Linköping University

TDDC17: Intro to Automated Planning

Classical Planning

Jonas Kvarnström
Artificial Intelligence and Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University

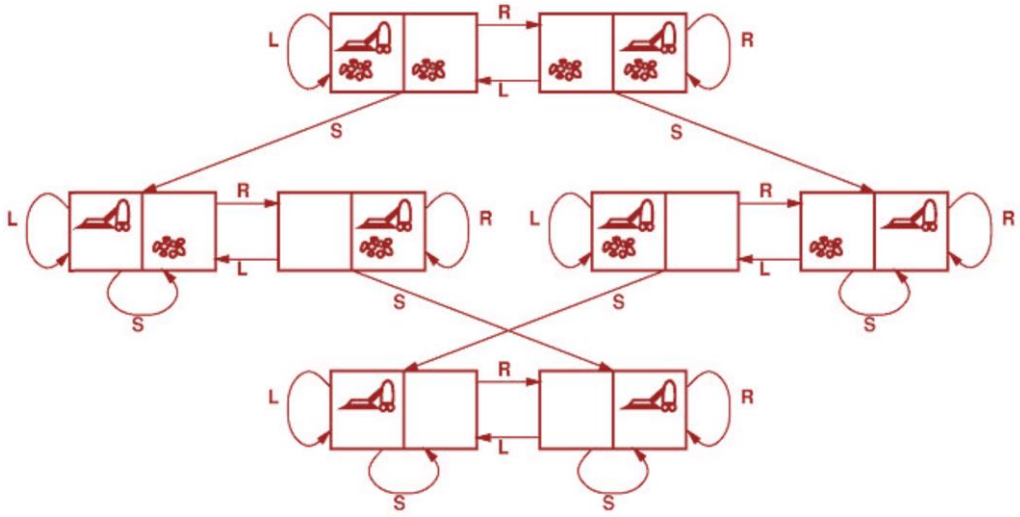
jonas.kvarnstrom@liu.se – 2021

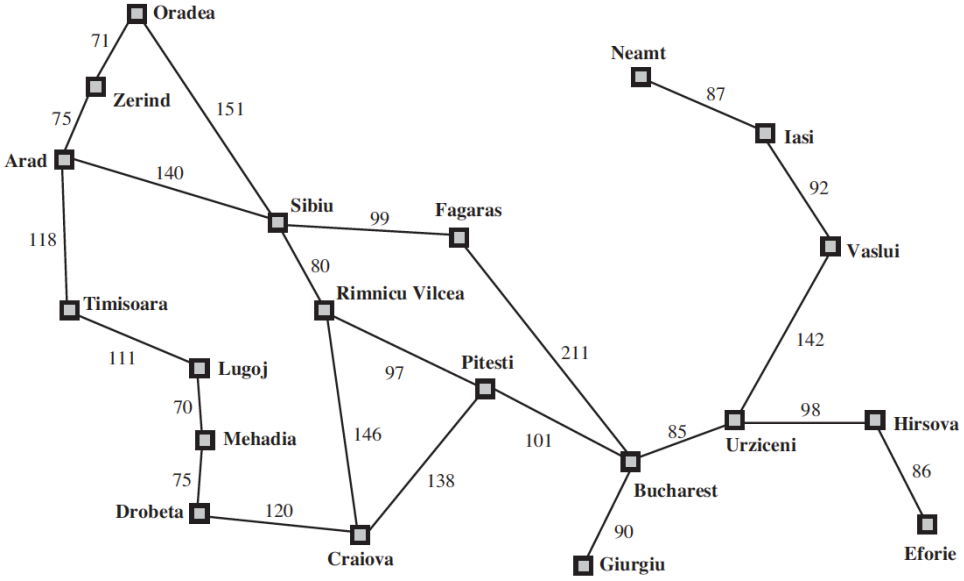
Introduction to Planning

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!

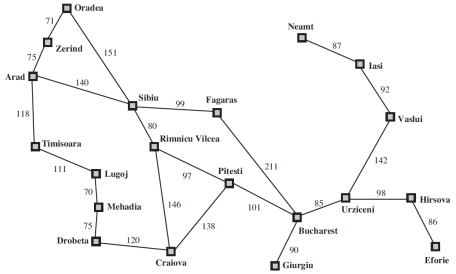


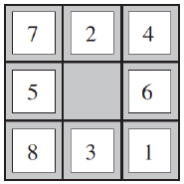


*Using **knowledge** about the world, including possible actions and their results, to **decide** what to do and when in order to achieve an **objective**, **before** you actually start doing it*

