TDDC17:
Introduction to Automated Planning

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

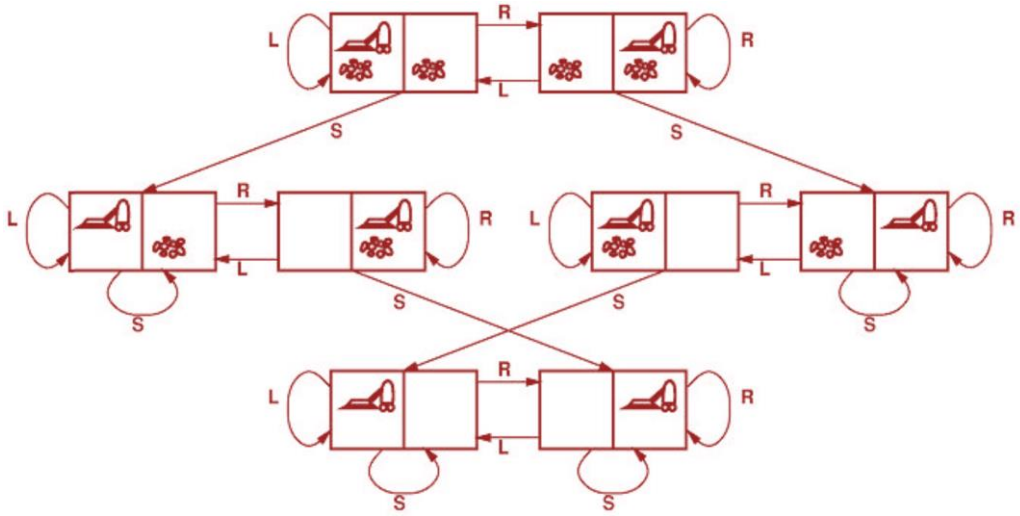
jonas.kvarnstrom@liu.se – 2018

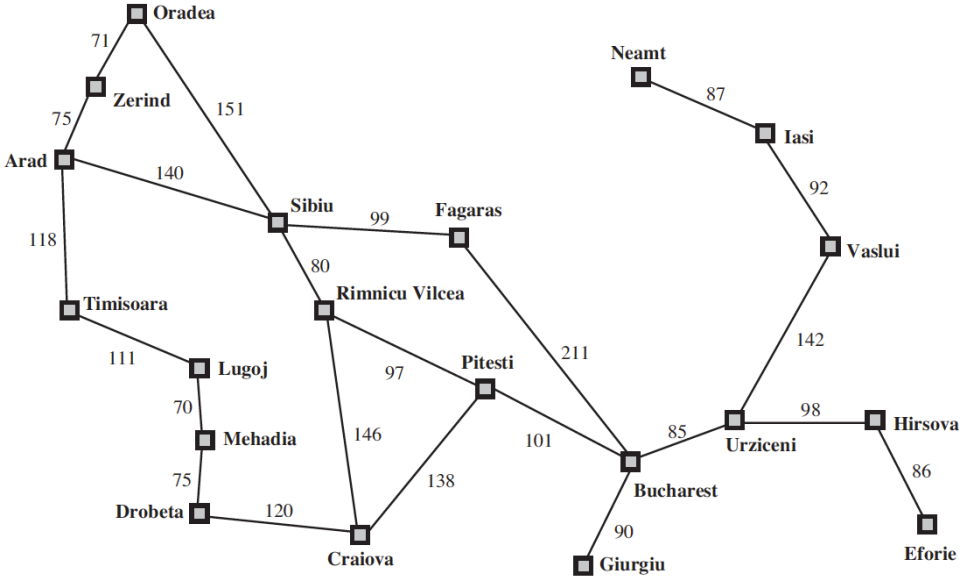
Introduction to Planning

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!

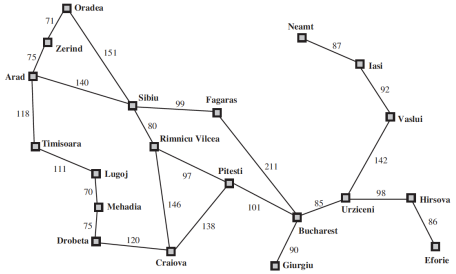




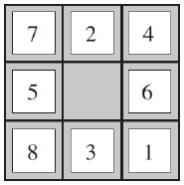
*Using **knowledge** about the world, including possible actions and their results, to **decide** what to do and when in order to achieve an **objective**, **before** you actually start doing it*

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 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.

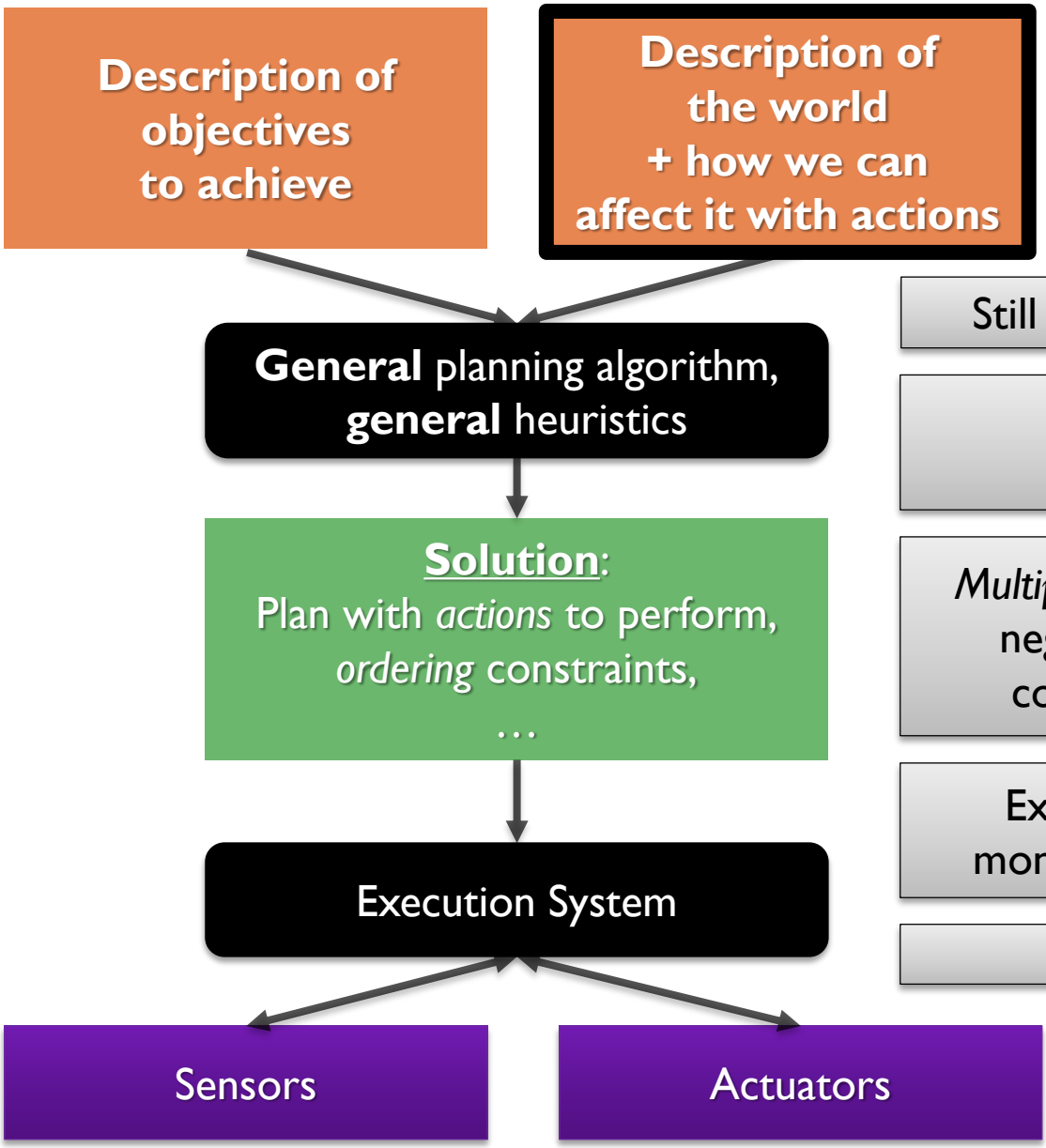
More complex ⟷ **???**

Planning: **<u>General</u> heuristics**	**Planning:** **Entirely different search spaces**
Pattern Databases **Landmarks** *FF, h^m, merge-and-shrink, …*	**Partial Order Causal Link** *SAT planning* *Planning Graphs, …*

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

AI Planning: A Simplified View

jonkv@ida



Still **simplified**, because in reality:

There must be an *agent* that defines the *objectives*

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

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

And so on…

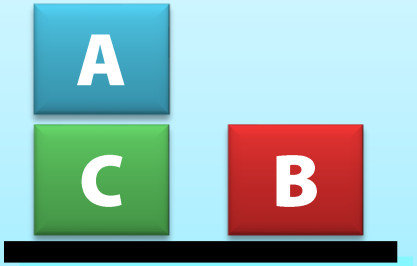
Domains 1: Blocks World

jonkv@ida

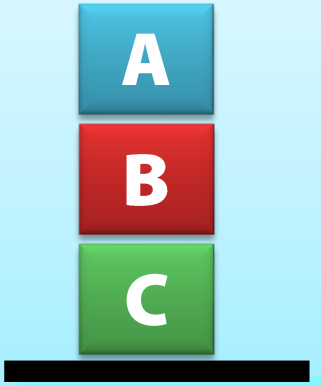
- Must support simple examples such as the **Blocks World**

You

Current state of the world



Your greatest desire



Domains 2: Towers of Hanoi

jonkv@ida

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

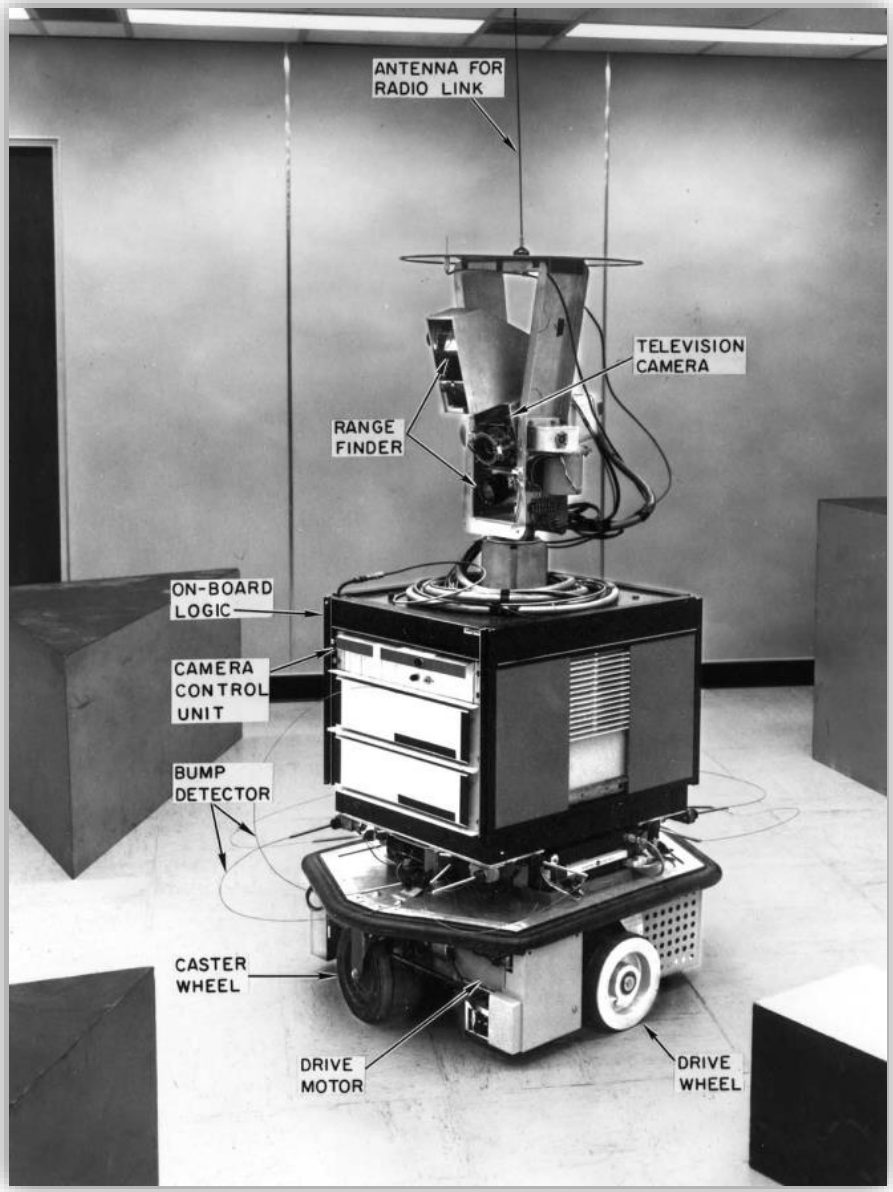
Specific Knowledge and Objective: Problem Instance

- 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

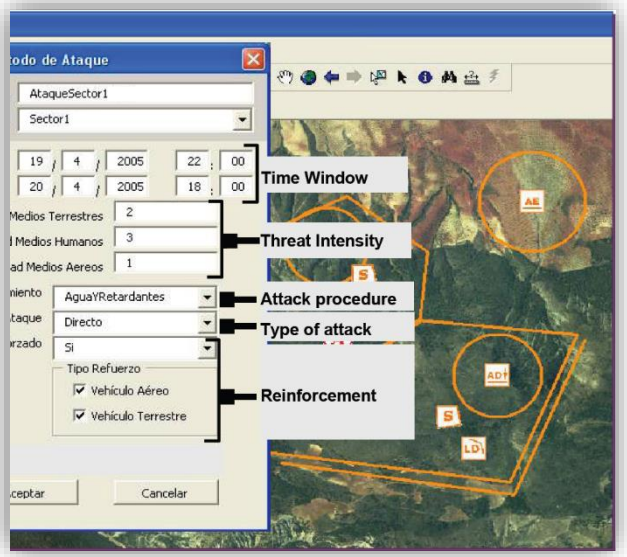
jonkv@ida

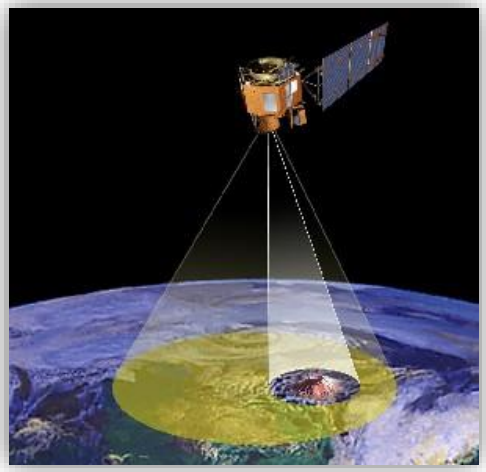
- Classical *robot* example: **Shakey** (1969)
  - Available **actions**:
    - *Moving* to another location
    - *Turning* light switches on and off
    - *Opening* and *closing* doors
    - *Pushing* movable objects around
    - …
  - **Goals**:
    - *Be in room 4 with objects A,B,C*
    - http://www.youtube.com/watch?v=qXdn6ynwpiI



Domains 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)

jonkv@ida

Containers shipped
in and out of a harbor

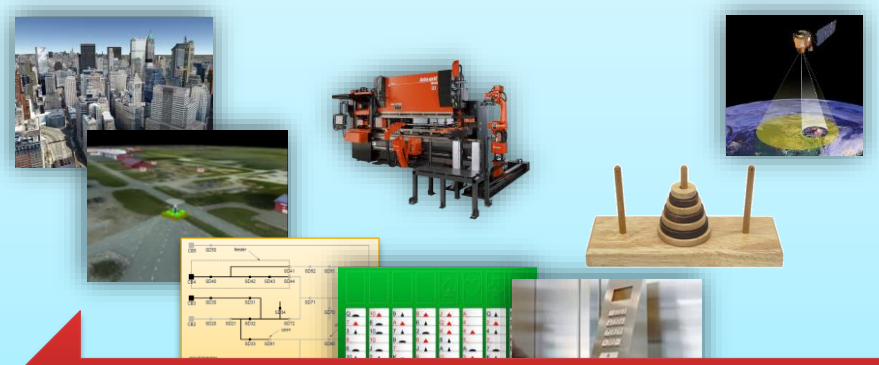
Cranes move containers
between ”piles” and robotic trucks

Classical Planning

jonkv@ida

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

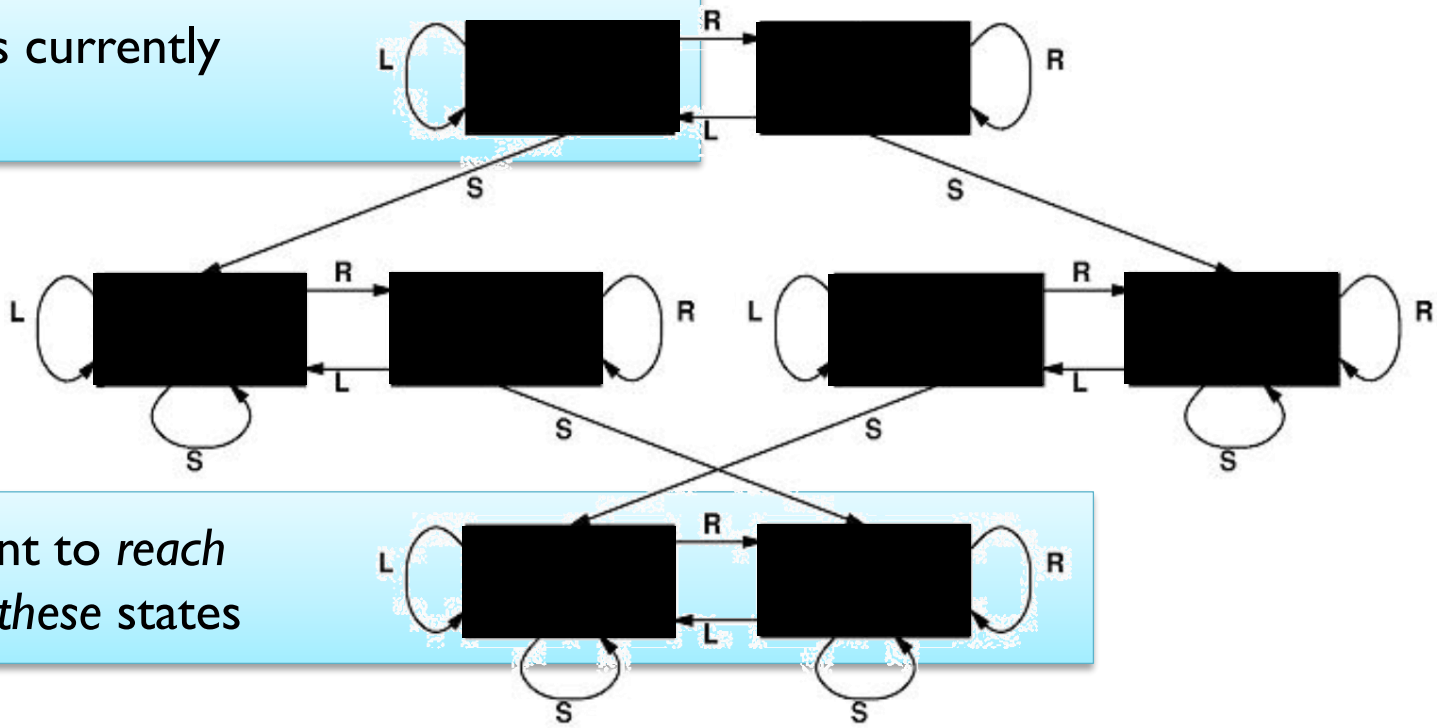
Modeling Classical Planning Problems

Facts and States: Introduction

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?

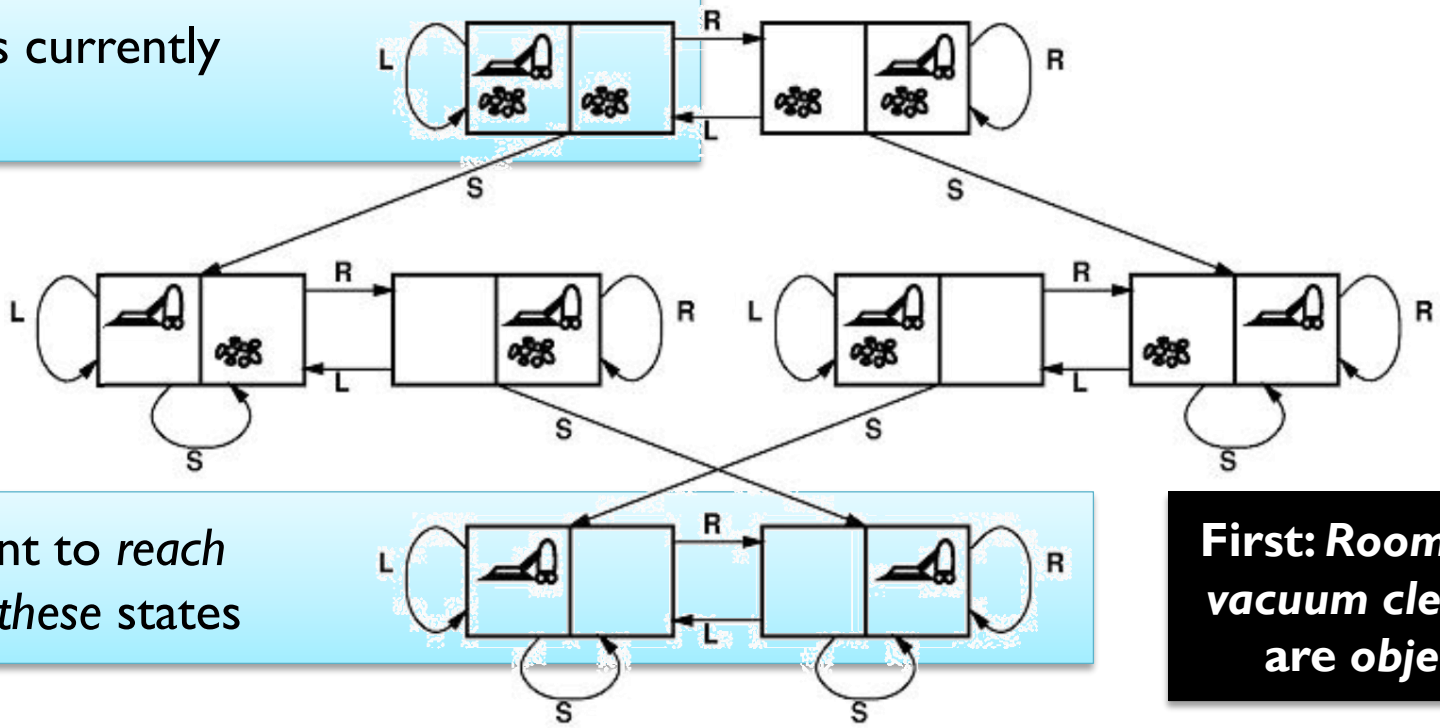
No – Need to ***analyze*** states,
find ***differences*** compared to goal states,
find ***relevant*** actions, …!

Facts and States: Introduction

jonkv@ida

- 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

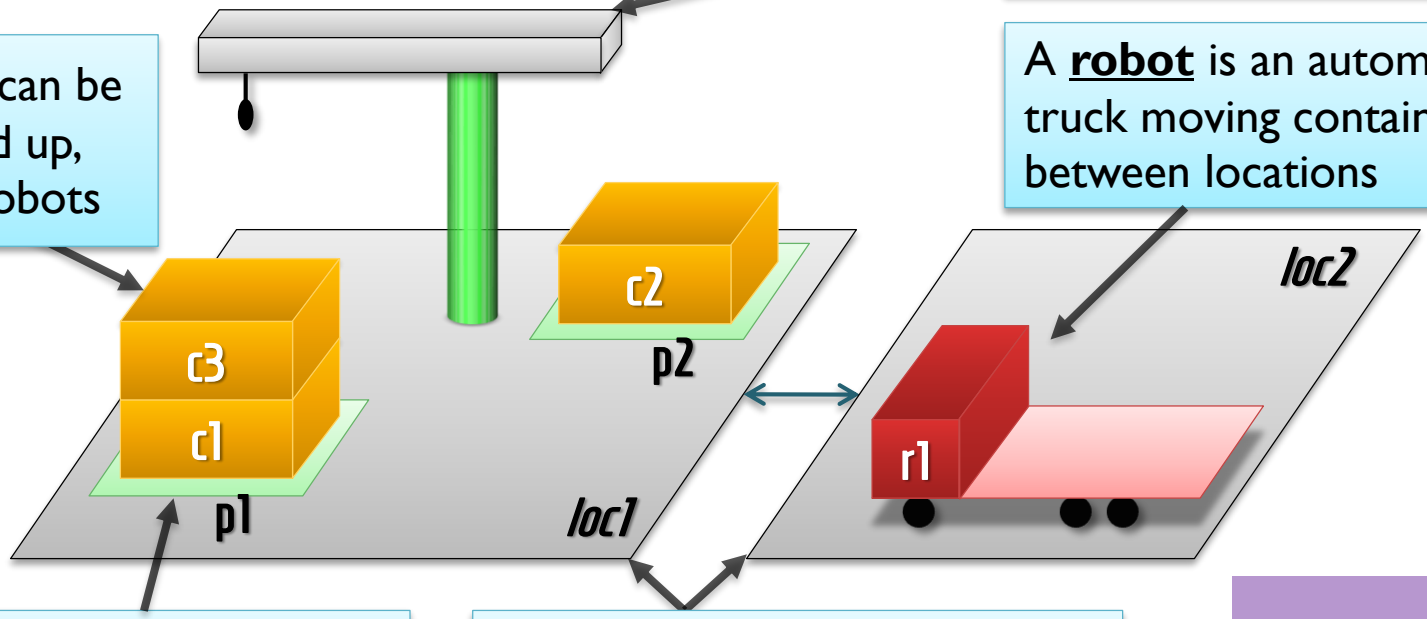
jonkv@ida

- We can often specify **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.

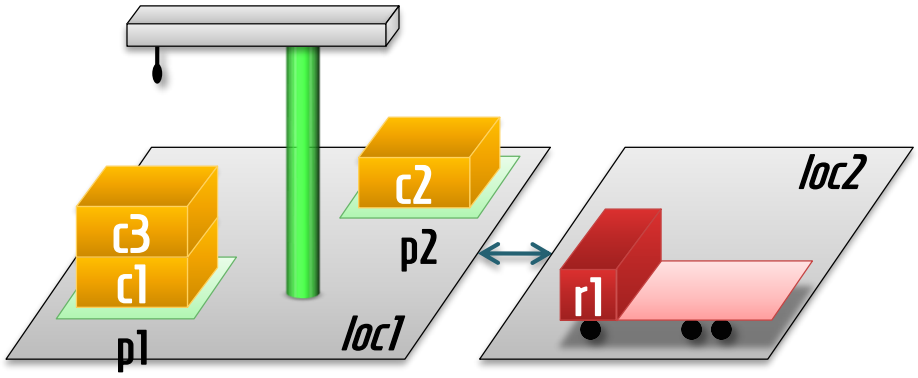
We can skip:
Drivers
Wheels
…

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

Objects 2: Actual objects

jonkv@ida

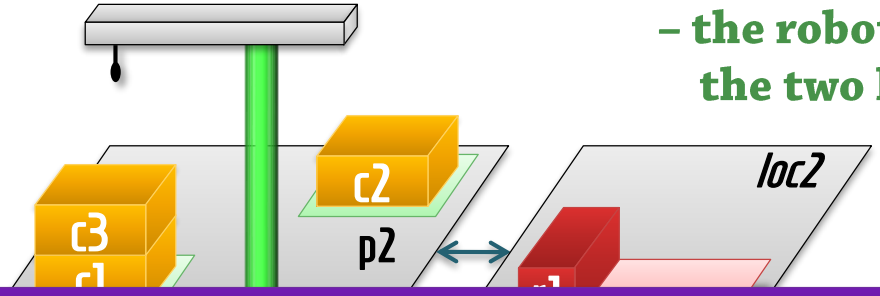
- We then specify sets of **actual objects**
  - **robot**: { r1 }
  - **location**: { loc1, loc2 }
  - **crane**: { k1 }
  - **pile**: { p1, p2 }
  - **container**: { c1, c2, c3, pallet }



Facts

jonkv@ida

- 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

jonkv@ida

- **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

jonkv@ida

- 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!

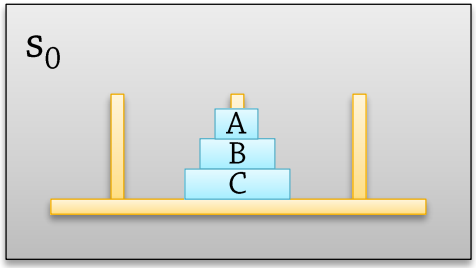
Every assignment of true/false to the ground atoms is a distinct state

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

States 2: Efficient Representation

jonkv@ida

- **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

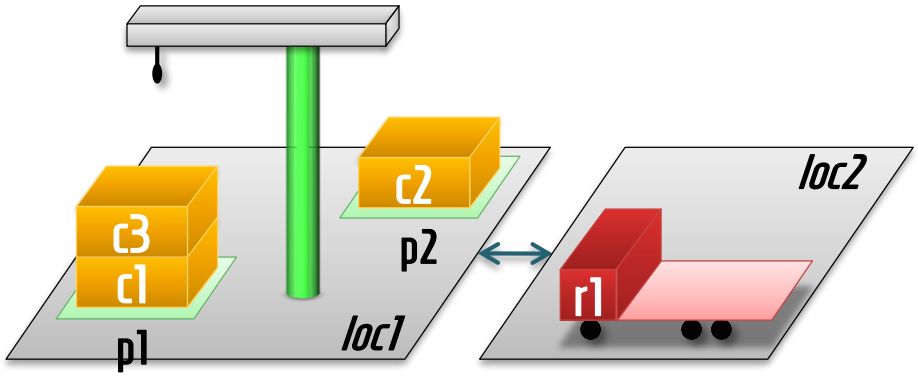
States 3: Initial State

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!

