAT planning
Planning Graphs, ...

Need a common, well-structured representation
for planners/heuristics to analyse!

AI Planning: A Simplified View



Domains 1: Blocks World

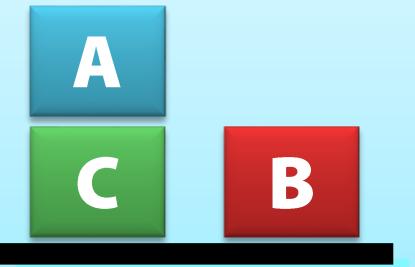


- Must support simple examples such as the **Blocks World**

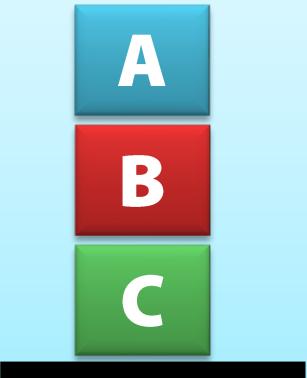
You



Current state of the world



Your greatest desire



Domains 2: Towers of Hanoi



Another "toy domain" – useful for examples!



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

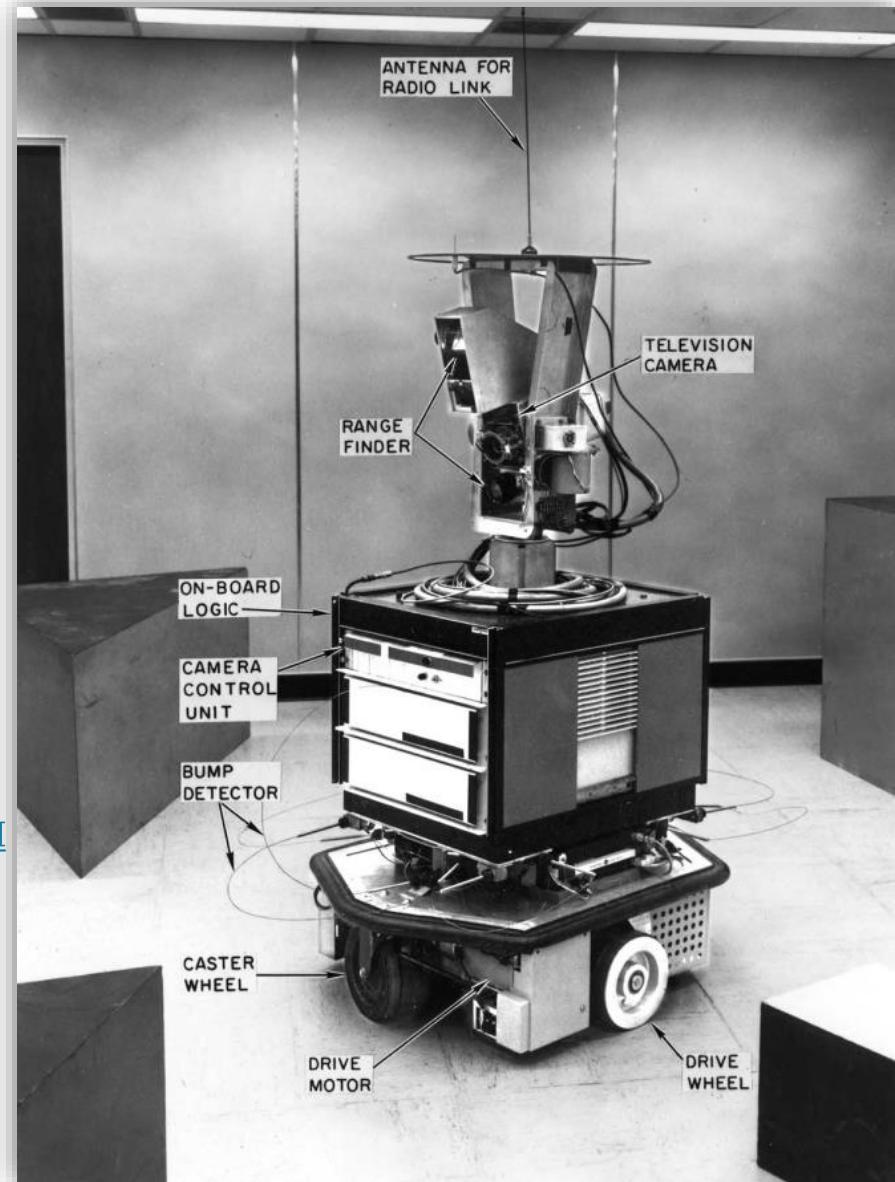
Domains 3: Shakey



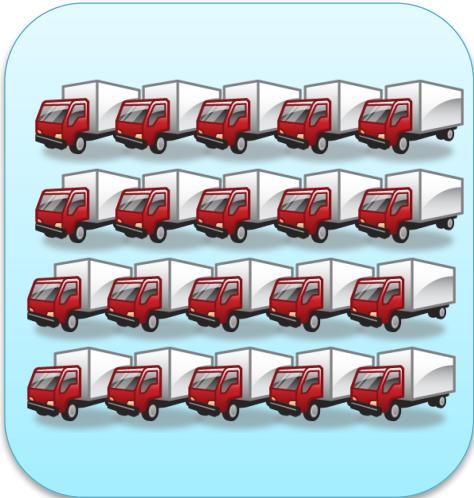
- Classical robot example:
Shakey (1969)

- 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=qXdn6ynwpI>

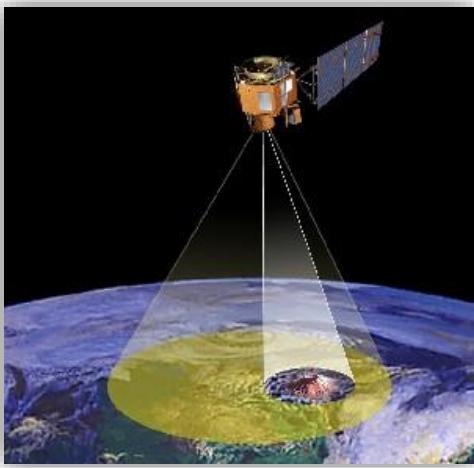


Domains 4



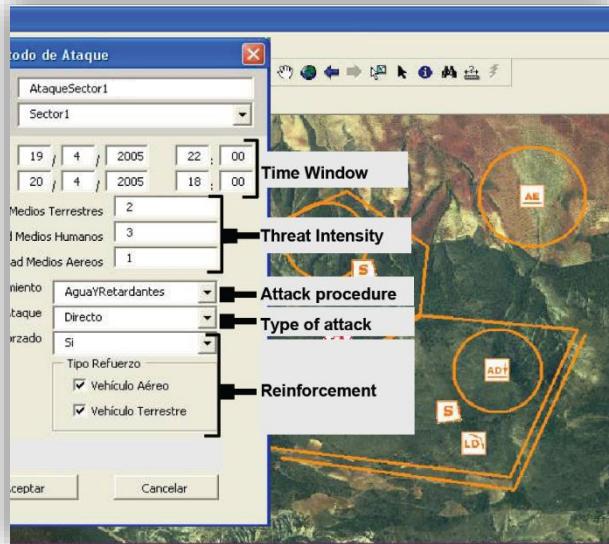
Logistics:

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 5: Dock Worker Robots (DWR)



Containers shipped
in and out of a harbor



Cranes move containers
between "piles" and robotic trucks



Classical Planning



How to define 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

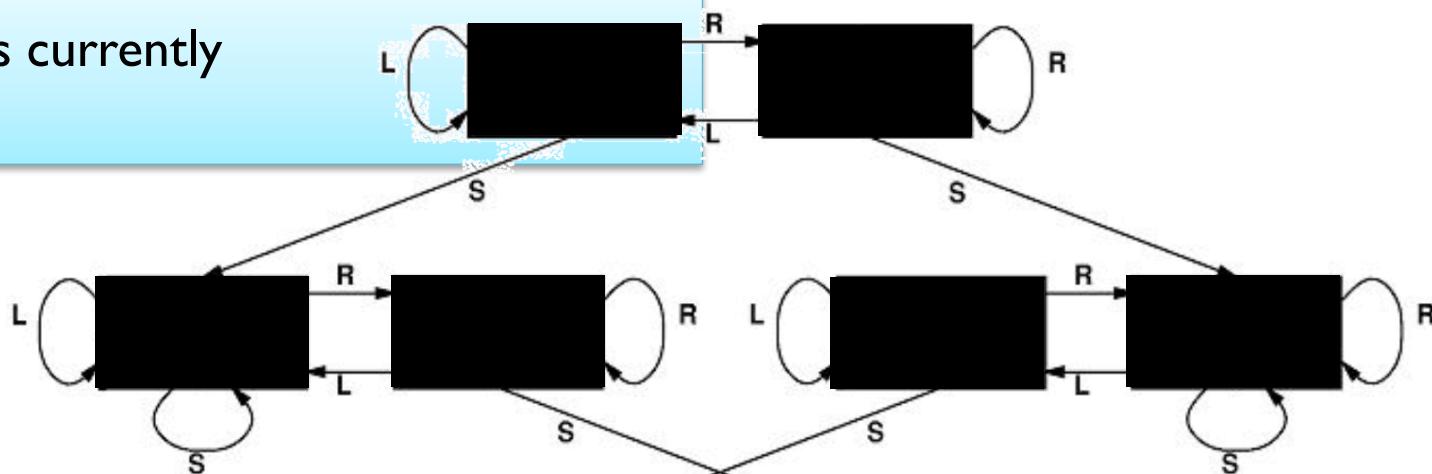
Facts and States: Introduction

14

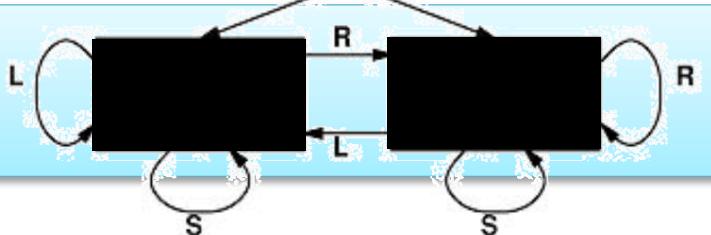
jonkv@ida

- As you saw before, we are interested in **states of the world**

The world is currently
in *this* state



We want to reach
one of *these* states



Is this information sufficient?

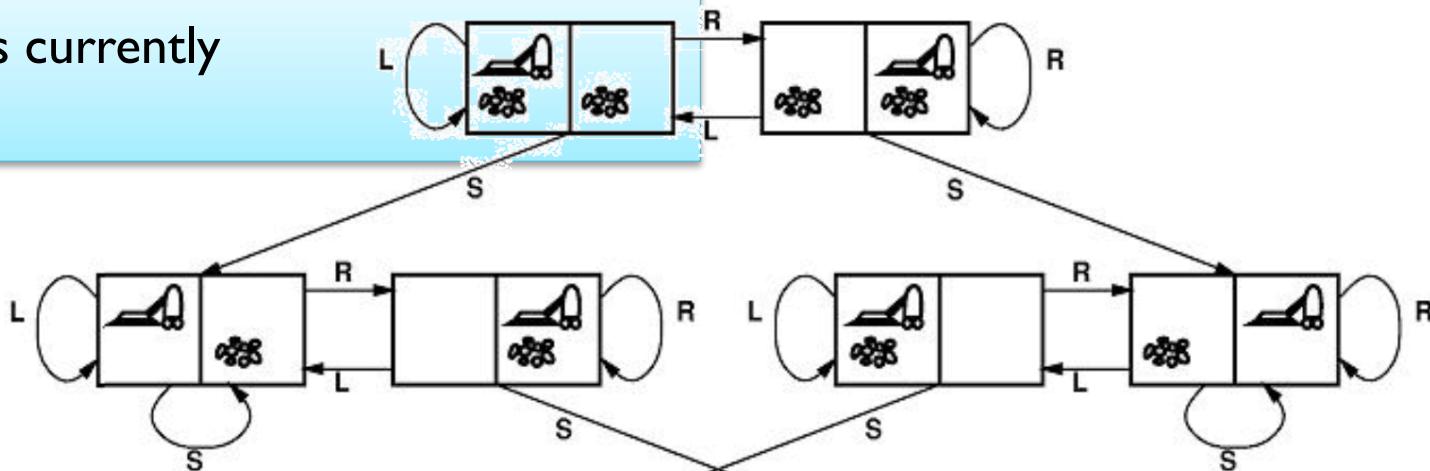
No – Need to **analyze** states,
find **differences** compared to goal states,
find **relevant** actions, ...!

Facts and States: Introduction

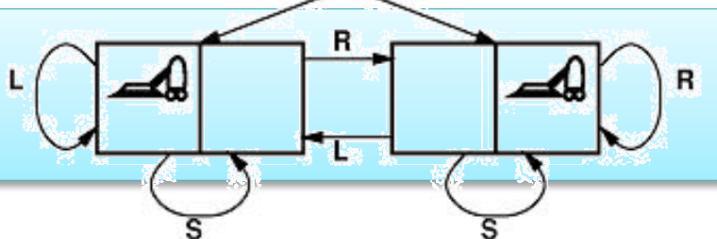


- We need **information about** every state:

The world is currently
in *this* state



We want to reach
one of *these* states



**First: Rooms and
vacuum cleaners
are objects**

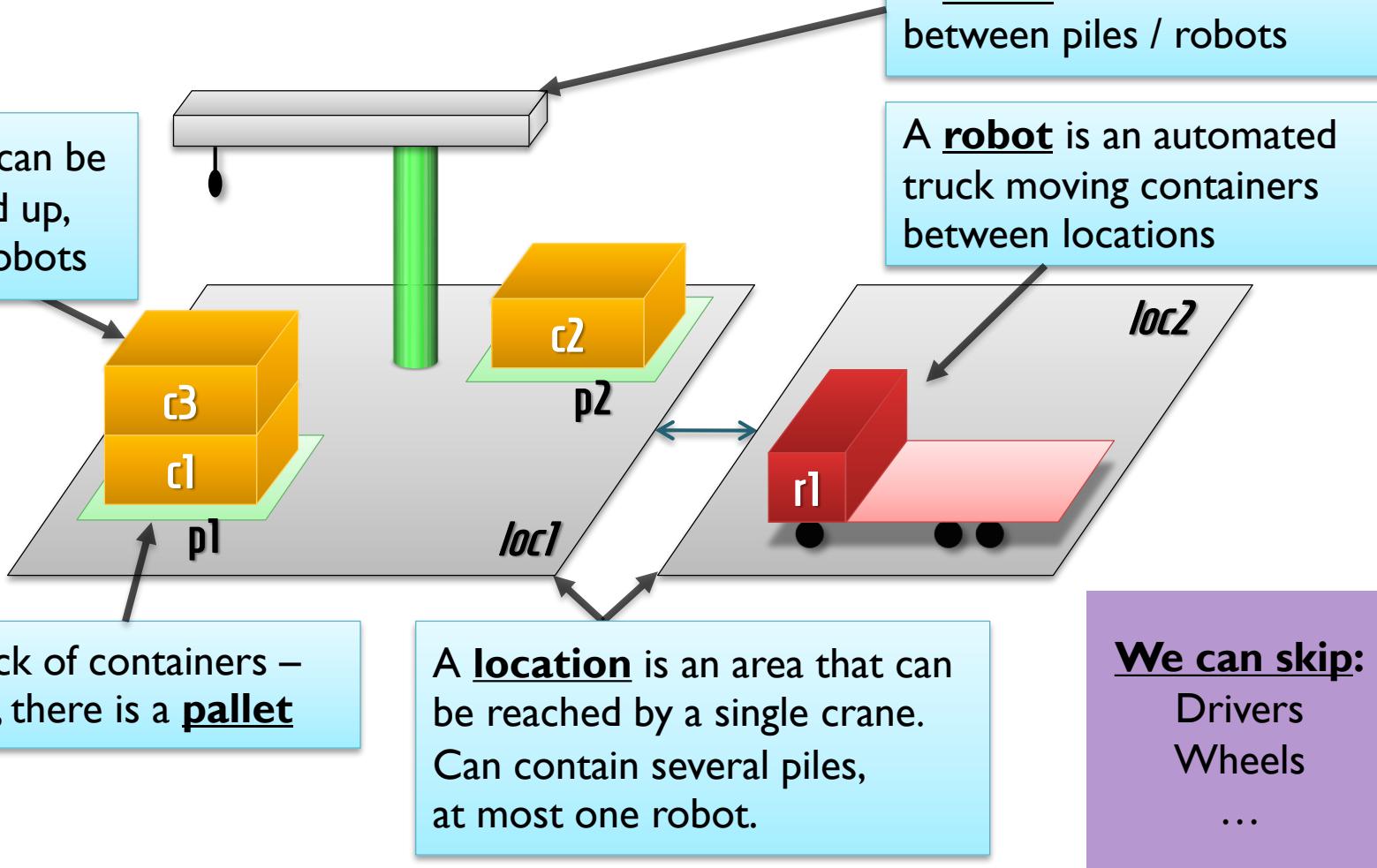
Efficient planning depends on describing states as collections of facts:

We are in a state where there is dirt in both rooms,
and the vacuum cleaner is in the leftmost room

Objects 1: Object Types



- We can often specify **object types**



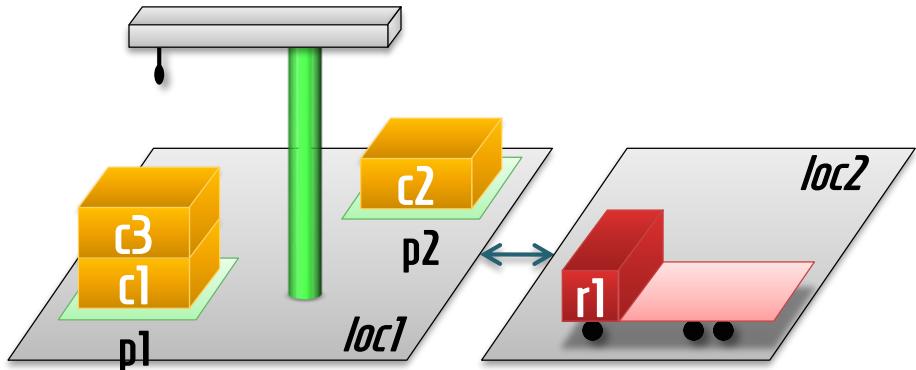
Essential: Determine which types are **relevant** for the **problem** and **objective**!