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 ⟷ **???**

Automated Planning: General heuristics

Pattern Databases, Landmarks

FF, h^m, merge-and-shrink, …

➔ More in **TDDD48**

Automated Planning: Entirely different search spaces

Partial Order Causal Link

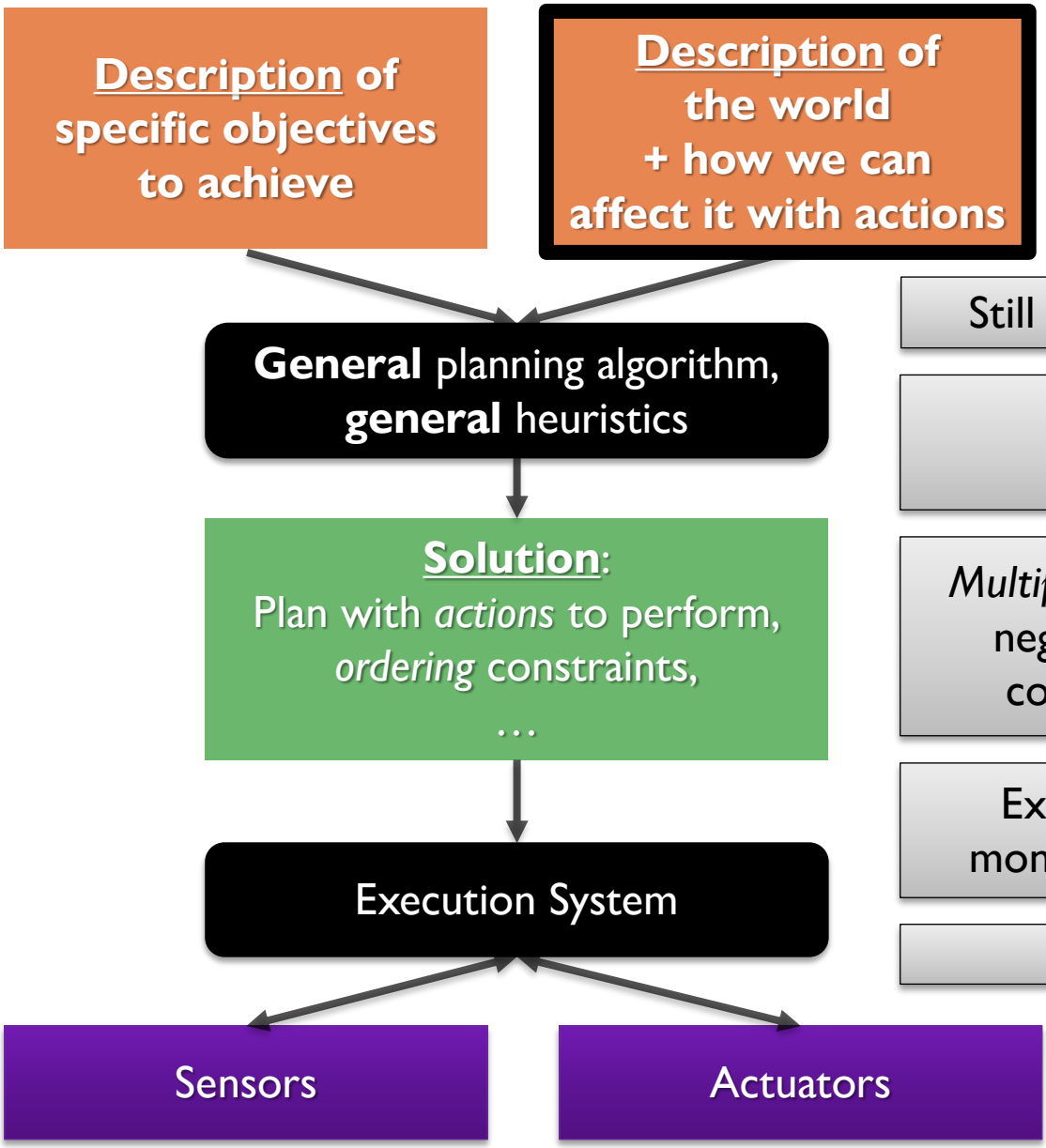
SAT planning, Planning Graphs, …

➔ More in **TDDD48**

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

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…

Modeling "Classical" Planning Problems

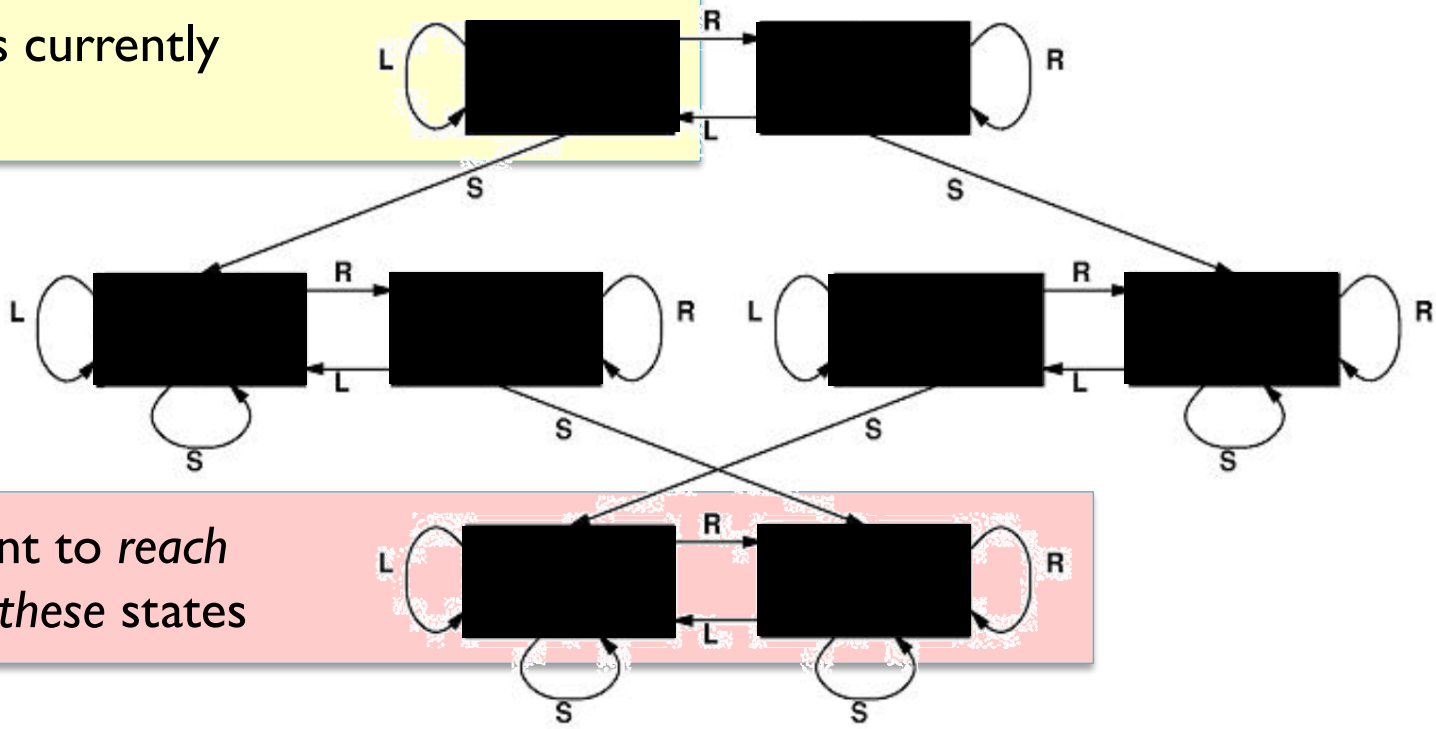
The basis for most extended forms of automated planning

Facts and States: Introduction

jonkv@ida

- Like 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 the information above sufficient?

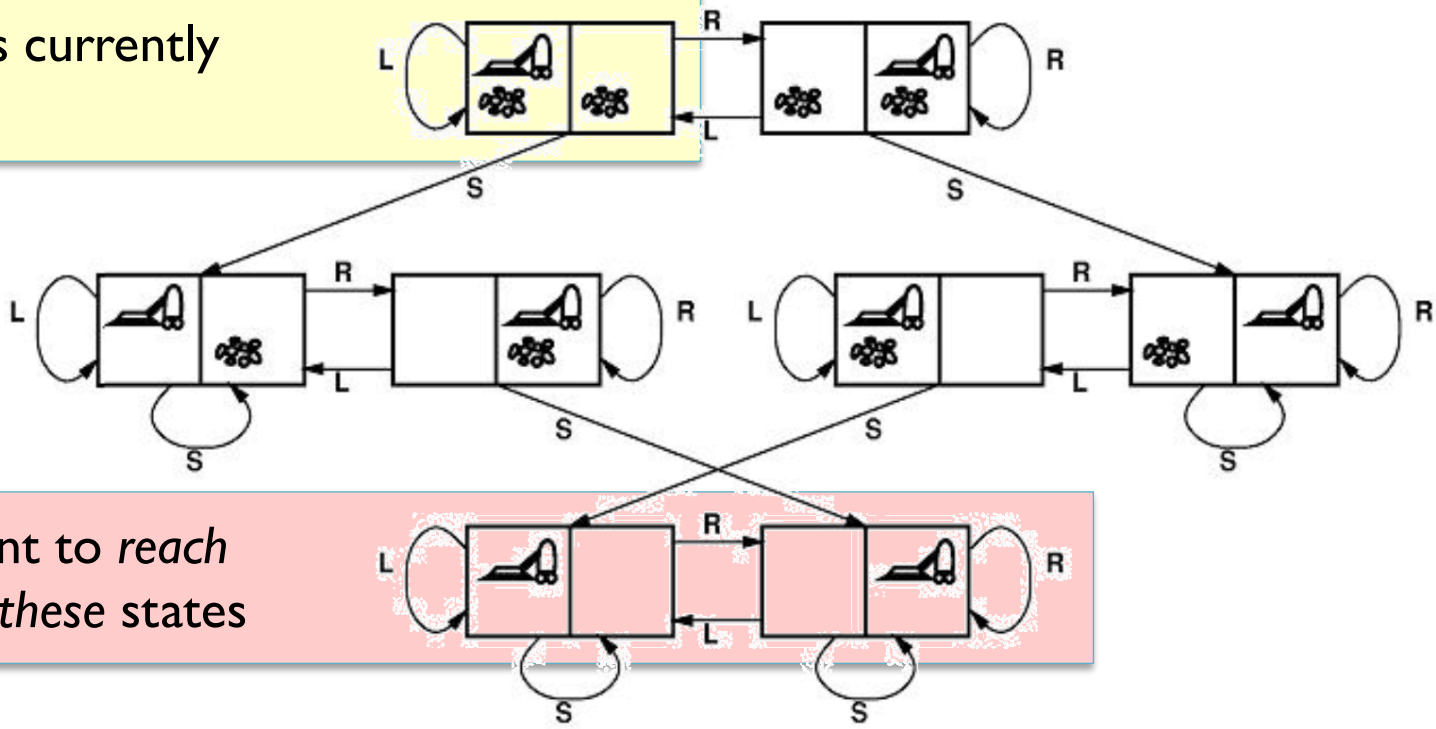
No – Need to ***analyze*** states,
find ***differences*** compared to goal states,
find ***relevant*** actions, …!

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

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

Let's use a representation based on first order logic!

Example: Dock Worker Robots (DWR)

jonkv@ida

Containers shipped in and out of a harbor

Cranes move containers between ”piles” and robotic trucks

Objects 1: Object Types

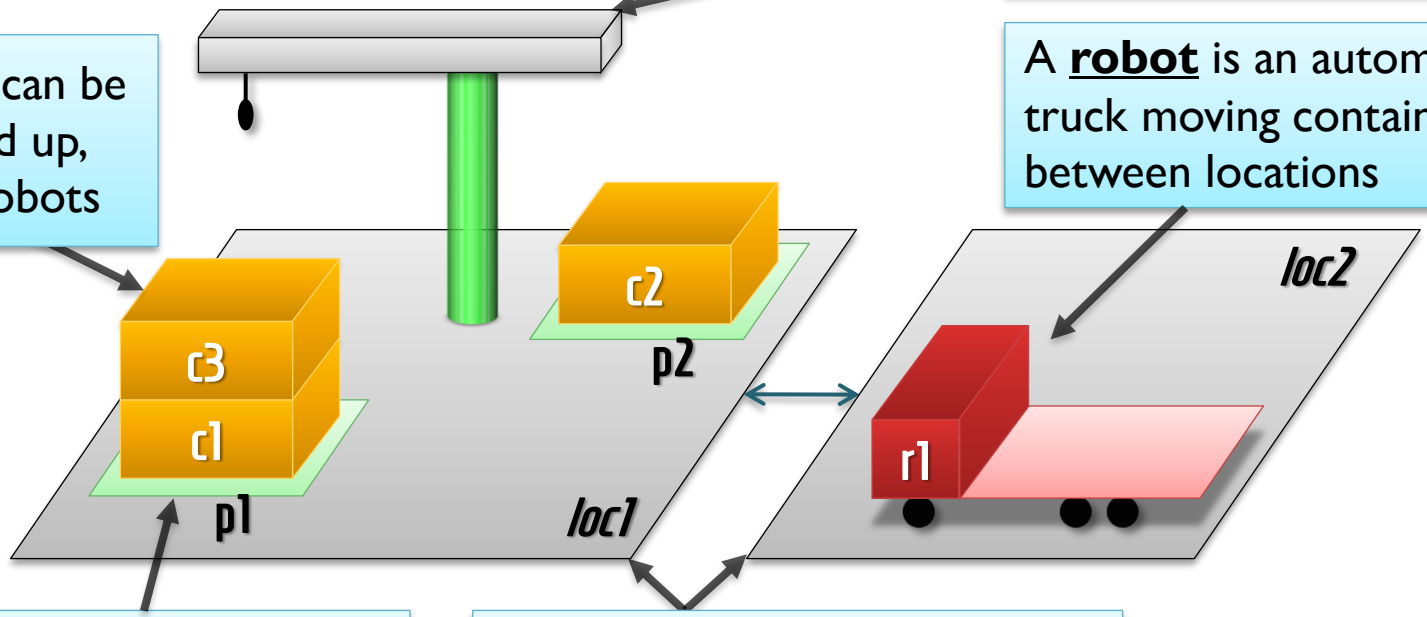
jonkv@ida

- Modern planners let us 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



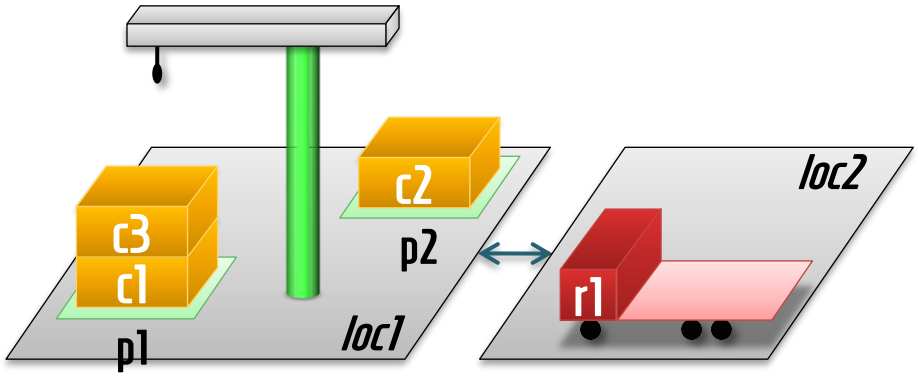
A **pile** is a stack of containers – at the bottom, there is a **pallet**