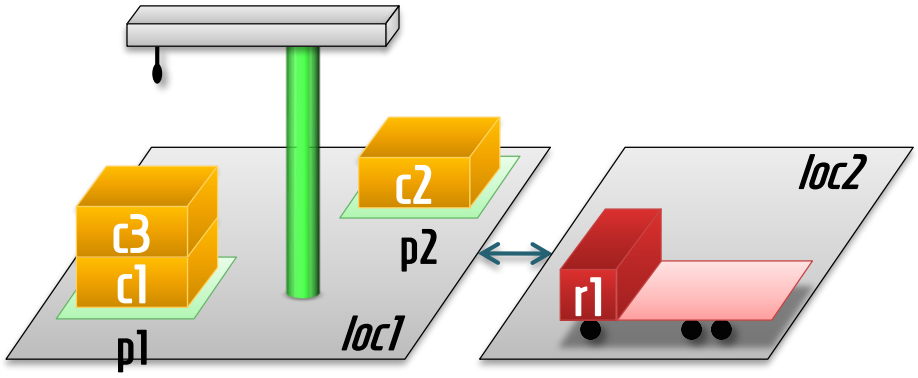
  - So we can use a set of true atoms
    - {
      attached(p1, loc1), in(c1, p1), on(c1, pallet), in(c3, p1), on(c3, c1), top(c3, p1),
      attached(p2, loc1), in(c2, p2), on(c2, pallet), top(c2, p2),
      belong(crane1, loc1), empty(crane1),
      at(r1, loc2), unloaded(r1), occupied(loc2) ,
    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



States 4: Goal States

jonkv@ida

- 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) }

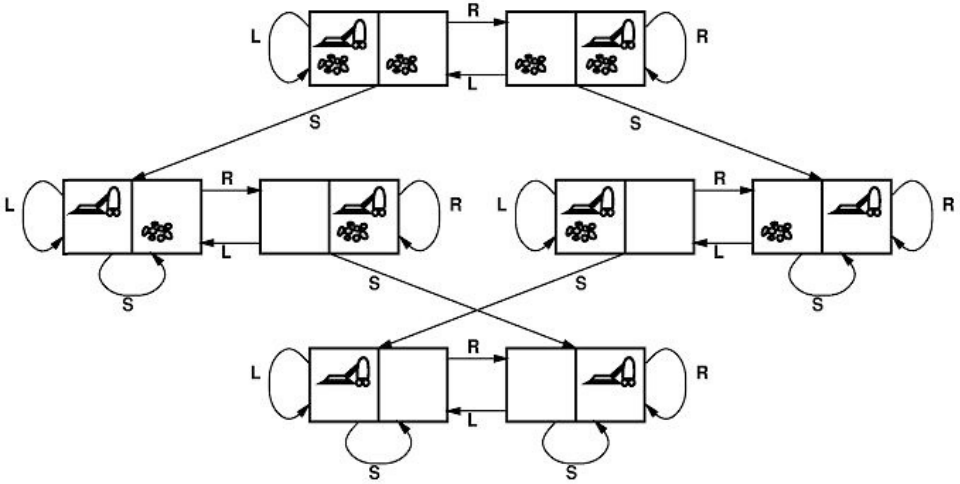


Actions 1: Intro

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

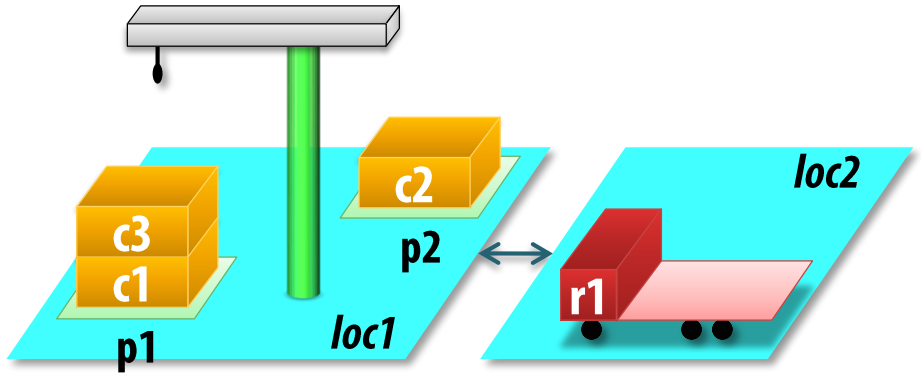
Actions 2: Operators

jonkv@ida

- 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



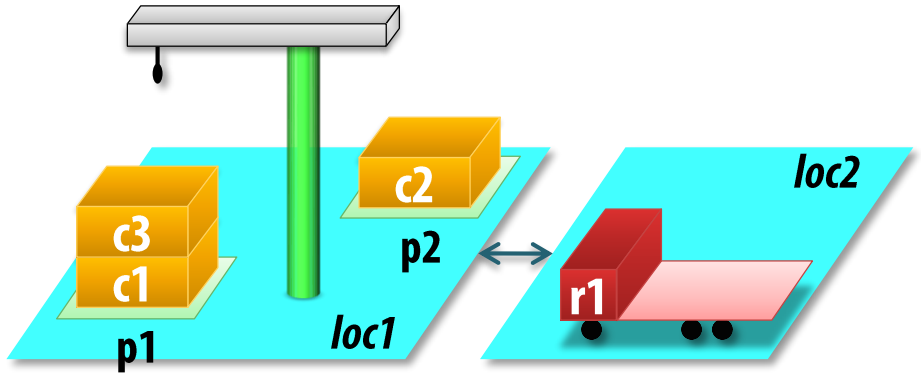
Actions 3: Instances

jonkv@ida

- The planner **instantiates** these schemas
  - Applies them to 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

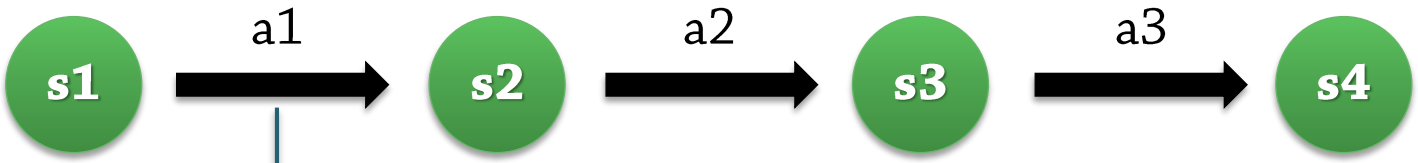
jonkv@ida

- 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

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?)

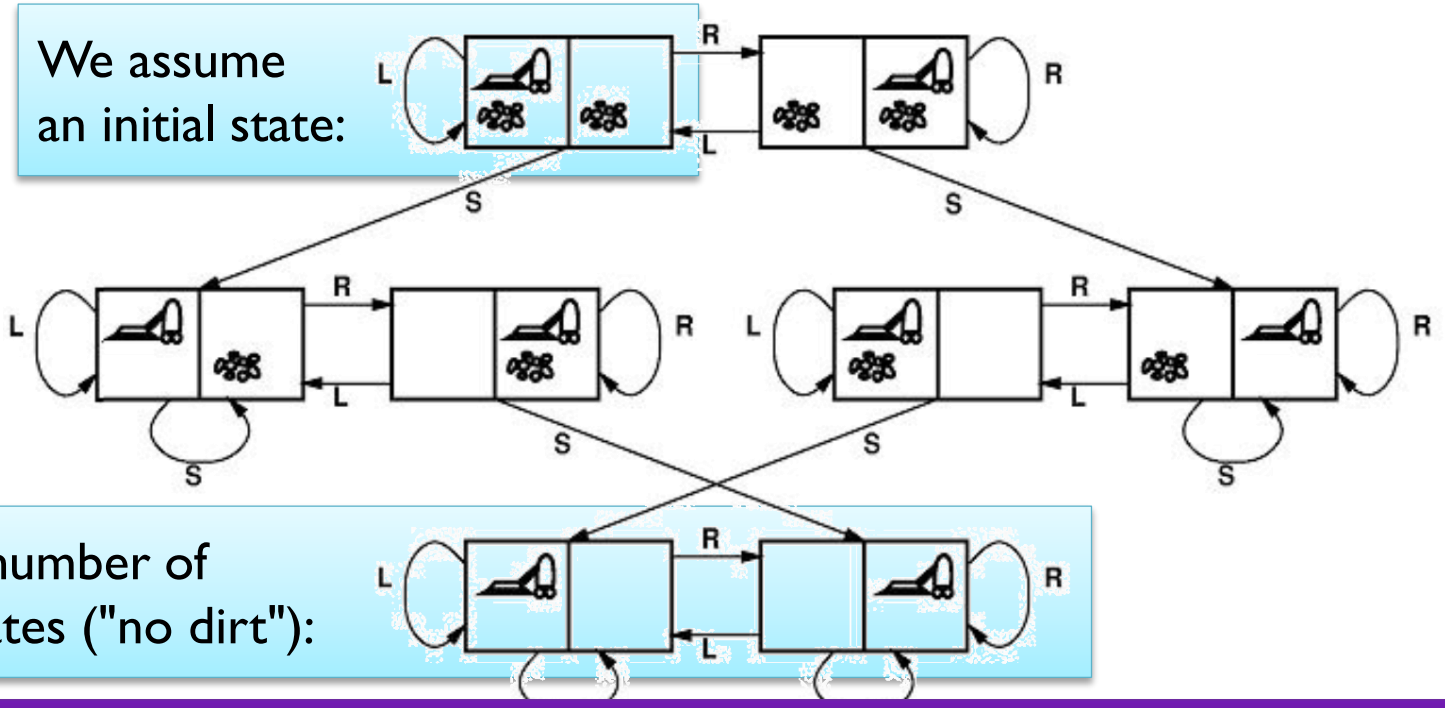
The State Space

State Spaces 1: Introduction

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)

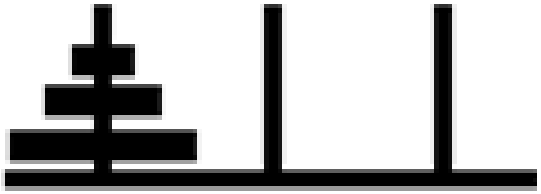


Example solutions: SRS, RSLS, LRLRLSSSRLRS, …

State Spaces 2: Our Intuitions

jonkv@ida

- Our **intuitions** often identify states that **we** think are:
  - ”Normal”
  - ”Expected”
  - ”Physically possible”

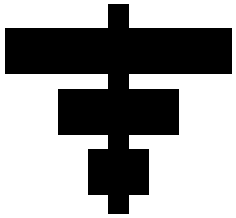


- Usually:
  - The **initial state** is one of those states
  - Mainly need to care about all states **reachable** from there (using the defined actions) – discussed later

State Spaces 3: Against our Intuitions

jonkv@ida

- 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:
    - on(A,A) is true
    - holding(A) and ontable(A) are true at the same time

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

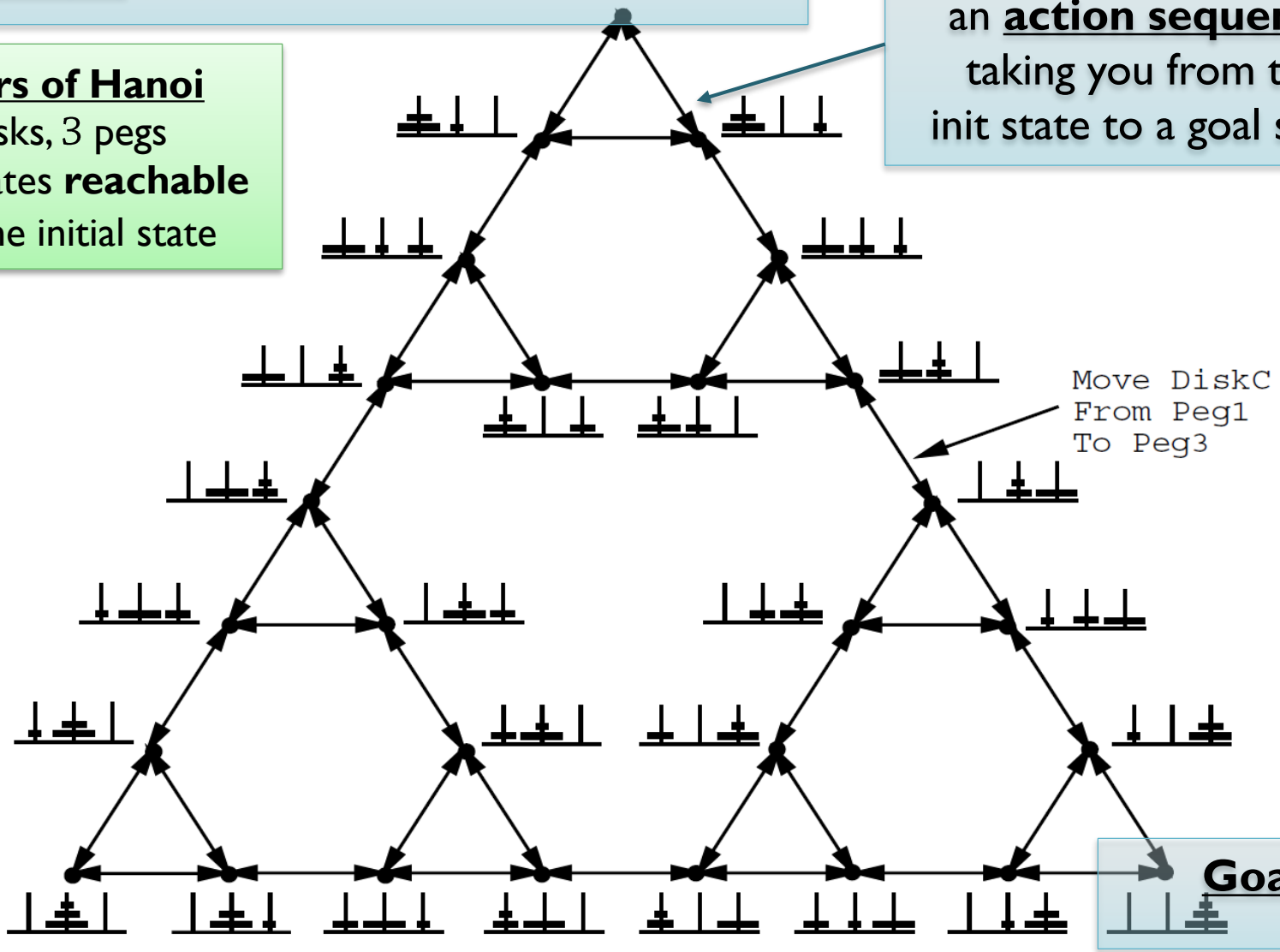
State Spaces 4: ToH, Actions

jonkv@ida

Initial/current state

Towers of Hanoi
3 disks, 3 pegs
➔ 27 states **reachable** from the initial state

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

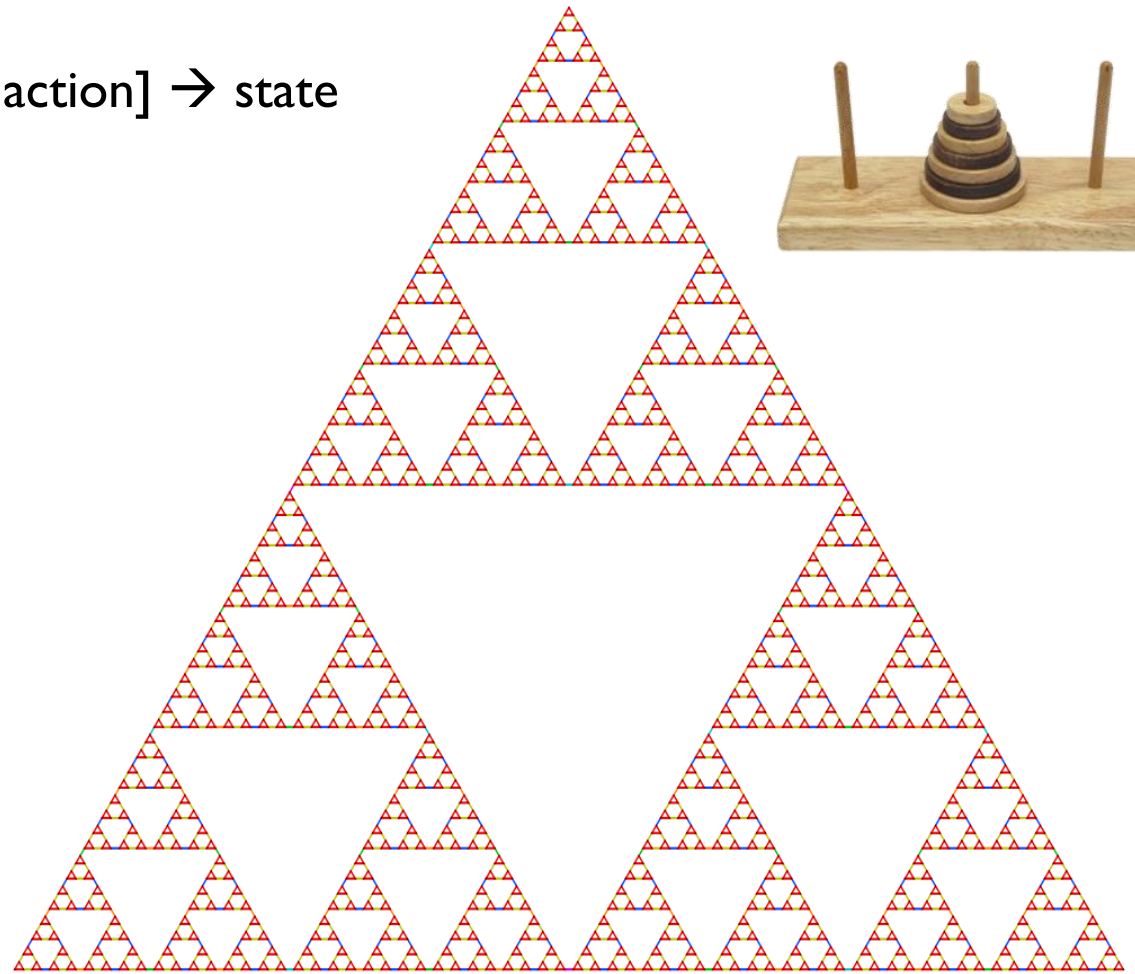


Goal state

State Spaces 4: Larger Example

jonkv@ida

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



State Spaces 5: Blocks World

jonkv@ida

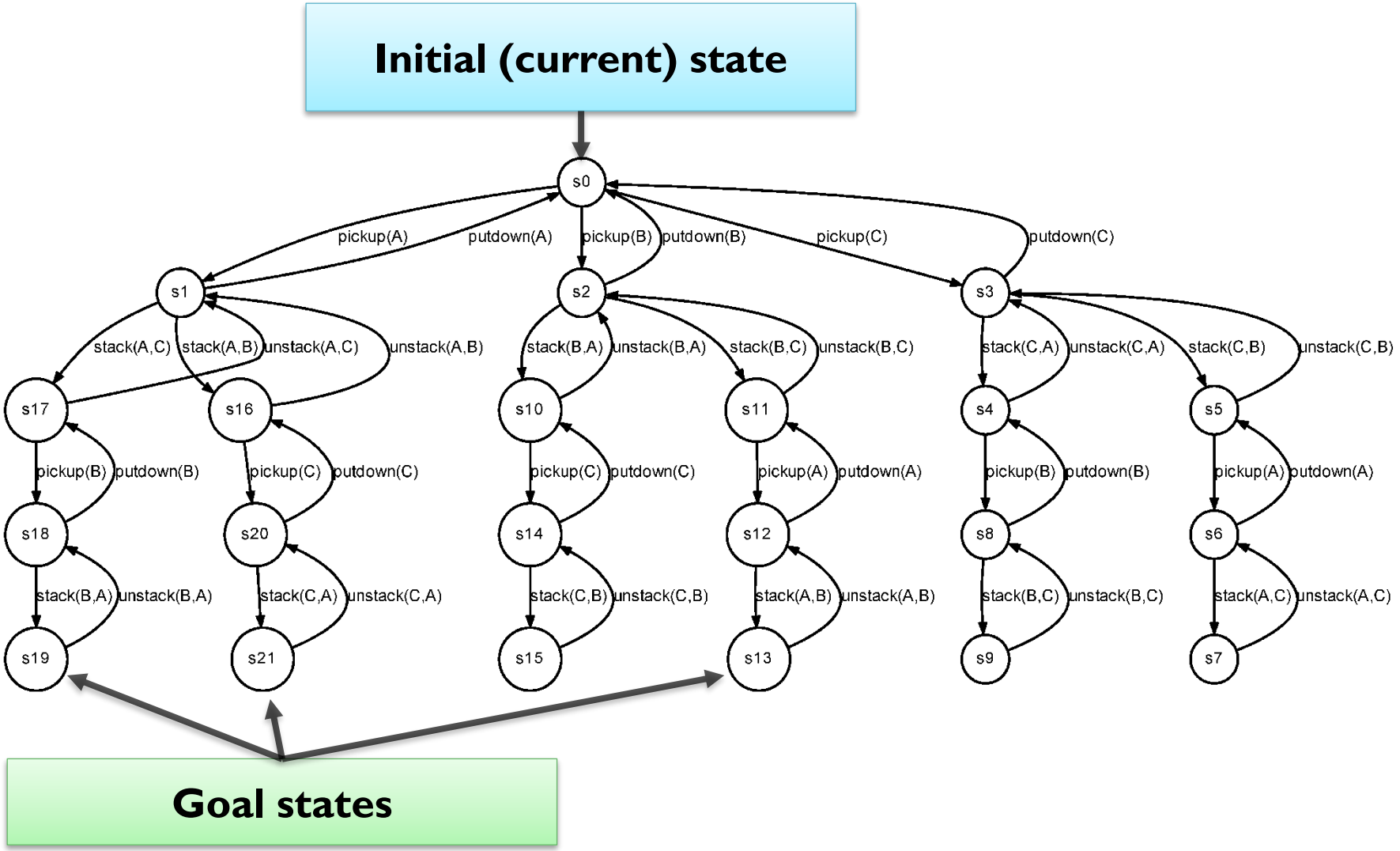
- A common blocks world version, with **4 operators**
  - **pickup**(x) – takes x from the table
  - **putdown**(x) – puts x on the table
  - **unstack**(x, y) – takes x from on top of ?y
  - **stack**(x, y) – puts x on top of y

- 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

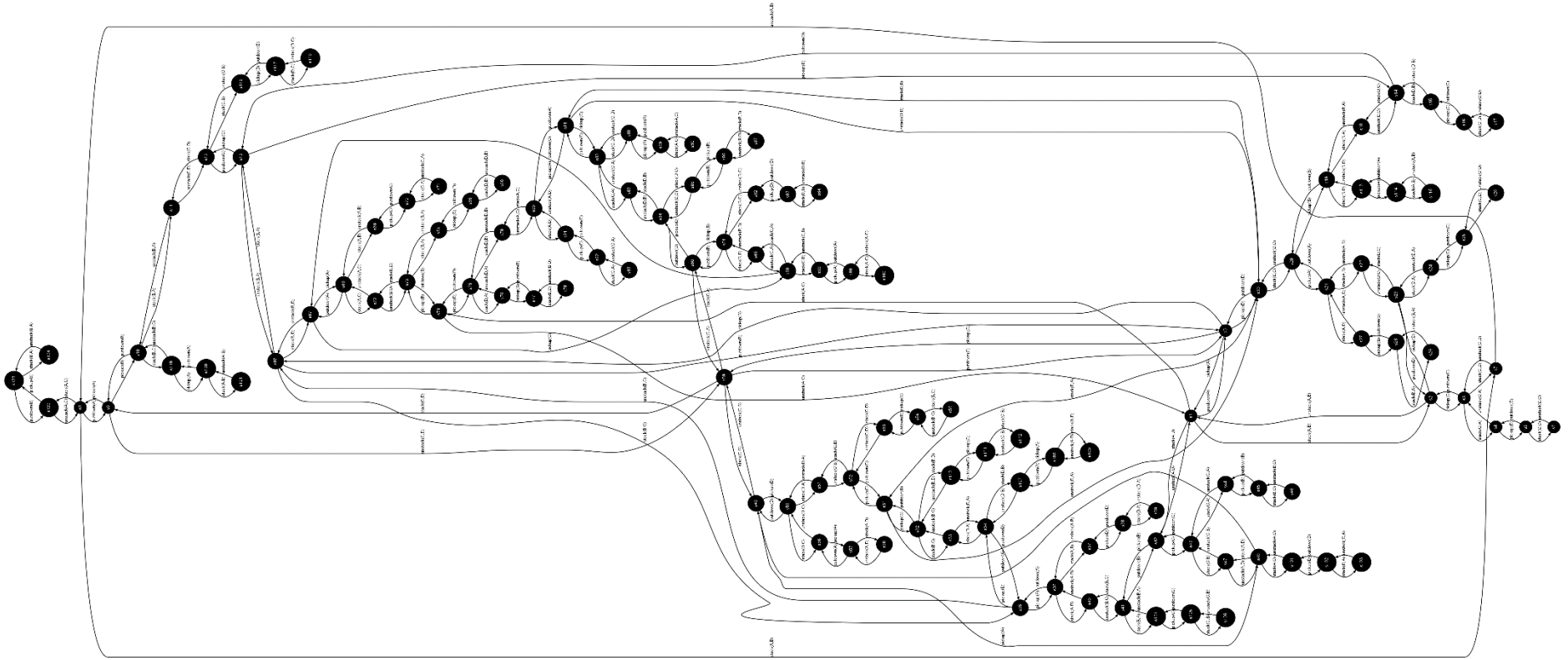
jonkv@ida



State Spaces 7: Blocks World, 4 blocks

jonkv@ida

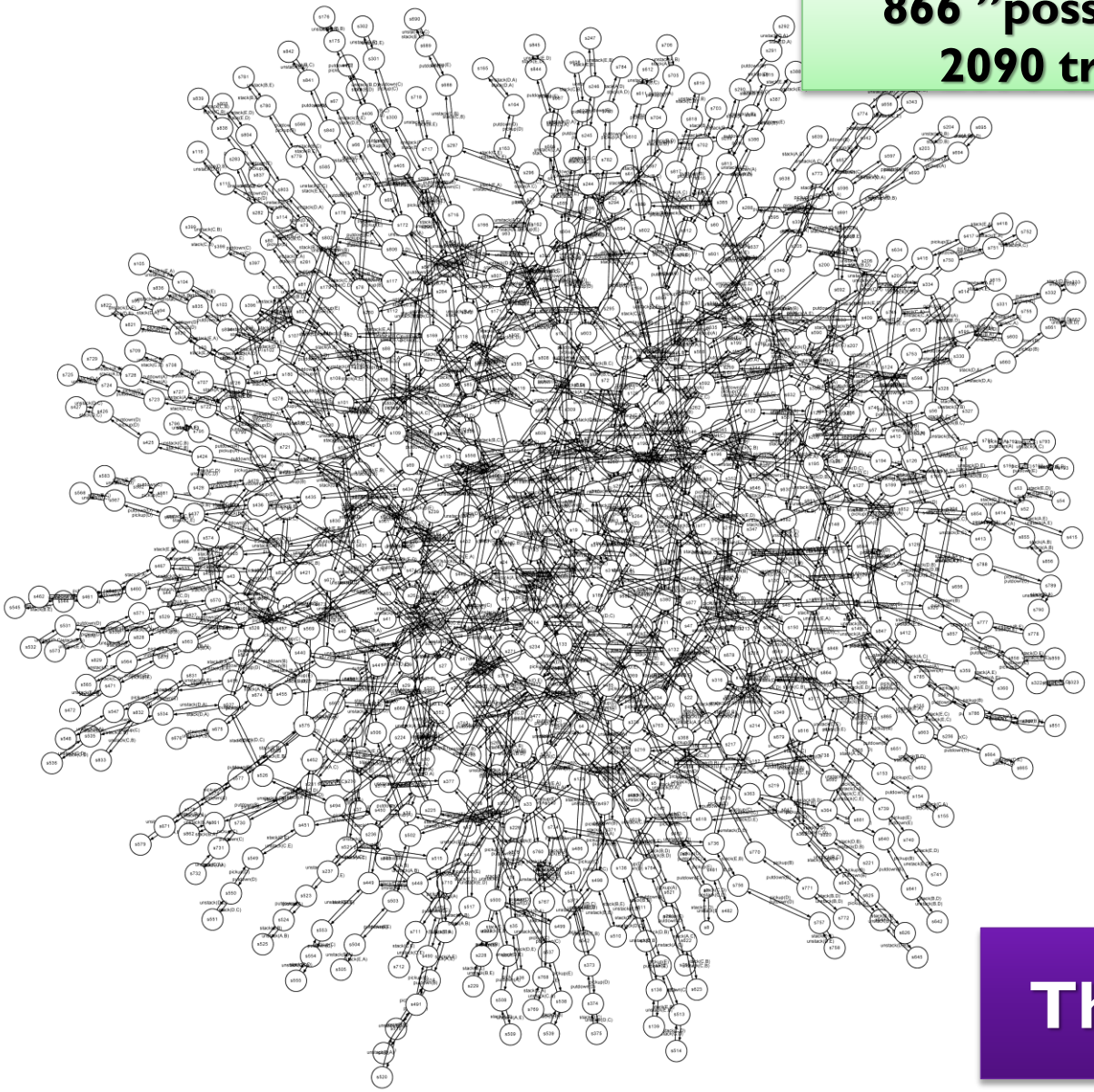
125 ”possible” states
272 transitions



State Spaces 8: Blocks World, 5 blocks

jonkv@ida

866 "possible" states
2090 transitions



This is tiny!