Objects 2: Actual objects

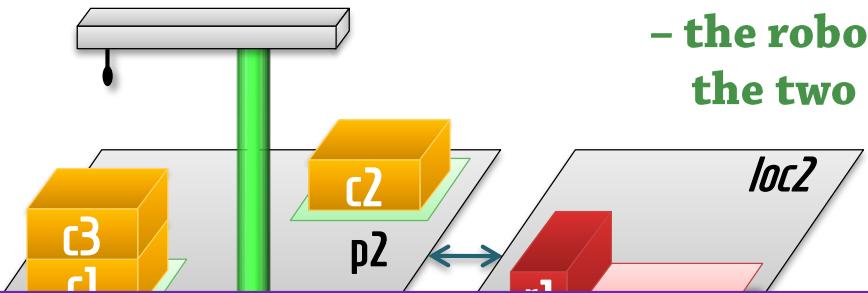


- We then specify sets of **actual objects**

- **robot:** { r1 }
- **location:** { loc1, loc2 }
- **crane:** { k1 }
- **pile:** { p1, p2 }
- **container:** { c1, c2, c3, pallet }



- Most planners use a **first-order representation**:
 - Every **fact** is represented as a logical **atom**: Predicate symbol + arguments
 - **Properties of the world**
 - **raining** – **it is raining [not in the standard DWR domain!]**
 - **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 in DWR]**



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

Facts / Predicates in DWR



- **Reference:** 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

Ground = without variables

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

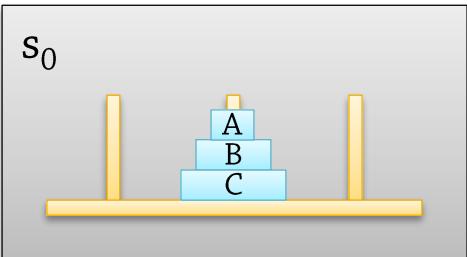
Number of states: $2^{\text{number of atoms}}$ – enormous, but finite (for classical planning!)

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
 - $s_0 = \{ \text{on}(A,B), \text{on}(B,C), \text{in}(A,2), \text{in}(B,2), \text{in}(C,2), \text{top}(A), \text{bot}(C) \}$



s_0	$\text{top}(A)$	$\text{in}(A,2)$
	$\text{on}(A,B)$	$\text{in}(B,2)$
	$\text{on}(B,C)$	$\text{in}(C,2)$
	$\text{bot}(C)$	

$\text{top}(A) \in s_0 \rightarrow \text{top}(A) \text{ is true in } s_0$
 $\text{top}(B) \notin s_0 \rightarrow \text{top}(B) \text{ is false in } s_0$

States 3: Initial State

22

jonkv@ida

Initial states in classical STRIPS planning:

- We assume *complete information* about the **initial state** s_0 (before any action)

Complete **relative to the model**:

We must know everything
about those predicates and objects
we have specified...

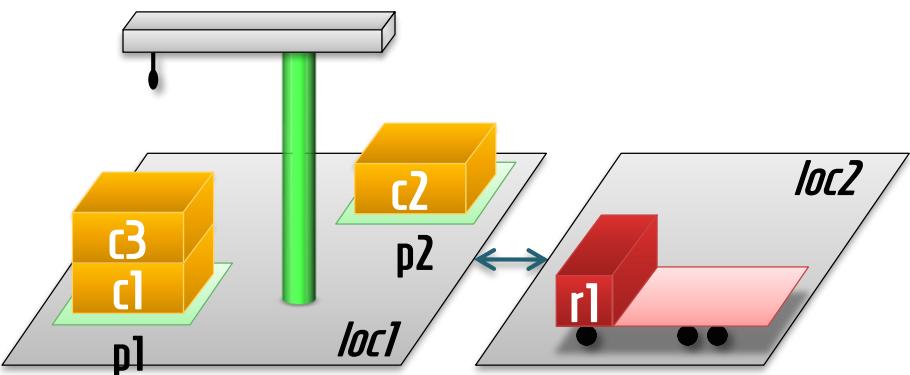
But not whether it's raining!

- 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),

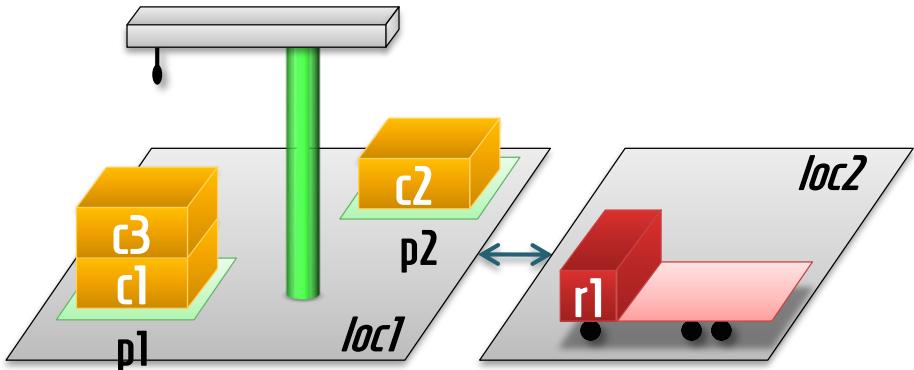
}



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) }** }



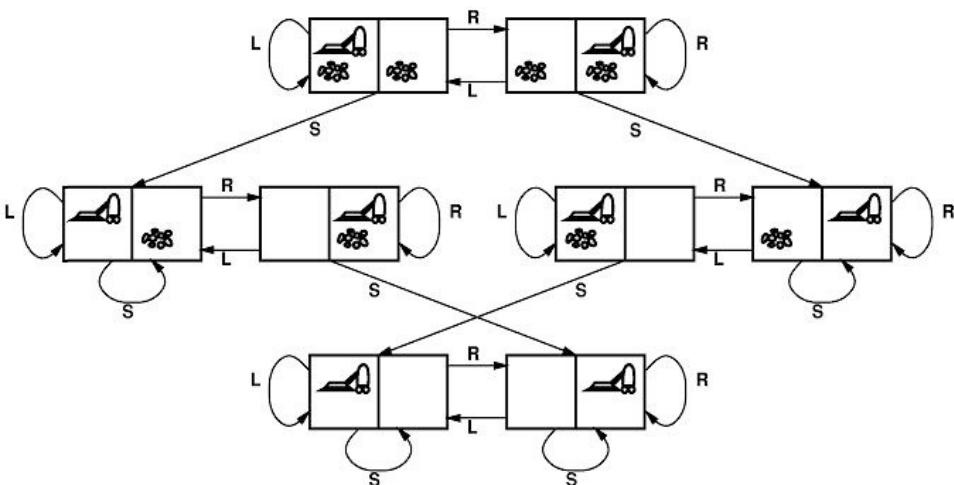
Actions 1: Intro

24

jonkv@ida

- 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?
 - States consist of facts
 - Actions add or remove facts

Actions 2: Operators

25

jonkv@ida

- Define operators or action schemas:

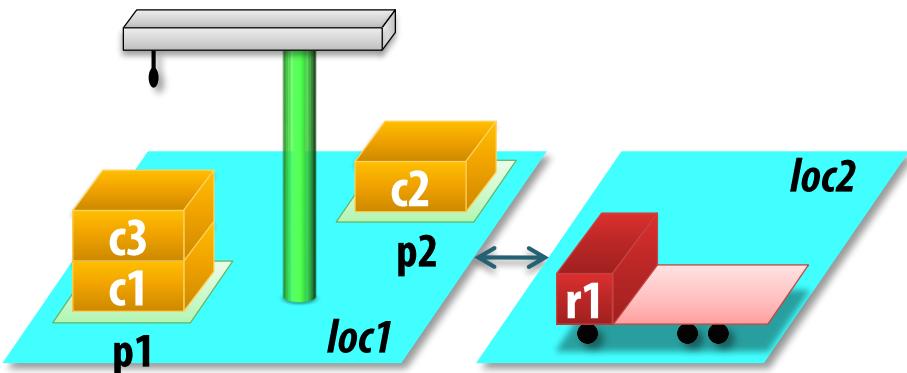
- **move**(*robot*, *location1*, *location2*)

- Precondition: **at**(*robot*, *location1*) \wedge
adjacent(*location1*, *location2*) \wedge
 \neg **occupied**(*location2*)

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

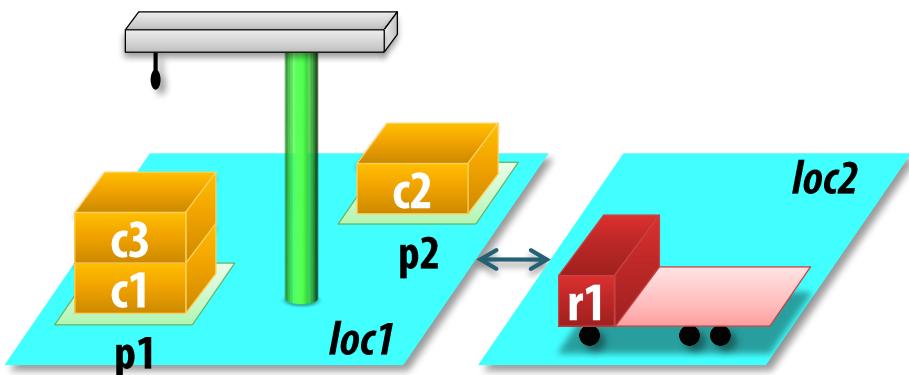
- Effects:
 \neg **at**(*robot*, *location1*),
at(*robot*, *location2*),
 \neg **occupied**(*location1*),
occupied(*location2*)

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



Actions 3: Instances

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



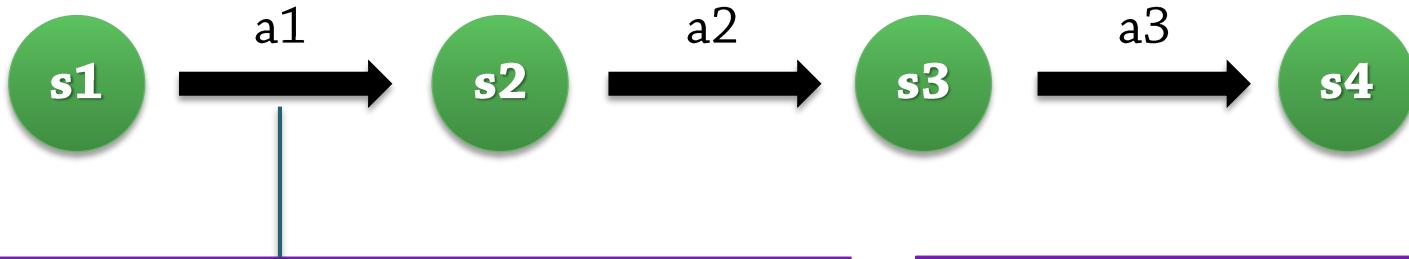
Actions 4: Step by Step

- 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

Planning Domain, Problem Instance

28

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?)

The State Space

State Spaces 1: Introduction

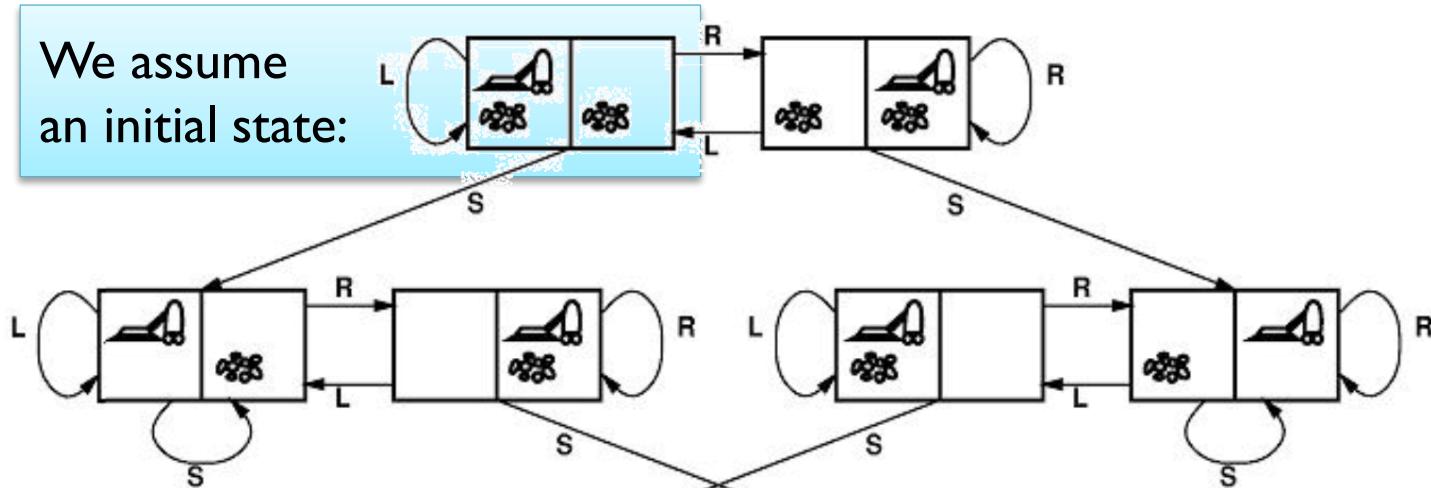
30

jonkv@ida

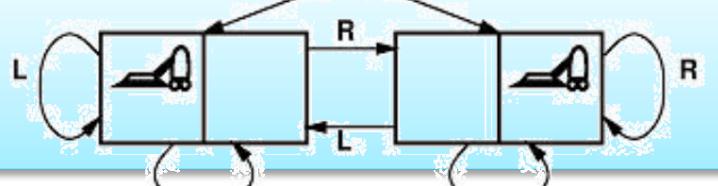
- Every classical planning problem has a state space – a **graph**
 - A node for every world state
 - An edge for every executable action

The planning problem: Find a path (not necessarily shortest)

We assume
an initial state:



And a number of
goal states ("no dirt"):



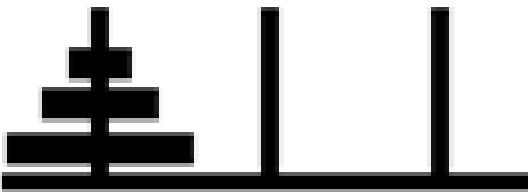
Example solutions: SRS, RSLS, LRLRLSSSRLRS, ...

State Spaces 2: Our Intuitions



- Our intuitions often identify states that we think are:

- "Normal"
- "Expected"
- "Physically possible"



- Usually:

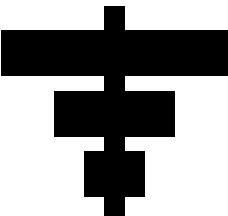
- The initial state is one of those states
- Mainly need to care about all states reachable from there (using the defined actions) – discussed later

State Spaces 3: Against our Intuitions



- But now that we introduced **facts**:

- **Every** combination of **facts** is a state
 - Some are "forbidden"
 - Towers of Hanoi:



- Some are "counter-intuitive" – in Blocks World, there are states where:
 - $\text{on}(A,A)$ is true
 - $\text{holding}(A)$ and $\text{ontable}(A)$ are true at the same time

These are like
"variables" that can
independently be
true or false!