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

Objects 2: Actual objects

jonkv@ida

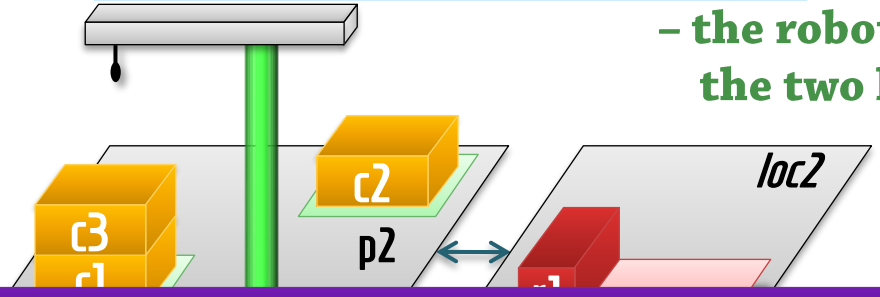
- We then specify sets of **actual objects**
  - **robot**: { r1 }
  - **location**: { loc1, loc2 }
  - **crane**: { k1 }
  - **pile**: { p1, p2 }
  - **container**: { c1, c2, c3, pallet }



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" (don'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

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

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

States 1b: How many ground atoms, states?

jonkv@ida

- If we have r robots, l locations, k cranes, p piles, c containers:
  - **adjacent** (*loc1*, *loc2*) ; l^2 ground atoms
    attached (*pile*, *loc*) ; pl ground atoms
    belong (*crane*, *loc*) ; kl ground atoms

    at (*rob*, *loc*) ; rl ground atoms
    occupied (*loc*) ; l ground atoms
    loaded (*rob*, *cont*) ; rc ground atoms
    unloaded (*rob*) ; r ground atoms

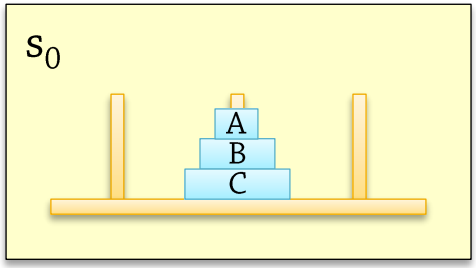
    holding (*crane*, *cont*) ; kc ground atoms
    empty (*crane*) ; k ground atoms

    in (*cont*, *pile*) ; cp ground atoms
    top (*cont*, *pile*) ; cp ground atoms
    on (*cont1*, *cont2*) ; c^2 ground atoms
- So:
  - $(l + p + k + r + 1)l + (c + 1)r + (c + 1)k + (c + 2p)c$ ground atoms
  - $2^{(l+p+k+r+1)l+(c+1)r+(c+1)k+(c+2p)c}$ states

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 **ground 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

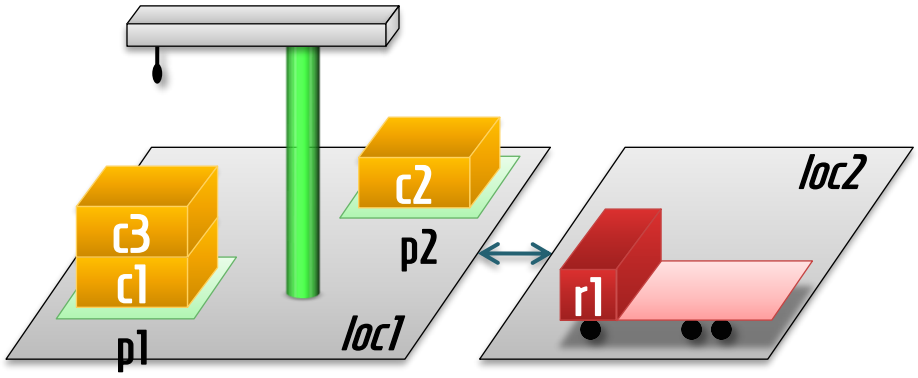
States 3: Initial State

jonkv@ida

- Initial states *in classical 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 still 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),
      }



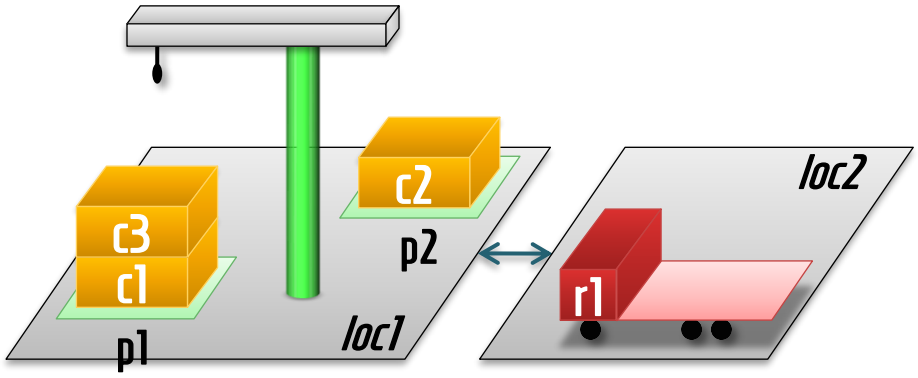
States 4: Goal States

jonkv@ida

- Classical planning: Reach one of possibly many **goal states**
  - Can be specified as a **set of literals** that must hold

Literals = positive or negated atoms

  - Example: **Containers 1 and 3 should be in pile 2; container 12 should not be in pile 2**
    - We don't care about their order, or any other fact
    - **{ in(c1, p2),**
      in(c3, p2),
      ¬in(c12, p2) }



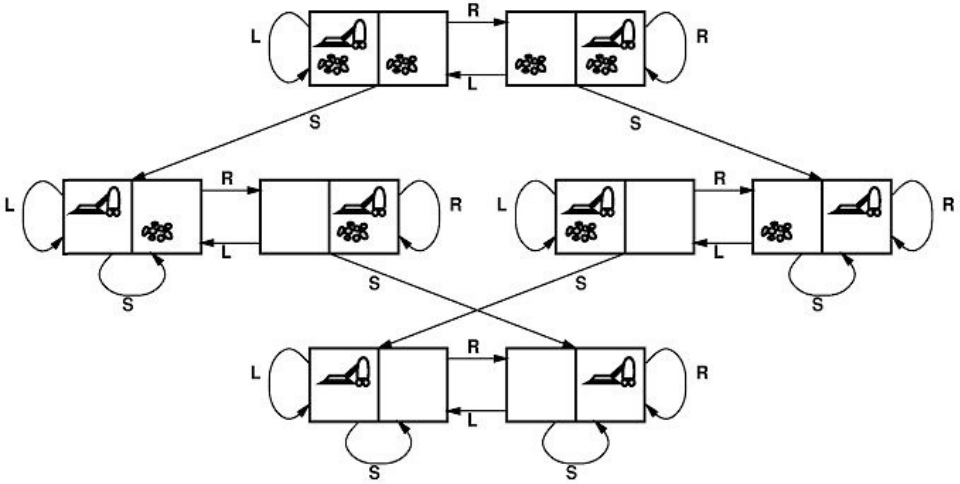
Actions, Operators

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**?

Actions 2: Operators

jonkv@ida

- Define **operators** or **action schemas**:
  - **move**(*robot*, *location1*, *location2*)
    - Precondition ***tests facts***, depending on ***parameters***:

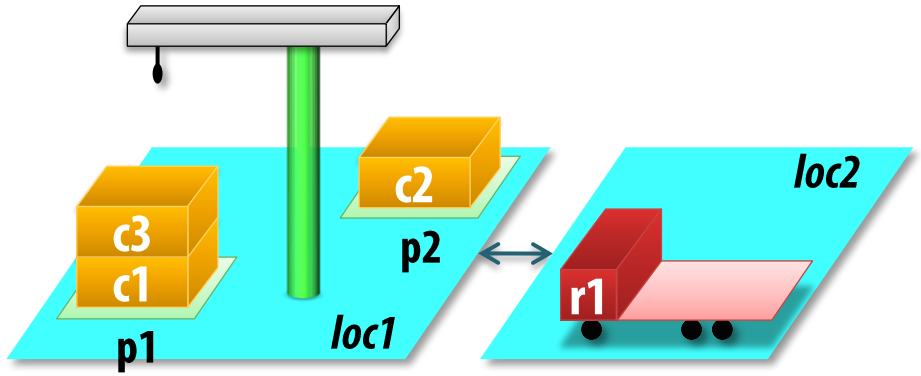
      at(*robot*, *location1*) ∧
      adjacent(*location1*, *location2*) ∧
      ¬ **occupied**(*location2*)

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

    - Effects ***add or remove facts***:

      ¬**at**(*robot*, *location1*),
      at(*robot*, *location2*),
      ¬**occupied**(*location1*),
      occupied(*location2*)

      The result of applying the action in state *s*:
      s – {negated effect facts}
      + {positive effect facts}