State Spaces 9: Reachable States

jonkv@ida

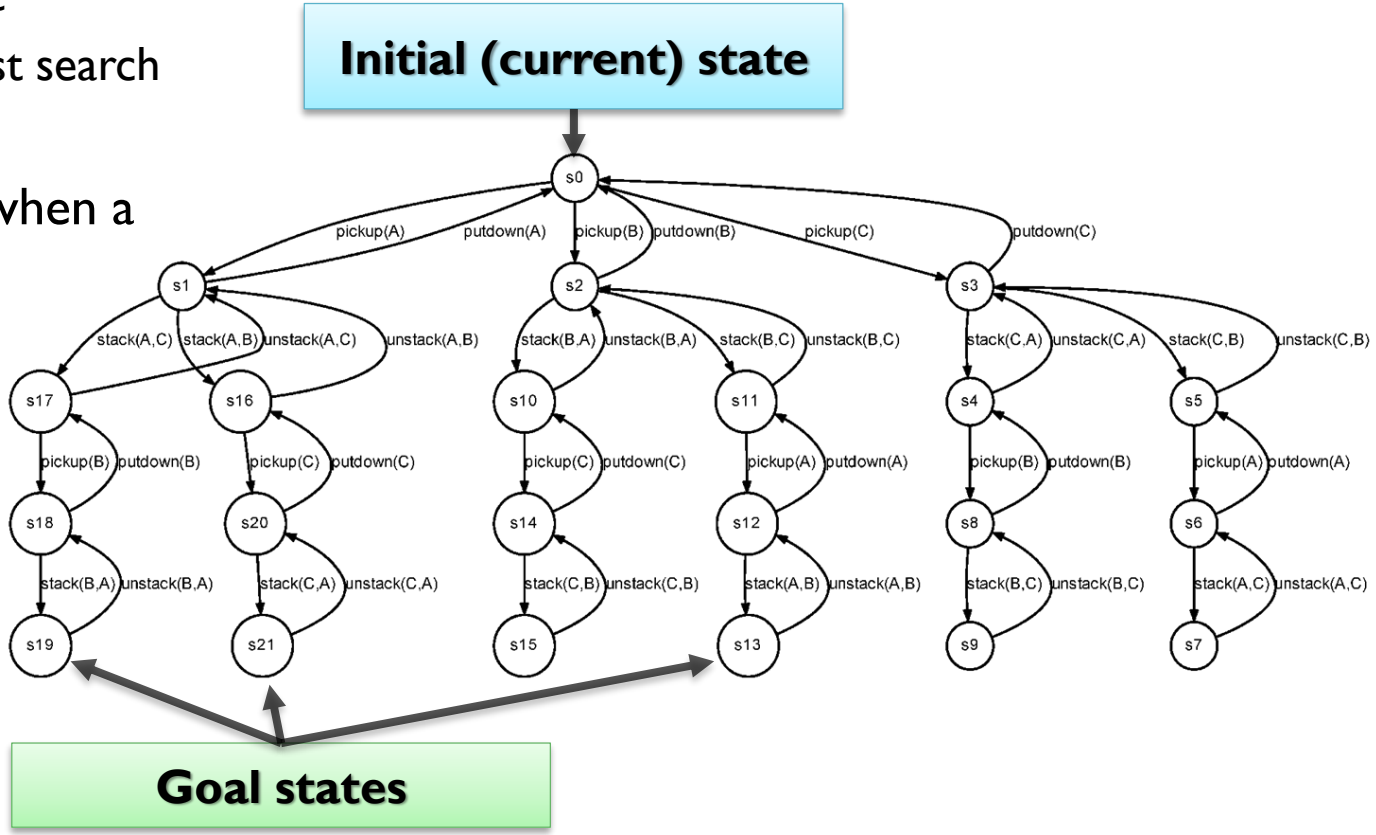
Blocks	States reachable from "all on table"	Transitions (edges) in reachable part
0	1	0
1	2	2
2	5	8
3	22	42
4	125	272
5	866	2090
6	7057	18552
7	65990	186578
8	695417	2094752
9	8145730	25951122
10	…	…
…30	>197987401295571718915006598239796851	

Plan Generation Method 1: Forward State Space Search

Forward Search 1

jonkv@ida

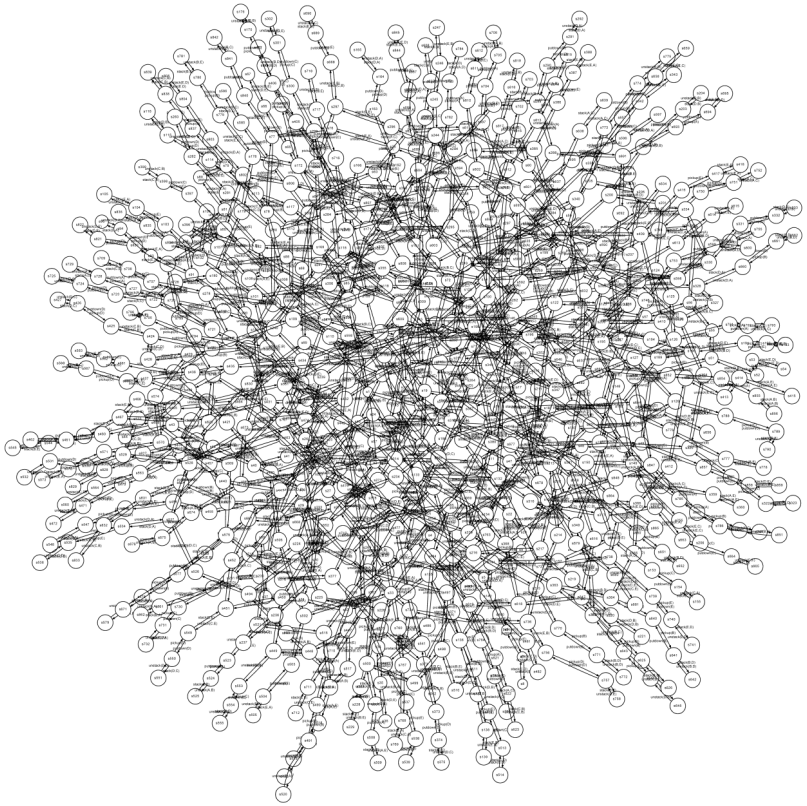
- Straight-forward planning: **Forward search in the state space**
  - **Start** in the initial state
  - Apply a **search** algorithm
    - Depth first
    - Breadth first
    - Uniform-cost search
    - …
  - **Terminate** when a goal state is found



Forward Search 2: Don't Precompute

jonkv@ida

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

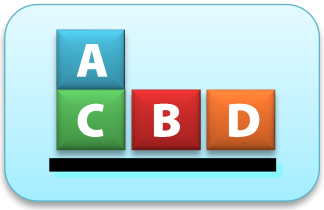


Usually too large!
➔ Generate as we go,
hope we don't actually need the *entire* graph

Forward Search 3: Initial state

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

jonkv@ida

- Given **any search node**…

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

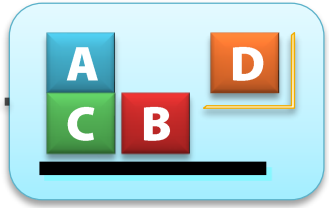
- …we can find **successors** – by applying **actions**!
  - **action** pickup(D)
    - Precondition: **ontable**(D) ∧ **clear**(D) ∧ **handempty**
      Effects: ¬**ontable**(D) ∧ ¬**clear**(D) ∧ ¬**handempty** ∧ **holding**(D)

- 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 }

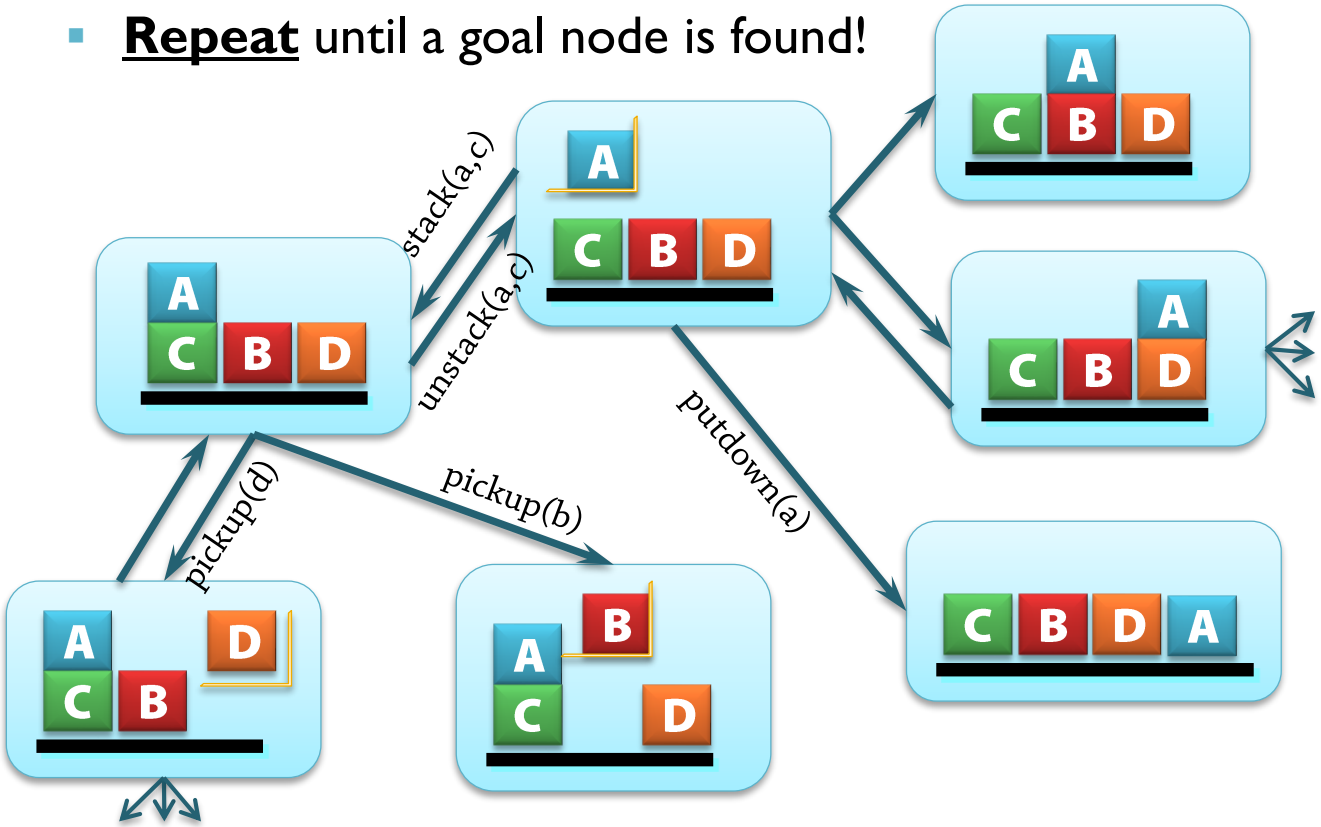
{ clear(A), on(A,C), ontable(C), clear(B), ontable(B), holding(D) }



Forward Search 5: Step by step

jonkv@ida

- 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"

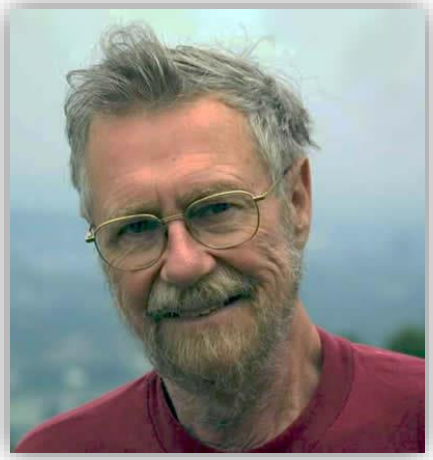
Not true for all domains!

Uninformed Forward State Space Search

Dijkstra

jonkv@ida

- 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!

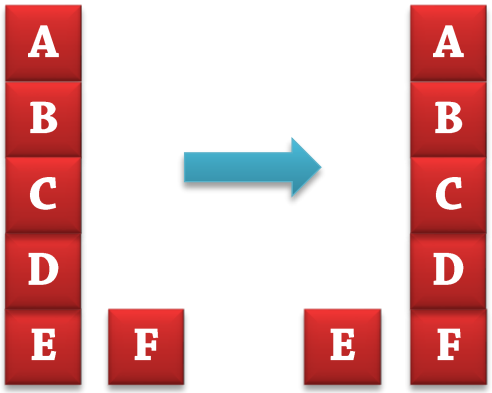


Would this algorithm be a solution?

Dijkstra's Algorithm: Example

jonkv@ida

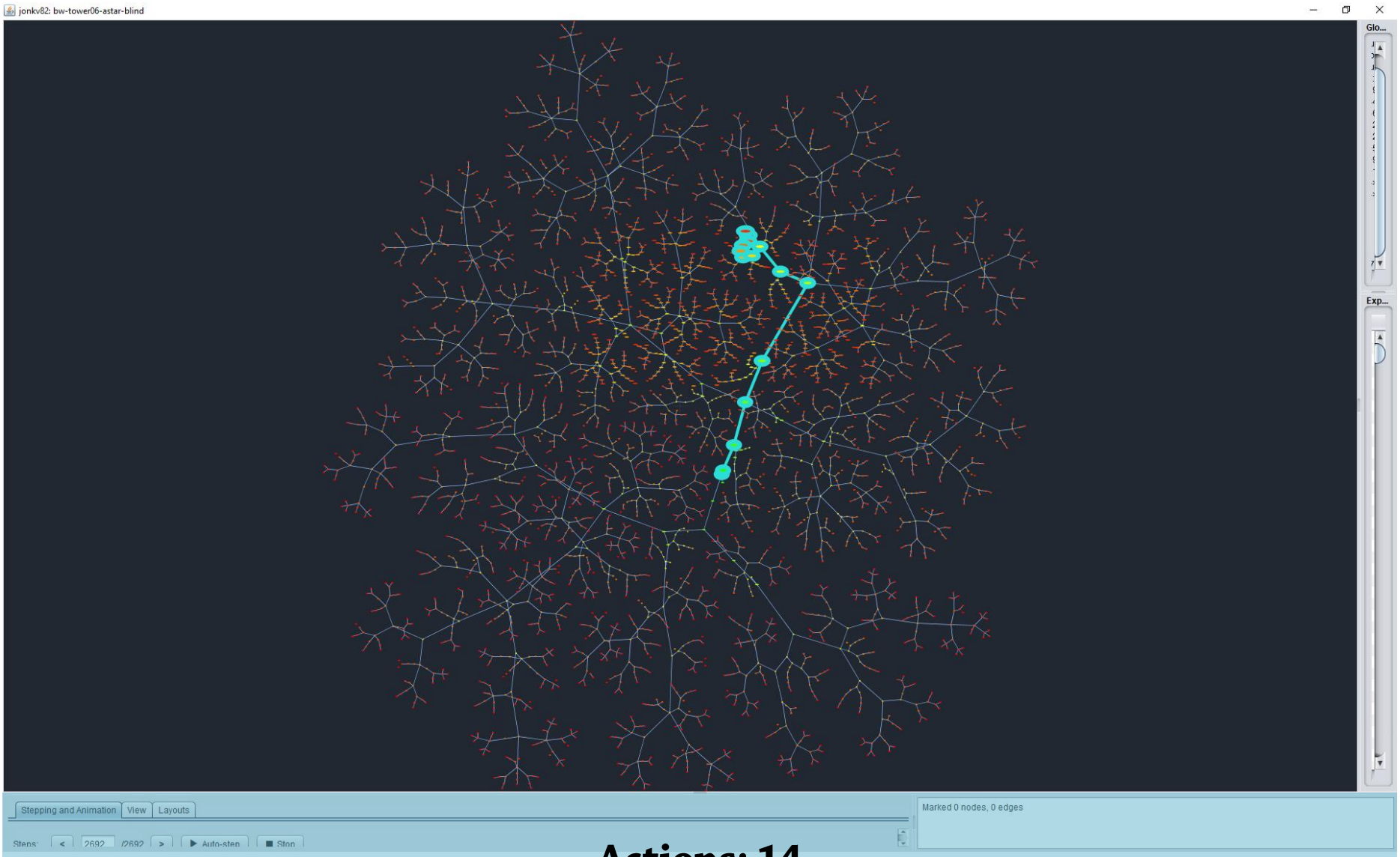
- A simple problem:



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

jonkv@ida

- 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
549738459404452586503693169345[illegible]0912754853265795909113444084441755664211796
274320256992992317773749830375[illegible]4882657444844563187930907779661572990289194
810585217819146476629300233605[illegible]1372350568748665249021991849760646988031691
394386551194171193333144031545[illegible]1302649432305620215568850657684229678385177
725358933986112127352452988035[illegible]3087201742432360729162527387508073225578630
777685901637435541458440833878709344174983977437430327535753441762912244883519172107733387523069568148099086710 9
051332104820413607822206465635272711073906611800376194410428900071013695438359094641682253856394743335678545824
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

jonkv@ida

- 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

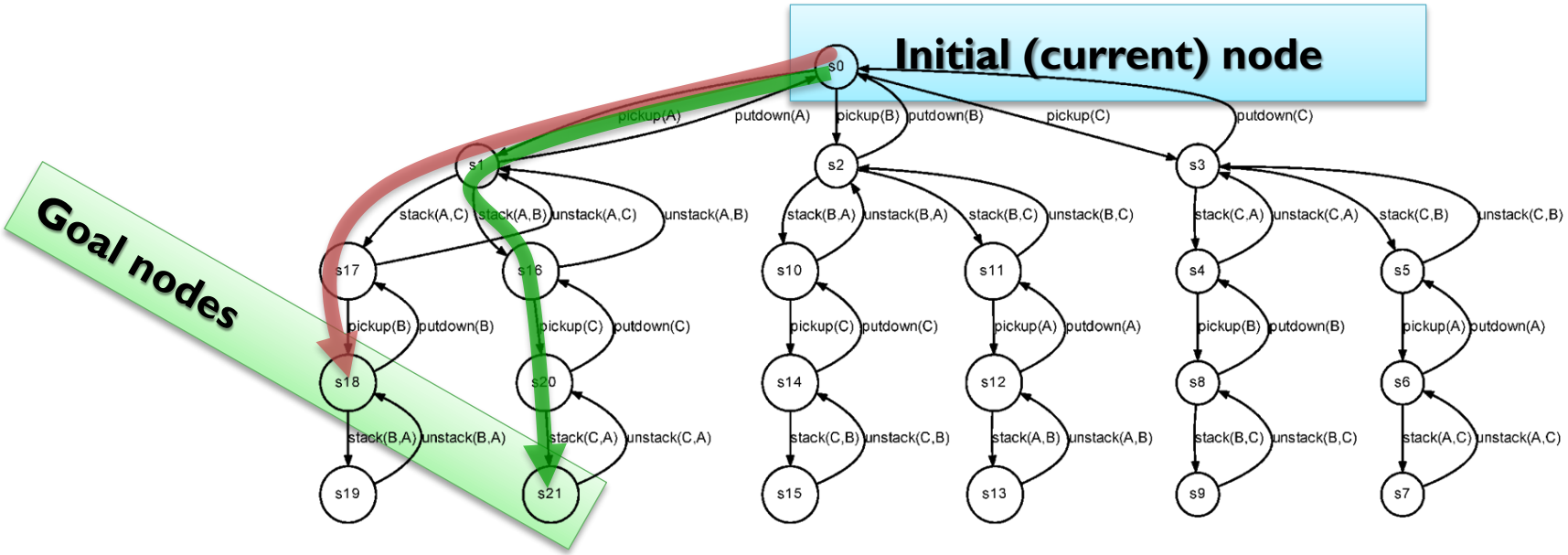
Hopeless? No: We need informed search!

Informed Forward State Space Search

Intro (1)

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!**

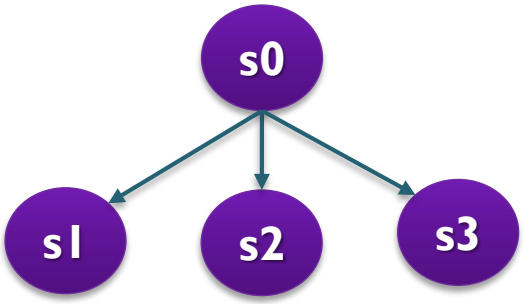


Intro (2)

jonkv@ida

- To **use** the heuristic, we will typically:
  - Pick some state (let's say s0):

  - **Expand** it one step:



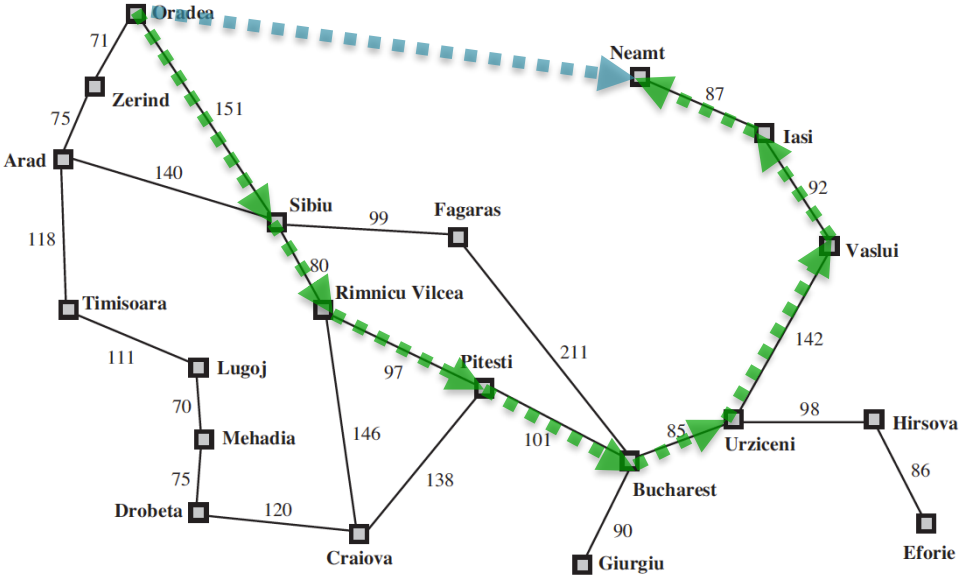
  - **Compute** h(n) for all the expanded nodes
    - h(s1), h(s2), h(s3)
  - Apply some **search strategy** using h(n) as part of its selection criteria, to:

Intro (3)

jonkv@ida

- 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



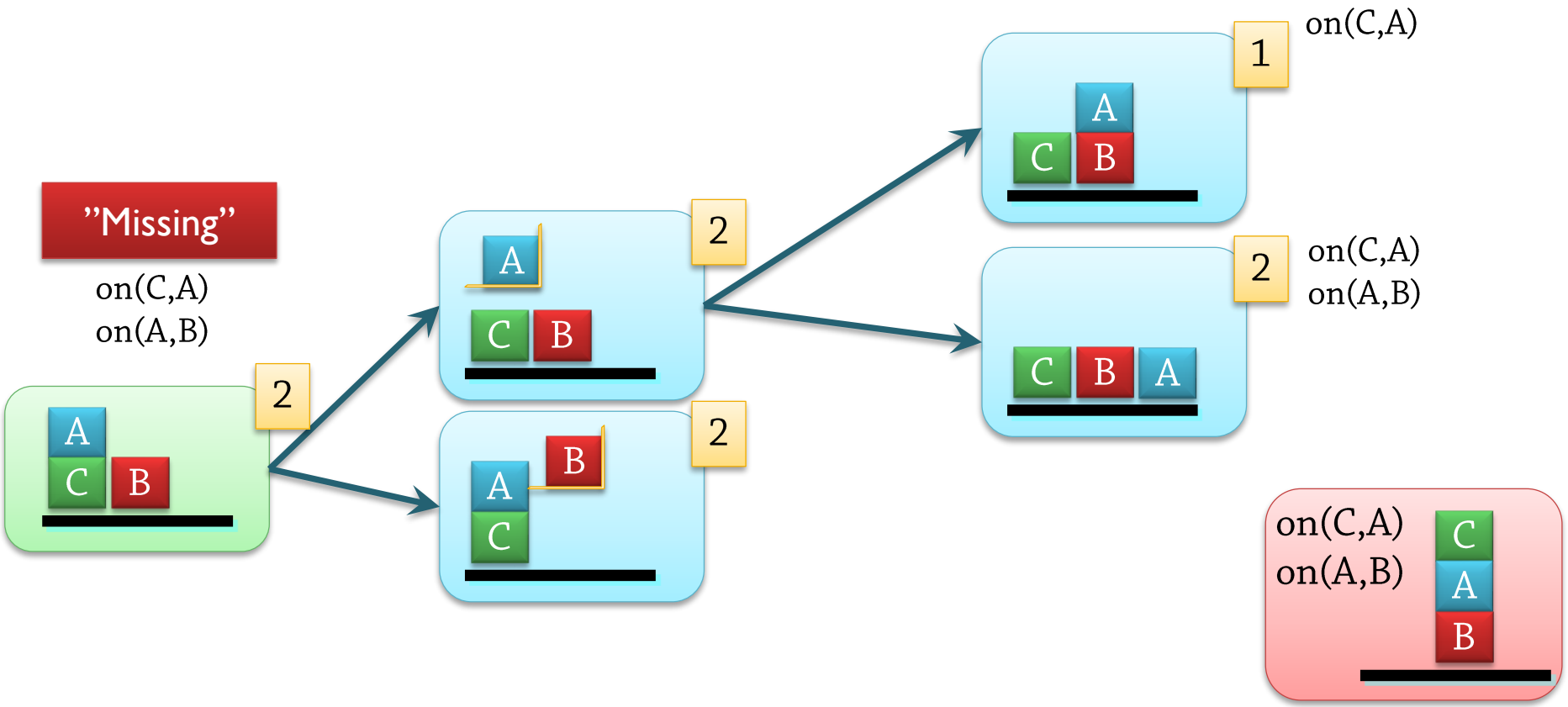
- Now: Want a **general** heuristic function
  - *Without knowing what planning problem is going to be solved!*

Domain-Independent Heuristics (1)

jonkv@ida

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

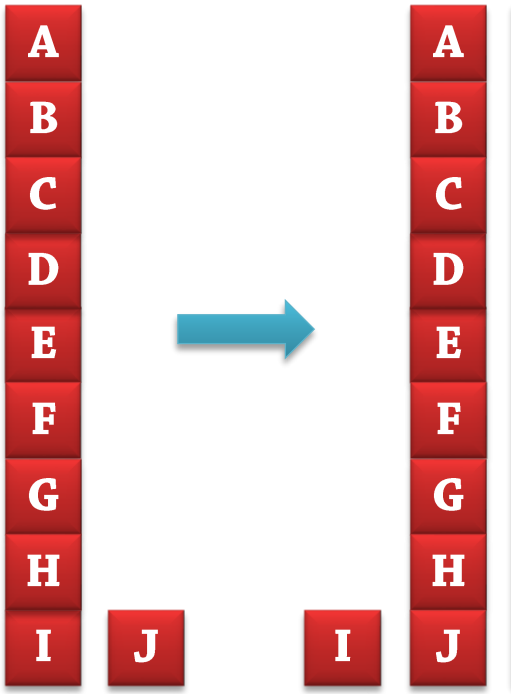
Optimal:
unstack(A,C)
stack(A,B)
pickup(C)
stack(C,A)



Domain-Independent Heuristics (2)

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)

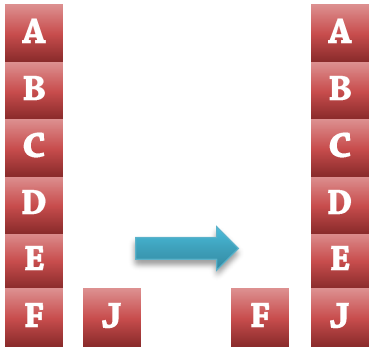
Optimal solution:

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

Only unachieved goal fact: on(H,J)

Must *remove* many "good facts" to reach the goal

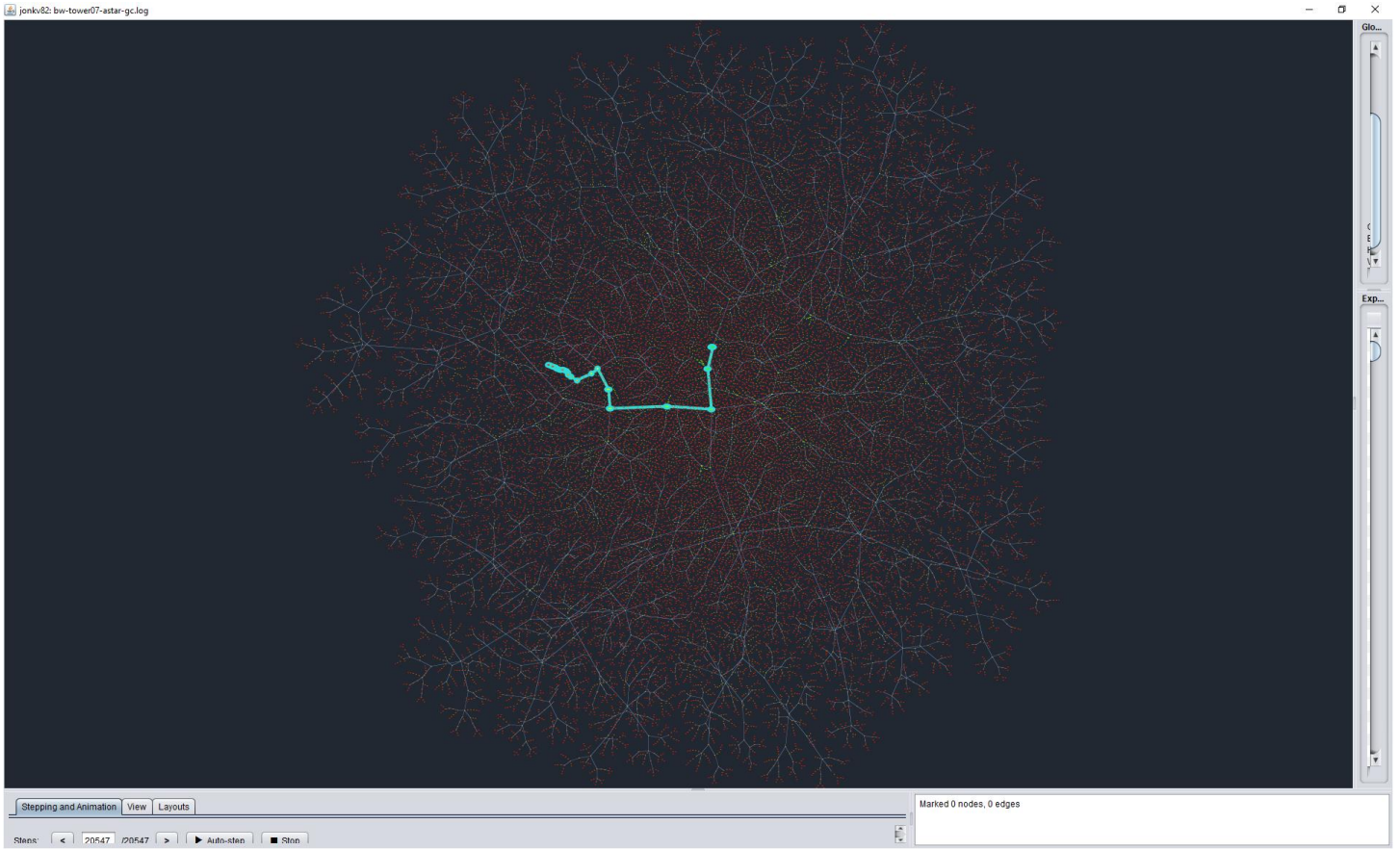
bw-tower07-astar-gc: Only 7 blocks, A* search, based on goal count



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

jonkv@ida

- **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):

Optimal 2: A*

jonkv@ida

- 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	12 = 10 + 2	12 = 12 + 0	12 = 11 + 1	13 = 11 + 2
Pop – not a solution	Pop – not a solution	Pop – solution!	Ignore: g is *known*, h is an *underestimate*, so solutions found by expanding these nodes will cost ≥ g+h (and we *have* one of cost ≤ g+h)	

Creating Admissible Heuristic Functions: The General Relaxation Principle

How does relaxation apply to **planning**?