State Spaces 4: ToH, Actions

33

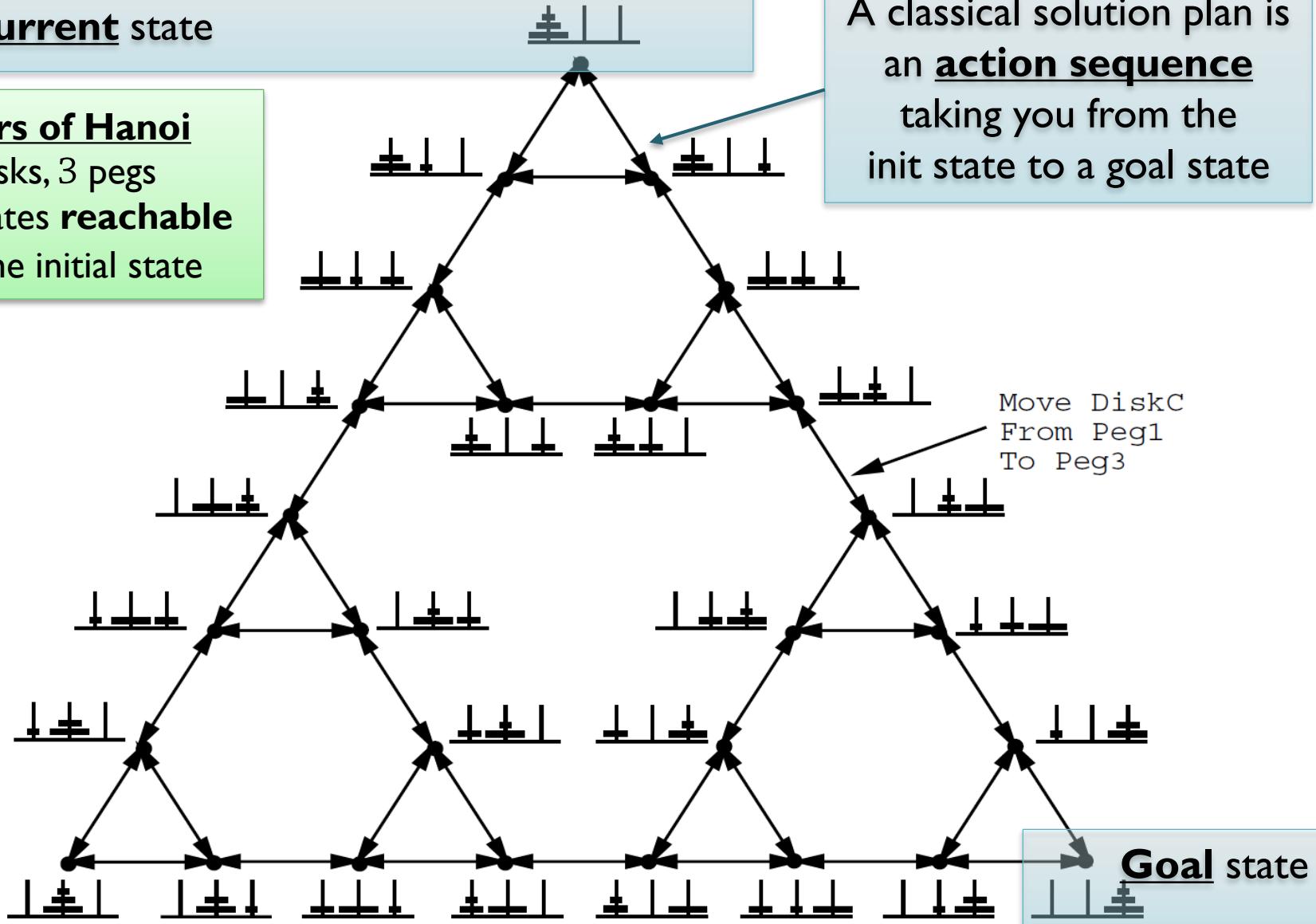


Initial/current state

Towers of Hanoi

3 disks, 3 pegs

→ 27 states **reachable**
from the initial state



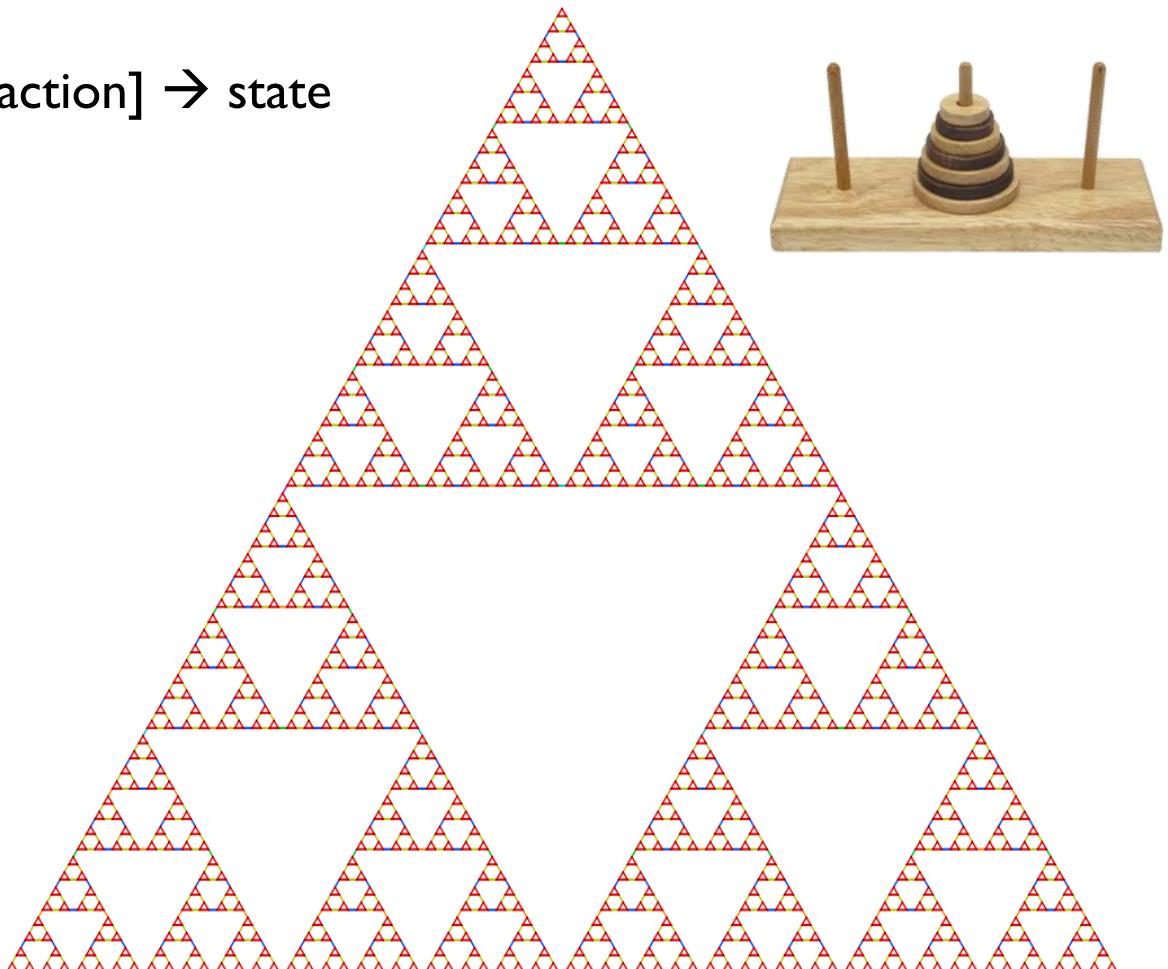
State Spaces 4: Larger Example

34

jonkv@ida

- Larger state space – interesting symmetry

- 7 disks
- 2187 “possible” states
- 6558 transitions, [state, action] → state



State Spaces 5: 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

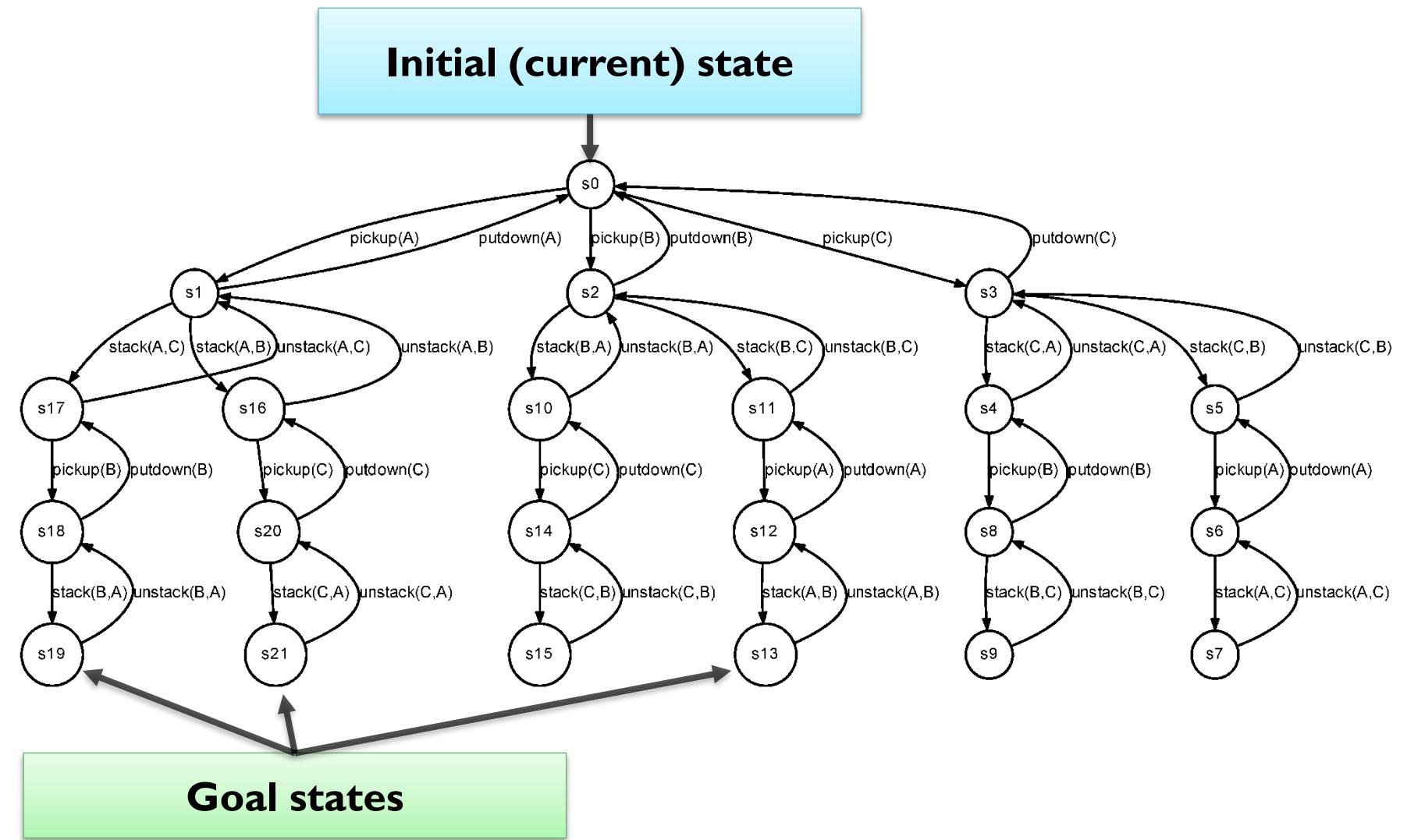


- Predicates (relations) used:

- **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 6: Blocks World, 3 blocks



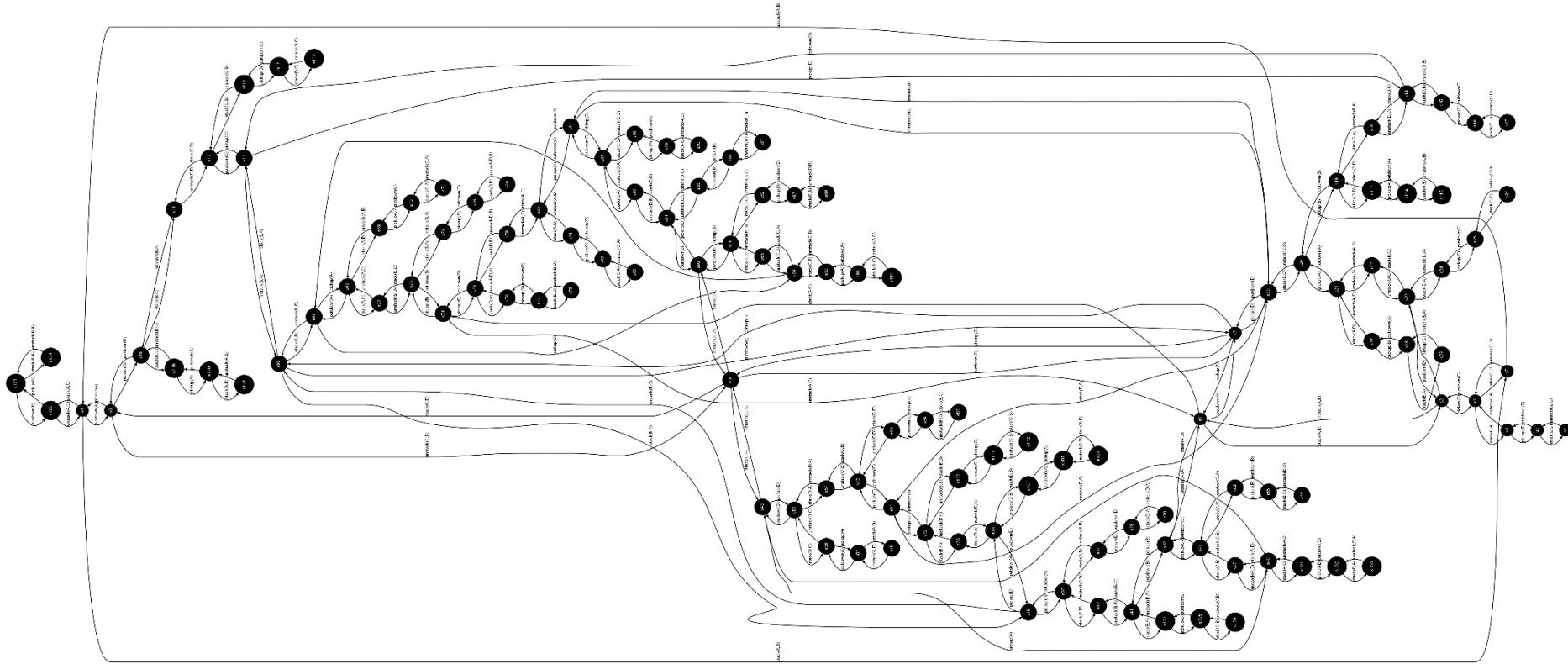
State Spaces 7: Blocks World, 4 blocks



37

jonkv@ida

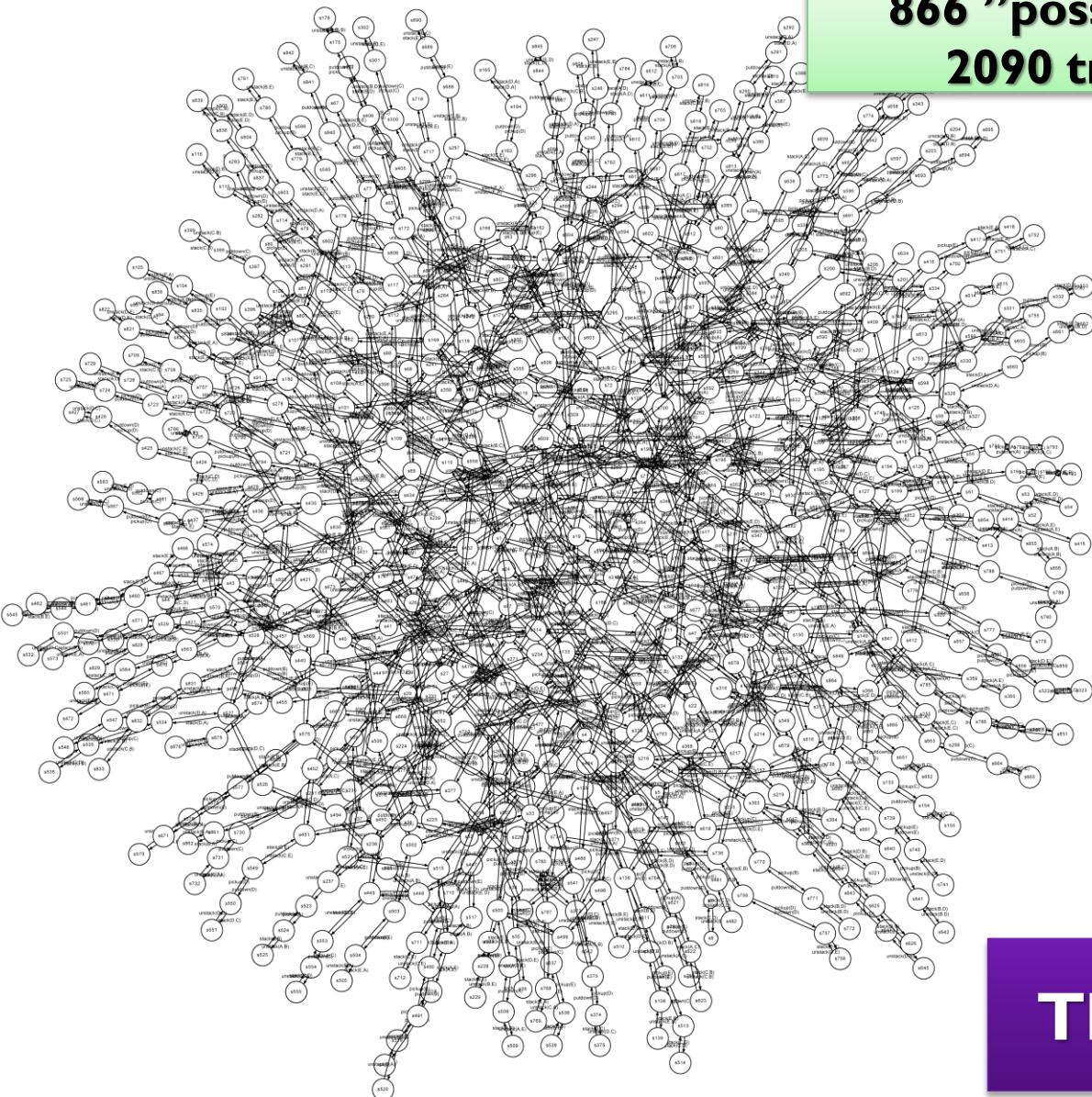
125 "possible" states
272 transitions



State Spaces 8: Blocks World, 5 blocks

38

jonkv@ida



**866 "possible" states
2090 transitions**

This is tiny!

State Spaces 9: Reachable States



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	

Plan Generation Method 1: Forward State Space Search

Forward Search 1

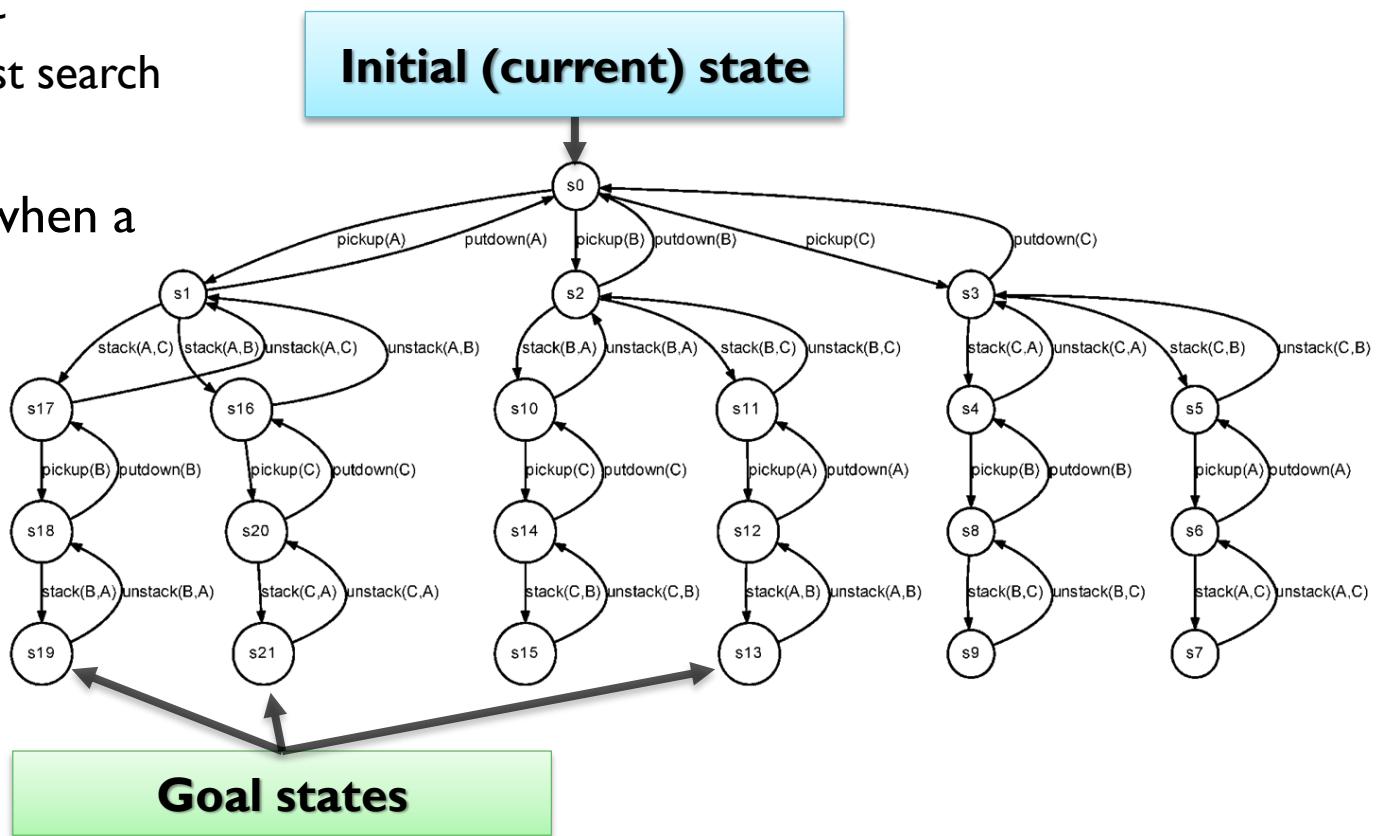
41

jonkv@ida

- Straight-forward planning: **Forward search in the state space**

- Start in the initial state
- Apply a **search** algorithm
 - Depth first
 - Breadth first
 - Uniform-cost search
 - ...