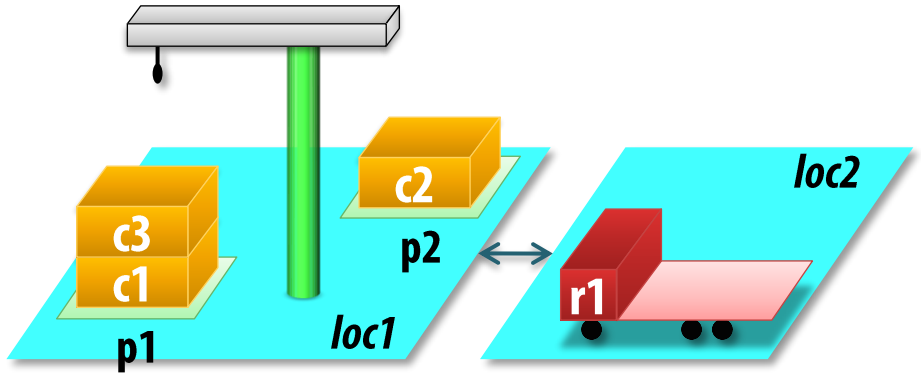
Actions 3: Instances

jonkv@ida

- The planner **instantiates** the schemas
  - Applies them to objects 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)



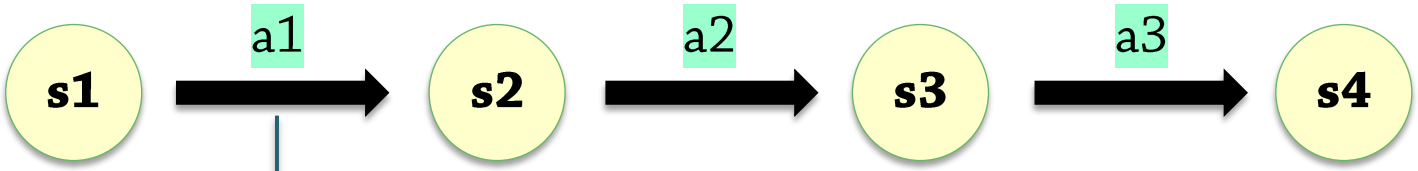
Actions 4: Step by Step

jonkv@ida

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

We know the initial state

Each action corresponds to **one** state update



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
  - 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?)

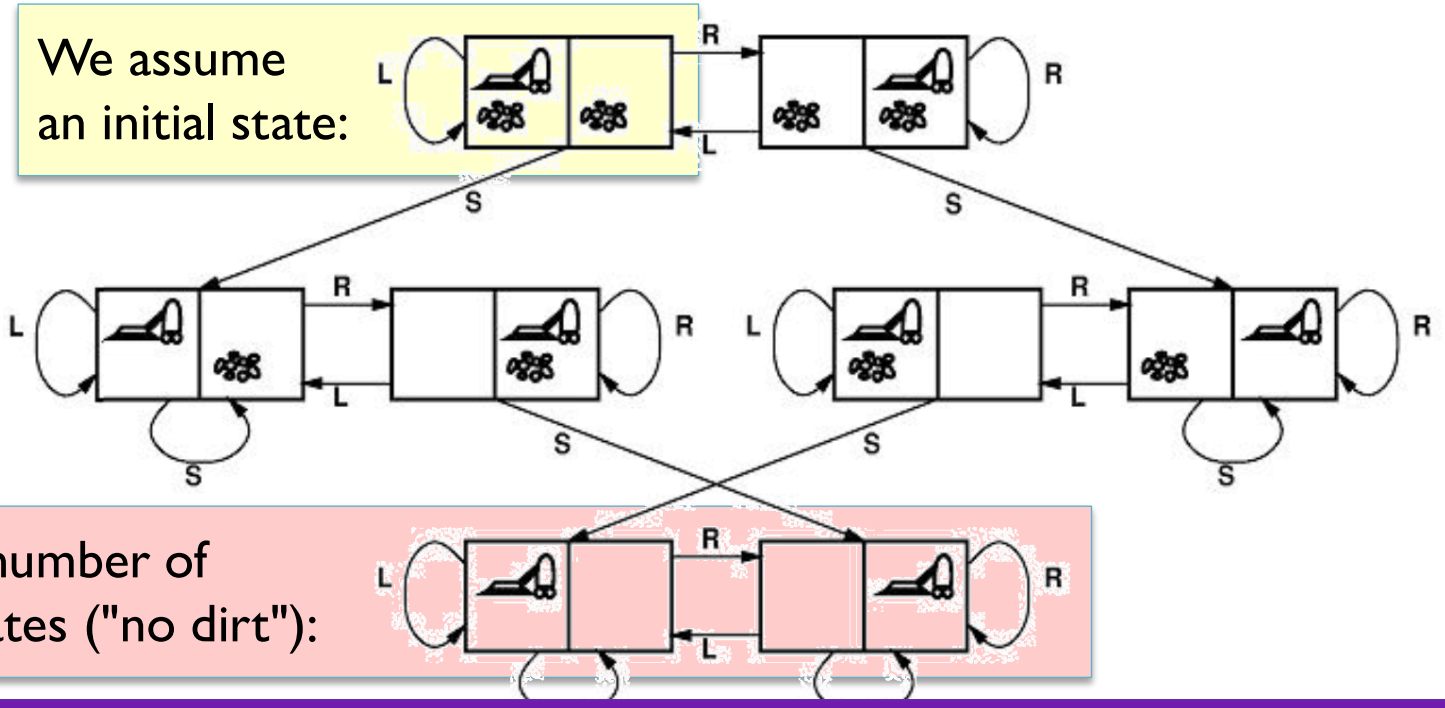
The State Space

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: Intuitions?

jonkv@ida

- Now that we have a general model of **facts:**
  - **Every** combination of **facts** is a state
    - { at(robot1,loc1), at(robot1, loc2) }
    - { adjacent(loc1, loc2) }
      [but *not* adjacent(loc2, loc1)!]

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

- But our **intuitions** often identify states that **we** think are:
  - "Normal"
  - "Expected"
  - "Physically possible"

- Usually:
  - The **initial state** is "normal"
    - We *never specify* { at(robot1,loc1), at(robot1, loc2) }
  - Preconditions/effects ensure that we can only **reach** other "normal" states
  - Mainly need to care about "normal" states… so let's focus on those!