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

The Problem

jonkv@ida

- **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

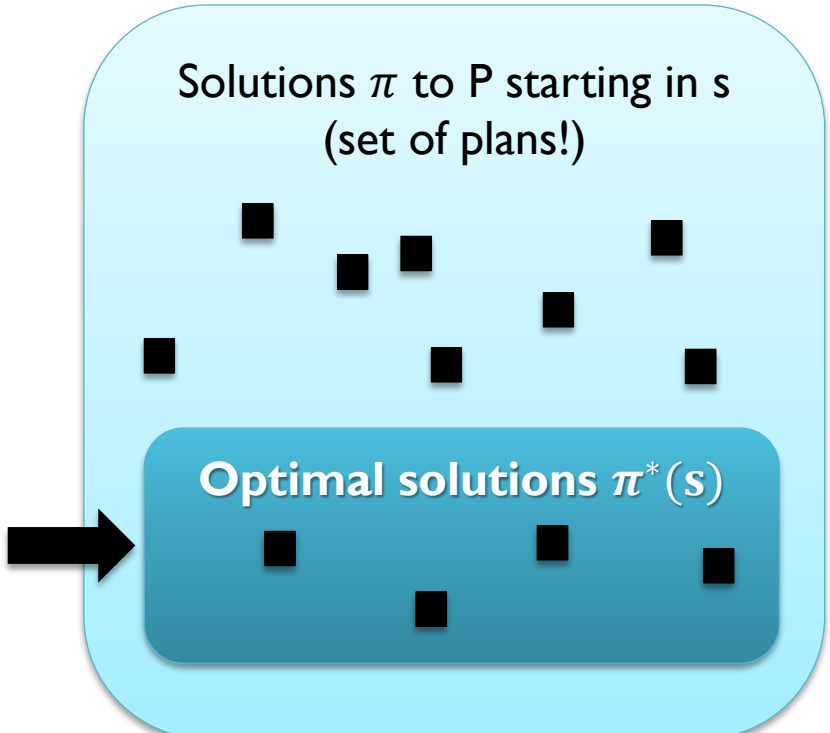
jonkv@ida

- **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)



Transforming a Problem

jonkv@ida

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

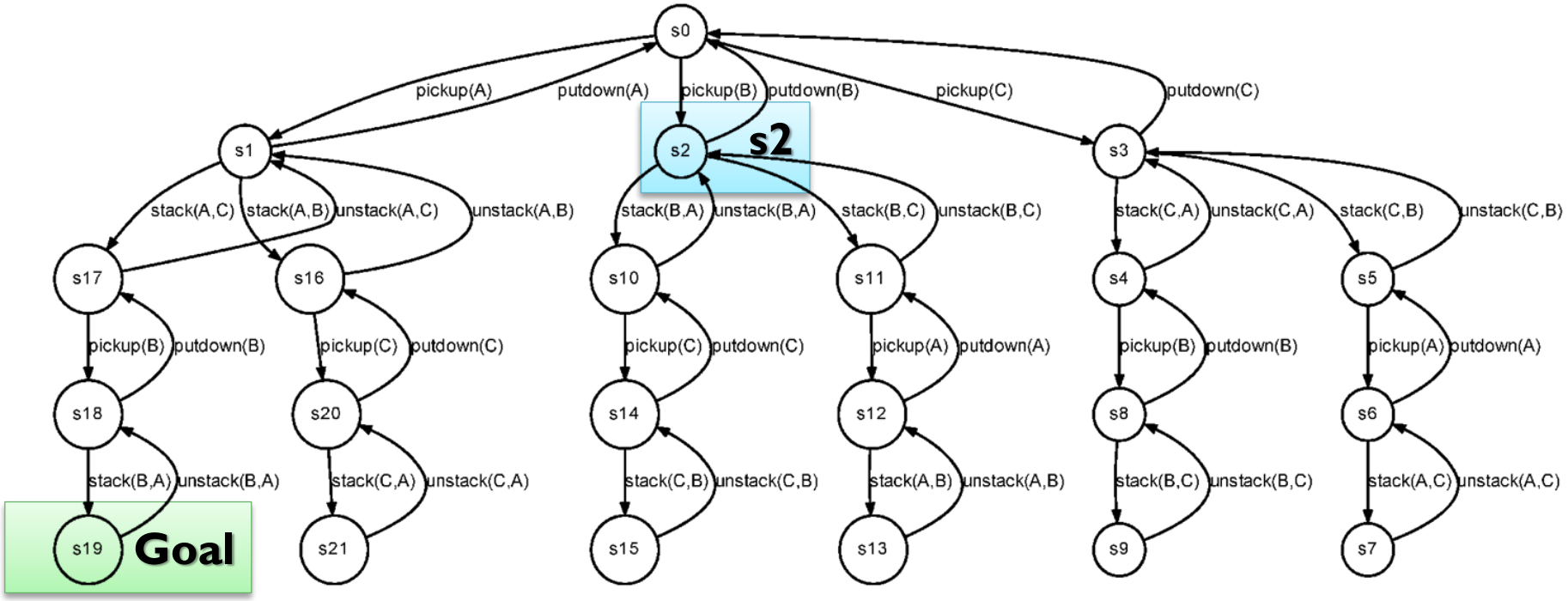
Original problem P, current state s (finding a solution: slow)

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

Simplification Example: Basis

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)

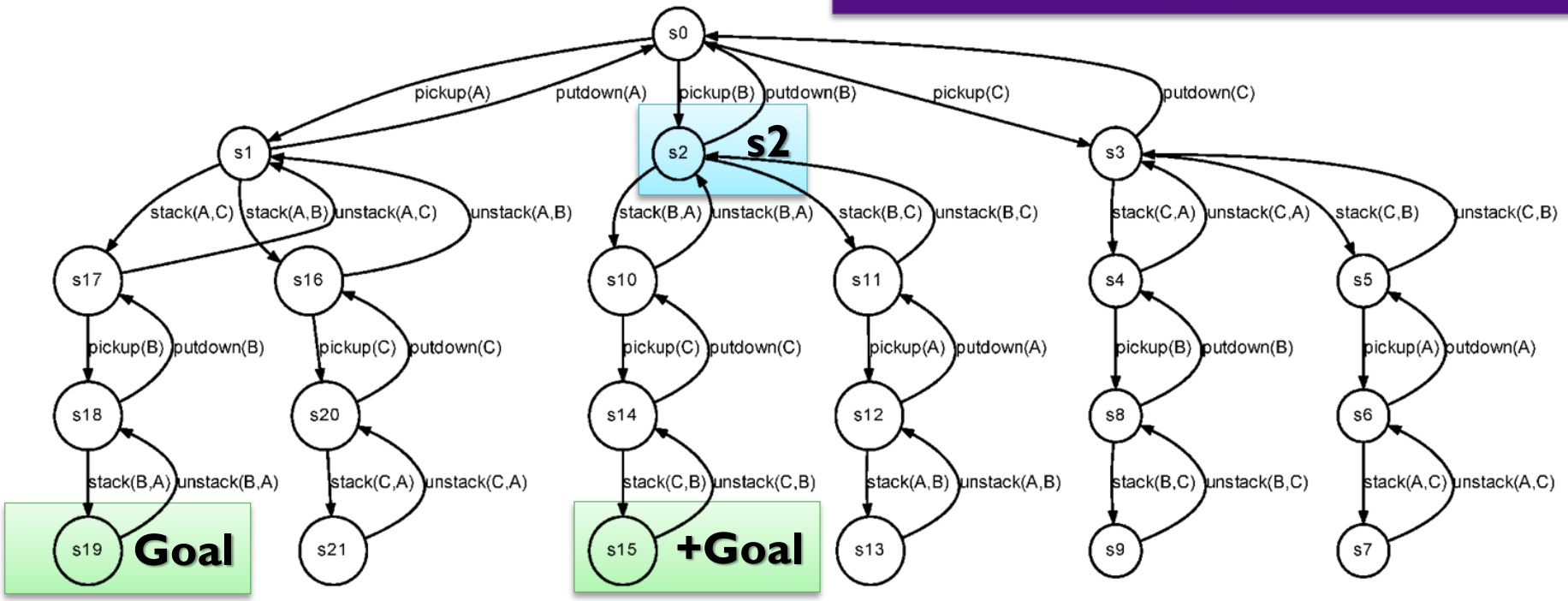


Simplification Example 1

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 s2, on average

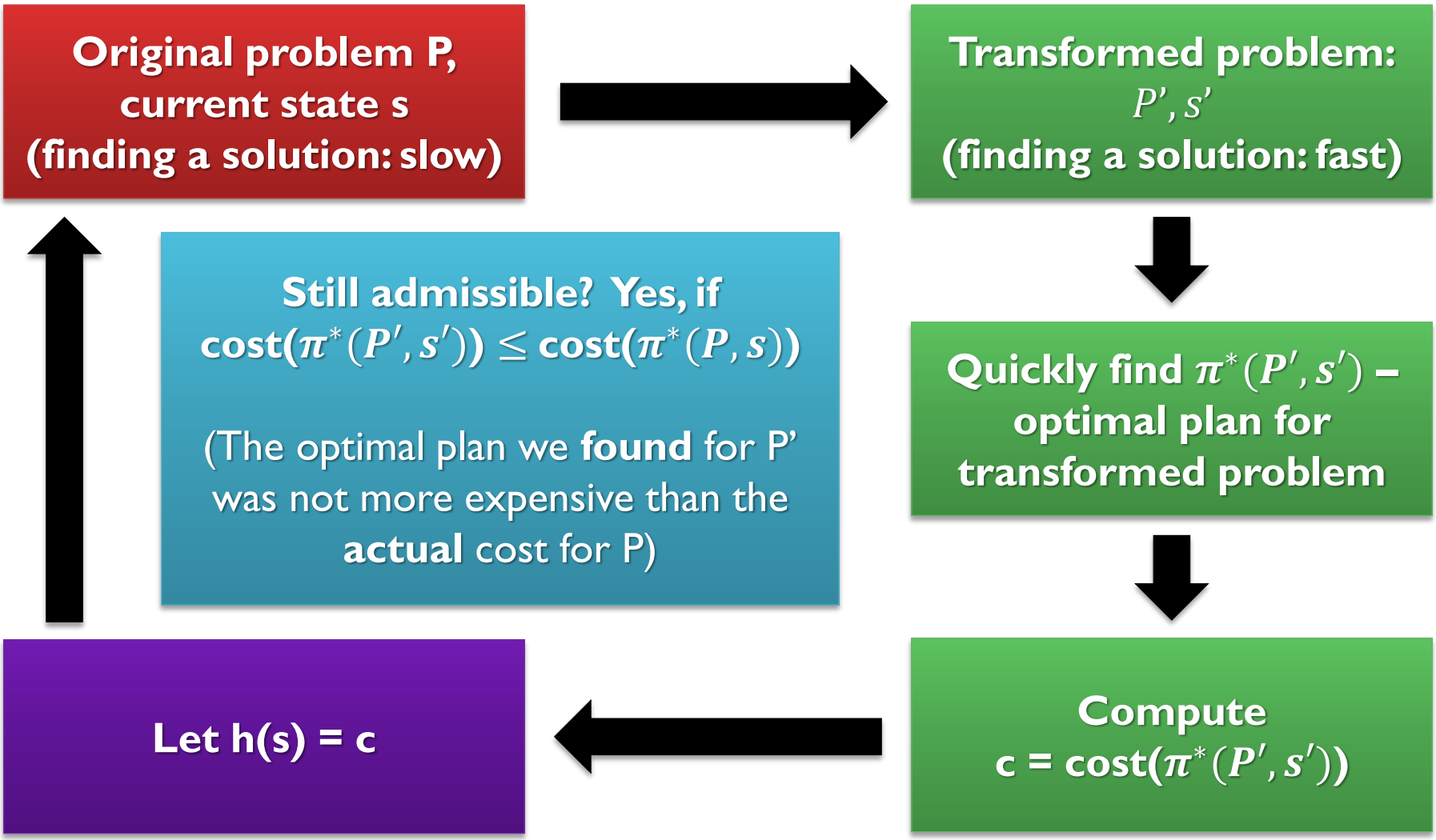
May need more transformations: Just showing a general idea!



Computing a Heuristic Value

jonkv@ida

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

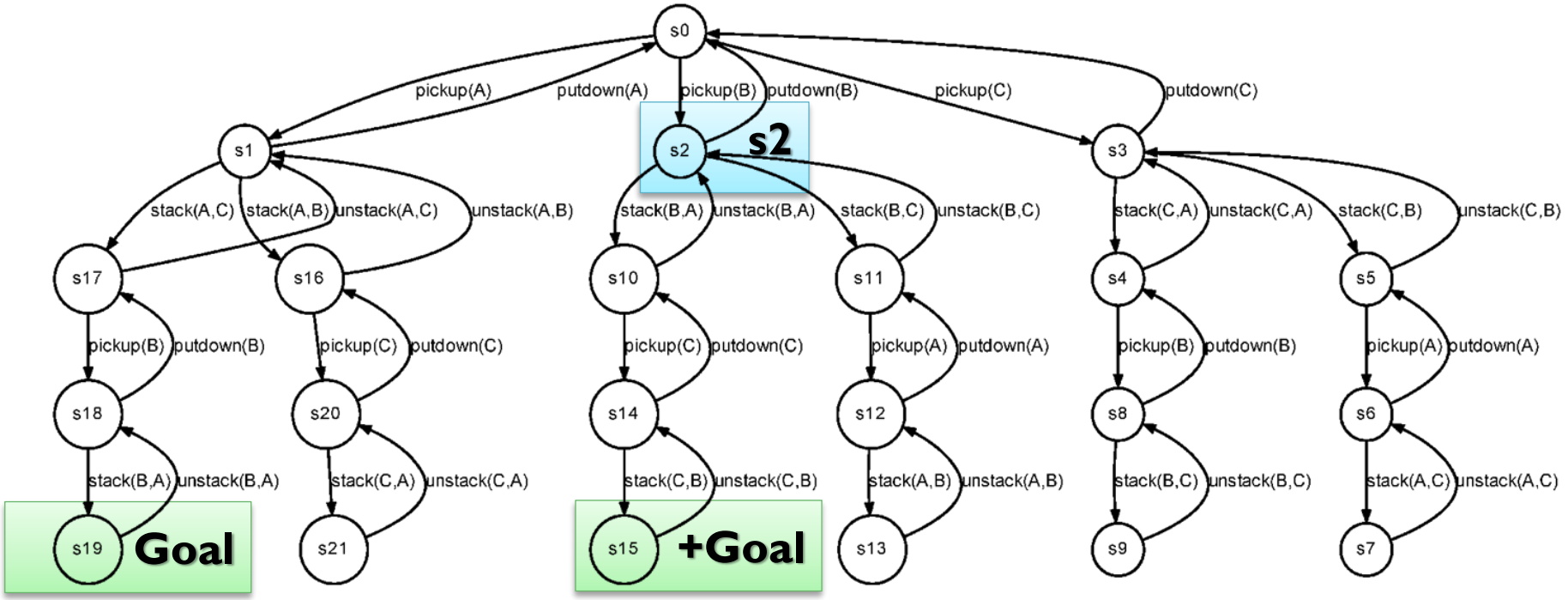


Simplification Example 1

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?

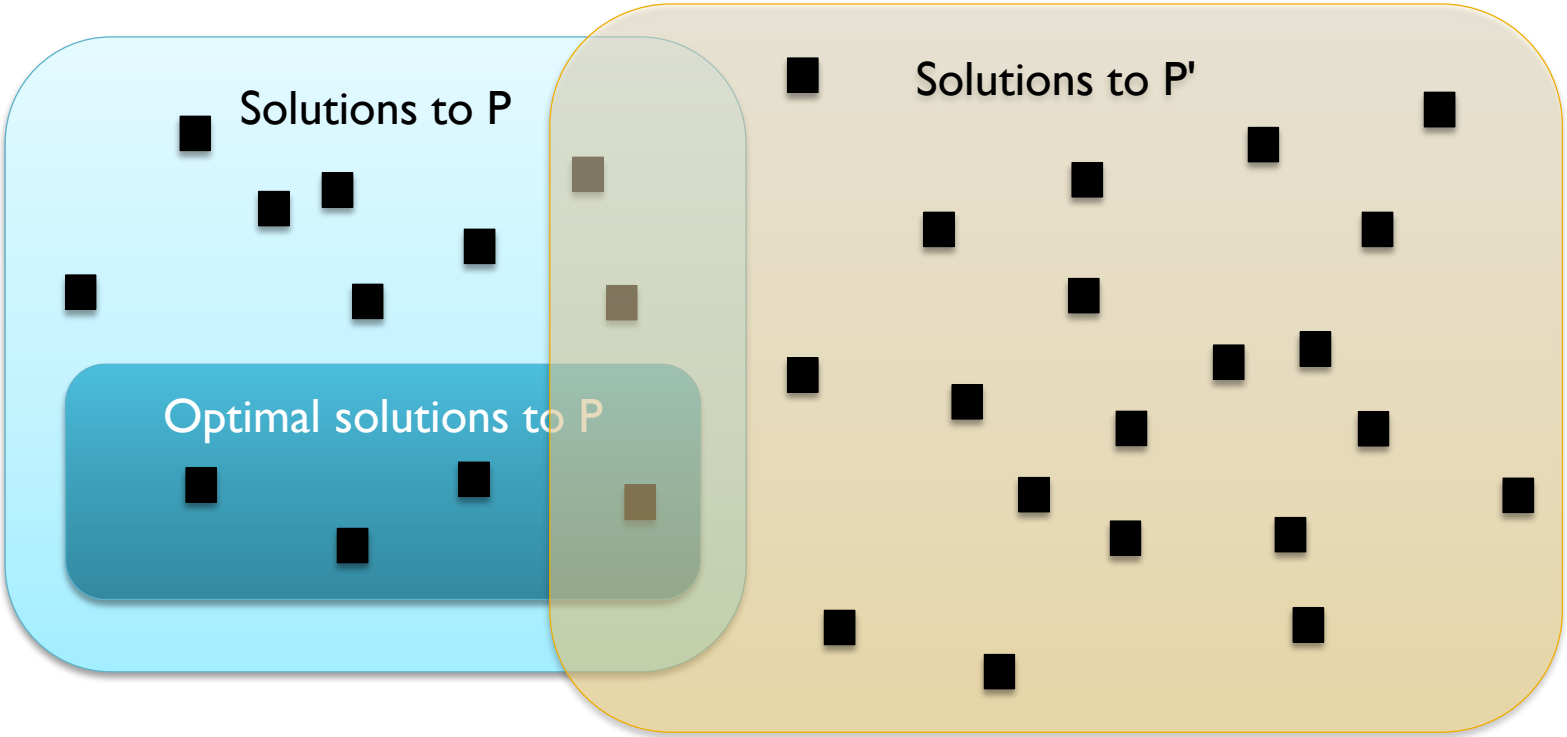


Solution Sets (1)

jonkv@ida

- How to prove cost(optimal-solution(P')) ≤ cost(optimal-solution(P))?
  - Sufficient criterion: **<u>One optimal solution</u>** to P **<u>remains</u>** a solution for P'
    - cost(optimal-solution(P')) = min { cost(π) | π is any solution to P' } <= cost(optimal-solution(P))

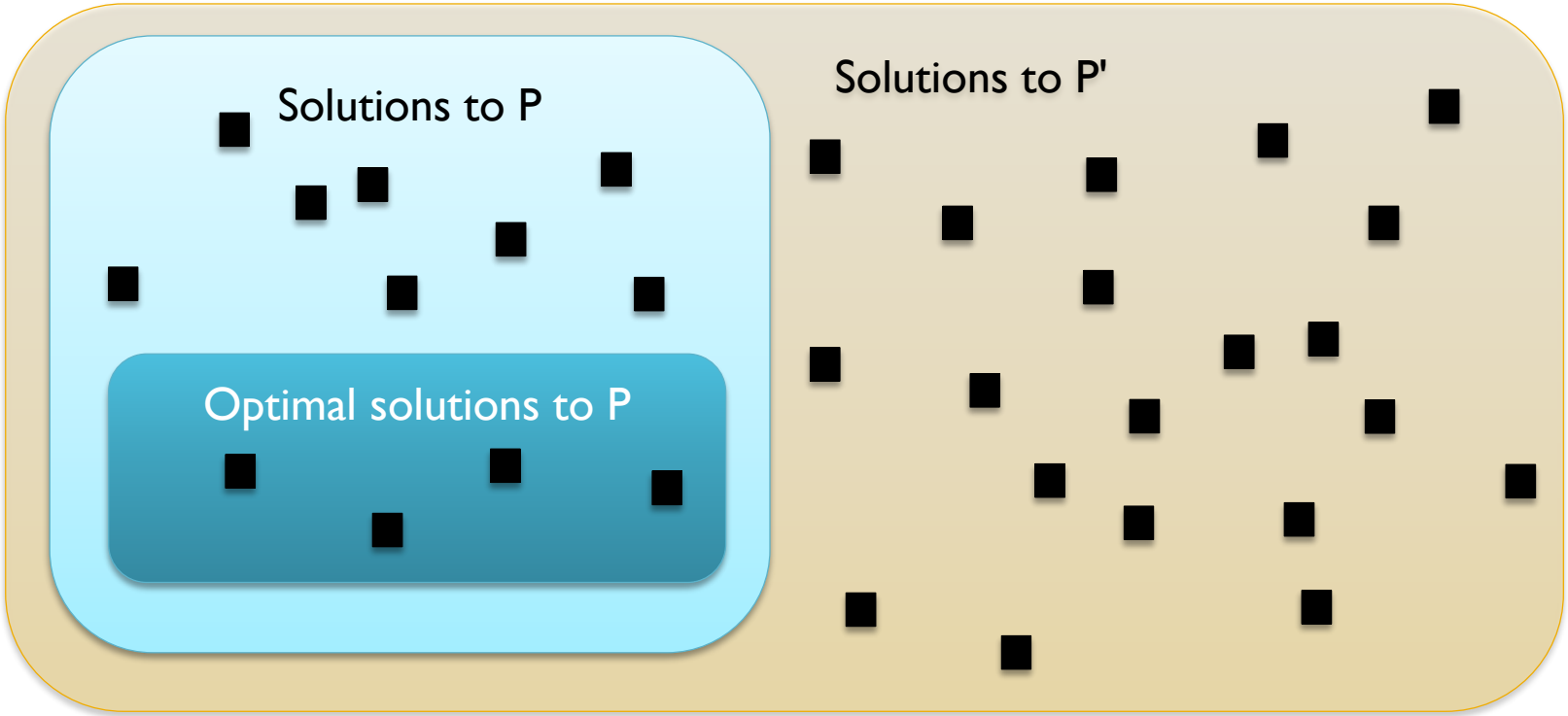
Includes the optimal solutions to P, so min {…} cannot be greater



Solution Sets (2)

jonkv@ida

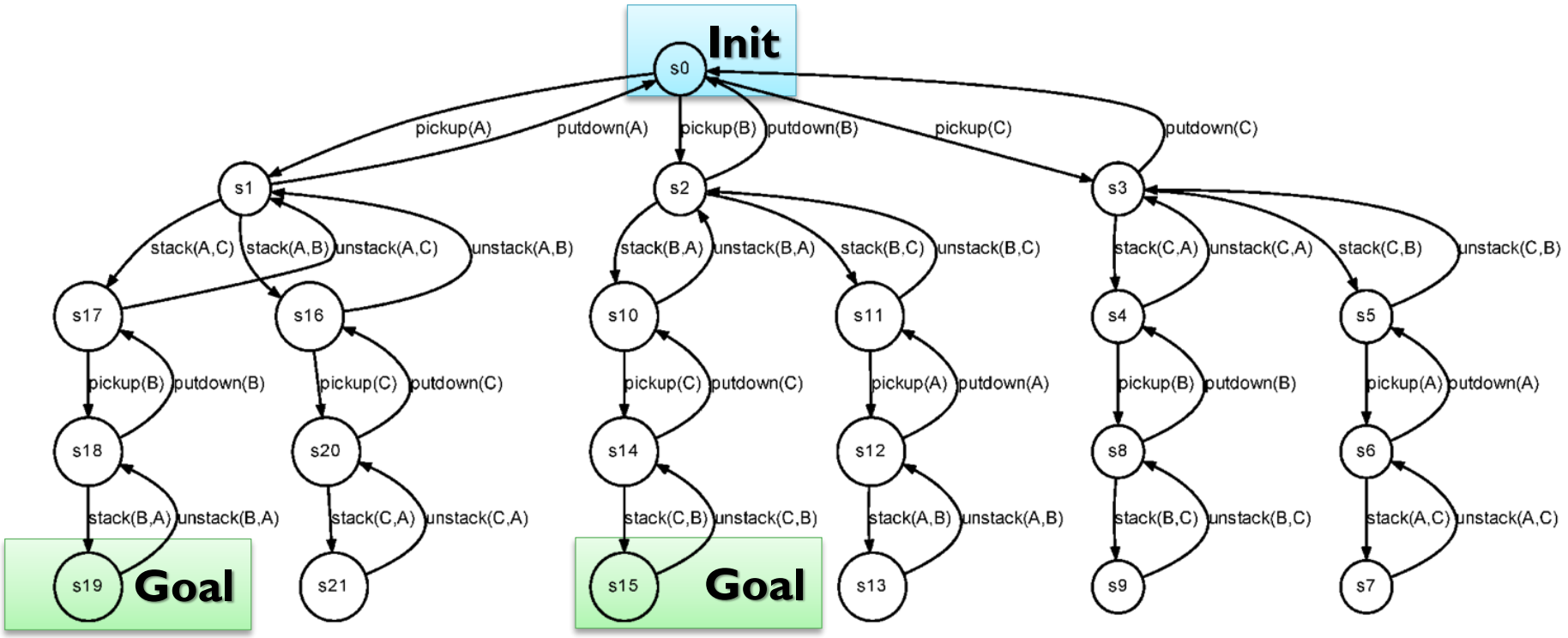
- Another sufficient criterion: **All solutions** to P **remain** solutions for P'
  - Stronger, but often **easier to prove**
  - **This** is called **relaxation**: P' is a relaxed version of P
  - **Relaxes** the constraint on what is accepted as a solution:
    The **is-solution(plan)?** test is "expanded,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



Relaxed Problem 1

jonkv@ida

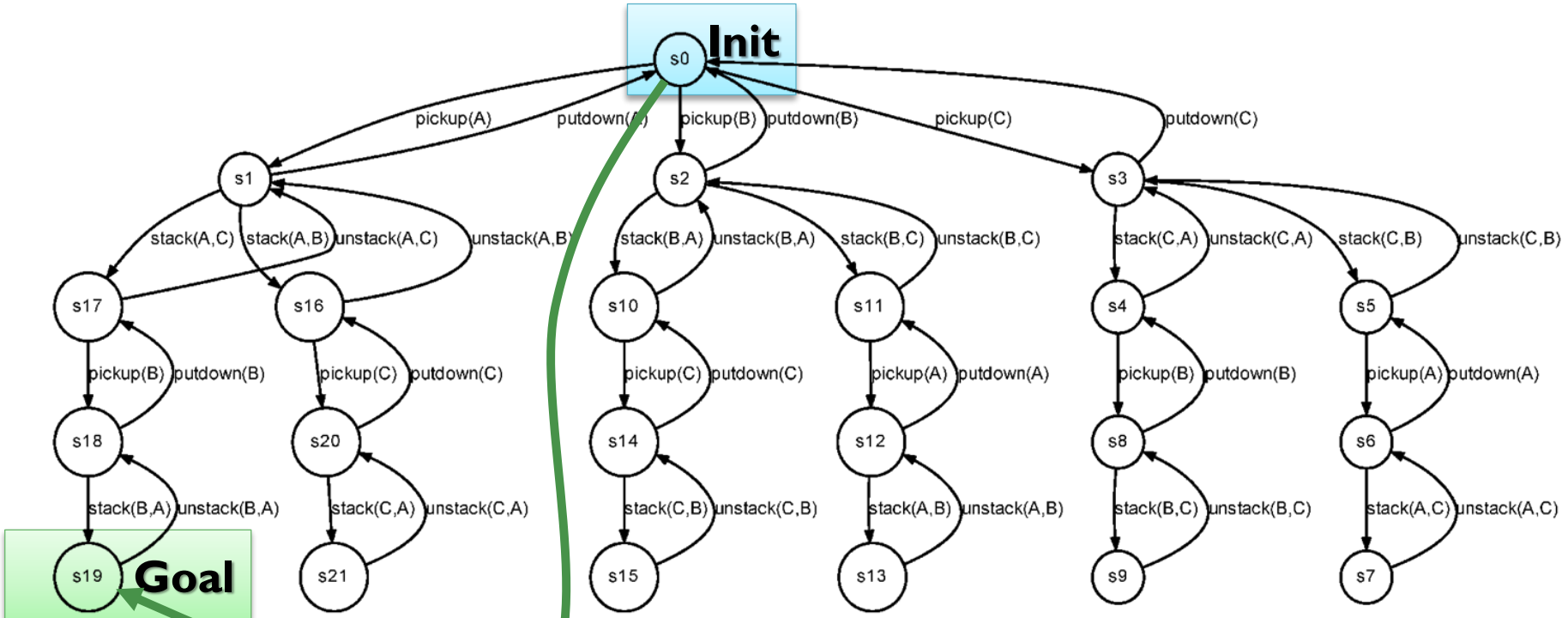
- Relaxation "method 1": **Adding goal states**
  - New goal formula: (and (on B A) **(or (on A C) (on C B))**)
    - All old solutions still valid
    - Some non-solution paths become solutions



Relaxed Problem 2

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



Other methods exist!

Understanding Relaxation

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)

jonkv@ida

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

jonkv@ida

- Main idea behind pattern databases:
  - Let's **ignore some facts** – everywhere
    - Initial state and goals
    - Preconditions and effects
  - Compute optimal costs **as if those facts didn't matter**
    - Simpler problem, smaller state space

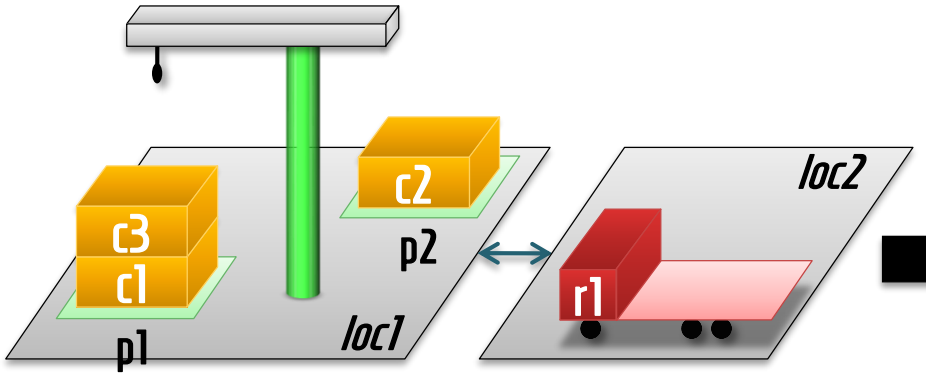
PDB 2: Dock Worker Robots

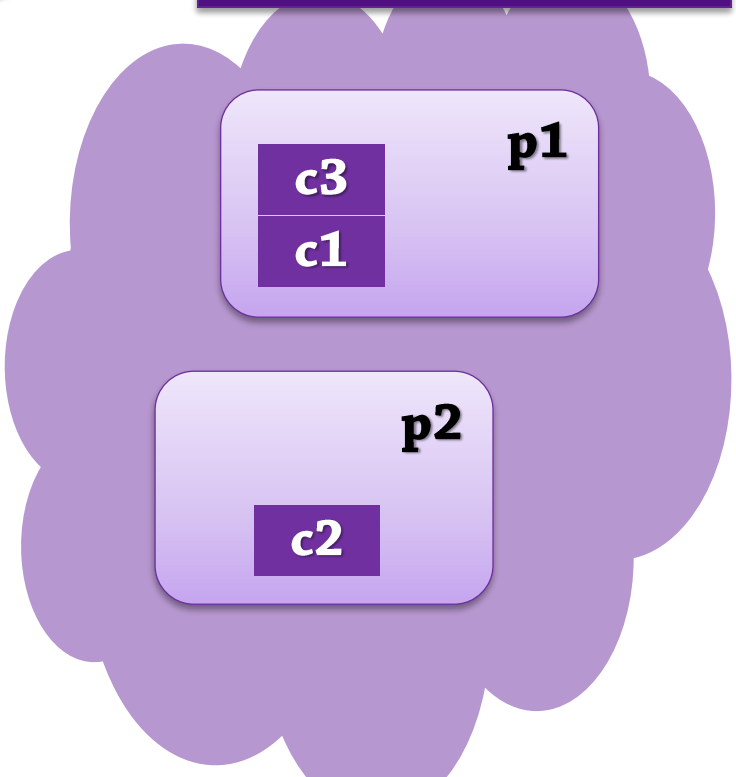
jonkv@ida

- 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



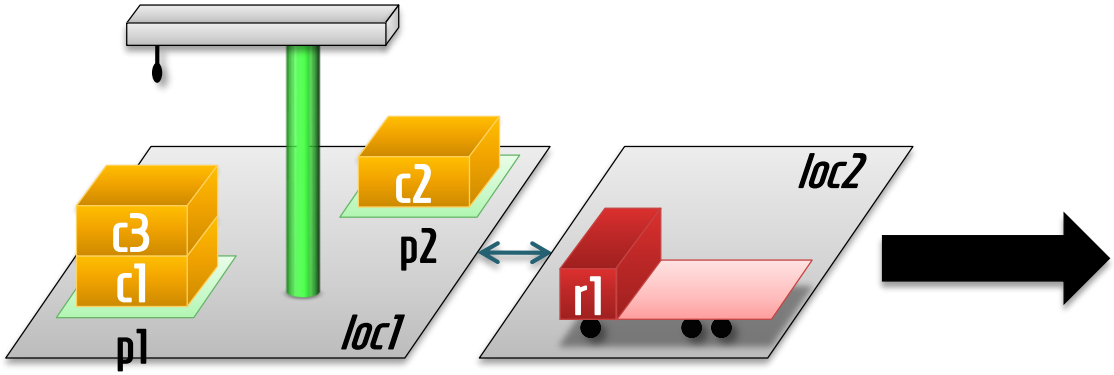


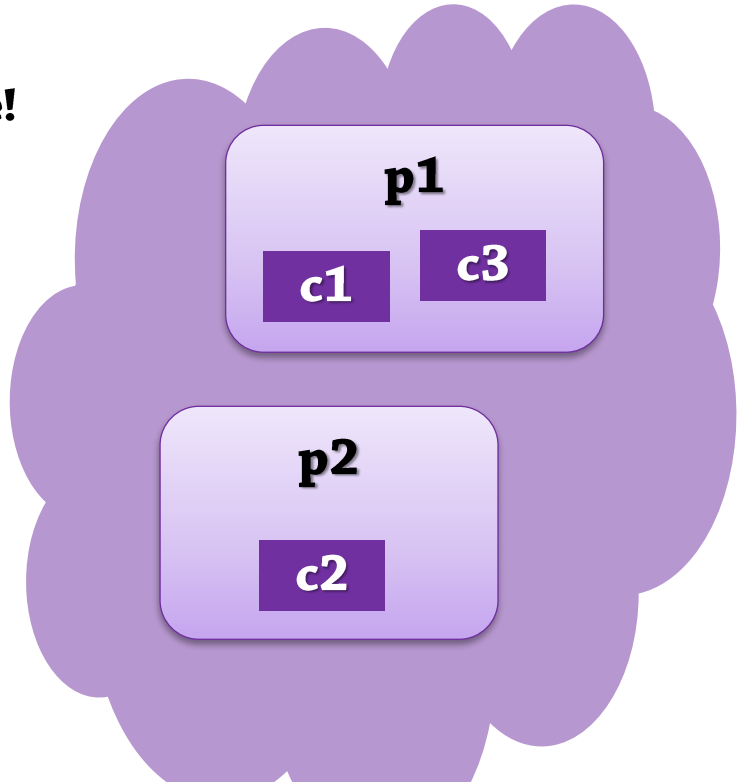
PDB 3: Planning in Patterns

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!