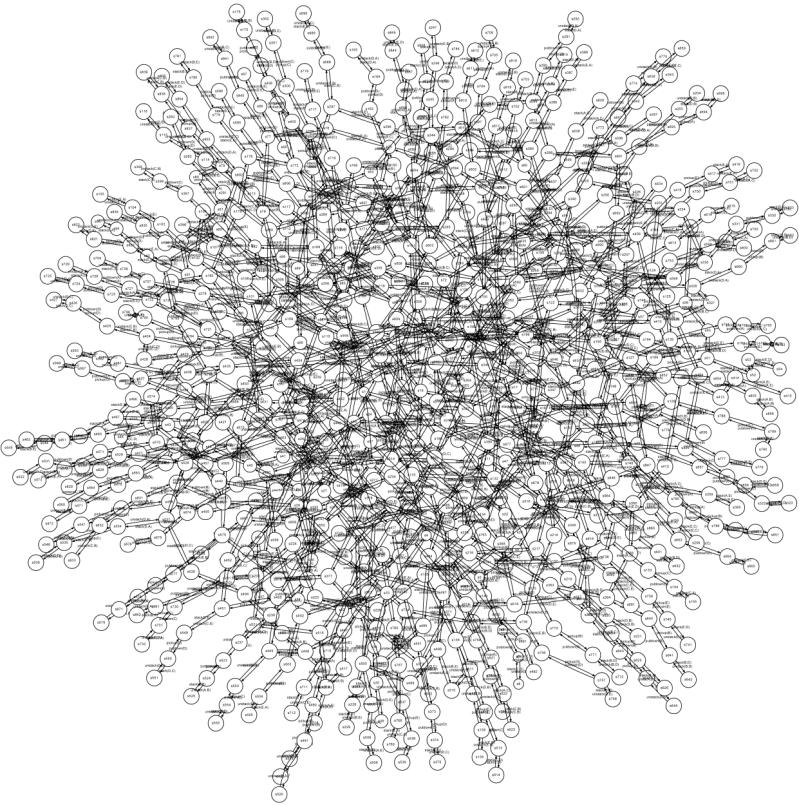
- Terminate when a goal state is found



Forward Search 2: Don't Precompute



- The planner is not given a complete precomputed search graph!



Usually too large!
→ Generate as we go,
hope we don't actually need the *entire* graph

Forward Search 3: Initial state

43

jonkv@ida

- The user (robot?) observes the current state of the world
 - The *initial* state



- Must describe this using the specified formal state syntax...
 - $s_0 = \{ \text{clear}(A), \text{on}(A,C), \text{ontable}(C), \text{clear}(B), \text{ontable}(B), \text{clear}(D), \text{ontable}(D), \text{handempty} \}$
- ...and give it to the planner, which creates one search node

{ **clear(A), on(A,C), ontable(C),**
clear(B), ontable(B), clear(D), ontable(D), handempty } }

Forward Search 4: Successors

44

jonkv@ida

- Given any search node...

```
{ clear(A), on(A,C), ontable(C),
  clear(B), ontable(B), clear(D), ontable(D), handempty }
```

- ...we can find successors – by applying actions!

- action** pickup(D)

- Precondition: **ontable**(D) \wedge **clear**(D) \wedge **handempty**

- Effects: \neg **ontable**(D) \wedge \neg **clear**(D) \wedge \neg **handempty** \wedge **holding**(D)

- This generates new reachable states...

...which can also
be illustrated

```
{ clear(A), on(A,C), ontable(C),
  clear(B), ontable(B), clear(D), ontable(D), handempty }
```

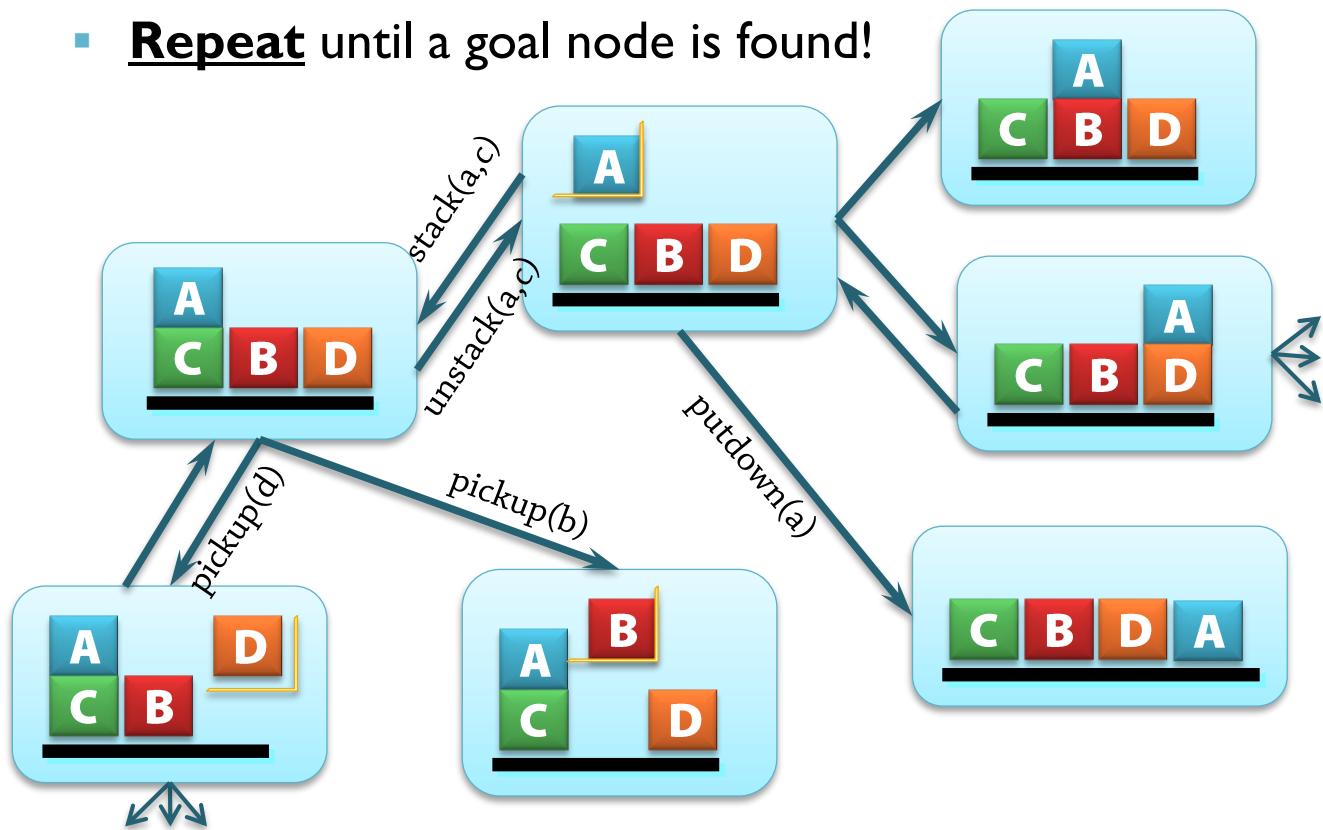
```
{ clear(A), on(A,C), ontable(C),
  clear(B), ontable(B), holding(D) }
```



Forward Search 5: Step by step

- A search strategy (depth first, A*, hill climbing, ...) will:

- Choose a node
- Expand the node, generating 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!



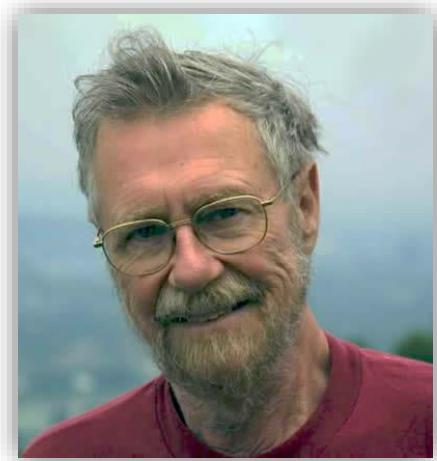
This is *illustrated* –
the planner works
with sets of facts

The blocks world is
symmetric: Can
always “return the
same way”
Not true for all
domains!

Uninformed Forward State Space Search

- One possible strategy: **Dijkstra's algorithm**

- Strategy can be described as *nearest-first*:
*“take an unexplored node
that can be reached at minimal cost from the initial node”*
 - **Efficient**: $O(|E| + |V| \log |V|)$
 - $|V|$ = the number of nodes
 - $|E|$ = the number of edges
 - And generates **optimal** (cheapest) paths/plans!



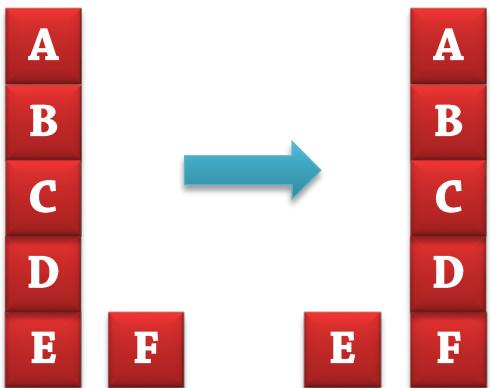
Would this algorithm be a solution?

Dijkstra's Algorithm: Example

48

jonkv@ida

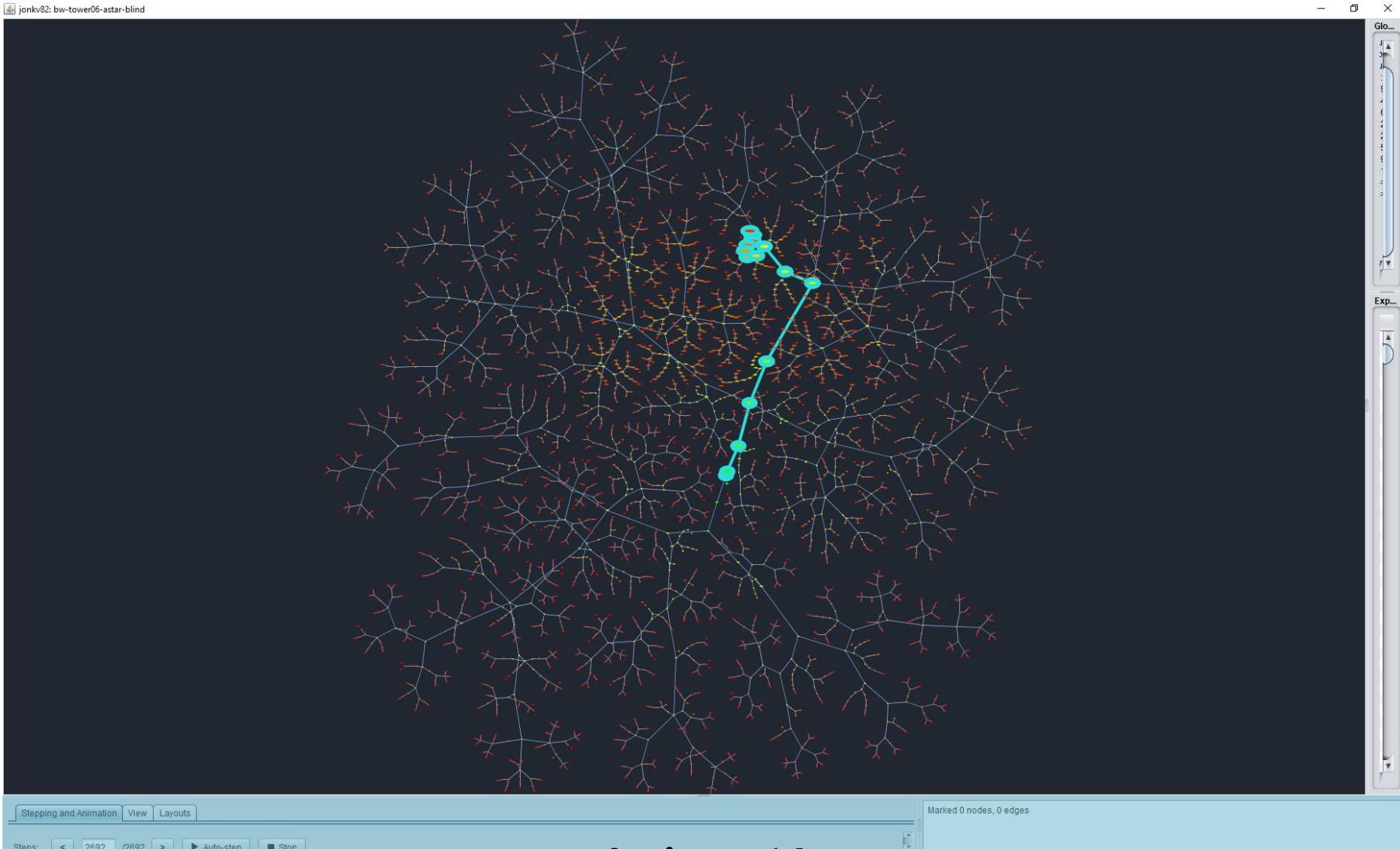
- A simple problem:



Goal
on(A,B)
on(B,C)
on(C,D)
on(D,F)
ontable(E)
ontable(F)

Optimal solution	
unstack(A,B)	pickup(D)
putdown(A)	stack(D,F)
unstack(B,C)	pickup(C)
putdown(B)	stack(C,D)
unstack(C,D)	pickup(B)
putdown(C)	stack(B,C)
unstack(D,E)	pickup(A)
stack(D,F)	stack(A,B)

bw-tower06-dijkstra: Only 6 blocks, Dijkstra search, no heuristic



Actions: 14

States: 8706 calculated, 2692 visited

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: > $400 \times 399 \times 398$ plans, ...
 - More than

163056983907893105864579679373347287756459484163478267225862419762304263994207997664258213955766581163654137118
163119220488226383169161648320459490283410635798745232698971132939284479800304096674354974038722588873480963719
240642724363629154726632939764177236010315694148636819334217252836414001487277618002966608761037018087769490614
847887418744402606226134803936935233568418055950371185351837140548515949431309313875210827888943337113613660928
318086299617953892953722006734158933276576470475640607391701026030959040303548174221274052329579637773658722452
54973845940445258650369316934
0912754853265795909113444084441755664211796
27432025699299231777374983037
488265744484456318793090777961572990289194
81058521781914647662930023360
1372350568748665249021991849760646988031691
39438655119417119333314403154
1302649432305620215568850657684229678385177
72535893398611212735245298803
3087201742432360729162527387508073225578630
777685901637435541458440833878709544174983977457450527557554417629122448835191721077333875230695681480990867109
051332104820413607822206465635272711073906611800376194410428900071013695438359094641682253856394743335678545824
320932106973317498515711006719985304982604755110167254854766188619128917053933547098435020659778689499606904157
077005797632287669764145095581565056589811721520434612770594950613701730879307727141093526534328671360002096924
483494302424649061451726645947585860104976845534507479605408903828320206131072217782156434204572434616042404375
21105232403822580540571315732915984635193126556273109603937188229504400

1.63 * 10¹⁷³⁵

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!

Informed Forward State Space Search

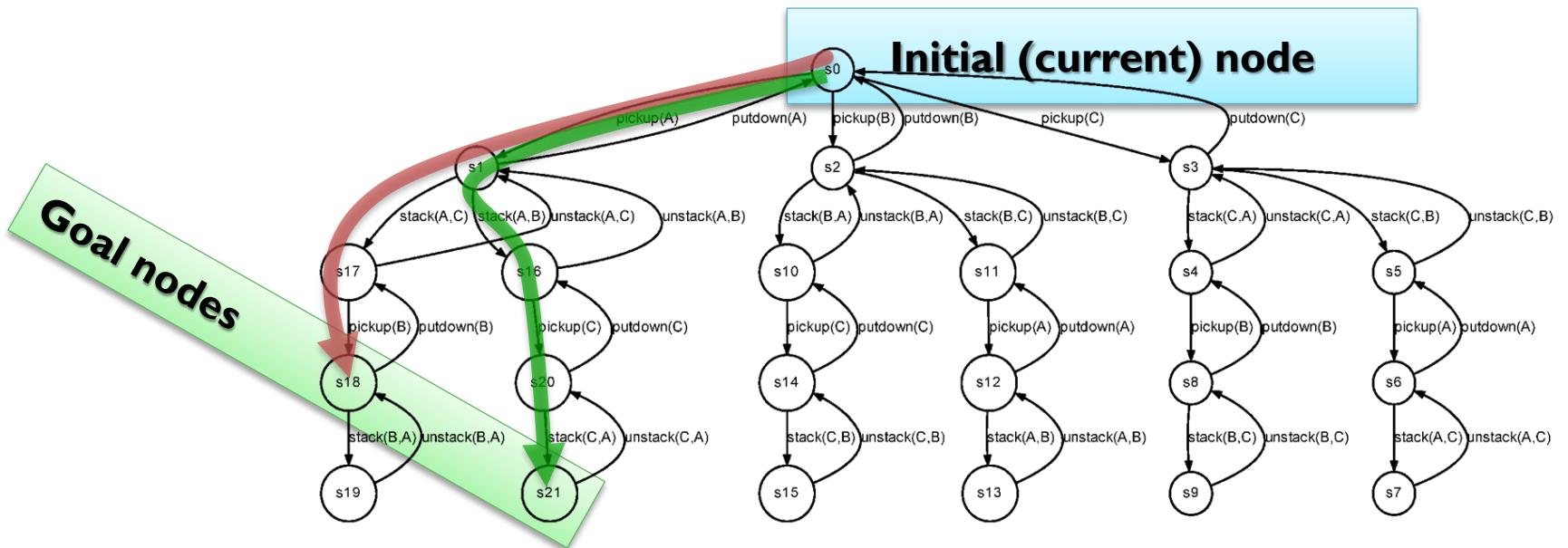
Intro (1)

53

jonkv@ida

We need:

- A heuristic function $h(n)$ estimating the cost of reaching a goal node from node n
 - Sometimes, we define **cost = number of actions**
 - More general: **each action has a cost $c(a)$ – longer plans may be cheaper!**