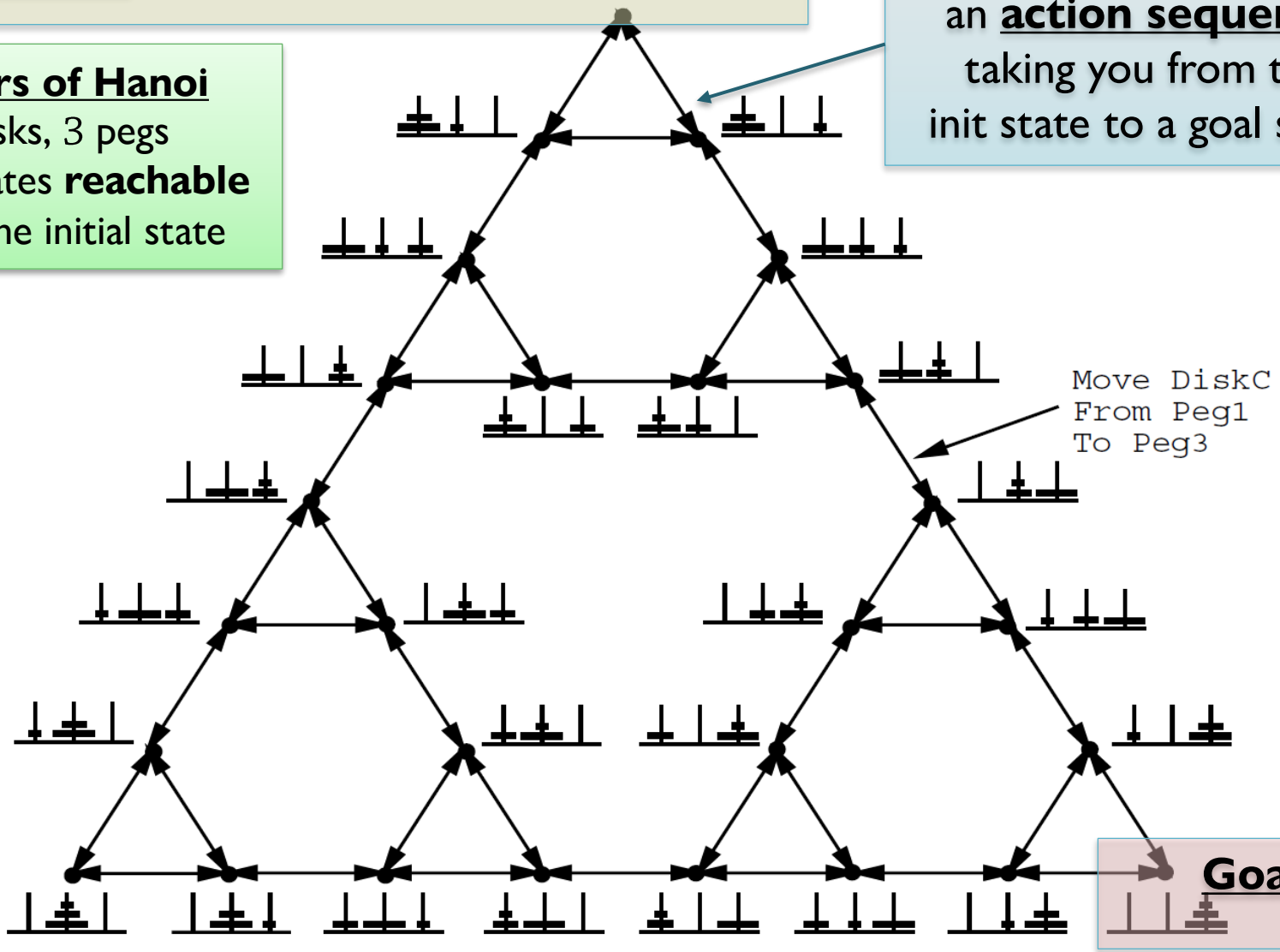
State Spaces 3: ToH, Actions

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

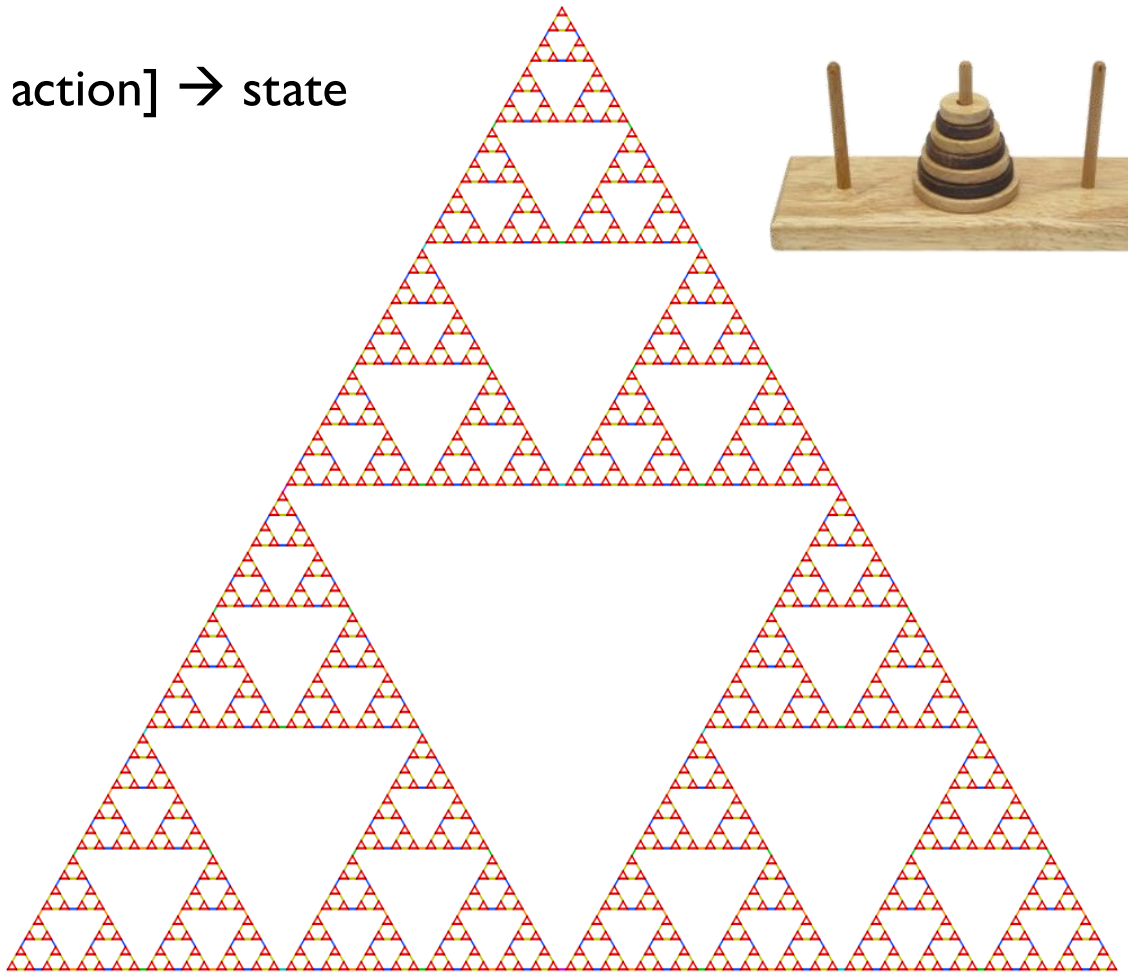


Goal state

State Spaces 4: Larger Example

jonkv@ida

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

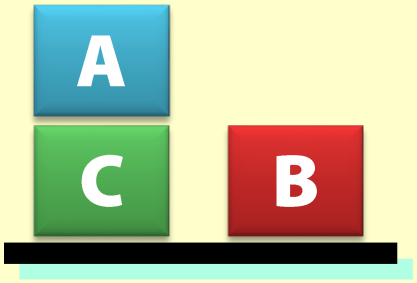


State Spaces 5: Blocks World

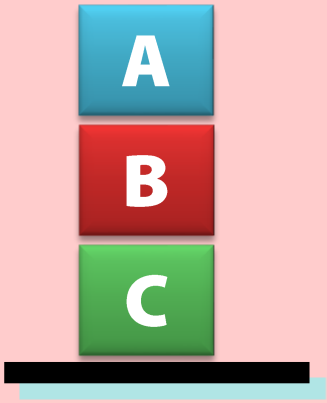
jonkv@ida

You

Current state of the world



Your greatest desire



Reference: 4 operators

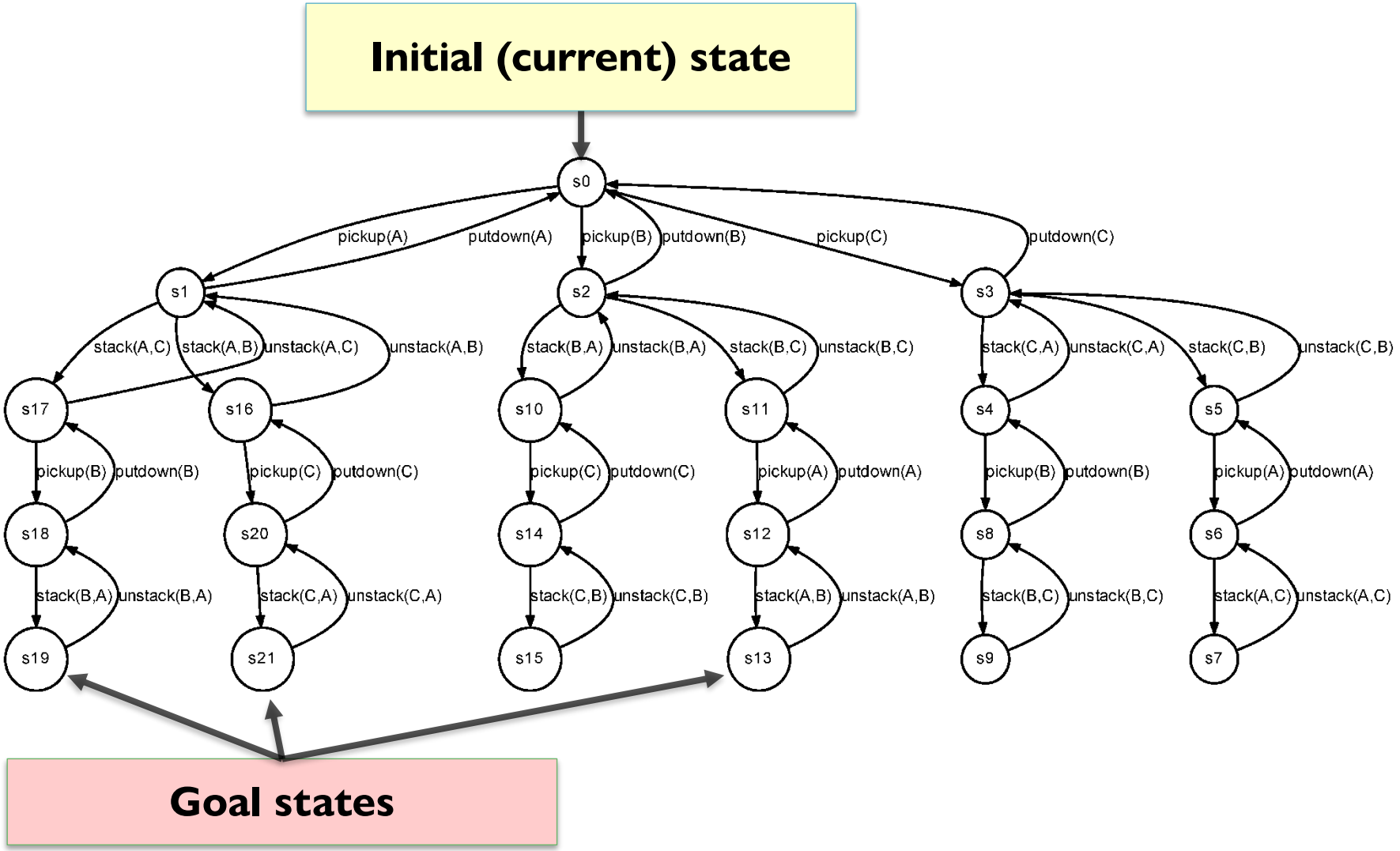
- pickup(x) – takes x from the table
- putdown(x) – puts x on the table
- unstack(x, y) – takes x from on top of ?y
- stack(x, y) – puts x on top of y

Reference: 5 predicates

- on(x, y) – block x is on block y
- ontable(x) – x is on the table
- clear(x) – we can place a block on x
- holding(x) – the robot is holding block x
- handempty – the robot arm is free