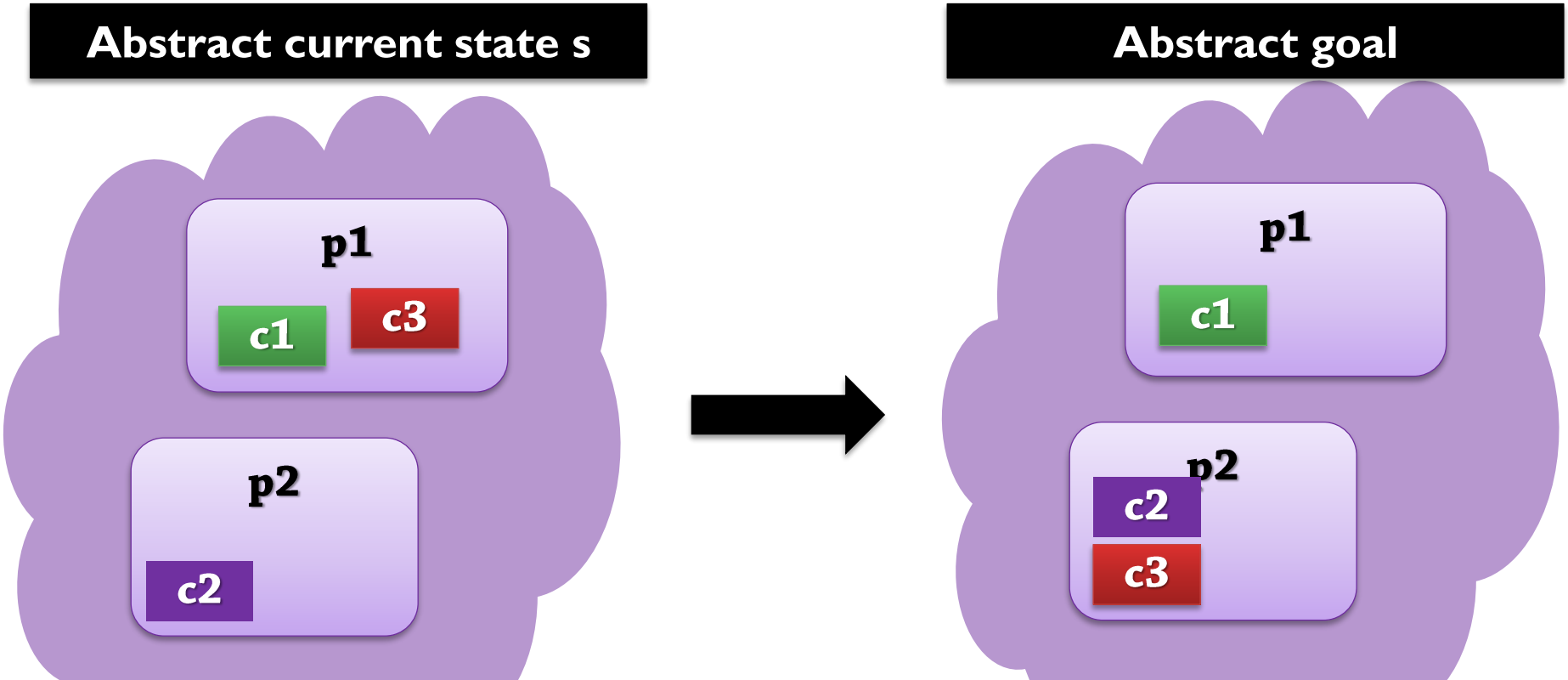


Solve optimally, compute cost ➔ admissible heuristic!

PDB 4: Computing a Heuristic Value

jonkv@ida

- **Solve** P'(s) **optimally:**
  - Take **c2** with the crane (it's in the way at the bottom)
  - Take **c3** with the crane [relaxation – not checking if the crane is busy]
  - Place **c3** at the bottom
  - Place **c2** on the top

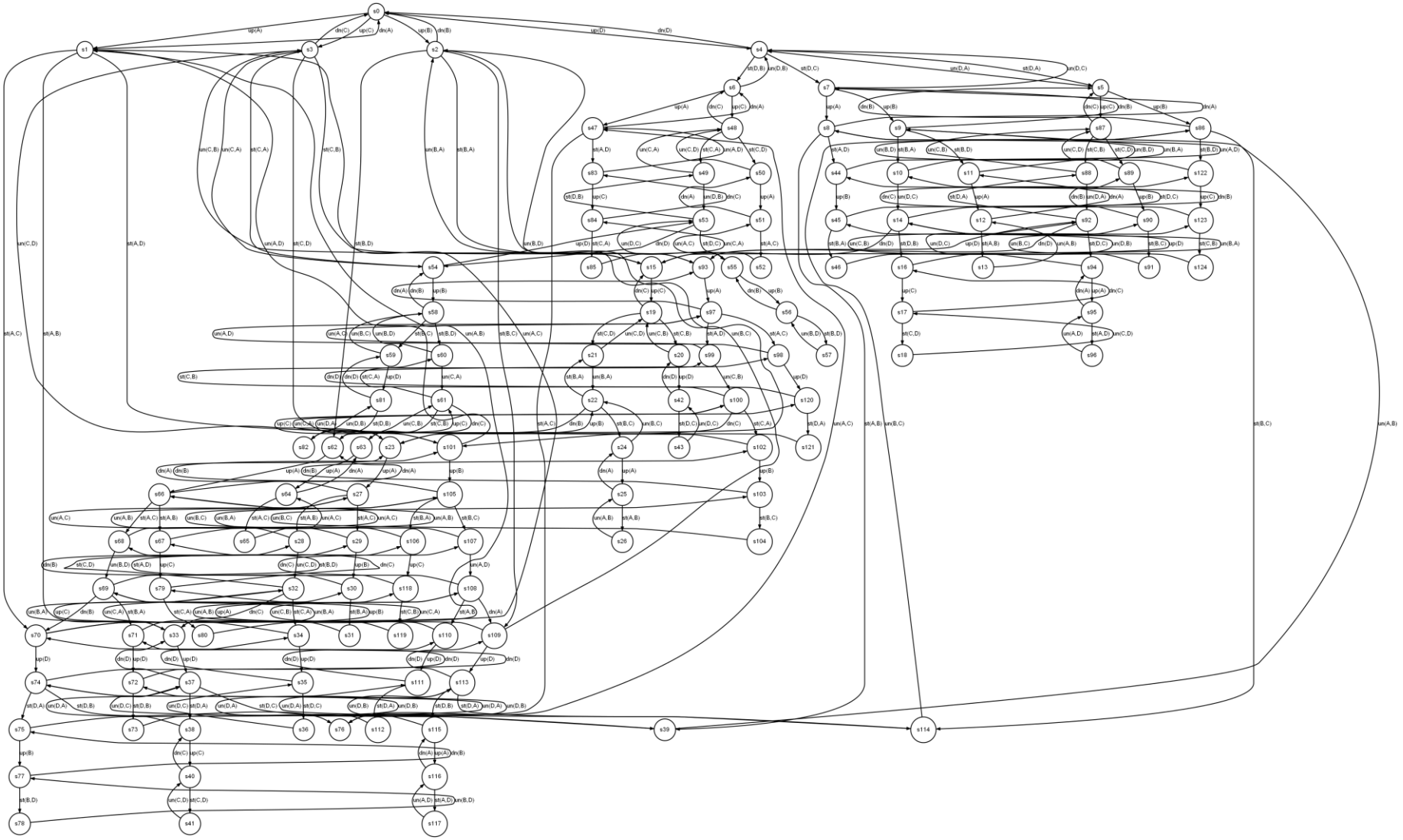


Let's formalize!

PDB: Blocks World size 4

jonkv@ida

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



PDB: Blocks World size 4, facts

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

jonkv@ida

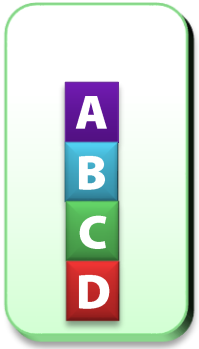
- Example: **only** consider 5 **ground facts** related to **block A**
  - **"Pattern":** p={**(on A B), (on A C), (on A D), (clear A), (ontable A)**}
  - **Initial state:**

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

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

jonkv@ida

- The **pattern** p is the **set of ground facts we care about**
  - 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

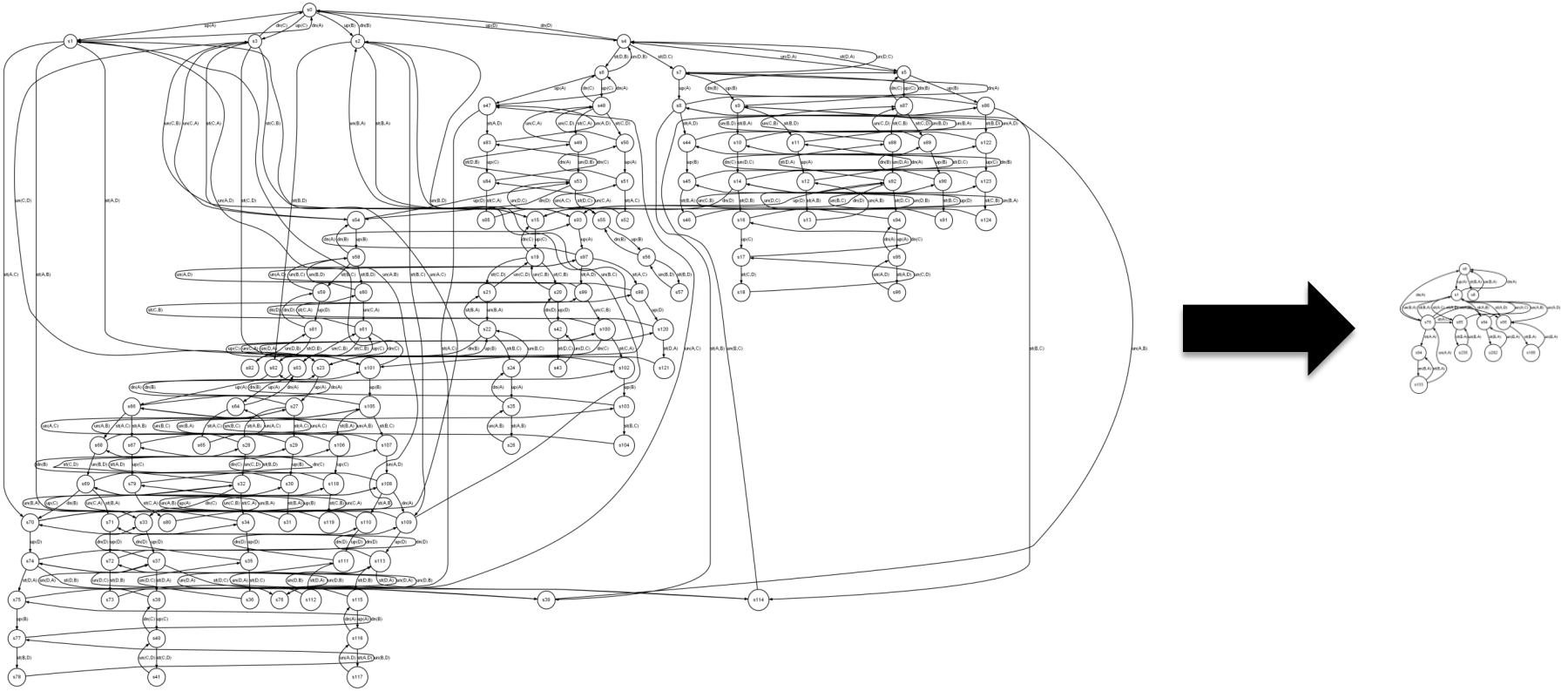
(clear A)
(ontable A)

A pattern generally contains few facts – for performance!

PDB: New State Space

jonkv@ida

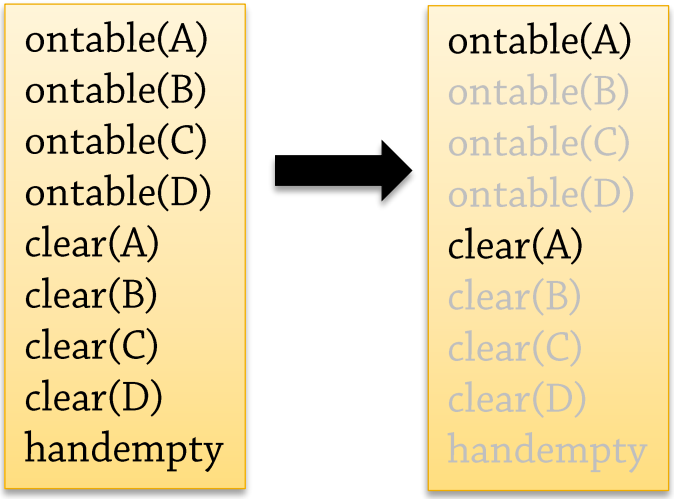
- Pattern p={**(on A B), (on A C), (on A D), (clear A), (ontable A)**}



PDB: Relaxation?

jonkv@ida

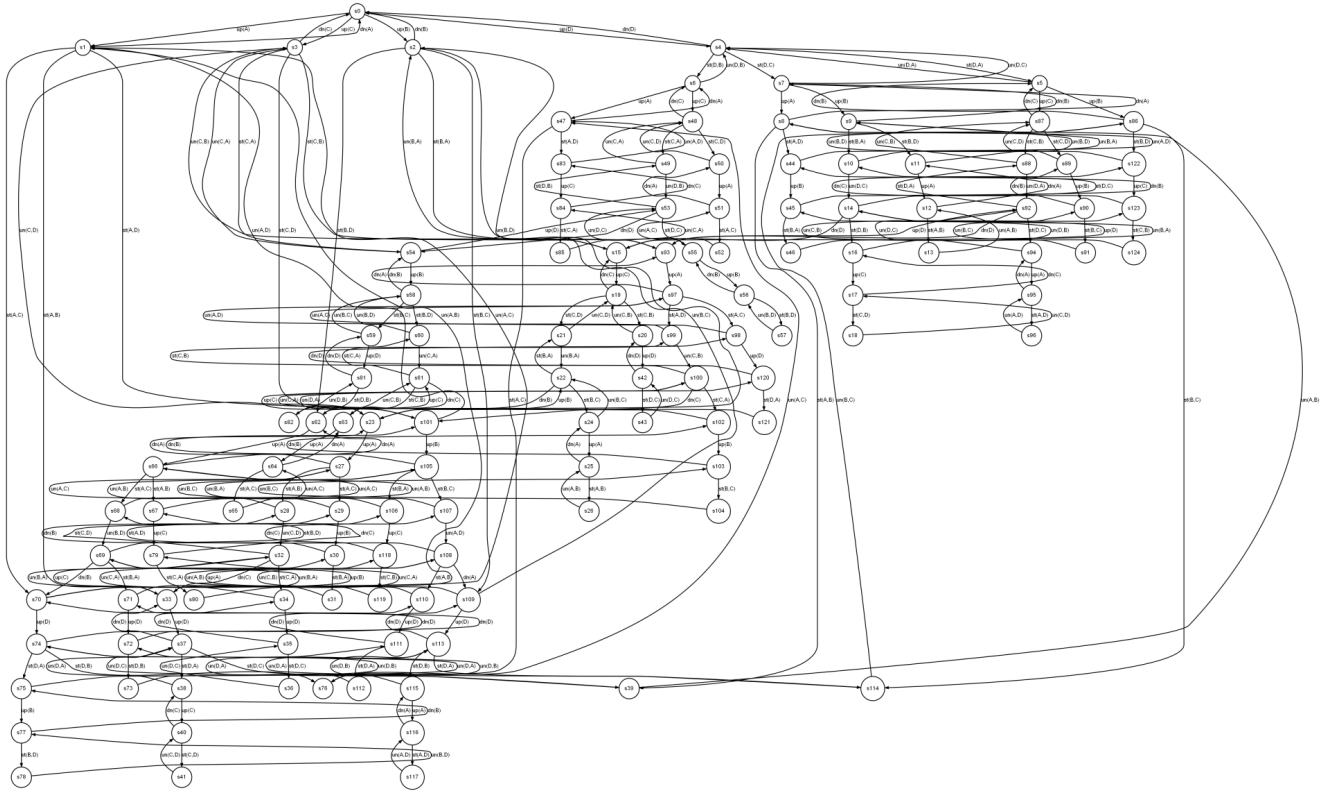
- Is this a **relaxation**?
  - **Yes**
  - Facts **disappear** from states…
    - $S' = \{s \cap p | s \in S\}$
  - But also from **precond/goal requirements**!
    - If a_i could be executed in s, $\text{transform}(a_i)$ can be executed in $s \cap p$
    - If $g \subseteq s$, then $g \cap p \subseteq s \cap p$
    - …



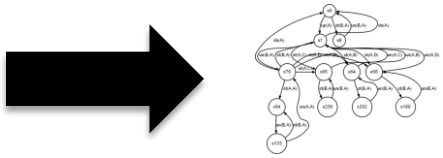
PDB: Preserving Solutions

jonkv@ida

Any path leading to a goal state here…



…also leads to an abstract goal state here!



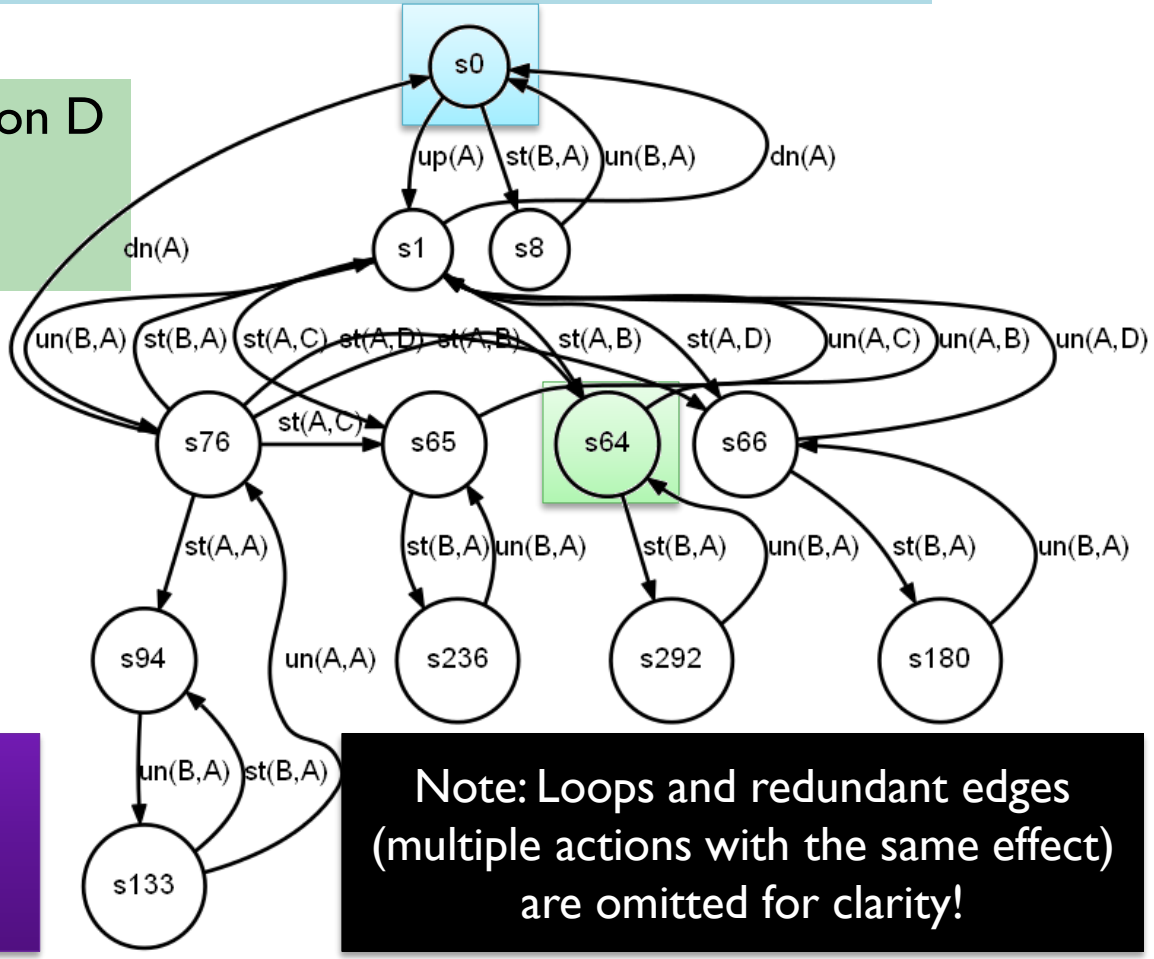
unstack(C,D)

**These paths can contain ”loops”, actions that do nothing in P’:
unstack(C,D) may be part of the original solution,
but does not affect any of the *interesting* facts**

PDB: State Transition Graph

jonkv@ida

- New **reachable state transition graph**:
  - **Current state**: Everything on the table, hand empty, all blocks clear
    - Abstract state: s0 = { (ontable A), (clear A) }
  - **Goal state**: A on B on C on D
    - Abstract goal: s64 = { (on A B), (clear A) }
  - Sufficiently few states to **quickly** compute **optimal** costs
    - Cost is *at least* 2: Shortest path s0 ➔ s64



Optimal cost of a relaxation ➔ admissible heuristic

Note: Loops and redundant edges (multiple actions with the same effect) are omitted for clarity!

PDB: More information

jonkv@ida

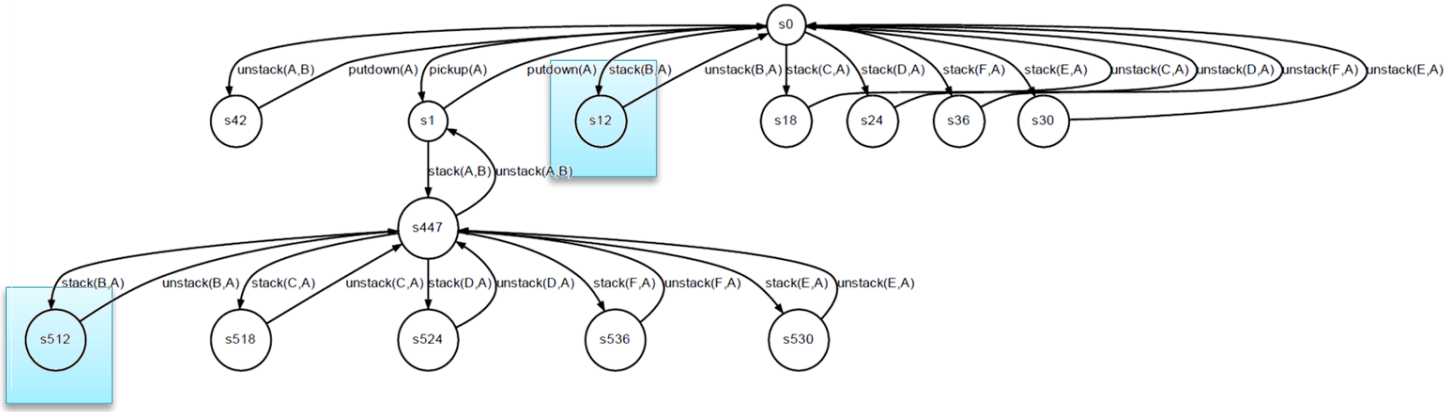
- To make PDB heuristics **more informative**:
  - Calculate costs for **several** patterns
    - Suppose we only care about **{clear(A), ontable(A)}**
    - Suppose we only care about **{on(A,B), on(C,D)}**
    - Suppose we…
  - Take the **maximum** of the computed heuristic values
    - Cost is at least c1 and at least c2 ➔
      cost is at least max(c1,c2)

- Real difficulty:
  - Choosing which patterns to use…

PDB: Databases

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

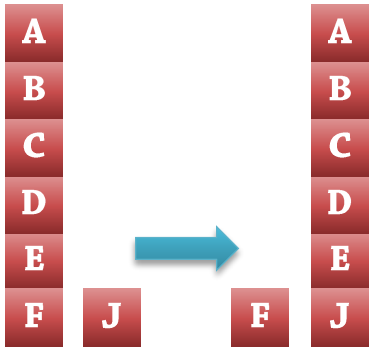


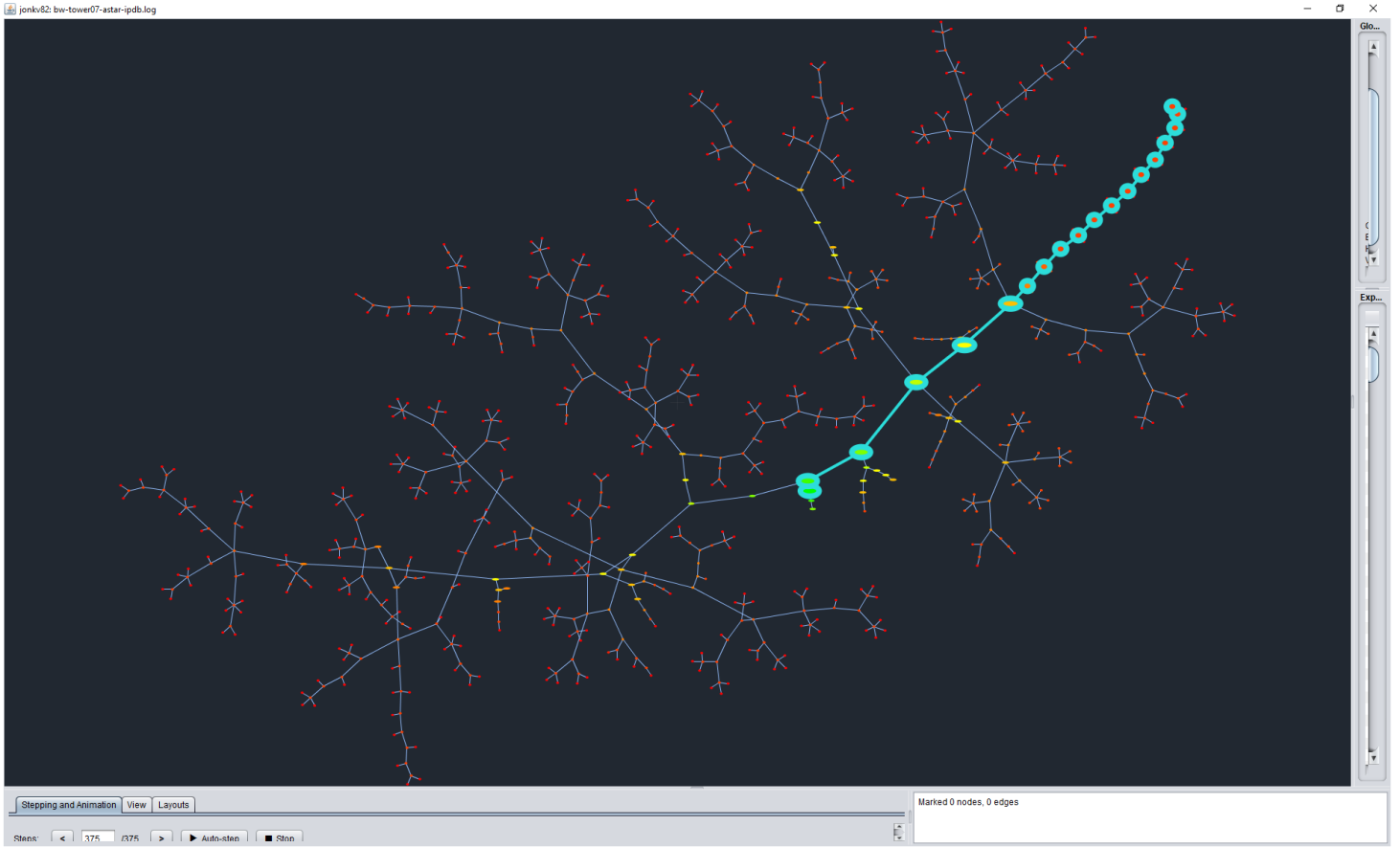
  - Determine which ones satisfy the **abstract goal**
    - Abstract: { aboveA = B }

  - 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!

Satisficing Planning

Satisficing Planning

jonkv@ida

- Optimal plans are **nice** but **often hard to find**
  - Larger problem instances ➔ too much time, memory (even with *good* heuristics)!