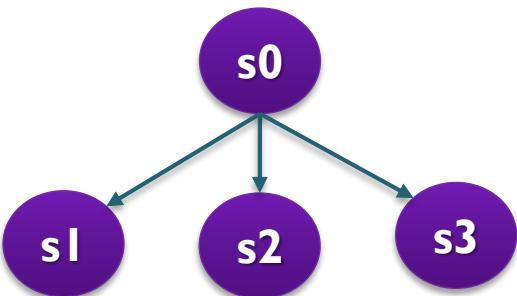
Intro (2)

- To use the heuristic, we will typically:

- Pick some state (let's say s_0):



- Expand it one step:



- Compute $h(n)$ for all the expanded nodes
 - $h(s_1), h(s_2), h(s_3)$
- Apply some search strategy using $h(n)$ as part of its selection criteria, to:

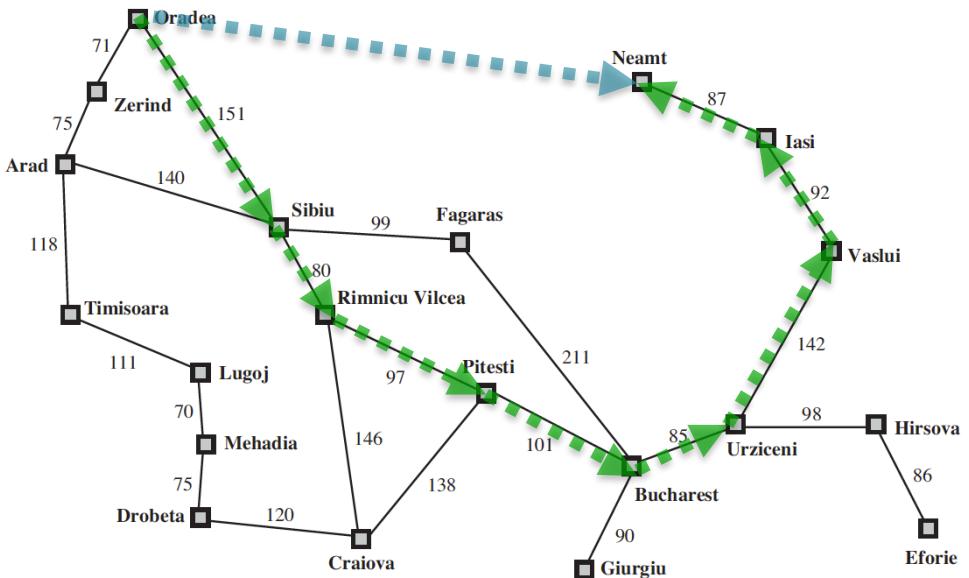


Intro (3)



- Previously we manually adapted heuristics to the problem
 - 8-puzzle → # pieces out of place, or sum of Manhattan distances
 - Romania Travel → straight line distance

7	2	4
5		6
8	3	1



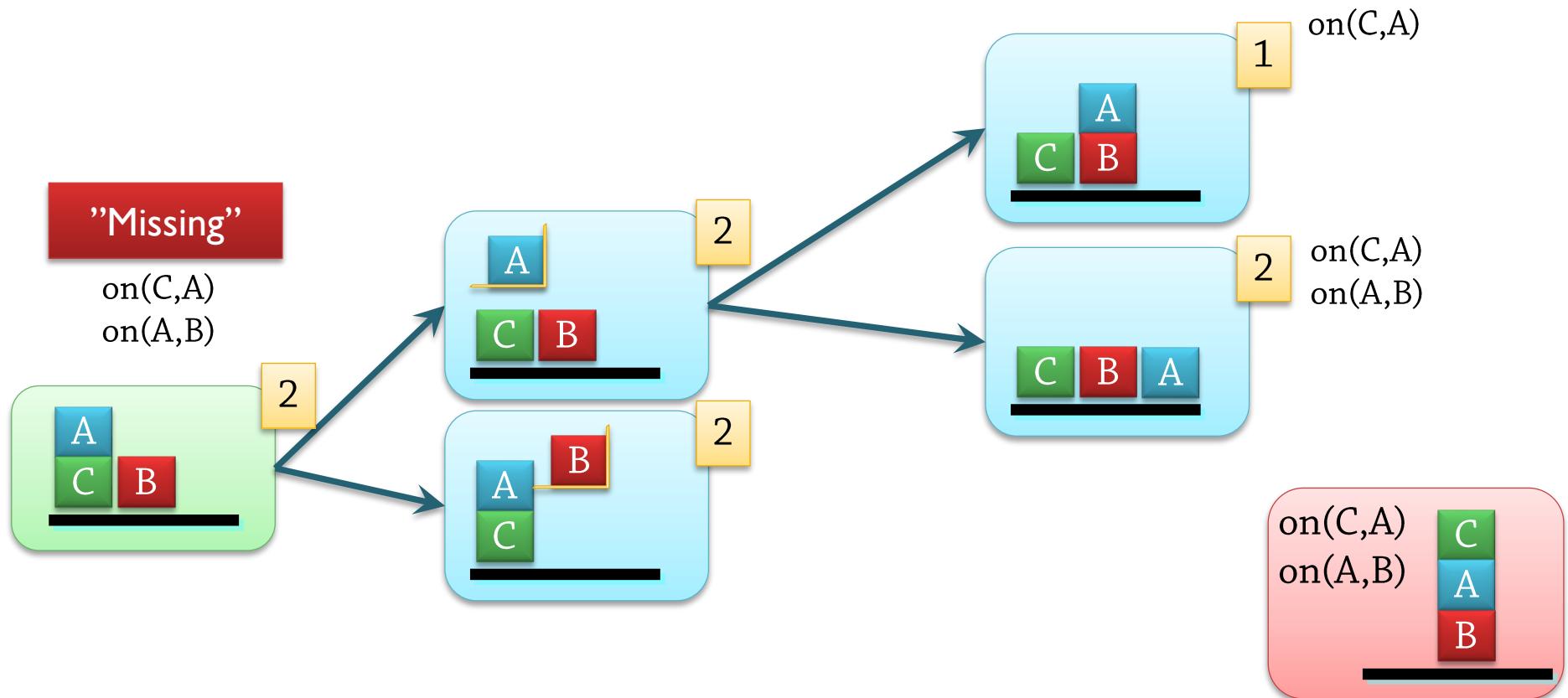
- Now: Want a general heuristic function
 - Without knowing what planning problem is going to be solved!

Domain-Independent Heuristics (1)



- A very simple domain-independent heuristic:
 - Count the number of unachieved goals in the current state
(similar to "number of pieces out of place")

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

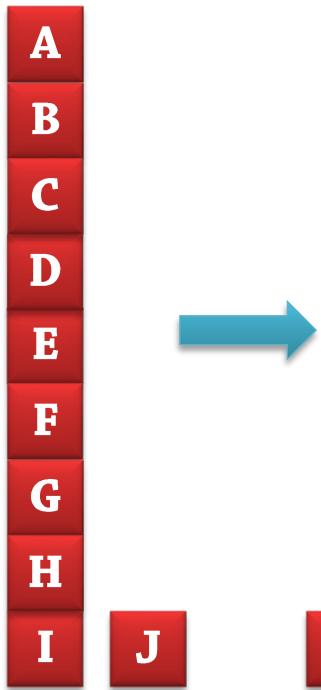


Domain-Independent Heuristics (2)

57

jonkv@ida

- Helpful, but not very...



Goal:

on(A,B)
on(B,C)
on(C,D)
on(D,E)
on(E,F)
on(F,G)
on(G,H)
on(H,J)

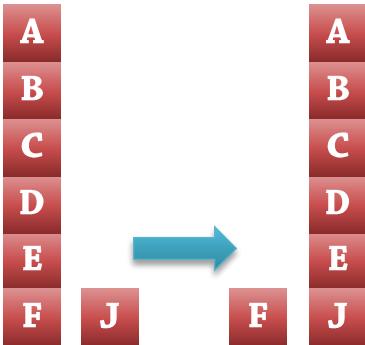
Optimal solution:

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)	

Only unachieved goal fact:
on(H,J)

Must remove many
"good facts"
to reach the goal

bw-tower07-astar-gc: Only 7 blocks, A* search, based on goal count

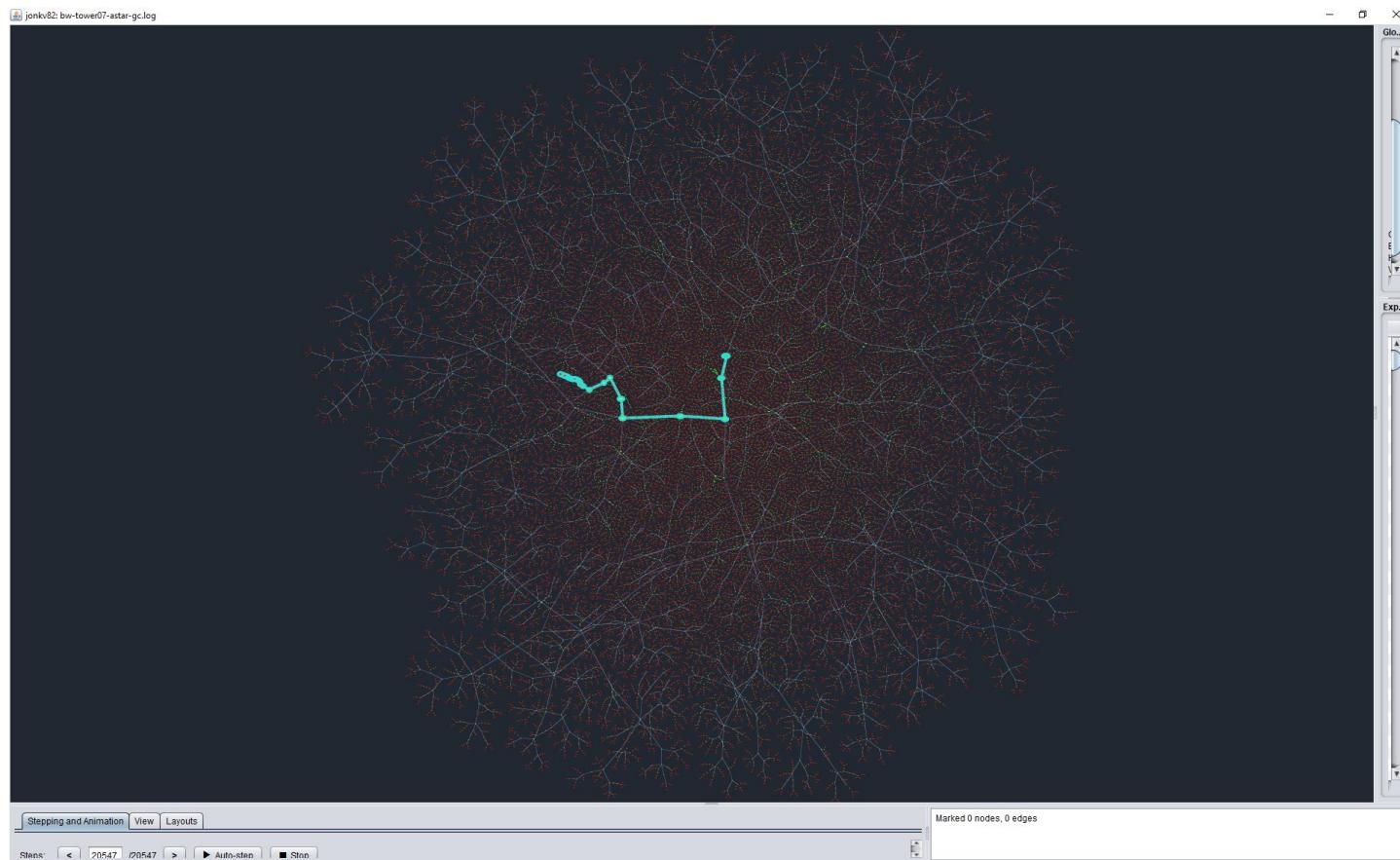


18 actions

States:

**6463 calculated,
3222 visited**

**(With Dijkstra,
43150 / 33436 –
improved, but we
can do better!)**



- $h(s_0) = 1$: Only one “missing” fact
- For a long time, all useful successors appear to increase remaining cost
 - Removing a block that must be moved
- And many useless successors appear to decrease remaining cost
 - Building towers that will need to be torn down

Optimal Classical Planning

Optimal 1: Introduction



■ Optimal plan generation:

- There is a quality measure for plans
 - Minimal number of actions
 - Minimal sum of action costs
 - ...
- We must find an optimal plan!



- Suboptimal plans
(0.5% more expensive):

Irrelevant

Optimal 2: A*



Optimal Plan Generation: Often uses A*

- A* focuses entirely on optimality
 - Find a **guaranteed optimal** plan as quickly as possible
 - But no point in trying to find a "reasonable" plan before the optimal one
 - Slowly expand from the initial state, systematically checking possibilities
- Requires admissible heuristics to guarantee optimality
 - Reason: Heuristic used for *pruning* (ignoring some search nodes)
 - Search queue ordered by $f = g$ (actual cost) + h (heuristic):

$$11 = 10 + 1$$

Pop – not a solution

$$12 = 10 + 2$$

Pop – not a solution

$$12 = 12 + 0$$

Pop – solution!

$$12 = 11 + 1$$

Ignore:
 g is known, h is an underestimate,
so solutions found by expanding
these nodes will cost $\geq g+h$
(and we have one of cost $\leq g+h$)

$$13 = 11 + 2$$

Creating Admissible Heuristic Functions: The General Relaxation Principle

How does relaxation apply to planning?

How can we make the definition more precise?

The Problem



■ We want:

- A general principle for developing different heuristic functions h(s)
 - Allowing us to explore different ideas for heuristics
- So that the heuristic functions we develop will:
 - Work for *all* classical planning problems, and *all* states s
 - Never overestimate the real cost of reaching the goal: h is admissible
 - Provide useful information – not simply $h(s) = 0$ for all s...

**What do we do?
Where do we start?
How do we think?**

An Obvious Method

64

jonkv@ida

■ One obvious(ish) method:

Every time we need $h(s)$ for some state s in a problem P :

1. Generate an optimal solution plan $\pi^*(s)$ starting in s

2. Let $h(s) = h^*(s) = \text{cost}(\pi^*(s))$
 - Admissible – why?
 - Informative – why?

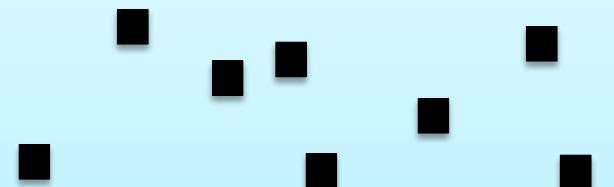
$h^*(s)$ = "the perfect heuristic" =
the actual cost of an optimal plan

■ Obvious, but stupid

- If we find $\pi^*(s)$, we're already done!

Also: These are hard to find
(or we wouldn't need
a heuristic)

Solutions π to P starting in s
(set of plans!)



Optimal solutions $\pi^*(s)$



Transforming a Problem



- Main problem: **performance** – we need something **faster**

Original problem P ,
current state s
(finding a solution: slow)



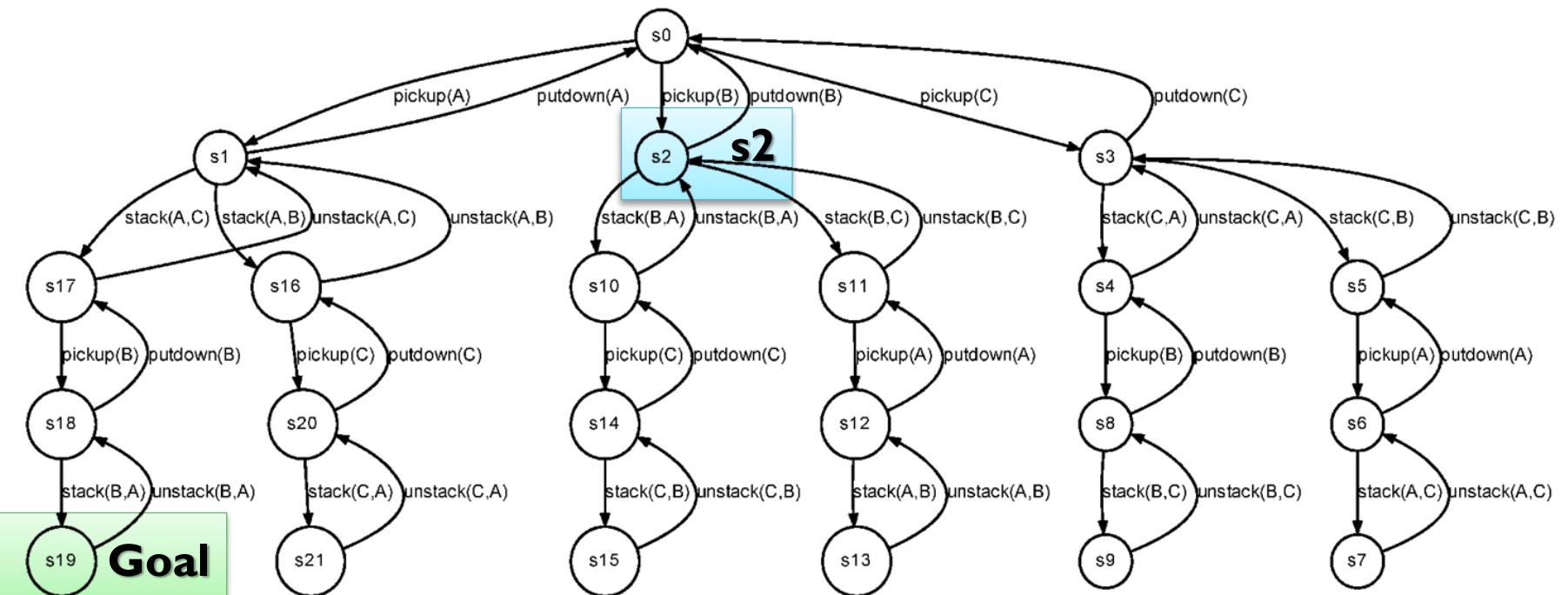
Transformed "simplified"
problem: P' , s'
(finding a solution: fast)

Simplification Example: Basis

66

jonkv@ida

- A simple planning problem (domain + instance)
- Blocks world, 3 blocks
- **Currently:** in state s2
- **Goal:** (and (on B A) (on A C)) (only satisfied in s19)
- Solutions from here:
All paths from s2 to goal (infinitely many – can have cycles)