State Spaces 6: Blocks World, 3 blocks

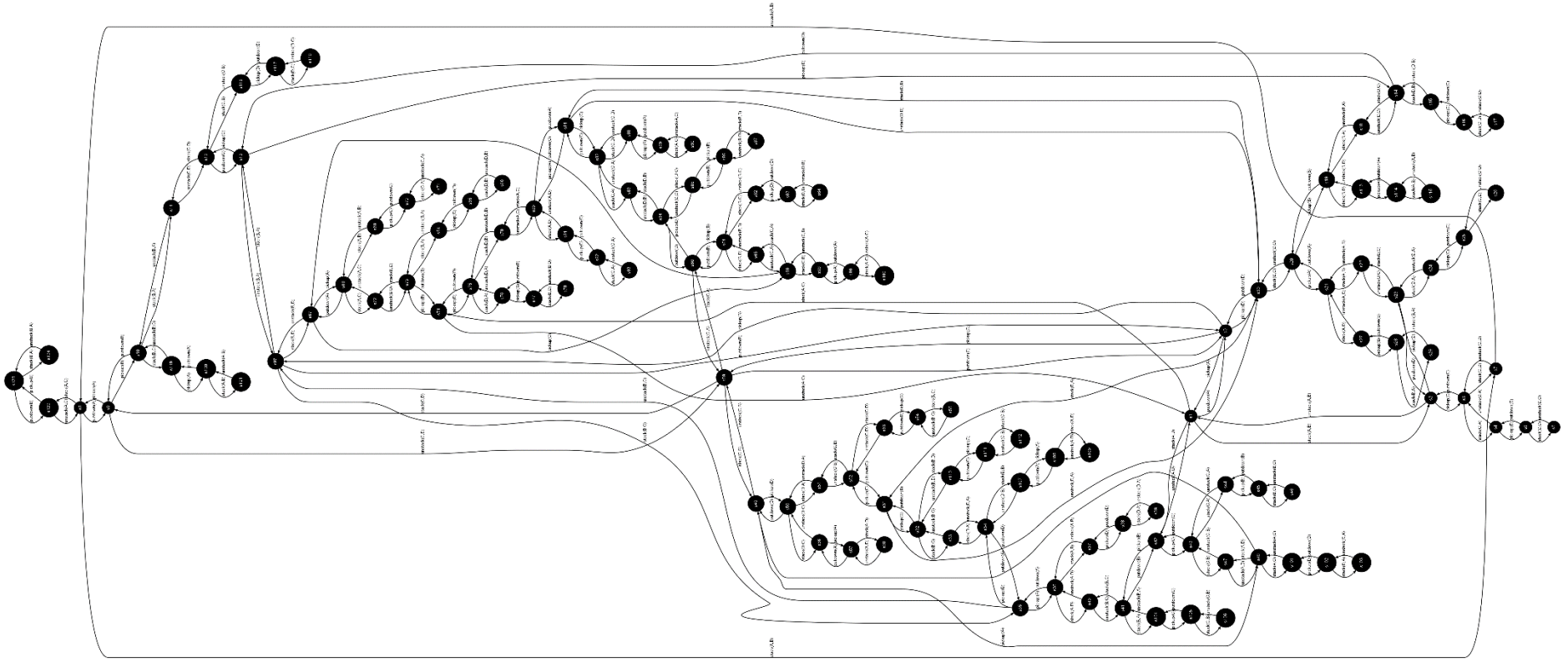
jonkv@ida



State Spaces 7: Blocks World, 4 blocks

jonkv@ida

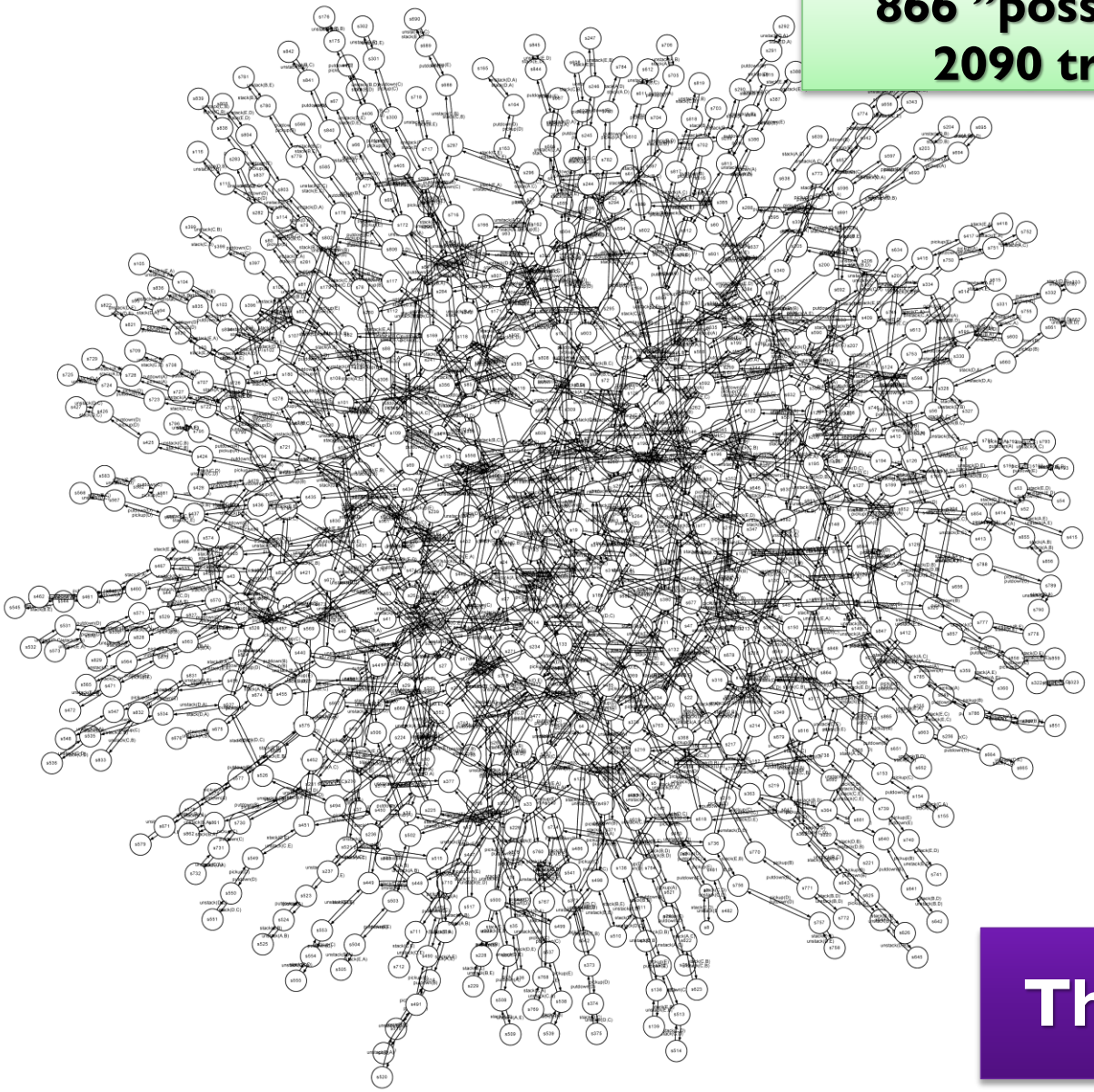
125 ”possible” states
272 transitions



State Spaces 8: Blocks World, 5 blocks

jonkv@ida

866 "possible" states
2090 transitions



This is tiny!

State Spaces 9: Reachable States

jonkv@ida

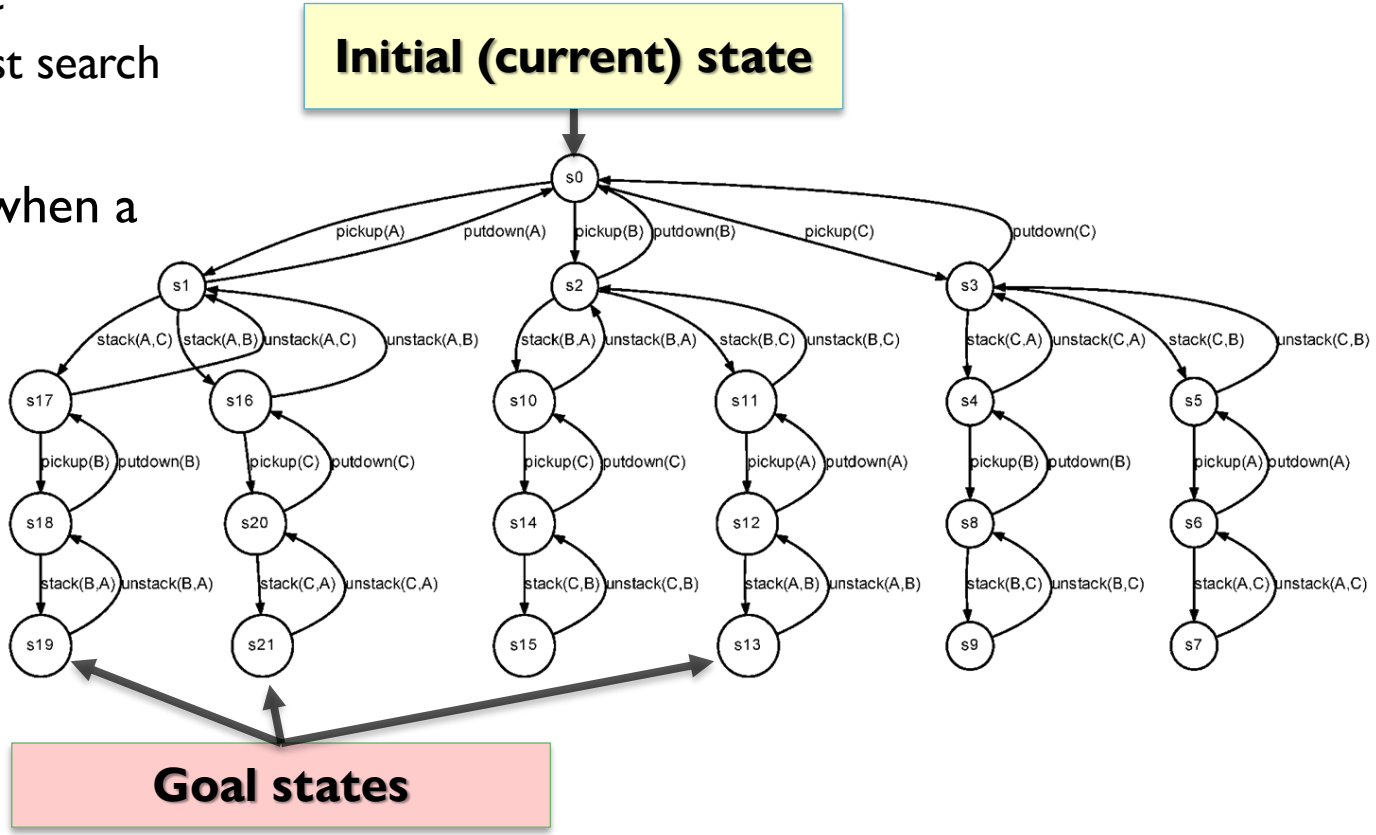
Blocks	States reachable from "all on table"	Transitions (edges) in this part of the space
0	1	0
1	2	2
2	5	8
3	22	42
4	125	272
5	866	2090
6	7057	18552
7	65990	186578
8	695417	2094752
9	8145730	25951122
10	…	…
…30	>197987’401295’571718’915006’598239’796851	