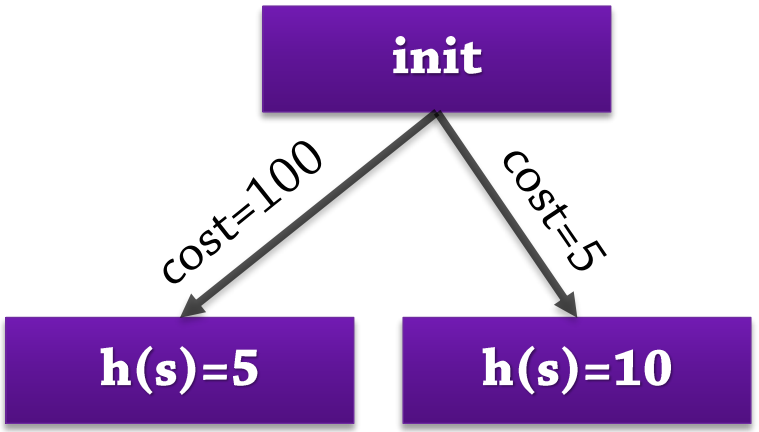
- **Satisficing** plan generation:
  - Find a plan that is sufficiently ***good***, sufficiently ***quickly***

- What's **sufficient**?
  - Usually not well-defined!
  - *"These strategies and heuristics seem to give pretty good results for the instances I tested…"*

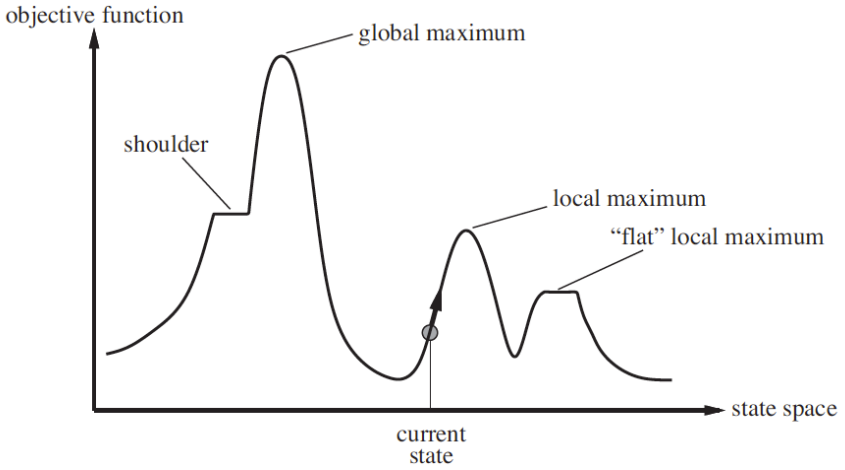
Speed: Strategies

jonkv@ida

- One reason for speed: Other informed **search strategies**
  - Simple: **Greedy best first search**
    - Always expand the node that seems to be closest to the goal
    - Who cares if getting there was expensive? At least I might find a way to the goal!
    - (Only care about h(s), not about g(s))



  - **Hill climbing**
    - Be stubborn: If one direction seems promising, ***continue*** in this direction

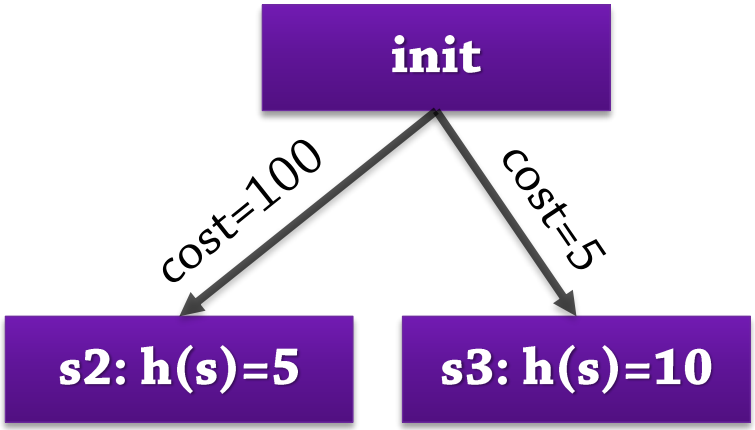


  - **Many others!**

Speed: Heuristics

jonkv@ida

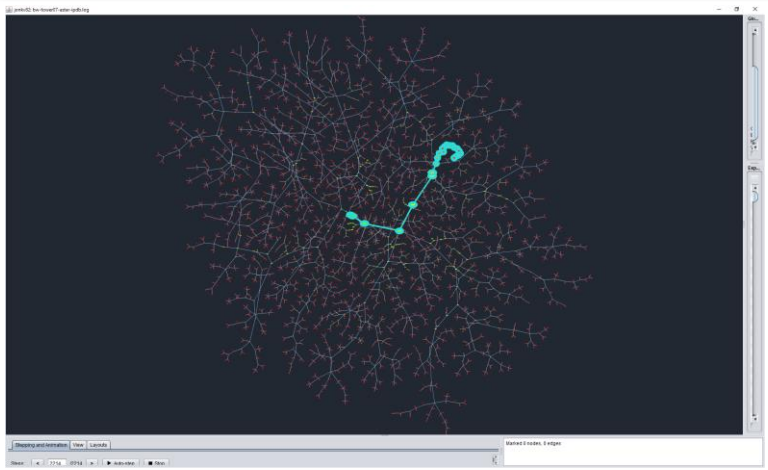
- One reason for speed: Often **more informative heuristics**
  - **Optimal planning**:
    - Often requires **admissibility**: ”We *must never* overestimate, *ever*!”
    - Result: Usually *underestimates* (a lot!)
  - **Satisficing planning:**
    Extreme example – ***greedy best first***
    - Only important that the ”best” successor has a low heuristic value
    - For GBF, what the value *is* doesn’t matter!

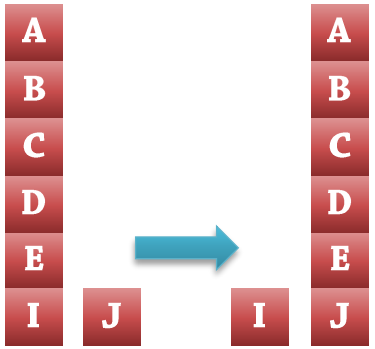


  - In many cases:
    - Admissibility is not required
    - Lack of admissibility is not ”only slightly harmful”
    - Lack of admissibility is *irrelevant*

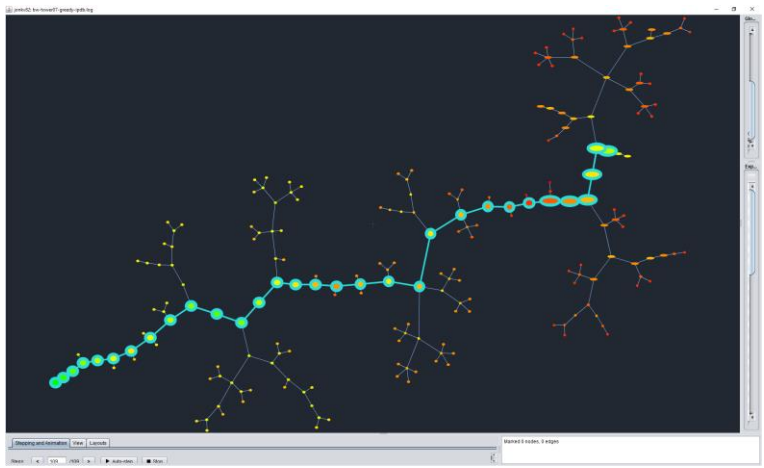
7 blocks (tiny problem)

A*, IPDB: 1321 states, 18 actions





Greedy, IPDB: 171 states, 22 actions



Larger problem instances → the difference <u>increases</u>

Example:
Landmark Heuristics

Landmark Heuristics (1)

Landmark:

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

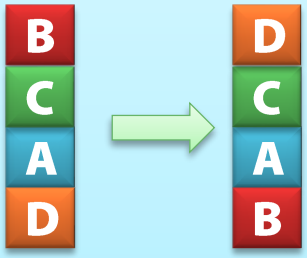
Landmark Heuristics (2)

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*



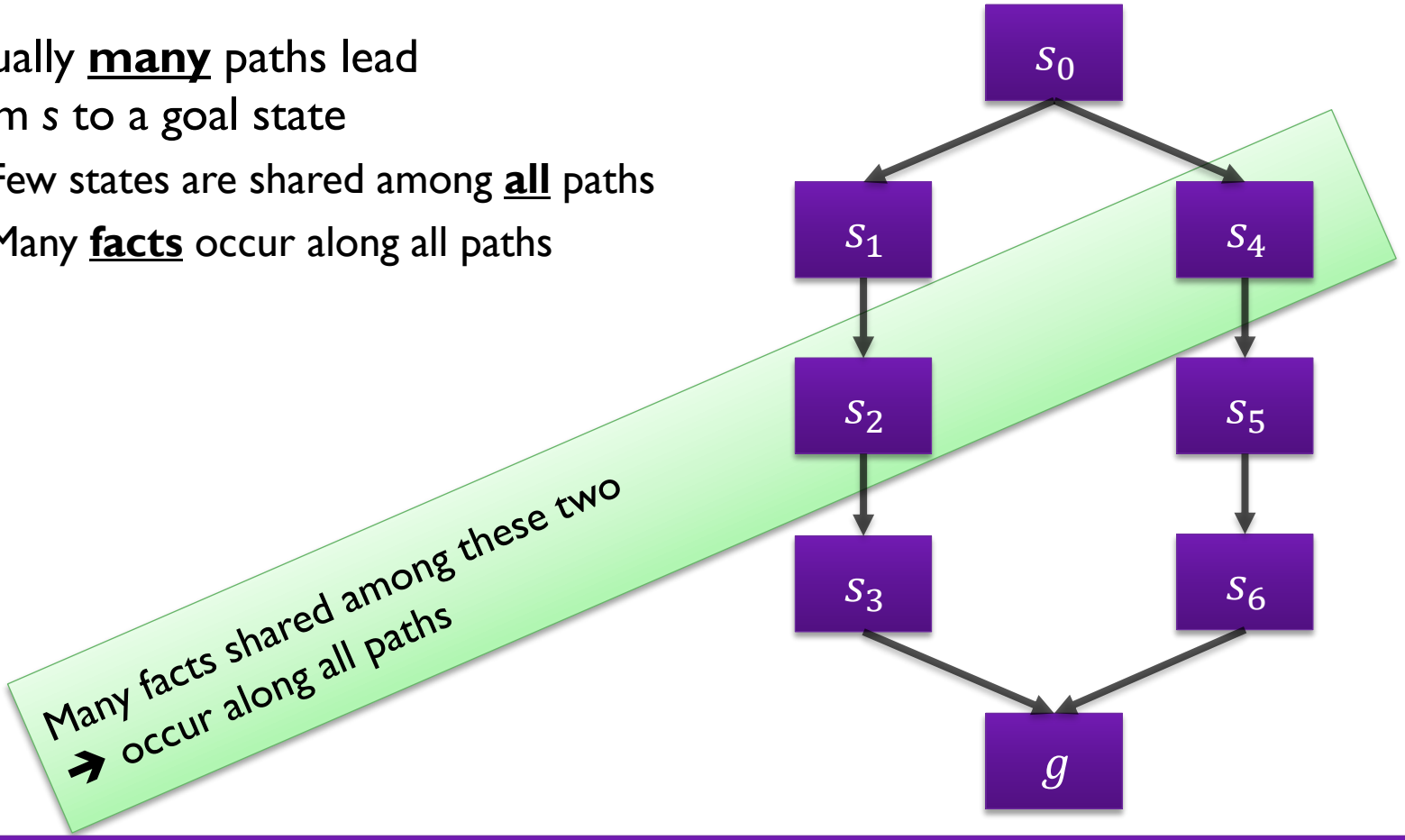
clear(A)
holding(C)
…

Landmark Heuristics (3)

jonkv@ida

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

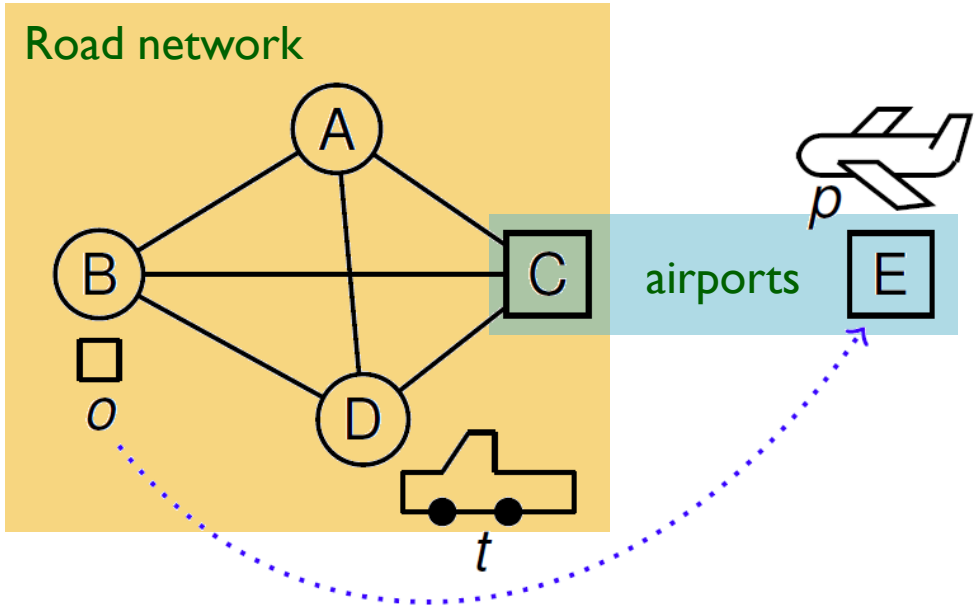


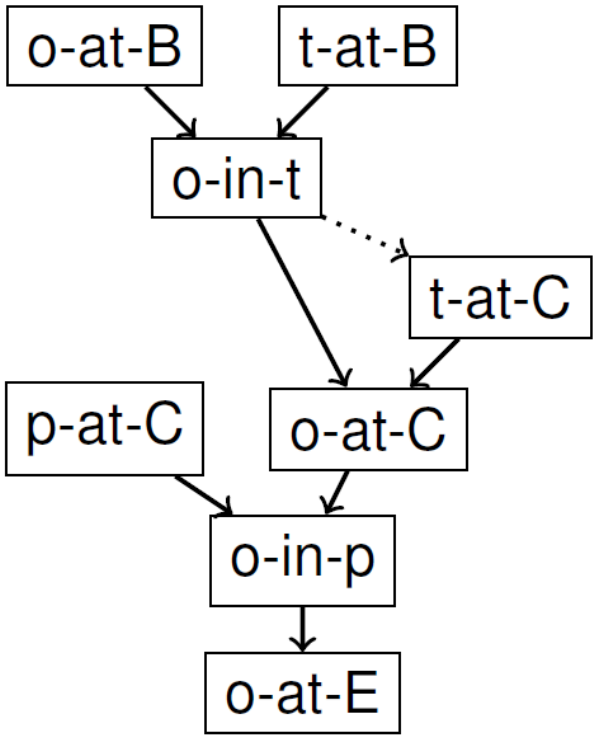
Finding landmarks: Complicated; not covered here

Landmarks as Subgoals

jonkv@ida

- 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*

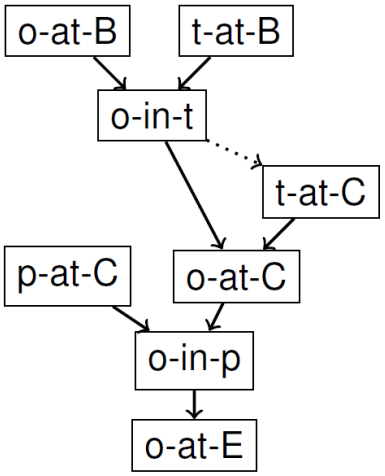




Landmark Counts and Costs

jonkv@ida

- Another 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

One action can achieve multiple landmarks at once
➔ not admissible (though this can be ”adjusted”)

Example:
Hill Climbing

Search Algorithms for non-admissible heuristics

jonkv@ida

- 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

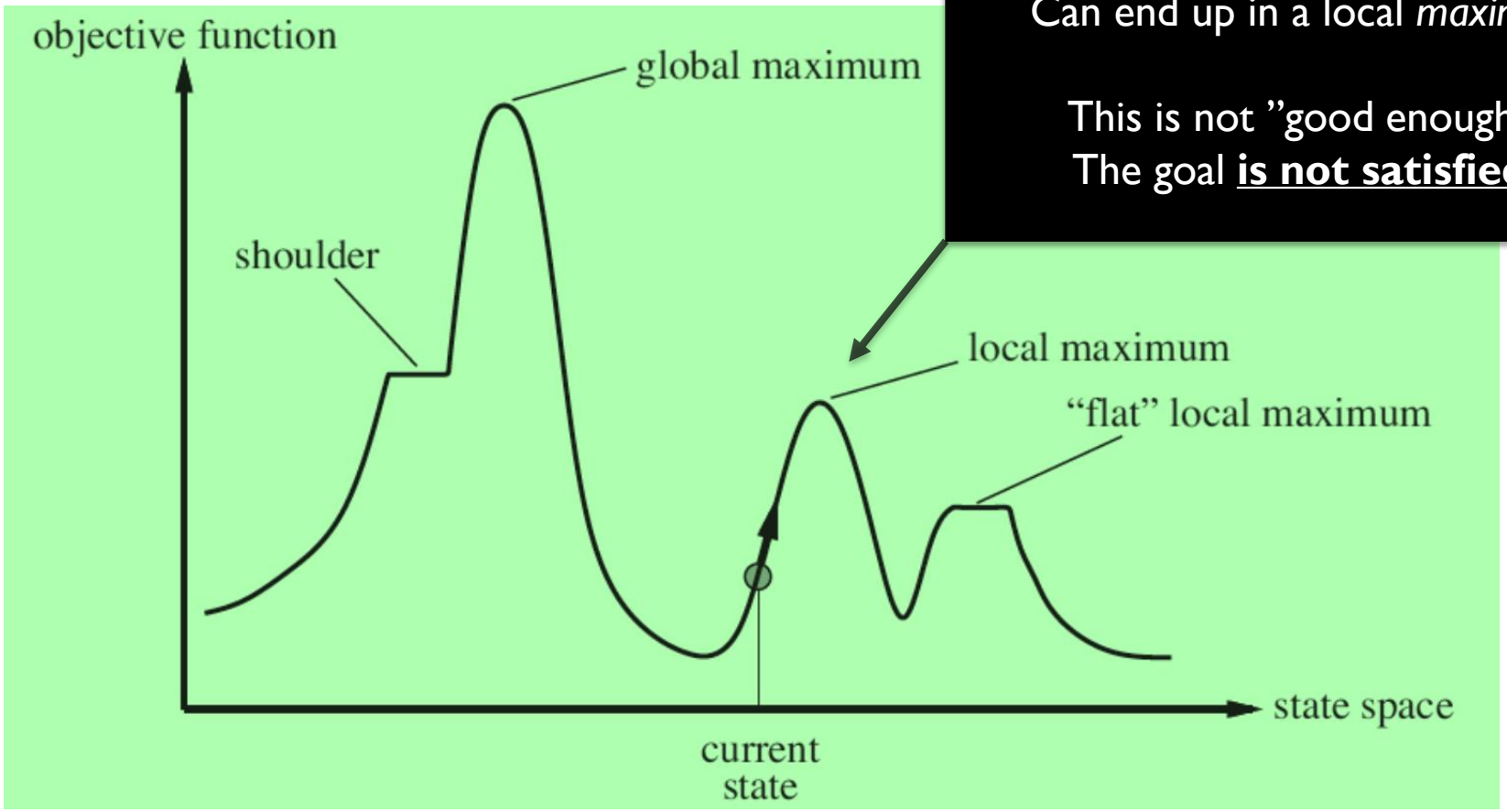
Value: Calculated as –h(s)
(maximize –h(s) ➔ minimize h(s))

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

jonkv@ida

- Problem:



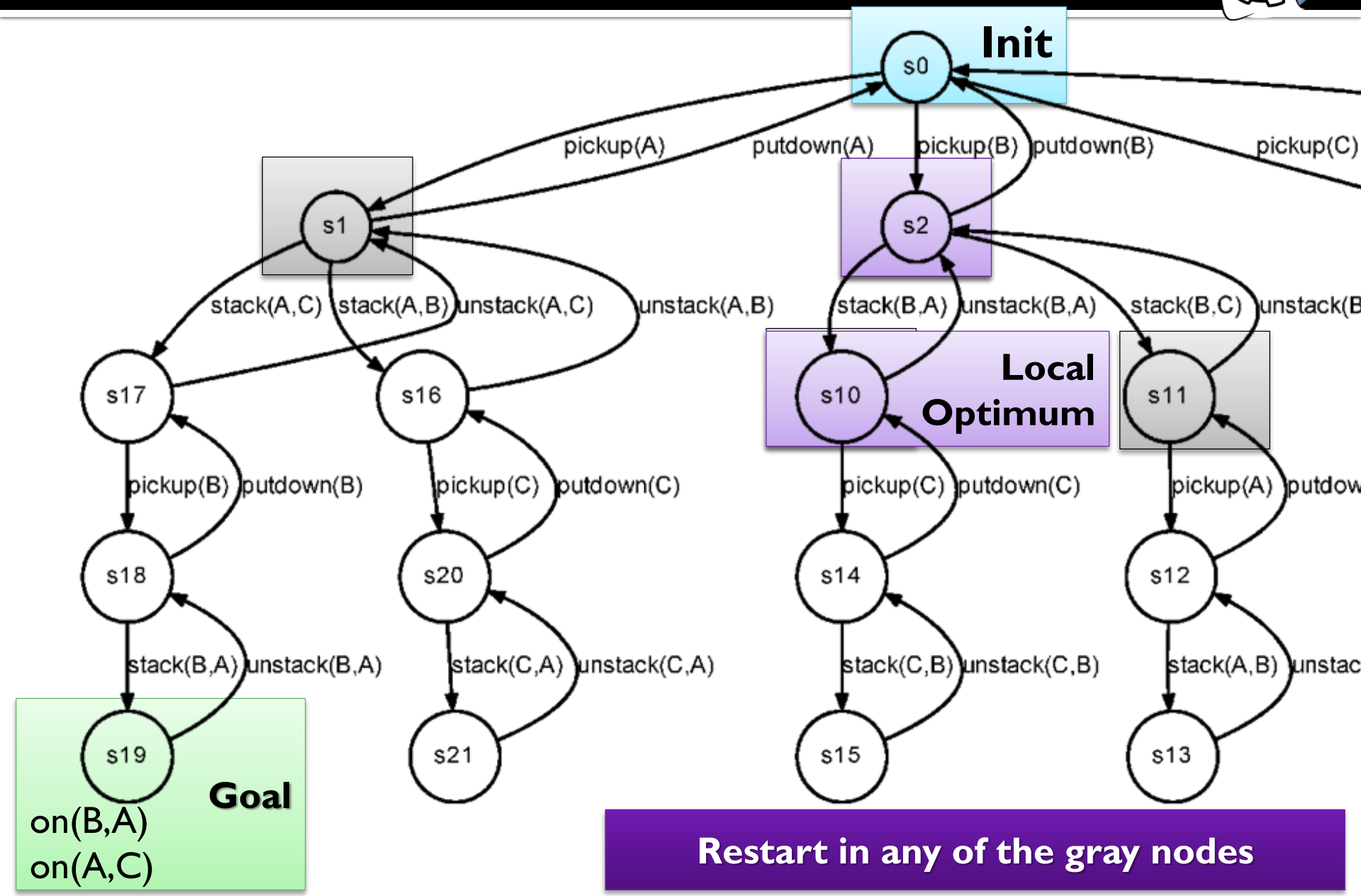
Can end up in a local *maximum*

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

Possible solution:
Restart search from another expanded state

Local Optima: Restarting

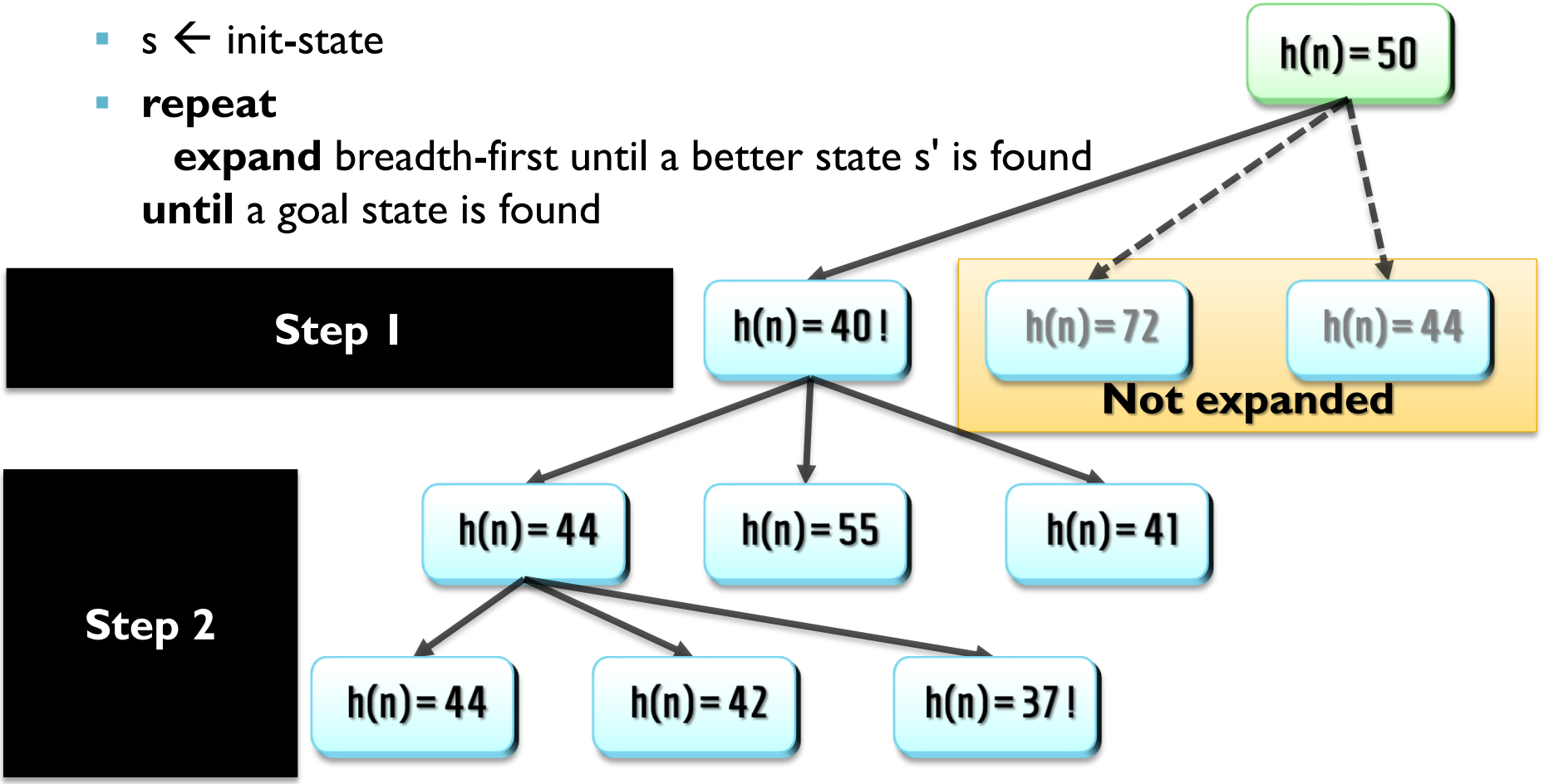
jonkv@ida



Enforced Hill Climbing

jonkv@ida

- *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

jonkv@ida

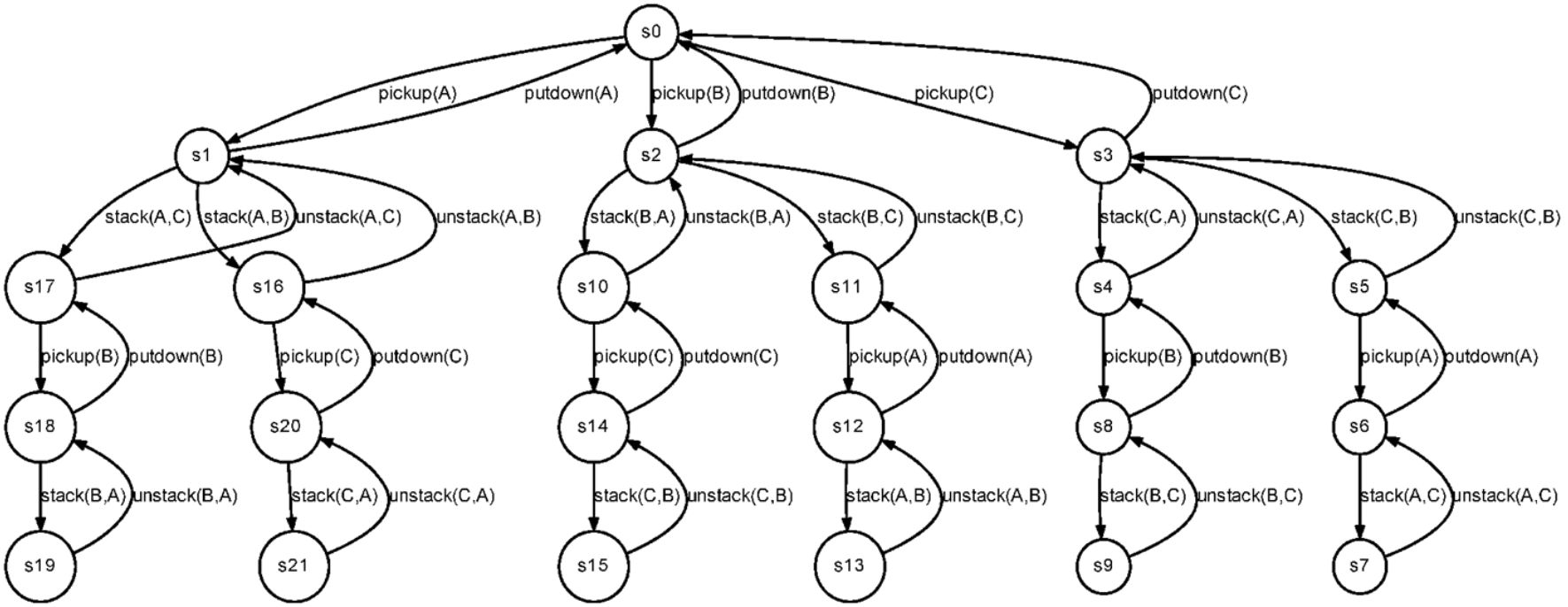
- Forward State Space Search:
  - Very common planning framework
  - **Many** heuristic functions defined
    - TDDC17: Focus on understanding the basics,
      and general principles for defining general heuristics
      without knowing the problem to be solved (blocks, helicopters, …)
    - TDDD48: 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

Introduction

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!