Simplification Example 1

67

jonkv@ida

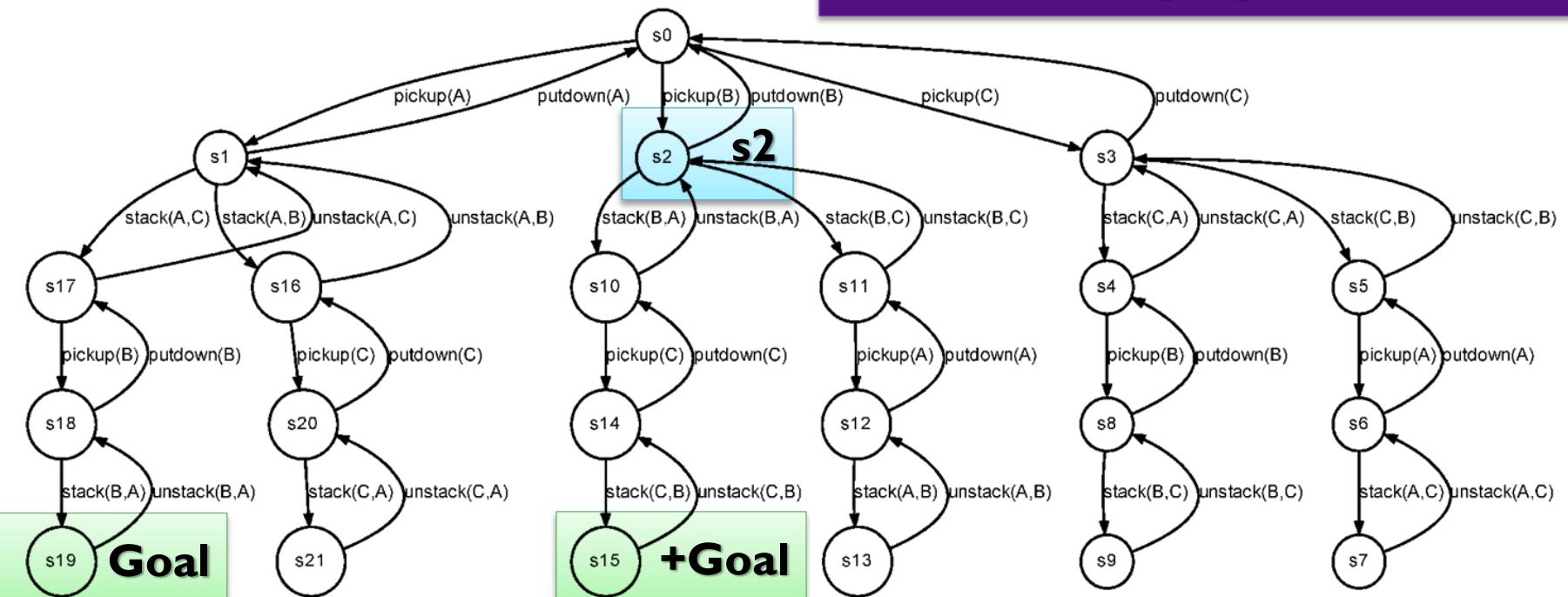
■ Adding goal states

- New goal formula: (and (on B A) **(or (on A C) (on C B))**)

- Informally:

More goal states → should be easier to find one from s_2 , on average

May need more transformations:
Just showing a general idea!



Computing a Heuristic Value



- **What do we do** with the transformed problem?

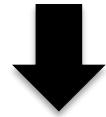
Original problem P ,
current state s
(finding a solution: slow)



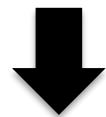
Transformed problem:
 P' , s'
(finding a solution: fast)



Still admissible? Yes, if
 $\text{cost}(\pi^*(P', s')) \leq \text{cost}(\pi^*(P, s))$
(The optimal plan we **found** for P'
was not more expensive than the
actual cost for P)



Quickly find $\pi^*(P', s')$ –
optimal plan for
transformed problem



Let $h(s) = c$



Compute
 $c = \text{cost}(\pi^*(P, s))$

Simplification Example 1

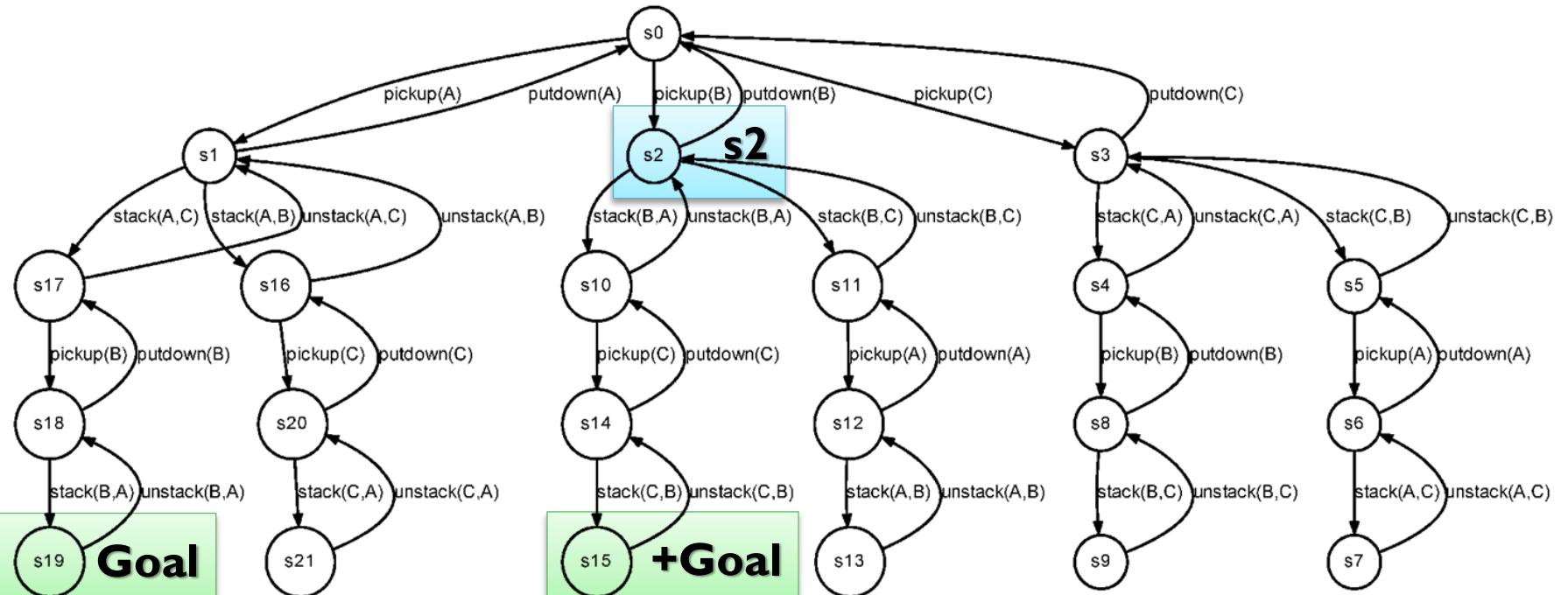
69

jonkv@ida

■ Adding goal states

- Can adding goal states make optimal plans more expensive?
- No, only cheaper from some states

How can we generalize this result?

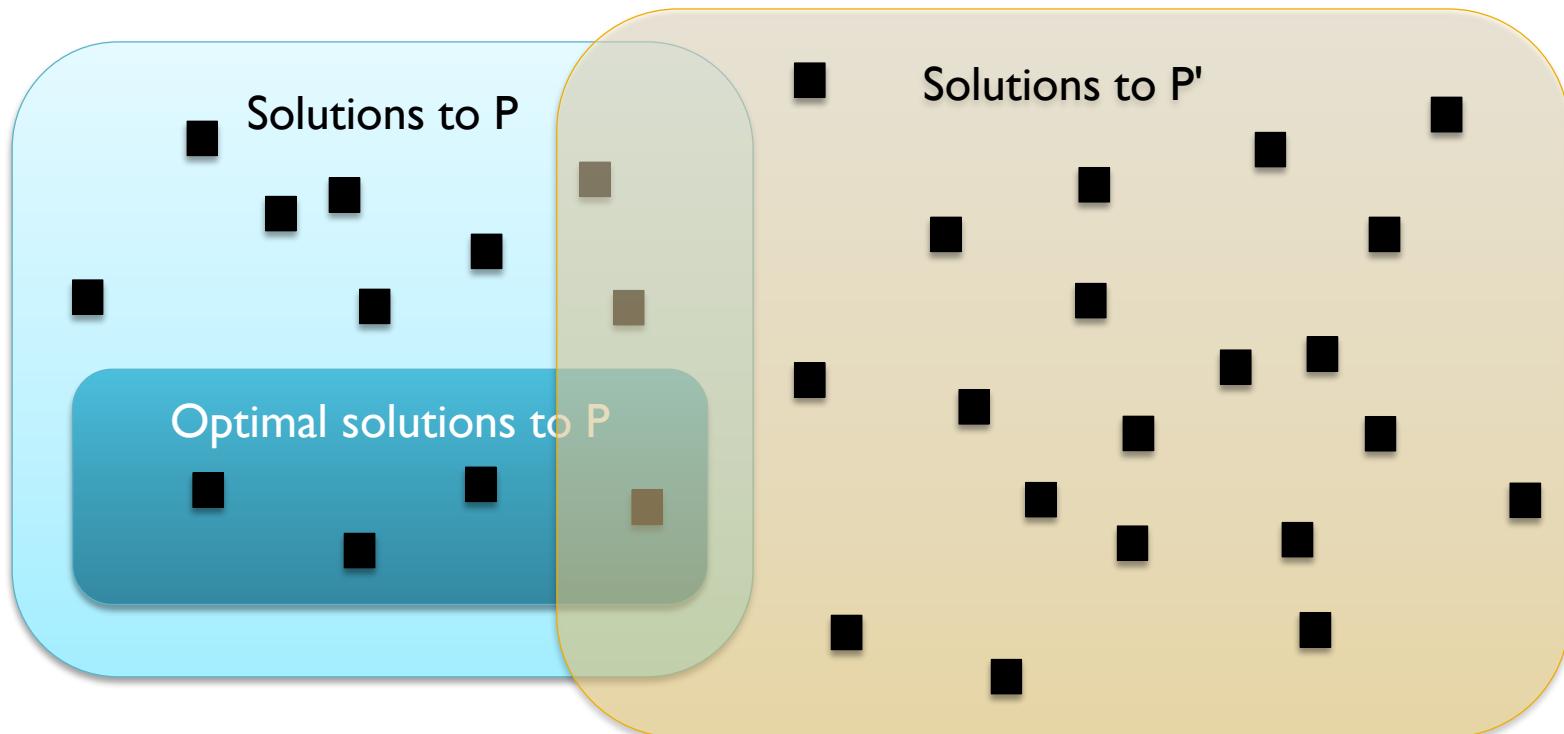


Solution Sets (1)



- How to prove $\text{cost}(\text{optimal-solution}(P')) \leq \text{cost}(\text{optimal-solution}(P))$?
 - Sufficient criterion: One optimal solution to P remains a solution for P'
 - $\text{cost}(\text{optimal-solution}(P')) = \min \{ \text{cost}(\pi) \mid \pi \text{ is any solution to } P' \} \leq \text{cost}(\text{optimal-solution}(P))$

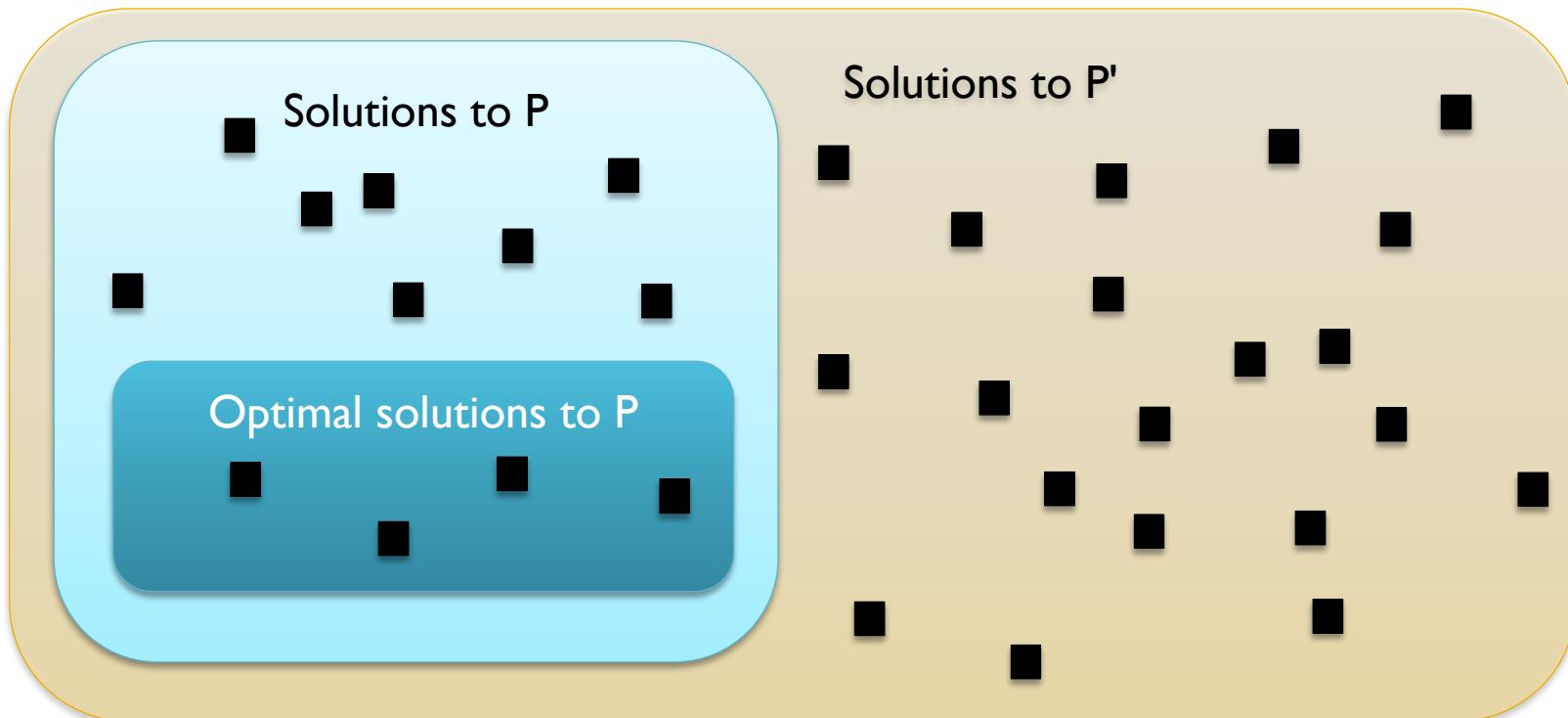
Includes the optimal solutions to P ,
so $\min \{ \dots \}$ cannot be greater



Solution Sets (2)



- Another sufficient criterion: All solutions to P remain solutions for P'
 - Stronger, but often easier to prove
 - This is called relaxation: P' is a relaxed version of P
 - Relaxes the constraint on what is accepted as a solution:
The **is-solution(plan)?** test is "expanded, relaxed" to cover additional plans