Plan Generation Method 1: Forward State Space Search

Forward Search 1

jonkv@ida

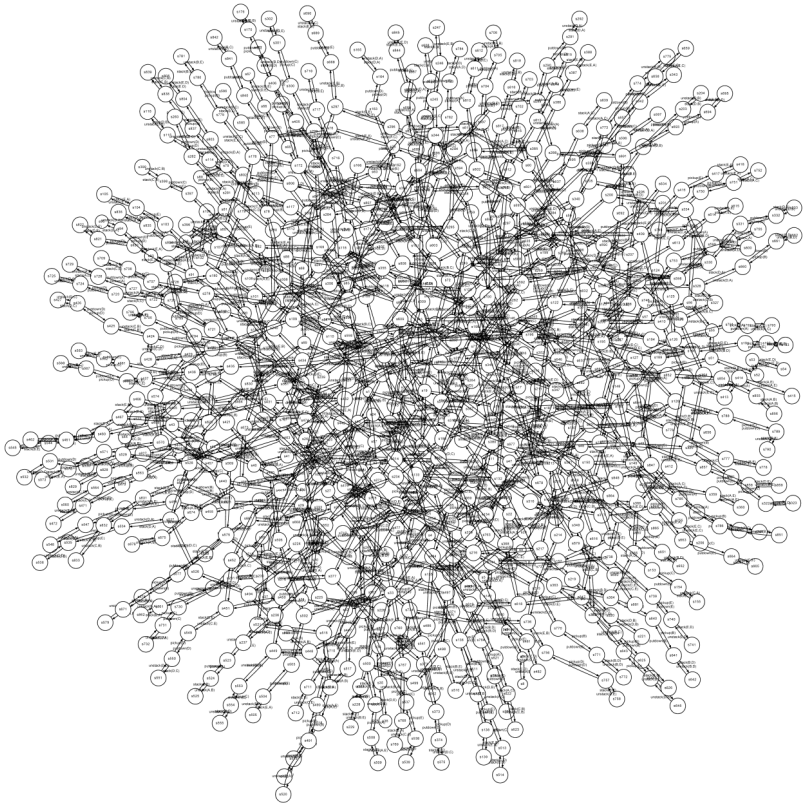
- Straight-forward planning: **Forward search in the state space**
  - **Start** in the initial state
  - Apply a **search** algorithm
    - Depth first
    - Breadth first
    - Uniform-cost search
    - …
  - **Terminate** when a goal state is found



Forward Search 2: Don't Precompute

jonkv@ida

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

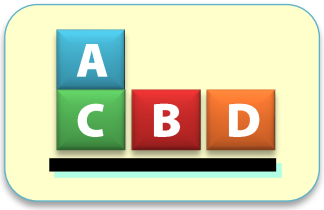


Usually too large!
➔ Generate as we go,
hope we don't actually need the *entire* graph

Forward Search 3: Initial state

jonkv@ida

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



- 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
  - Here we show the path used to reach each 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 }

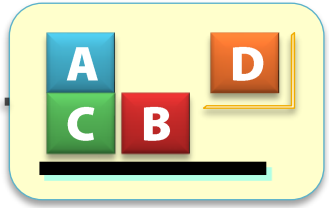
- …we can find **successors** – by applying **actions**!
  - **action** pickup(D)
    - Precondition: **ontable**(D) ∧ **clear**(D) ∧ **handempty**
      Effects: ¬**ontable**(D) ∧ ¬**clear**(D) ∧ ¬**handempty** ∧ **holding**(D)

- This generates **new reachable nodes/states**…

…which can also be illustrated

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

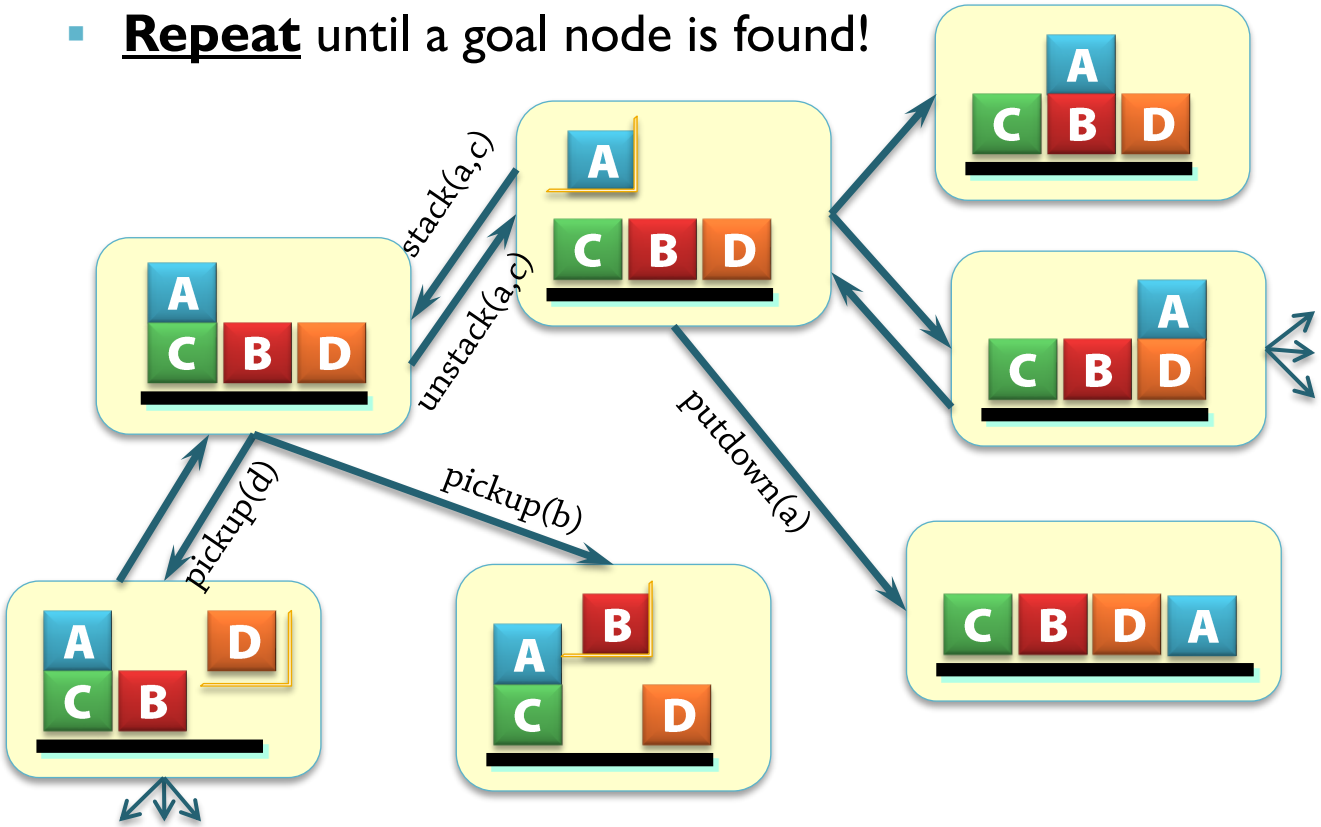
[pickup(D)] ➔ { clear(A), on(A,C), ontable(C), clear(B), ontable(B), holding(D) }



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"

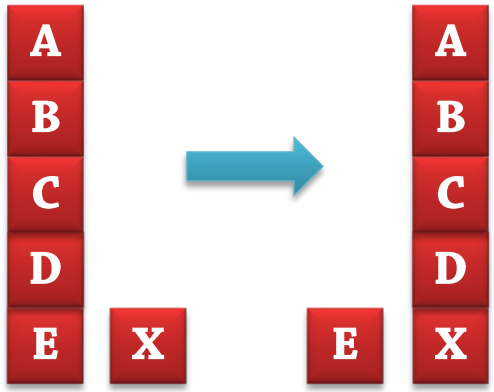
Not true for all domains!

Uninformed or Informed
Forward State Space Search?