Motivating Problem

jonkv@ida

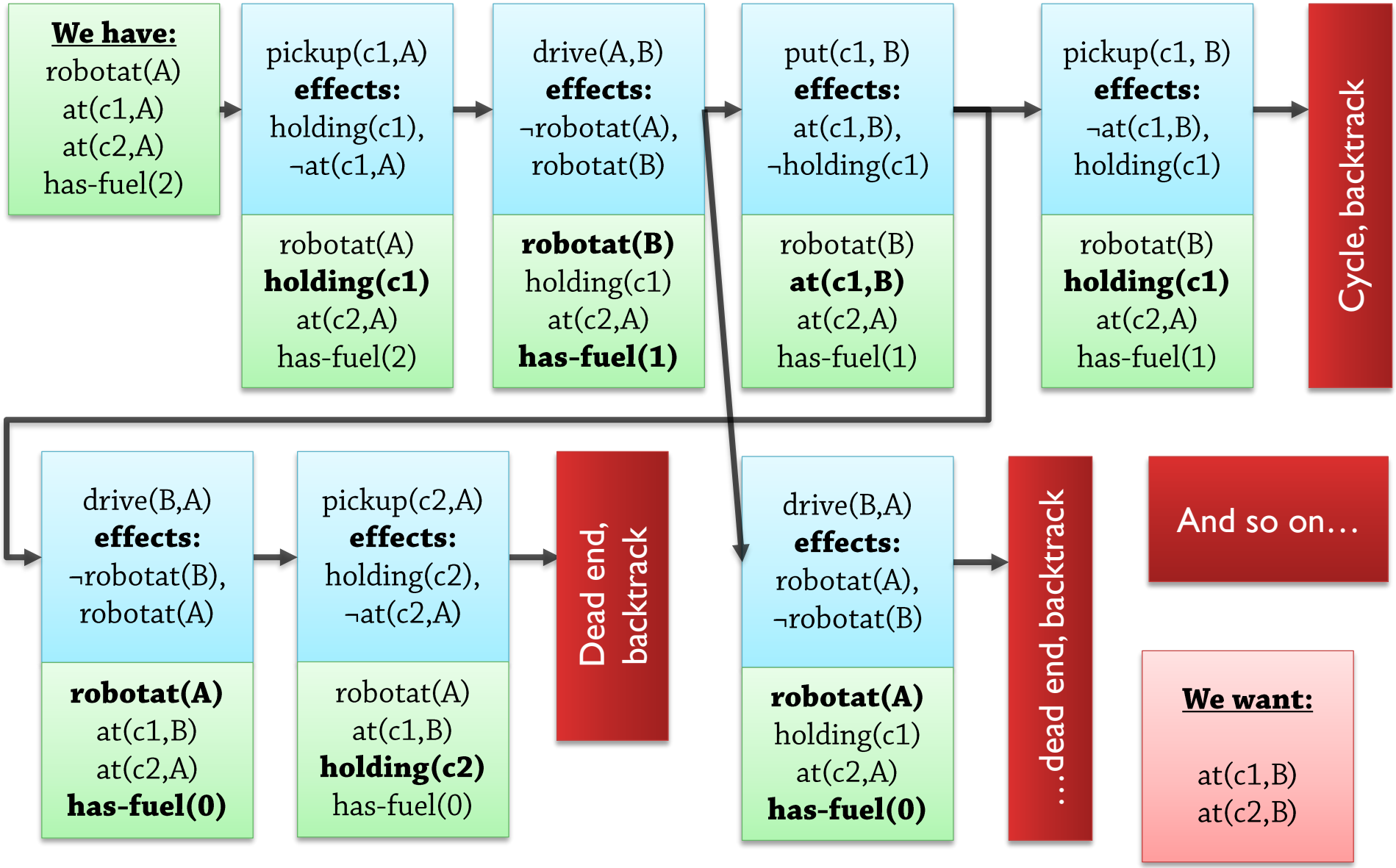
- Simple planning problem:
  - Two **crates**
    - At A
    - Should be at B

  - One **robot**
    - Can **carry** up to two crates
    - Can **move** between locations, which requires one unit of **fuel**
    - Has only two units of fuel

Let's see what a forward-chaining planner *might* do (depending on heuristics)…

Motivating Problem 2: Forward Search

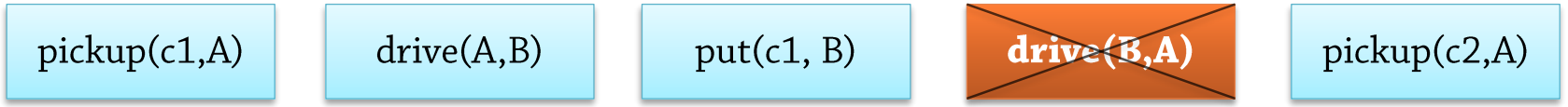
jonkv@ida



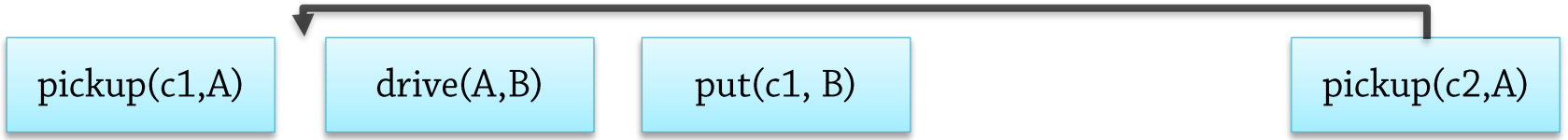
Motivating Problem 3

jonkv@ida

- Observations:
  - **Most** actions we added before backtracking were **useful** and **necessary**!



  - At first, we added them in the **wrong order**

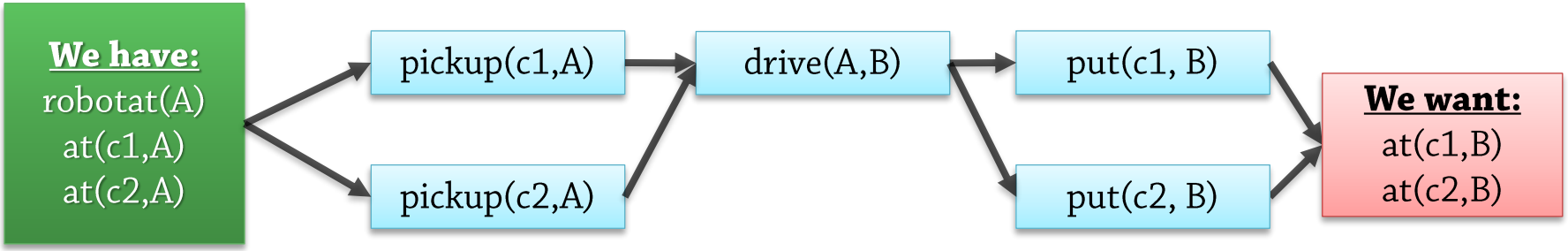


  - **Forward** planning **commits** immediately to action order
    - Puts each action in its **final place** in the plan
  - State space **heuristics** must be smart enough tell us:
    - Which actions are **useful**
    - ***When*** *to add them to the plan*

Partial Order Planning

jonkv@ida

- What if we could **insert actions anywhere?**
  - Requires different plan structures, book-keeping, heuristics, …
- Turns out we might as well take it **one step further**
  - Don't use sequences; change the plan structure to a **partial order**
  - Keep track of **necessary** and **possible** orderings



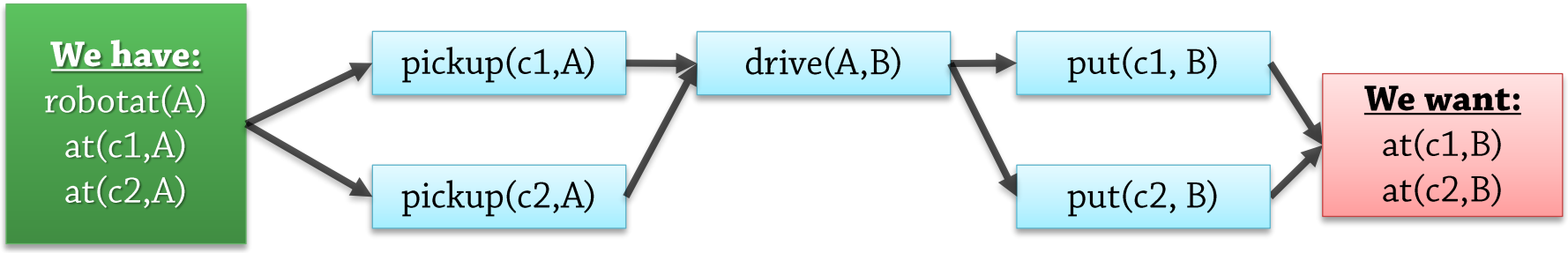
Haven't decided if we handle c1 or c2 first

Least/late commitment to ordering – "don't decide until you have to"

Partial Order Planning: Consequences (1)

jonkv@ida

- Consequence 1: If plans look like this…



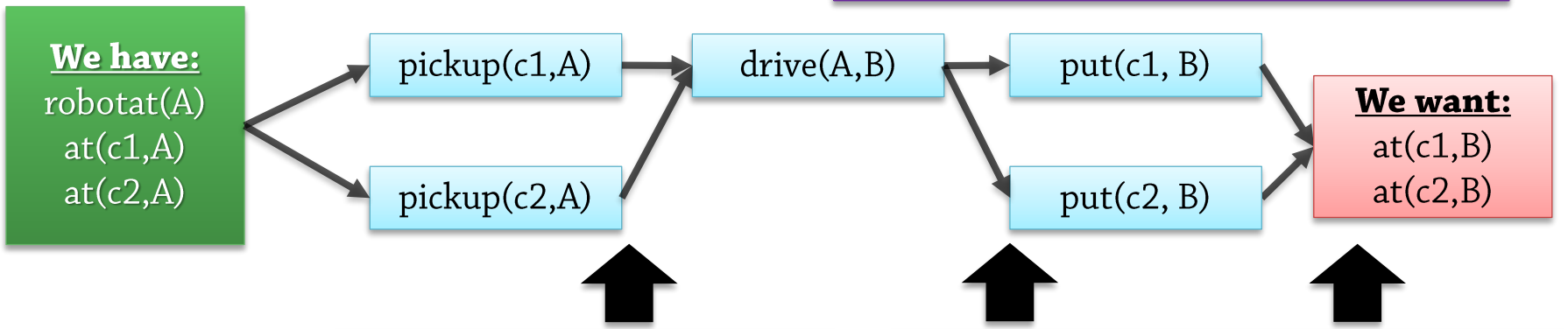
How do we define the set of successors?
Can we add any action anywhere?

Partial Order Planning: Consequences (2)

jonkv@ida

- Consequence 2:

Can the put actions interfere with each other in some way?



What's the state after pickup(c2,A)?

We don't know, exactly! Has p(c1,A) finished?

Are preconditions satisfied? Are goals satisfied?

More sophisticated "bookkeeping" required!

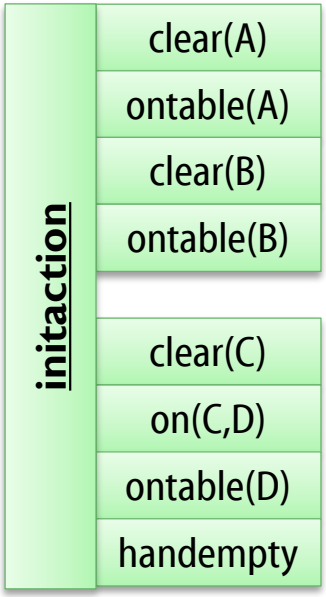
Partial-Order Planning

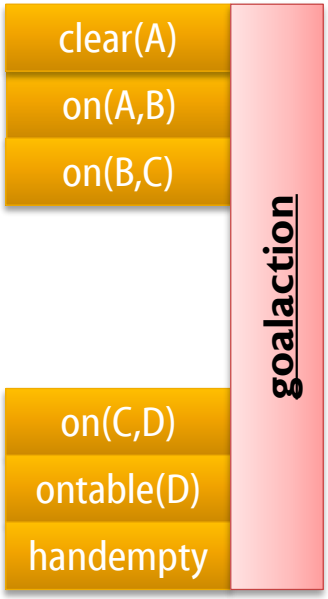
jonkv@ida

- 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

jonkv@ida

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

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

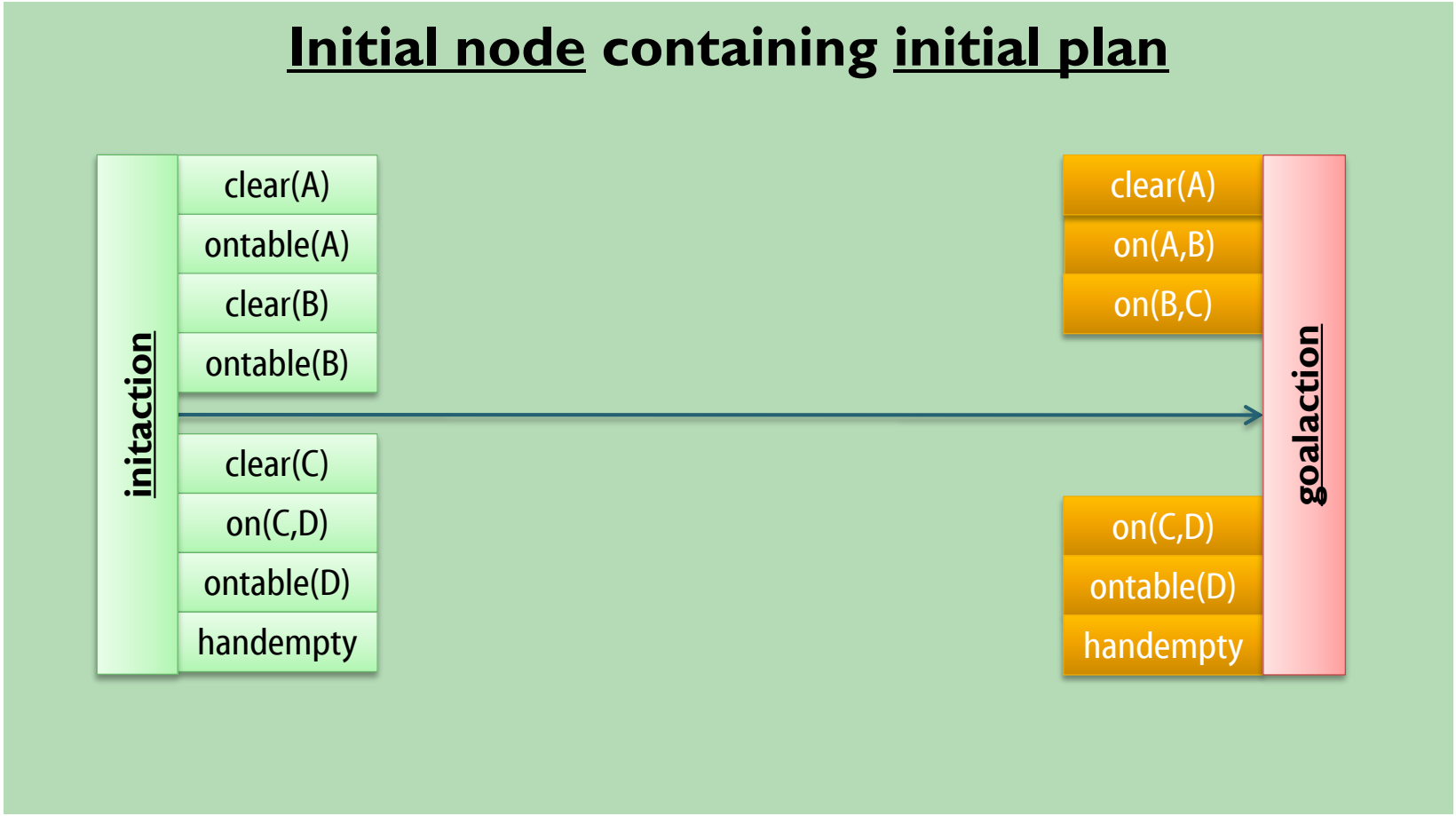
A "fake" goal action, whose preconditions are the conjunctive goal facts

initaction happens *before* goalaction

Partial-Order Planning

jonkv@ida

- This is the **initial node** in the search space – not a state!

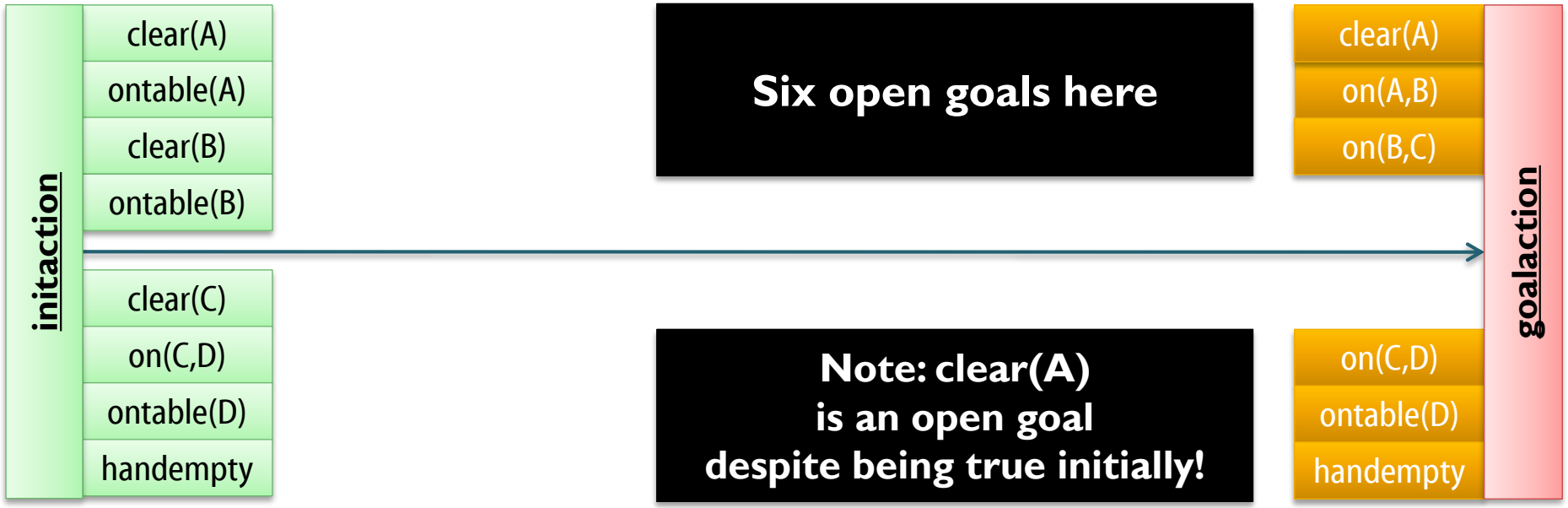


Partial-Order Planning

jonkv@ida

- No current state – what are the successors?
  - Don't need to "add any action anywhere"
  - We can identify a set of **issues to take care of** ("todos") called **flaws**

- First type of flaw: **Open goal**
  - When an ***action*** has a ***precondition*** that is not ***explicitly marked*** as handled



Resolving Open Goals 1

jonkv@ida

- To **resolve** an open goal:

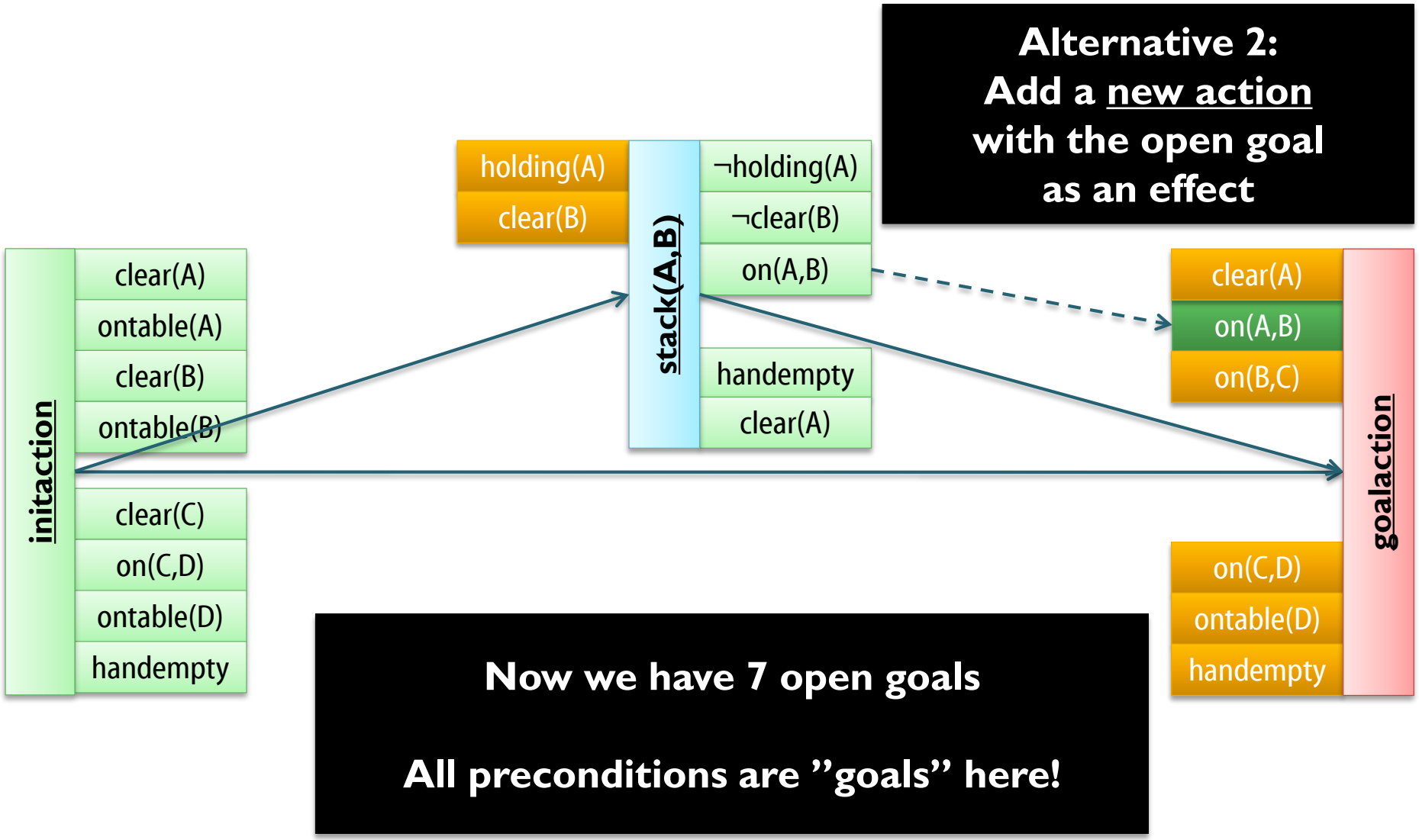
Alternative 1:
Add a causal link from an existing action that achieves it

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

jonkv@ida

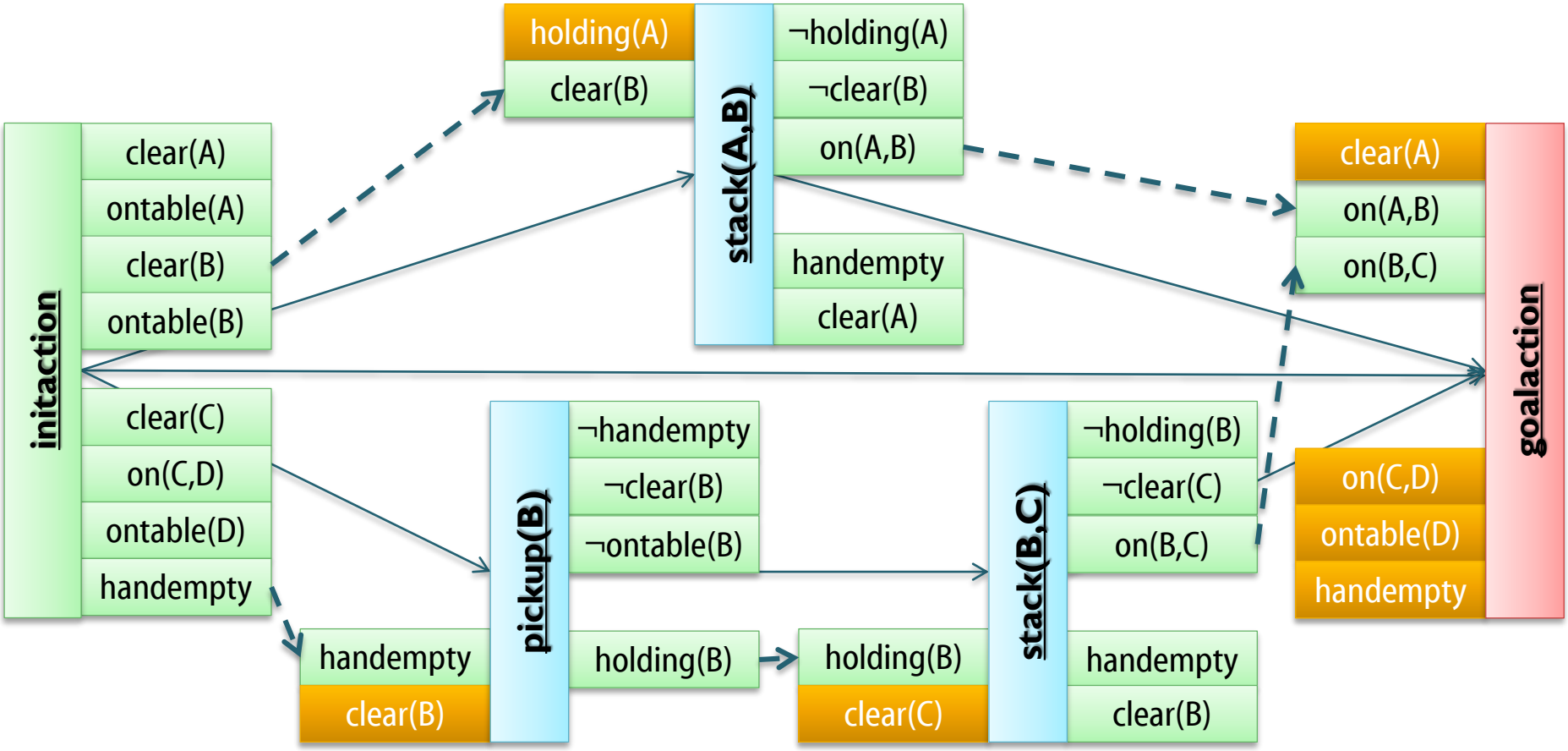
- To **resolve** an open goal:



Partial Orders

jonkv@ida

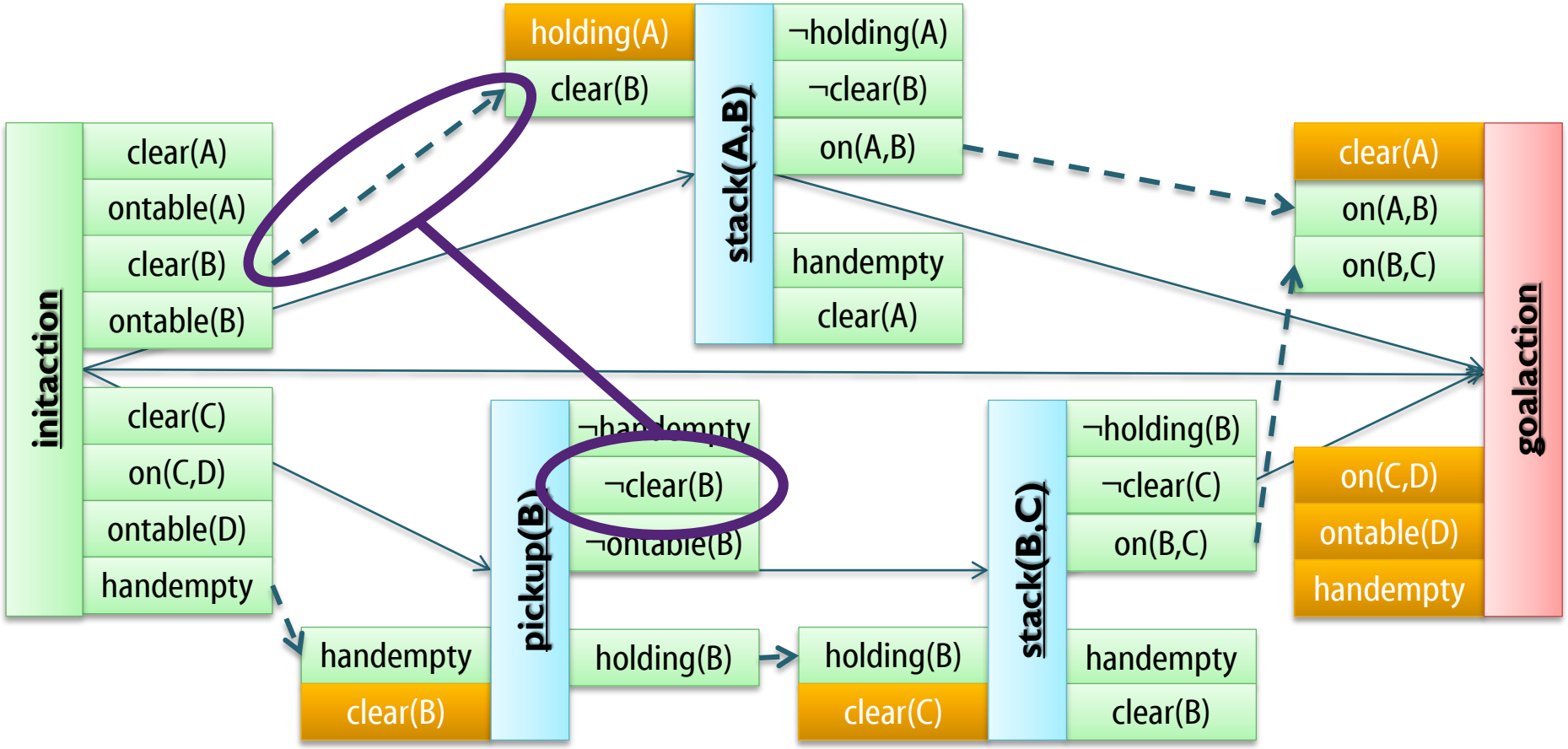
- Some pairs of actions can remain **unordered**



Threats

jonkv@ida

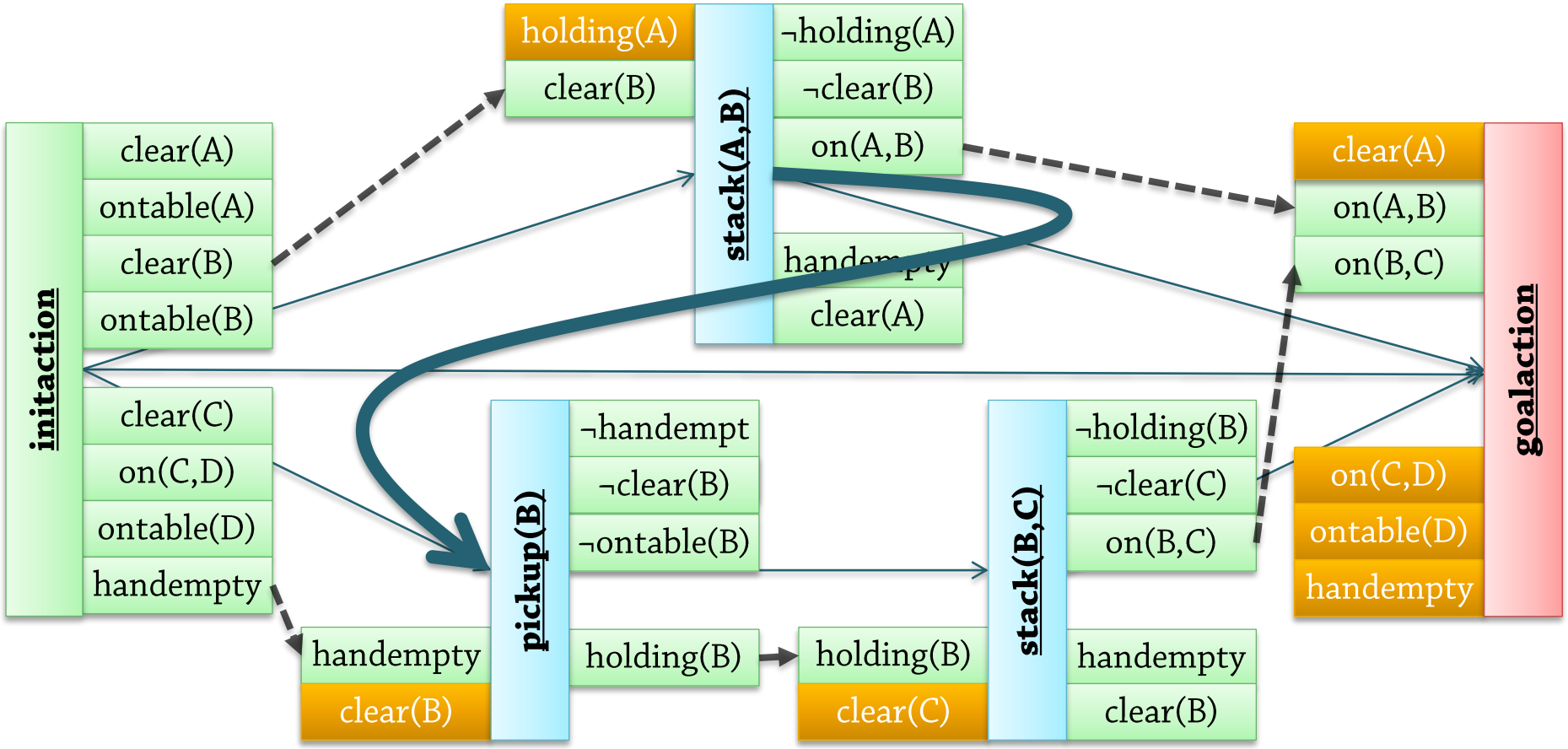
- Second flaw type: A **threat** to be resolved
  - initaction **supports** clear(B) for stack(A,B) – there's a **causal link**
  - pickup(B) **deletes** clear(B), and **may** occur **"during" the link**
  - We can't be certain that clear(B) still holds when stack(A,B) starts!