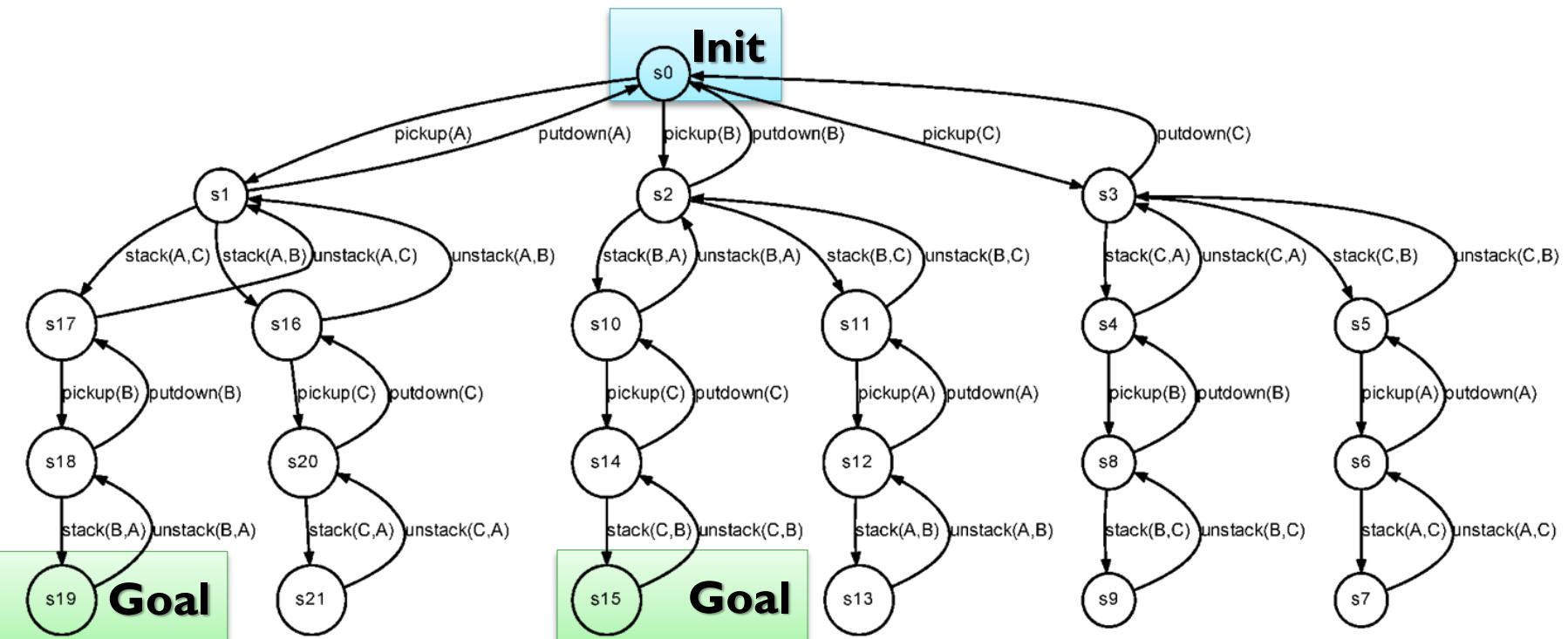


Relaxed Problem 1



■ 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

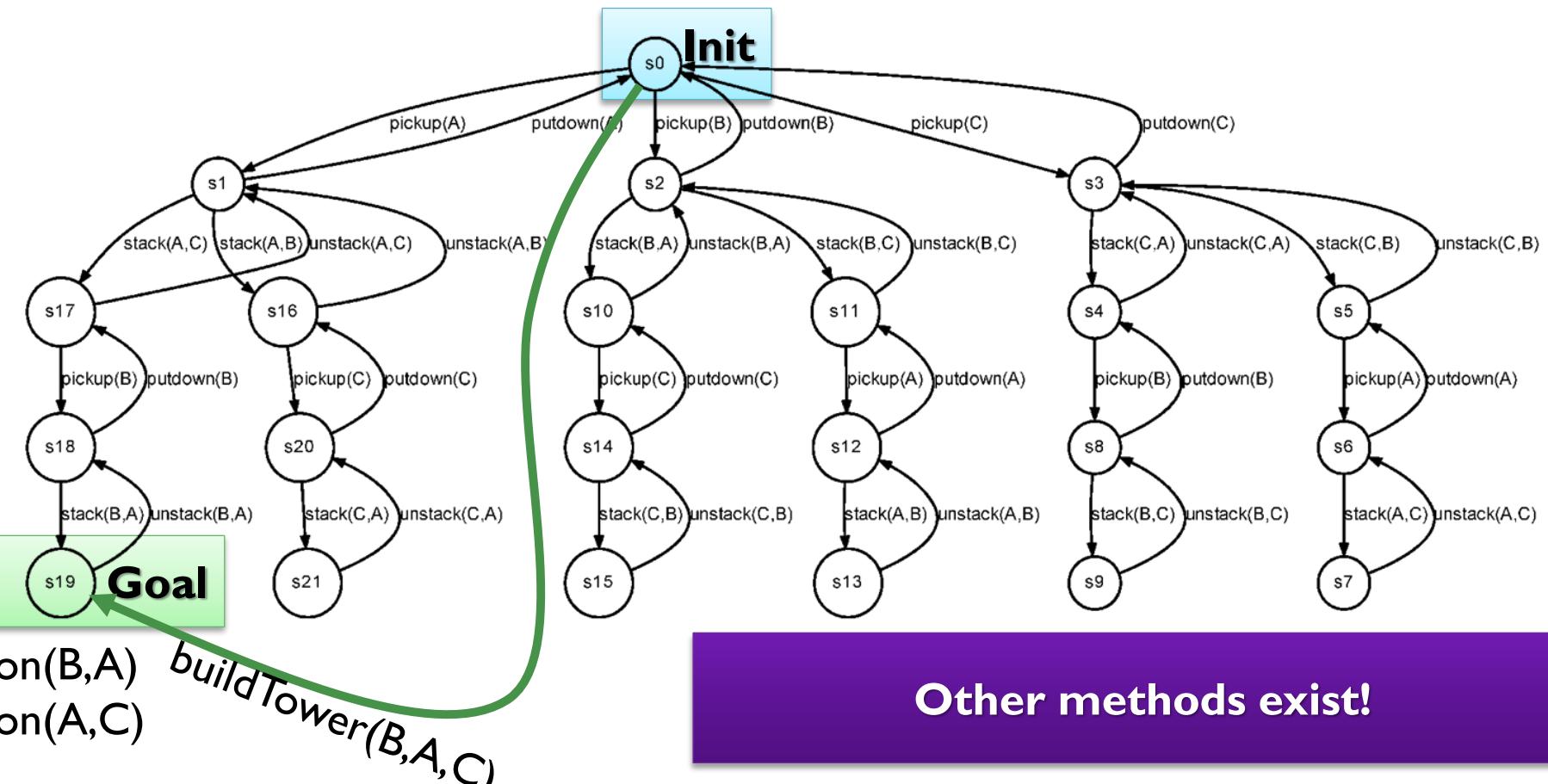


Relaxed Problem 2

73

jonkv@ida

- 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

74

jonkv@ida

■ Important:

Relaxation is not simply "removing preconditions"

There are many other relaxations

Definition: Transformation that preserves all 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)



Relaxation is just one specific way of (1) finding a simplifying transformation, and (2) proving "not-more-expensive"!

You've seen relaxations adapted to the 8-puzzle

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

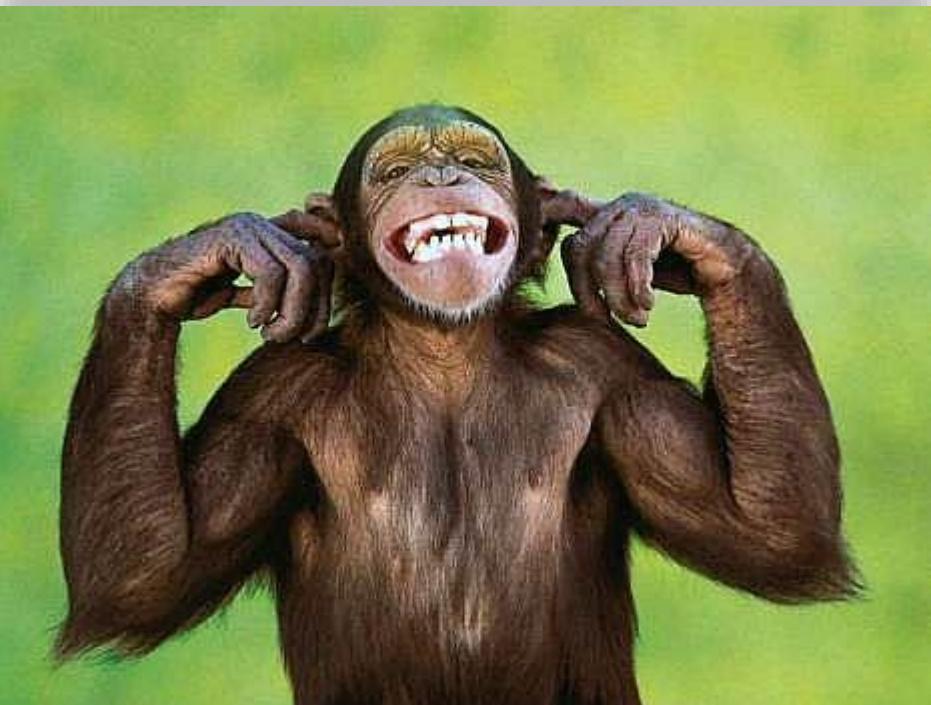
Example: Pattern Database Heuristics

PDB 1: Introduction

77

jonkv@ida

- Main idea behind pattern databases:
 - Let's ignore some facts – everywhere
 - In goals
 - In preconditions or effects
 - Compute costs as if those facts didn't matter



PDB 2: Dock Worker Robots

78

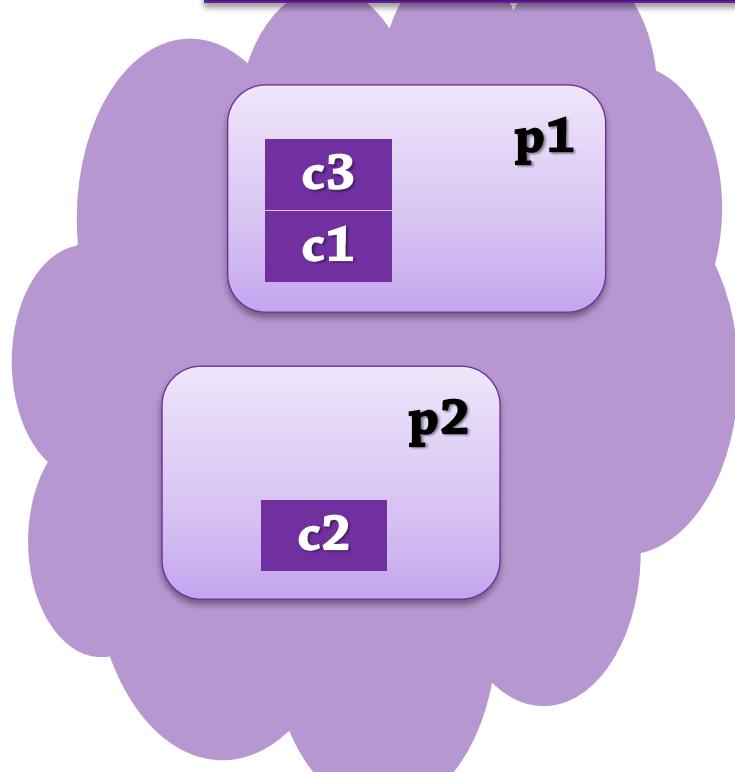
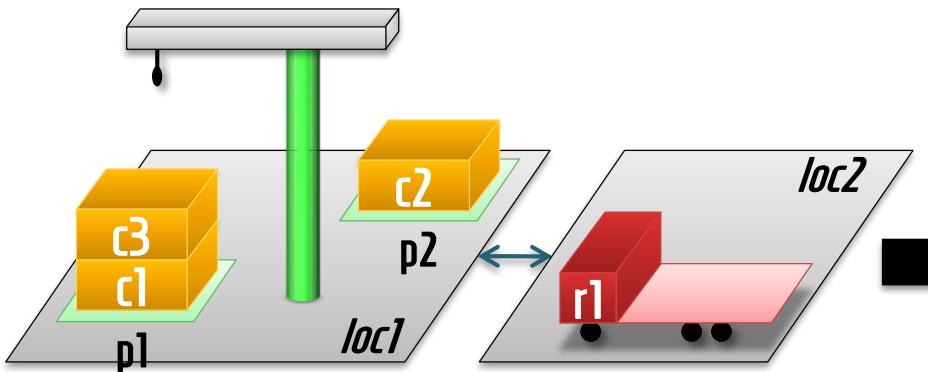
jonkv@ida

- Example: Dock Worker Robots
 - Care about facts related to **container locations**
 - $in(container, pile)$, $top(container, pile)$, $on(c1, c2)$, ...
 - Ignore robot locations, crane locations, ...
 - Original states are grouped together

Abstract state in **P'**,
represents *many*
states in **P** where
c3 is on **c1** in **p1**,

...

Ordinary state in **P**,
all facts defined



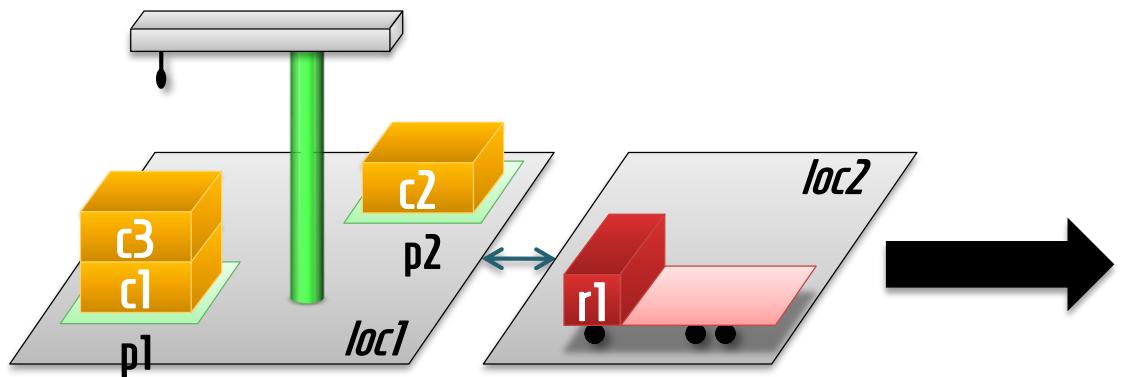
PDB 3: Planning in Patterns

79

jonkv@ida

- In P' we (pretend that we) can use the crane at p1 to:
 - pick up c3 (as we should)
 - place something on r1 (too far away, but we don't care)
 - place five containers on one truck
- But we can't:
 - pick up c1 (we do care about pile ordering)
 - immediately place c1 below c2, ...
 - → **Still a planning problem P' left to solve!**

New paths
to the goal!



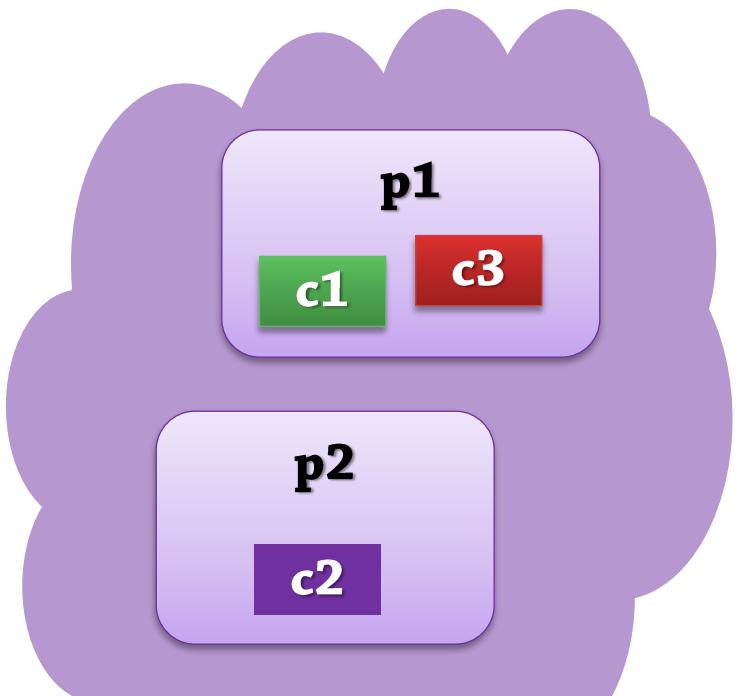
Solve optimally, compute cost
→ admissible heuristic!

PDB 4: Computing a Heuristic Value

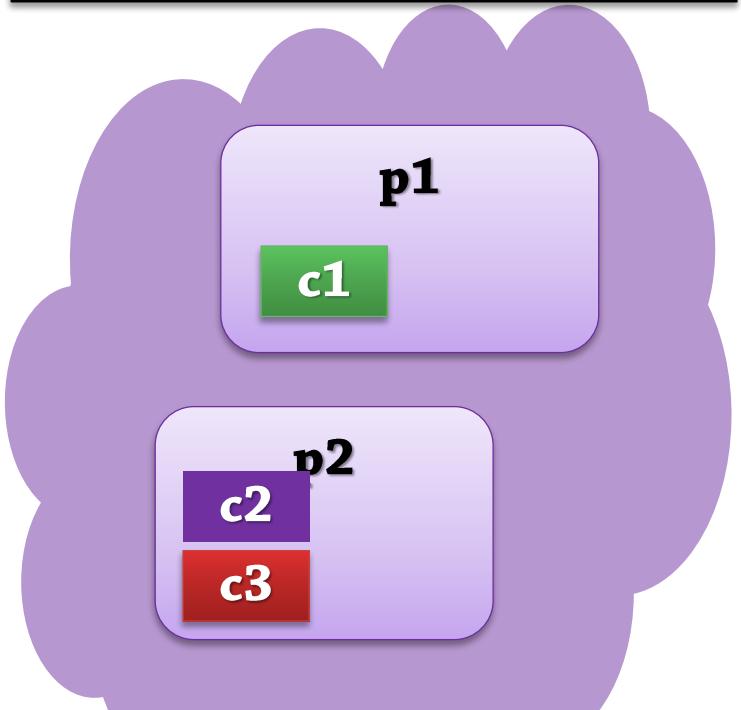


- **Solve $P'(s)$ optimally**, compute cost → admissible heuristic $h(s)!$
 - Take **c2** with the crane (it's in the way)
 - Take **c3** with the crane [relaxation – not checking if the crane is busy]
 - Place **c3** at the bottom
 - Place **c2** on the top

Abstract current state s



Abstract goal



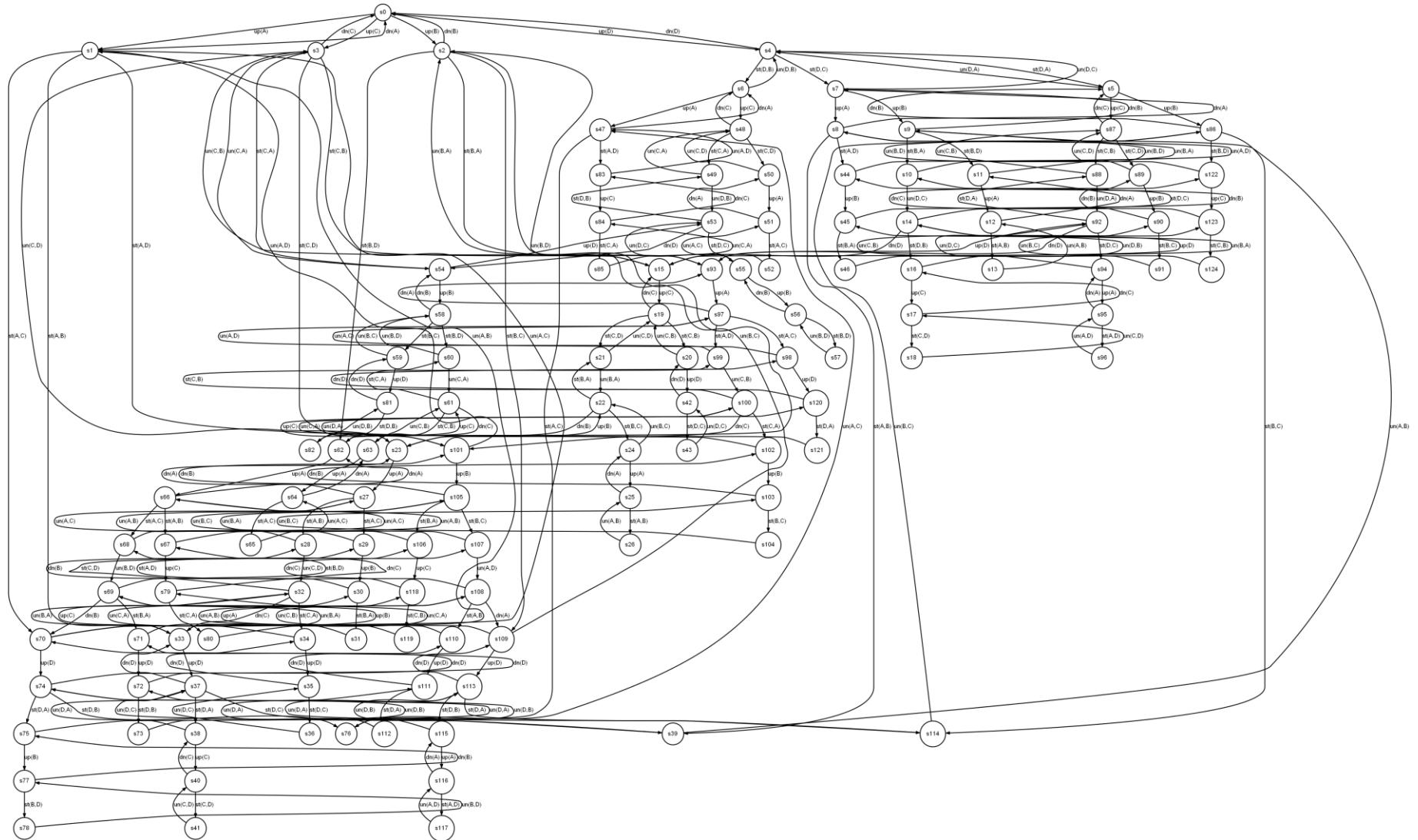
Let's formalize!

PDB: Blocks World size 4

82

jonkv@ida

- Consider physically achievable states in the blocks world, size 4:



PDB: Blocks World size 4, facts

83

jonkv@ida

- All **ground atoms (facts)** in this problem instance:

- **(on A A) (on A B) (on A C) (on A D)**
(on B A) (on B B) (on B C) (on B D)
(on C A) (on C B) (on C C) (on C D)
(on D A) (on D B) (on D C) (on D D)

(ontable A) (ontable B) (ontable C) (ontable D)

(clear A) (clear B) (clear C) (clear D)

(holding A) (holding B) (holding C) (holding D)

(handempty)

PDB: Ignoring Facts

84

jonkv@ida

- Example: only consider 5 **ground facts** related to **block A**

- "Pattern": $p=\{(on\ A\ B), (on\ A\ C), (on\ A\ D), (clear\ A), (ontable\ A)\}$

- Initial state:**



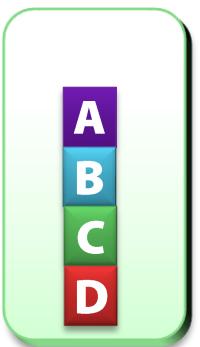
ontable(A)
ontable(B)
ontable(C)
ontable(D)
clear(A)
clear(B)
clear(C)
clear(D)
handempty



ontable(A)
clear(A)

An "abstract state"

- Goal:**



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



clear(A)
on(A,B)

An "abstract goal"

PDB: Transforming Actions

85

jonkv@ida

- Pattern $p = \{(on A B), (on A C), (on A D), (clear A), (ontable A)\}$

- Example action:** (unstack A B)

- Before transformation:**

```
:precondition (and (handempty) (clear A) (on A B))
```

```
:effect      (and (not (handempty)) (holding A) (not (clear A)) (clear B)  
                (not (on A B)))
```

- After transformation:**

```
:precondition (and (clear A) (on A B))
```

```
:effect      (and (not (clear A)) (not (on A B)))
```

Loses **some** preconditions
and effects

Let's call this action
 $transform(a, p)$

- Example action:** (unstack C D)

- Before transformation:**

```
:precondition (and (handempty) (clear C) (on C D))
```

```
:effect      (and (not (handempty)) (holding C) (not (clear C)) (clear D)  
                (not (on C D)))
```

- After transformation:**

```
:precondition (and)
```

```
:effect      (and)
```

Loses **all** preconditions and
effects → never used!

PDB: Patterns, Abstract States

86

jonkv@ida

- The set of ground facts we care about is called a pattern p

- A state s is represented by the abstract state $s \cap p$
- If $s \cap p = s' \cap p$, the two states are considered equivalent

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

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

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

represented
by a single
abstract
state

(**clear A**)
(on A B)

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

(**clear A**)
(ontable A)
(holding B)
(clear C)
(on C D)
(ontable D)

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

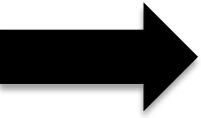
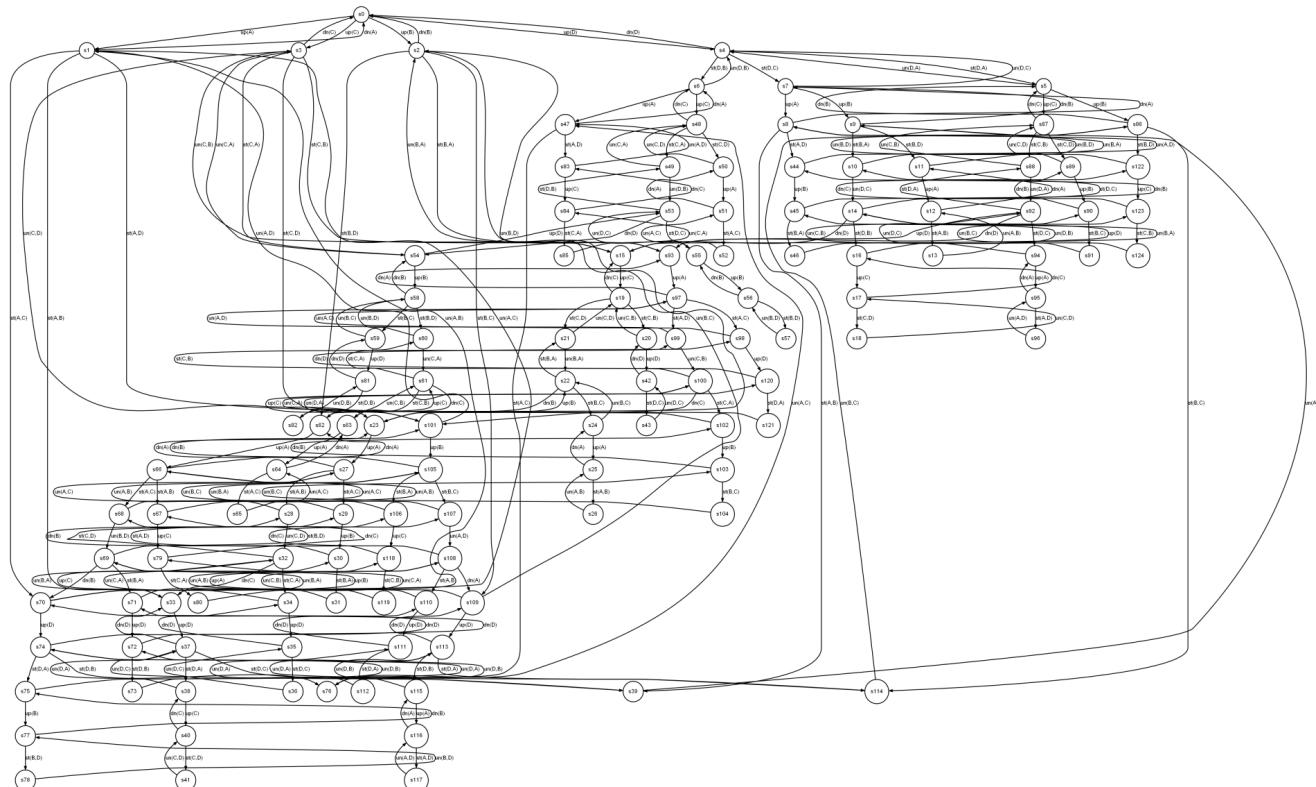
represented
by a single
abstract
state

(**clear A**)
(ontable A)

A pattern generally contains few facts – for performance!

PDB: New State Space

87



jonkv@ida

PDB: Relaxation?



■ Is this a relaxation?

- Yes
- Facts disappear from states...
 - $S' = \{s \cap p \mid s \in S\}$
- But also from precond/goal requirements!
 - If a_i could be executed in s ,
 $\text{transform}(a_i)$ can be executed in $s \cap p$
 - If γ' is the state transition function given transformed actions, then
$$\gamma'(\text{transform}(a_i), s \cap p) = \gamma(a_i, s) \cap p$$
 - → executable action sequences are preserved
- If $g \subseteq s$, then $g \cap p \subseteq s \cap p$
- So: Solutions are preserved (but new solutions may arise)

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



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

PDB: State Transition Graph

89

jonkv@ida

■ New reachable state transition graph:

- **Current state:** Everything on the table, hand empty, all blocks clear
 - Abstract state: $s_0 = \{ (\text{ontable } A), (\text{clear } A) \}$

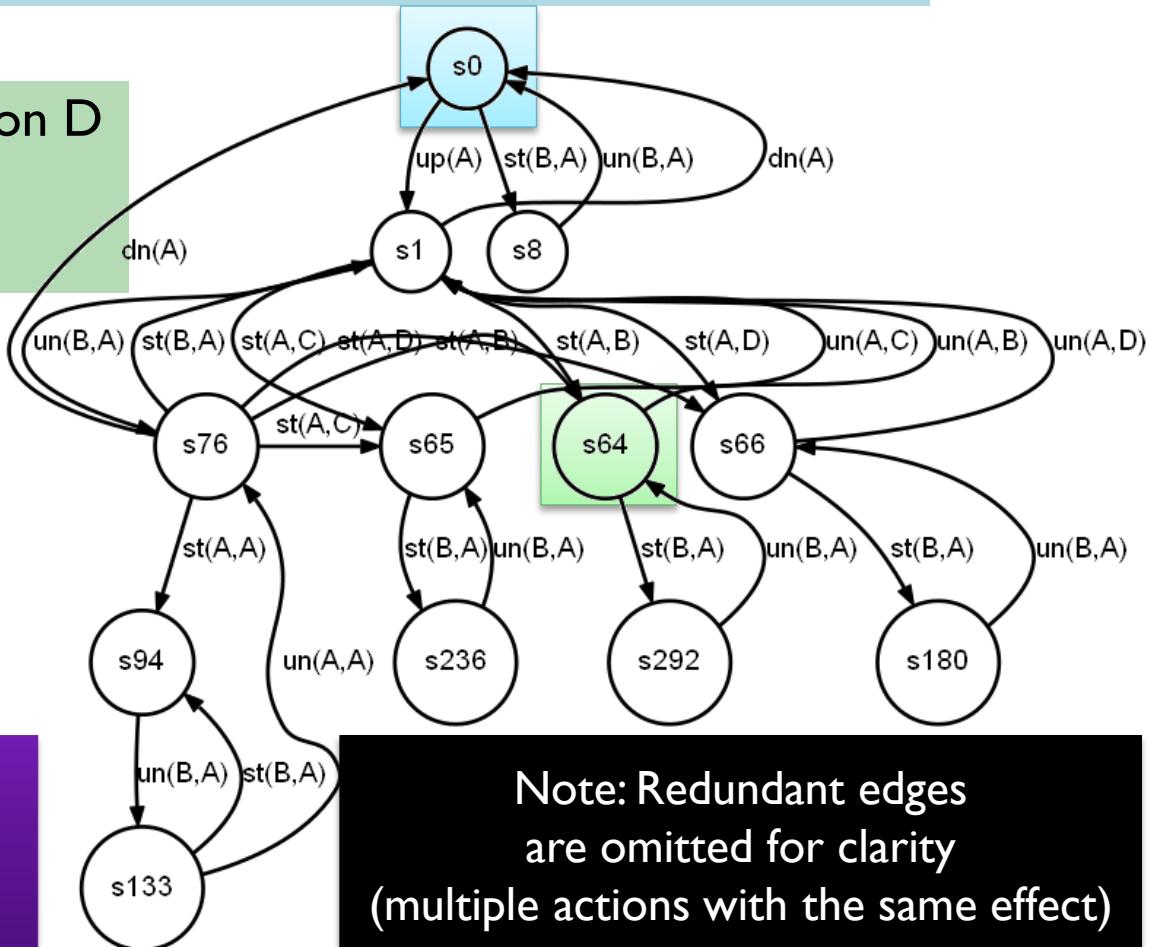
- **Goal state:** A on B on C on D

- Abstract goal: $s_{64} = \{ (\text{on } A B), (\text{clear } A) \}$

- Sufficiently few states to quickly compute optimal costs

- Cost is *at least* 2:
Shortest path $s_0 \rightarrow s_{64}$

Optimal cost of a relaxation
→
admissible heuristic



Note: Redundant edges are omitted for clarity
(multiple actions with the same effect)

PDB: More information



- To make PDB heuristics **more informative**:
 - Calculate costs for **several** patterns
 - Suppose we only care about **{clear(A), ontable(A)}**
 - Suppose we only care about **{on(A,B), on(C,D)}**
 - Suppose we...
 - Take the **maximum** of the computed heuristic values
- Real difficulty:
 - Choosing which patterns to use...

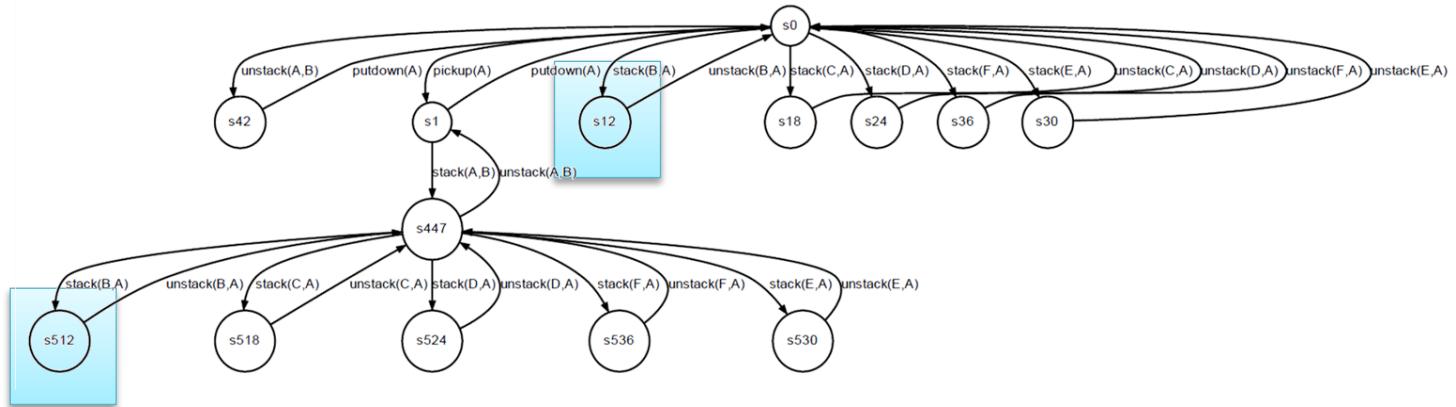
PDB: Databases

91

jonkv@ida

■ Where did the databases go?

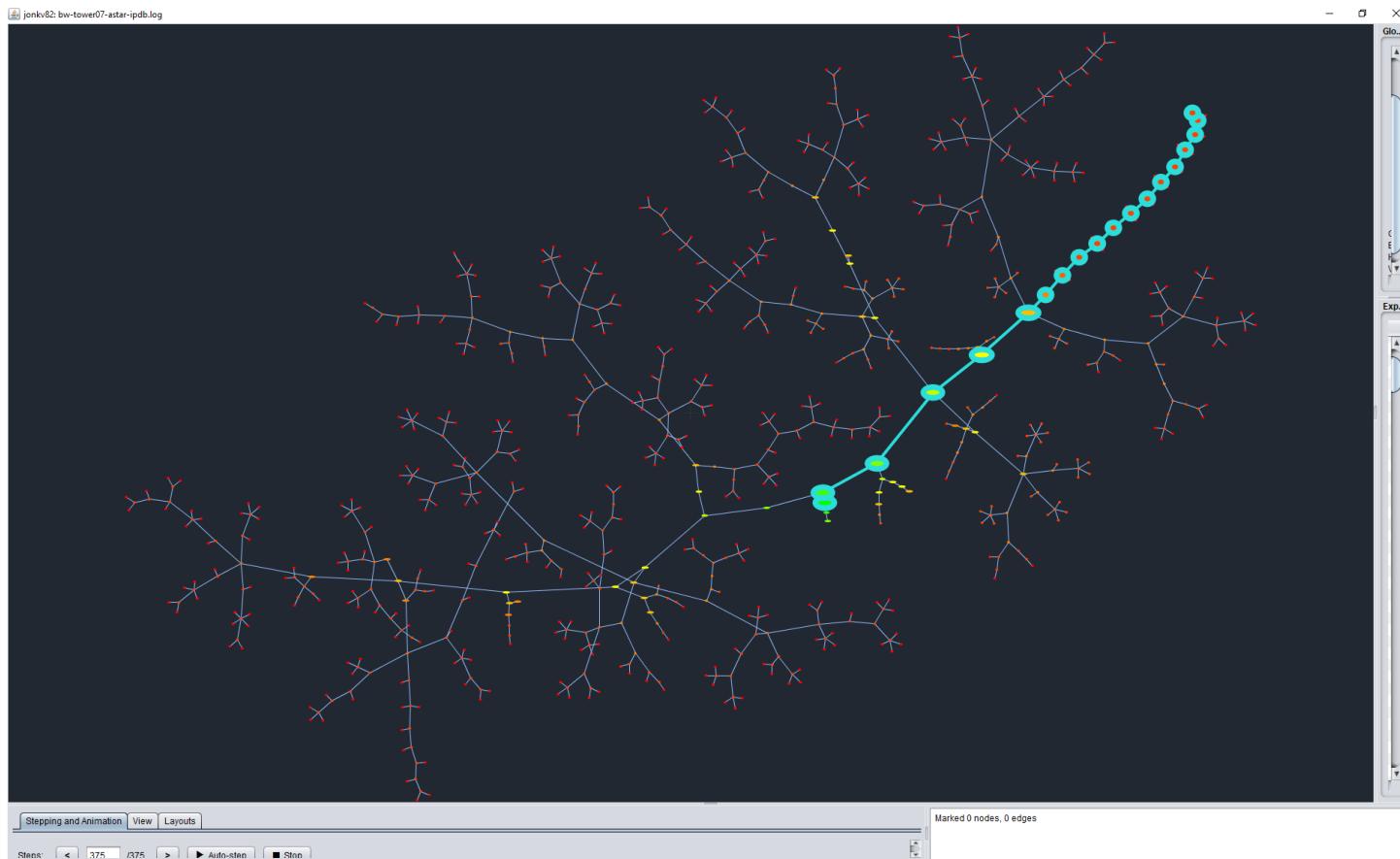
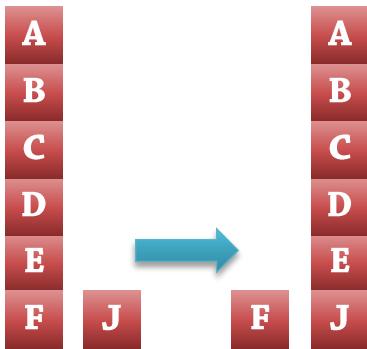
- Planning visits many **abstract** states over and over again
- For each pattern, we **precompute** all reachable abstract states



- 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*

Only for efficiency!
The principle is more important!

bw-tower07-astar-ipdb: Only 7 blocks, A* search, based on PDB variation



- Blind A*: 43150 states calculated, 33436 visited
- A* + goal count: 6463 states calculated, 3222 visited
- A* + iPDB: 1321 states calculated, 375 visited

No heuristic is perfect – visiting some additional states is fine!