Resolving Threats 1

jonkv@ida

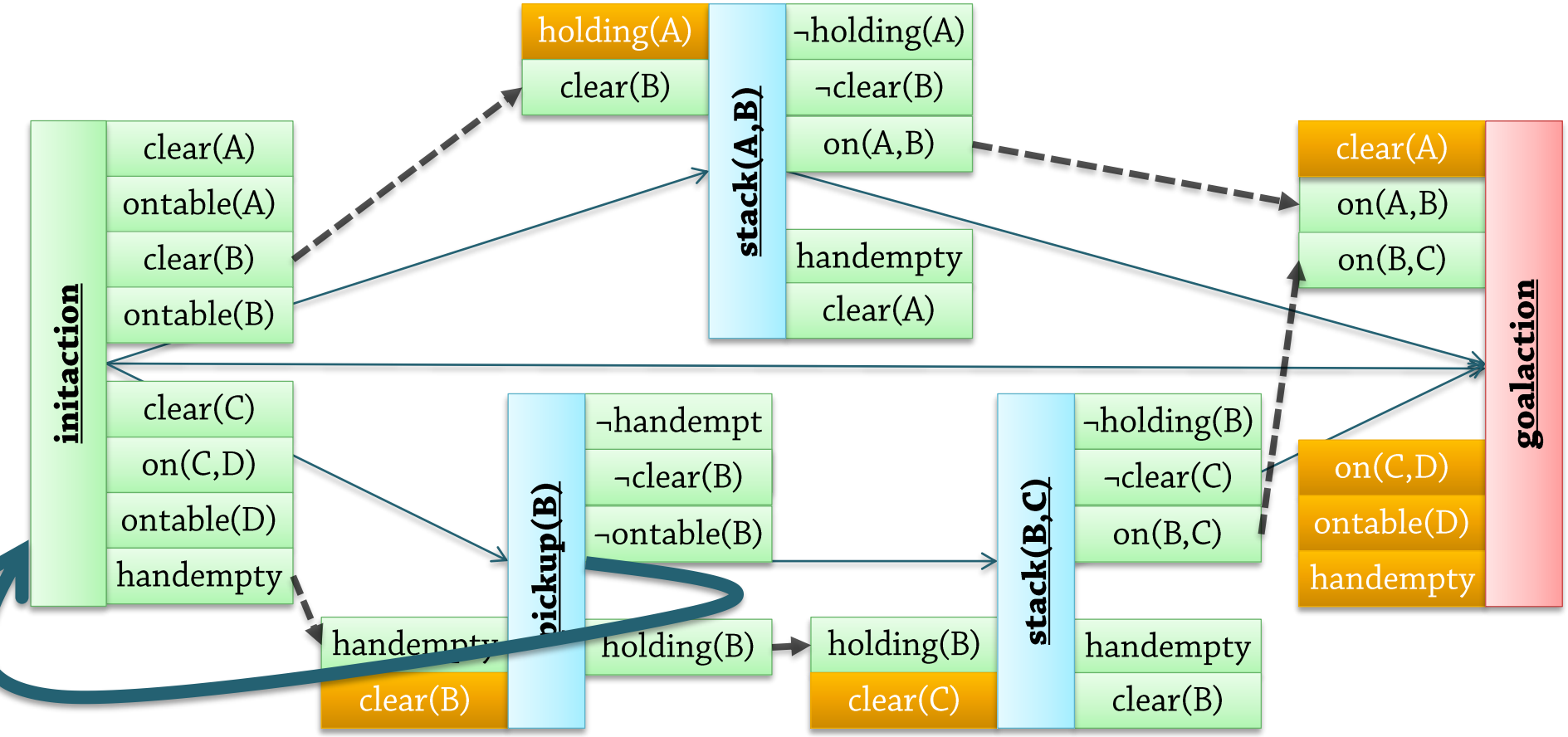
- How to **make sure** that clear(B) holds when stack(A,B) starts?
  - Alternative 1: The action that **disturbs** the precondition is placed **after** the action that **has** the precondition
    - Only possible if the resulting partial order is consistent (acyclic)!



Resolving Threats 2

jonkv@ida

- 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

jonkv@ida

- As usual, uninformed search is not sufficient
  - Heuristics are required

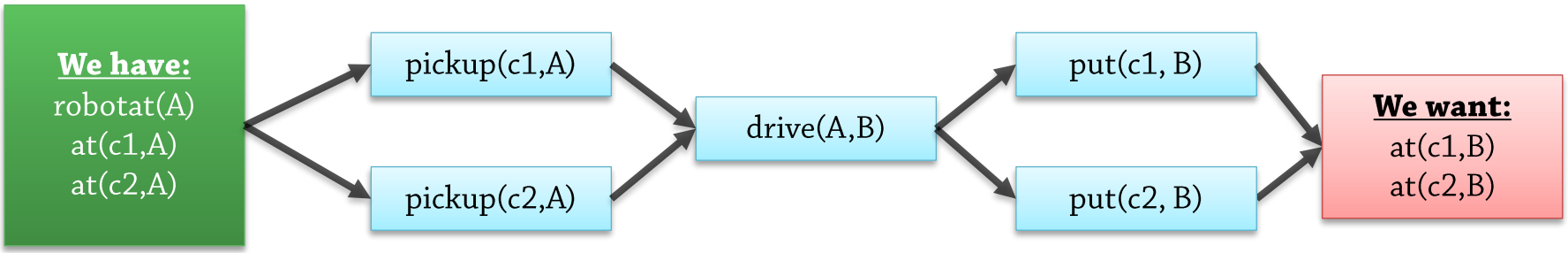
Remember what a flaw is!

- **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

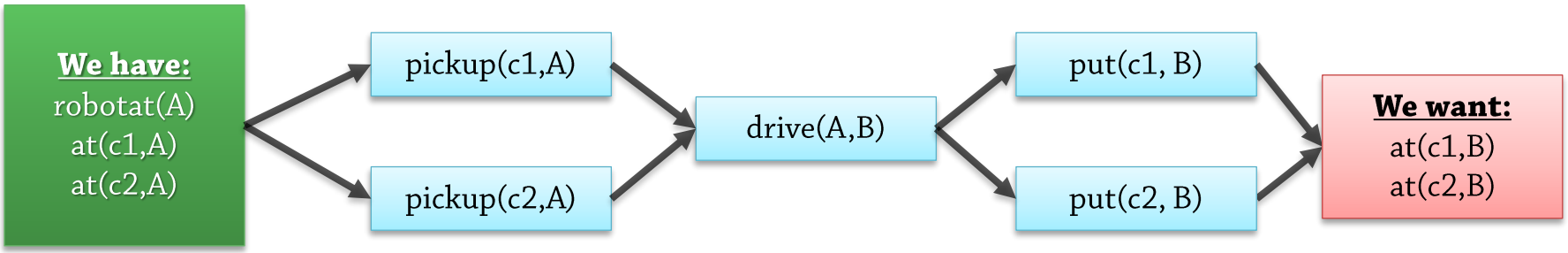
jonkv@ida

- 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

- 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: An Overview

Many alternative search spaces exist;
some examples here!

SAT Planning

jonkv@ida

- 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	–	

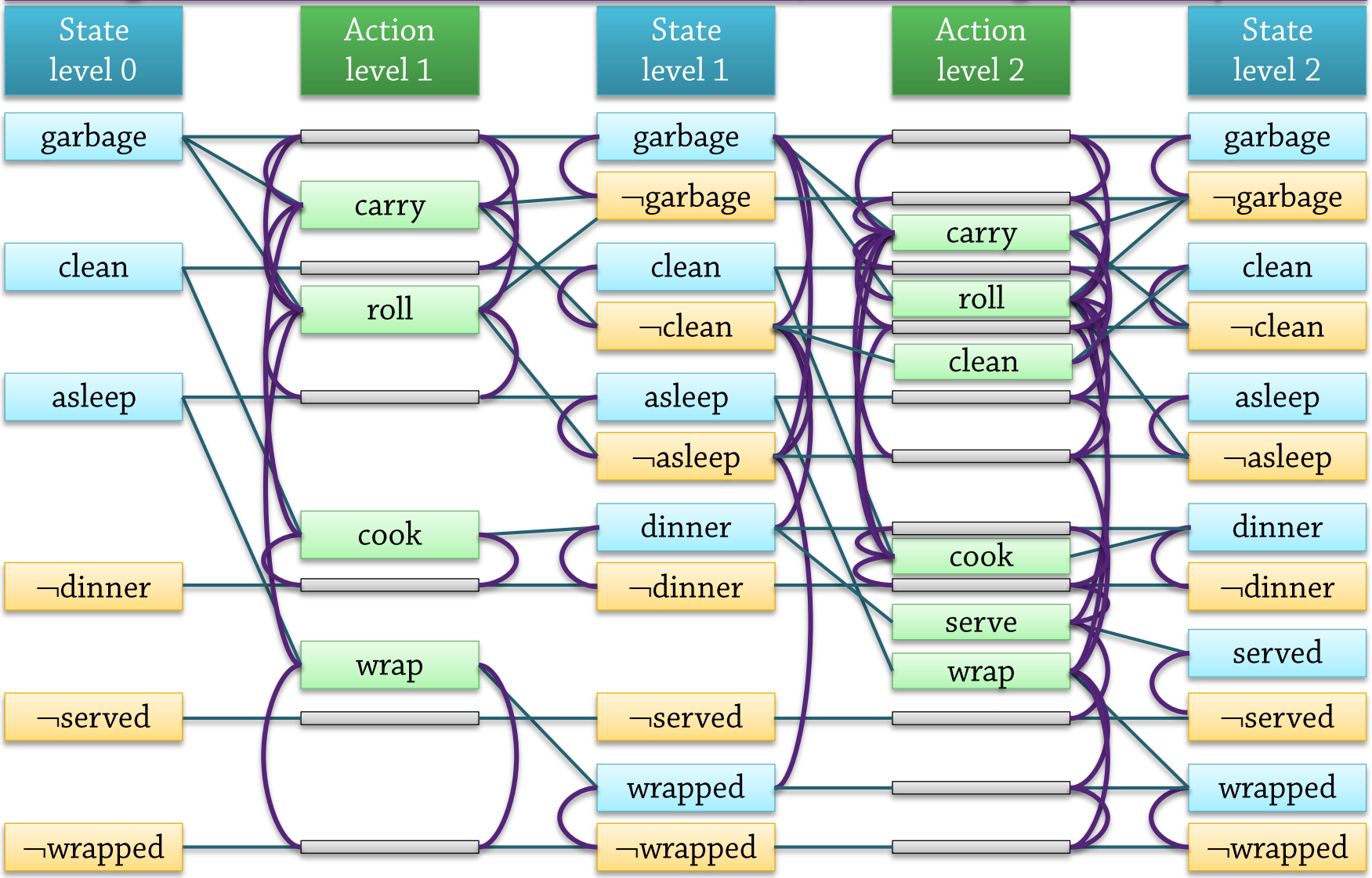
Seq. plan!

Seq. plan!

Planning Graphs

jonkv@ida

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



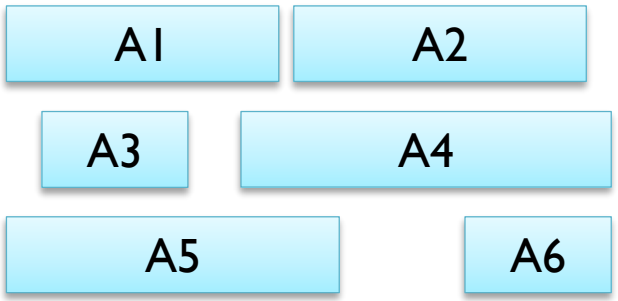
Beyond Classical Planning: Time, Concurrency, Resources

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)

jonkv@ida

- **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)

jonkv@ida

- 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)

jonkv@ida

- **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))))
```

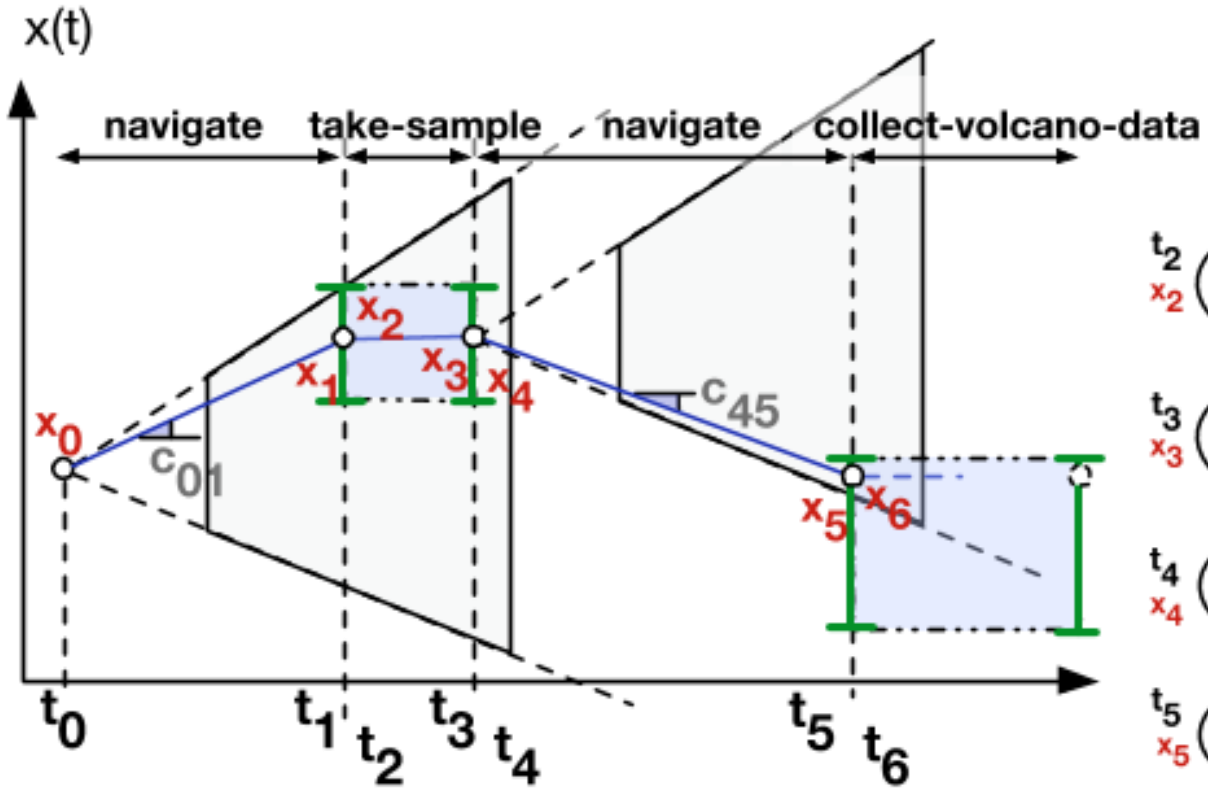
Can vary continuously during the action

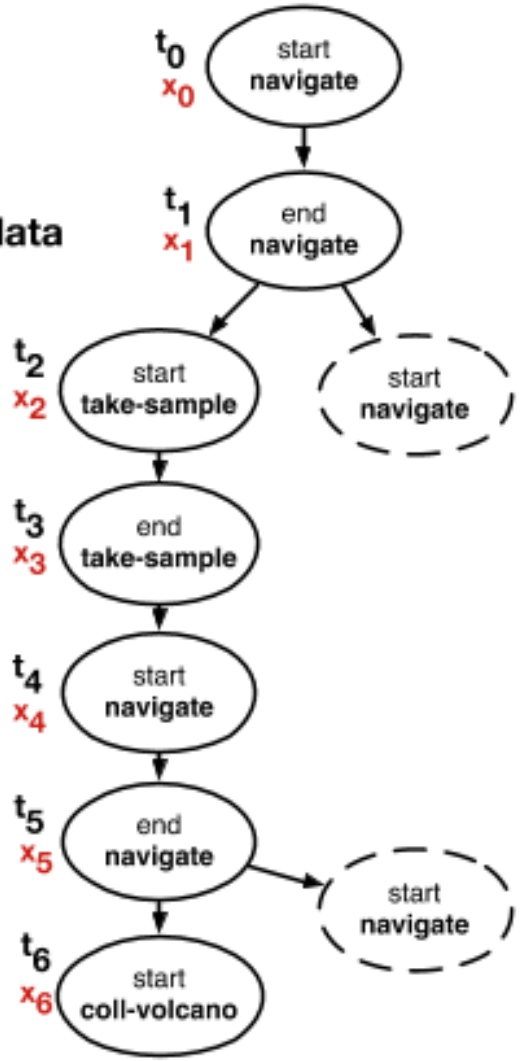
Continuous effects depend on the **instantaneous** values of control variables

Time, Concurrency and Resources (5)

jonkv@ida

- 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

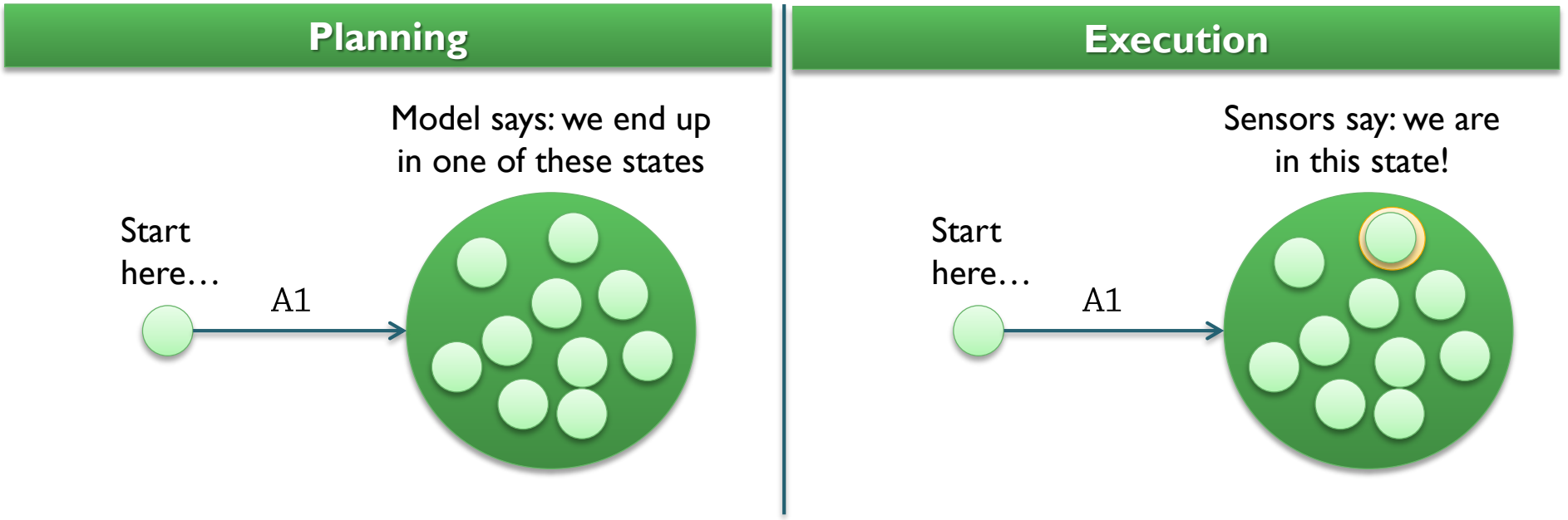
To a *planner*, there is generally no difference between these cases!

FOND Planning

jonkv@ida

- **Nondeterministic** planning:
  - Many possible outcomes for each action

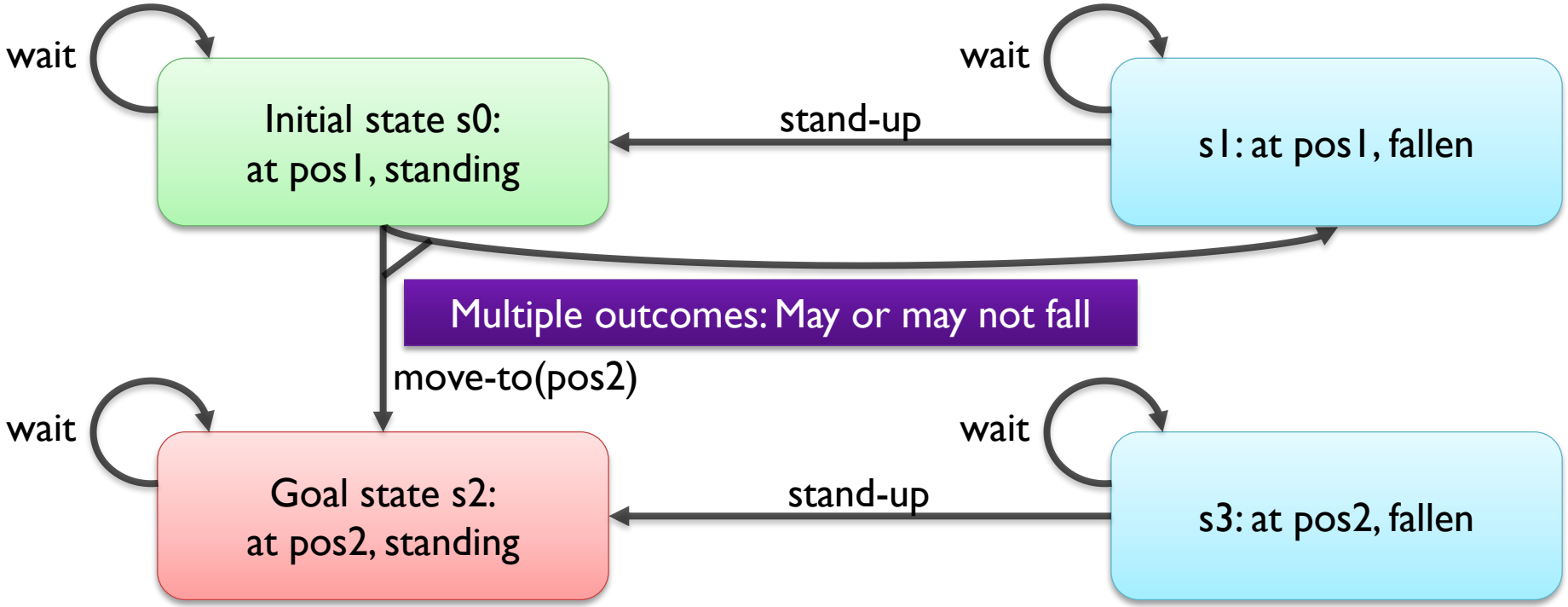
- FOND: **Fully Observable** Non-Deterministic
  - After executing an action, sensors determine exactly which state we are in



FOND Planning: Plan Structure (1)

jonkv@ida

- 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)

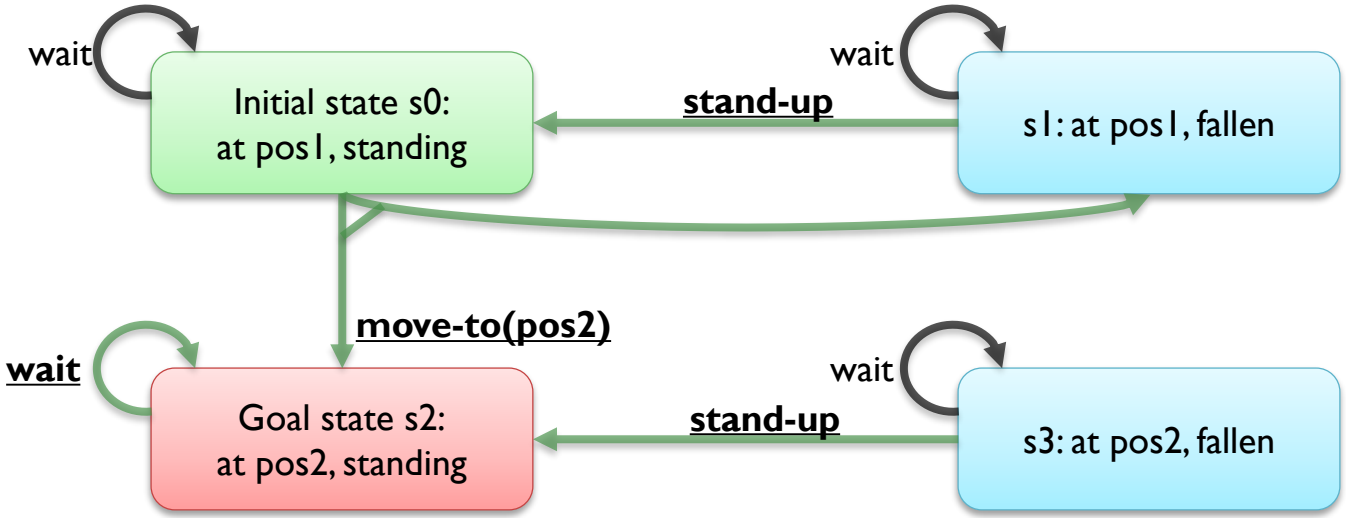
There may be no upper bound on how many actions we may have to execute!

FOND Planning: Plan Structure (2)

jonkv@ida

- 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*)

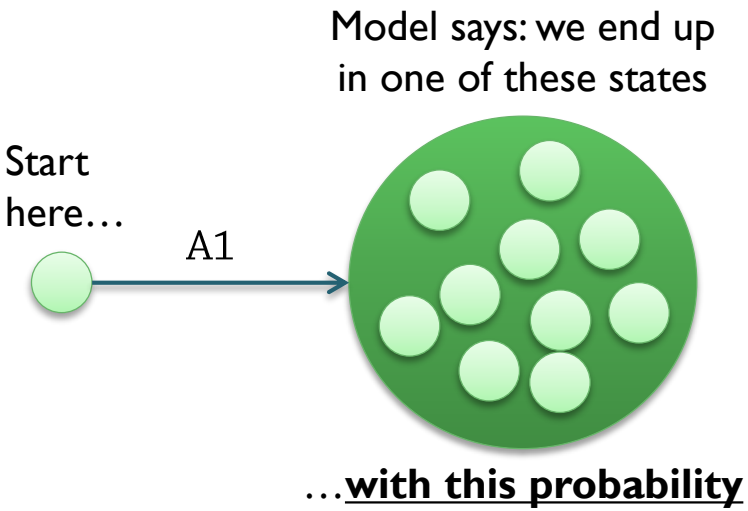


Stochastic Systems

jonkv@ida

- **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

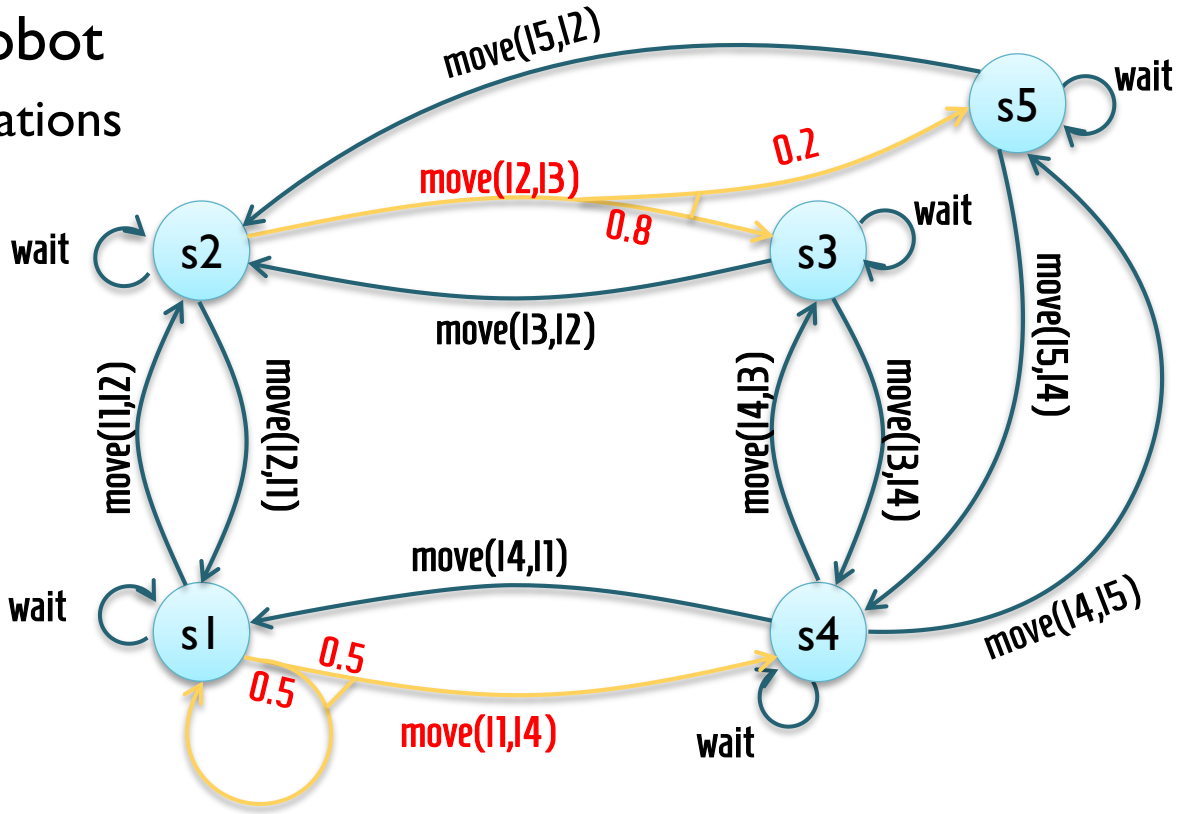
Planning



Stochastic System Example

jonkv@ida

- **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)

Overview

jonkv@ida

	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

jonkv@ida

- 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

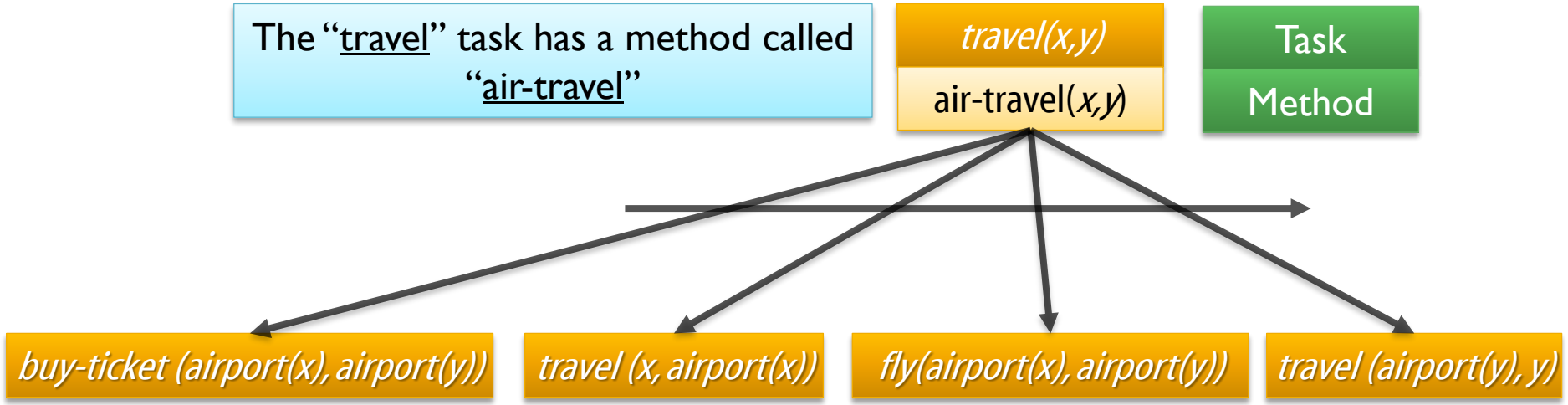
- Completely different **principles** for **heuristics**
  - Landmarks
  - Pattern databases

- 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

TDDD48 Automated Planning (2)

- 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…