Uninformed Search

jonkv@ida

- Can we use **uninformed** search algorithms?
  - With only 30 blocks, we have >1979874012955717189150065982397968 51 reachable states
  - **But what if we don't need to explore all the states?**
    - Suppose we need to tear down a 400-block tower and build it up on another "base"
    - Suppose we want **good** plans
      ➔ use a shortest-path algorithm such as Dijkstra's / Uniform Cost Search
    - Will explore **all** plans of lower length/cost than the optimal one



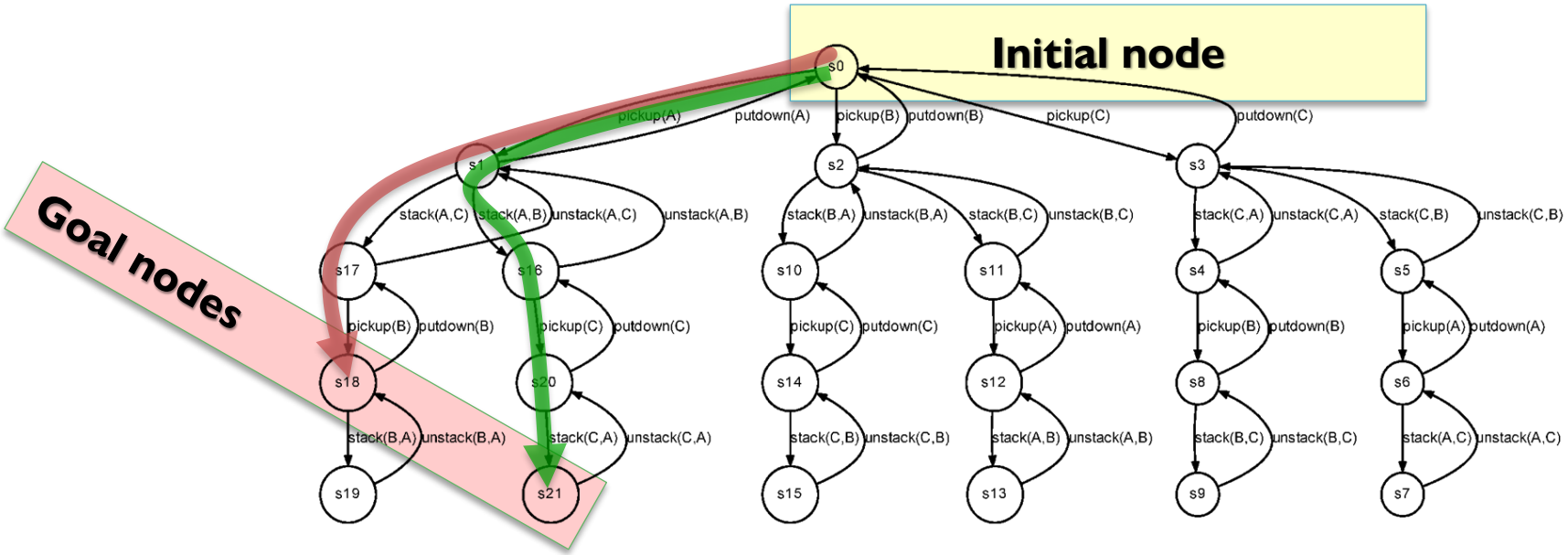
- Plans to test: More than…

1630569839078931058645796793734728775645948416347826722586241976230426399420799766425821395576658116365413711816311922048822638316916164832
0459490283410635798745232698971132939284479800304096674354974038722588873480963719240642724363629154726632939764177236010315694148636819334 2
1725283641400148727761800296660876103701808776949061484788741874440260622613480393693523356841805595037118535183714054851594943130931387521 0
8278889433371136136609283180862996179538929537220067341589332765764704756406073917010260309590403035481742212740523295796377736587224525497 3
8459404452586503693169340418435407383263781602533940396297139180912 6992992317773749830375100 7
1027115784612583228566467641071085488265744484456318793090777966151 4729483460905459057110164 2
4654413723505687486652490219918497606469880316913943865511941711931 4413026494323056202155688 5
0657684229678385177725358933986112127352452988033775364935611164107 7508073225578630777685901 6
3743554145844083387870934417498397743743032755753441762912244883519 1360782220646563527271107 3
9066118003761944104289000710136954383590946416822538563947433356785 7551101672548547661886191 2
8917053933547098435020659778689499606904157077005797632287669764145 0879307727141093526534328 6
7136000209692448349430242464906145172664594758586010497684553450747960540890382832020613107221778215643420457243461604240437521105232403822 5
8054057131573291598463519312655627310960393718822950440 0

Informed Search

jonkv@ida

- **We need guidance!**
  - For example, 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!**

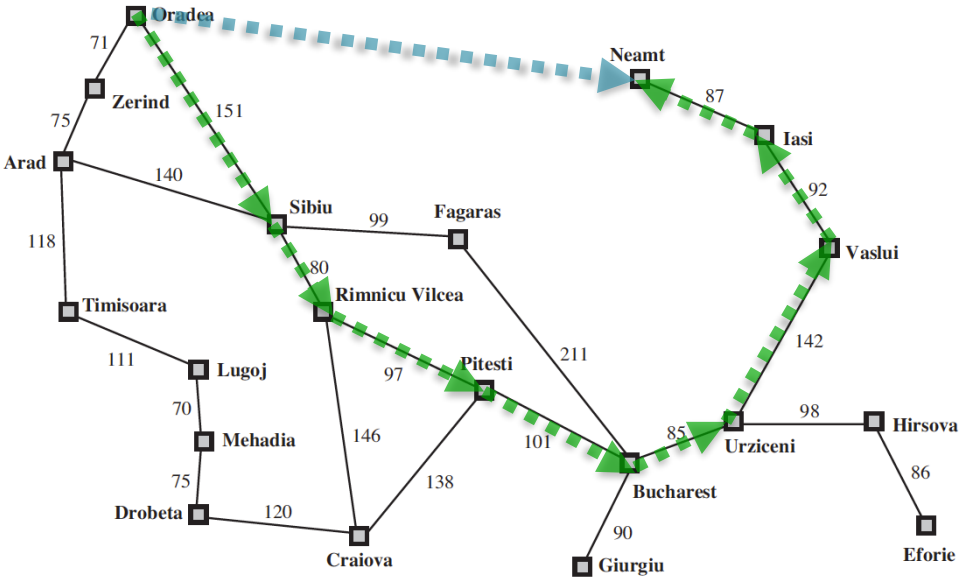


Informed Search (2)

jonkv@ida

- Previously we **manually designed** heuristics for a 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 to define **general** heuristic functions
  - *Without knowing what planning problem is going to be solved!*

Informed Search: Perfect Information?

jonkv@ida

- Given a **planning problem instance** and a **current state** s:
  - $\pi^*(s)$ denotes an **optimal solution** starting in s
  - $h^*(s) = \text{cost}(\pi^*(s))$ denotes the **cost** of an optimal solution

- ➔ h^* would be the "**perfect heuristic**"
  - Admissible – cannot overestimate
  - Informative – perfect information

- Great, but can we **compute** $h^*(s)$?
  - Theoretically, yes
  - Practically, *as difficult as finding an optimal plan in the first place!*

We need approximations

Desirable properties depend on the type of planning

Heuristics for Optimal Classical Planning

Optimal 1: Introduction

jonkv@ida

- In **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 node,
      systematically checking possibilities
  - A* requires **admissible** heuristics to guarantee optimality
    - Reason: Heuristic used for *pruning* (ignoring some search nodes)
    - Non-admissible ➔ can ignore some nodes *that would lead to optimal plans*

Optimal 3: Relaxation?

jonkv@ida

- **Relaxation** can be used to generate admissible heuristics…

Inventing Admissible heuristics: Problem relaxation

- A problem with fewer restrictions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is in fact an admissible heuristic to the original problem

If the problem definition can be written down in a formal language, there are possibilities for automatically generating relaxed problems automatically!

Sample rule:

A tile can move from square A to square B if
A is horizontally or vertically adjacent to B
and B is blank

Optimal 4: Computing h() using Relaxation

jonkv@ida

Original problem P,
we just generated state *s*,
want to compute *h(s)*

↓ Relax

Relaxed problem: P'
(finding a solution: fast)

↓ Solve

Find $\pi^*(P')$ –
optimal (!) plan
for relaxed problem

↓ Find cost

Compute h(s) = cost($\pi^*(P')$)

But how can you find relaxations that work for all classical planning problems?

Since P is just 'some classical planning problem', we can't expect to find a general shortcut such as 'sum of Manhattan distances'

Pattern Database Heuristics:
One of many relaxation heuristics

PDB 1: Introduction

jonkv@ida

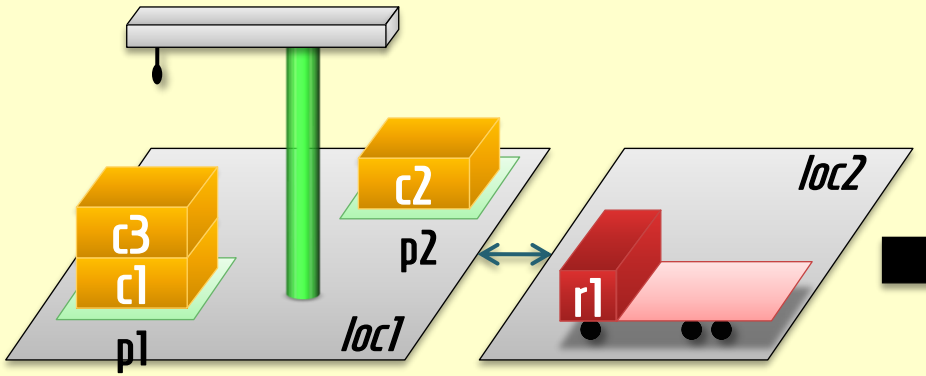
- Main idea behind **relaxation** in **pattern database heuristics**:
  - Let's **ignore some facts** – ground atoms – *everywhere*
  - Remove from ***preconditions and goals***
    - Clearly makes the problem easier – relaxation!
  - Remove from ***current state*** and from ***action effects***
    - No need to update these facts, when no conditions require them!

PDB 2: Dock Worker Robots

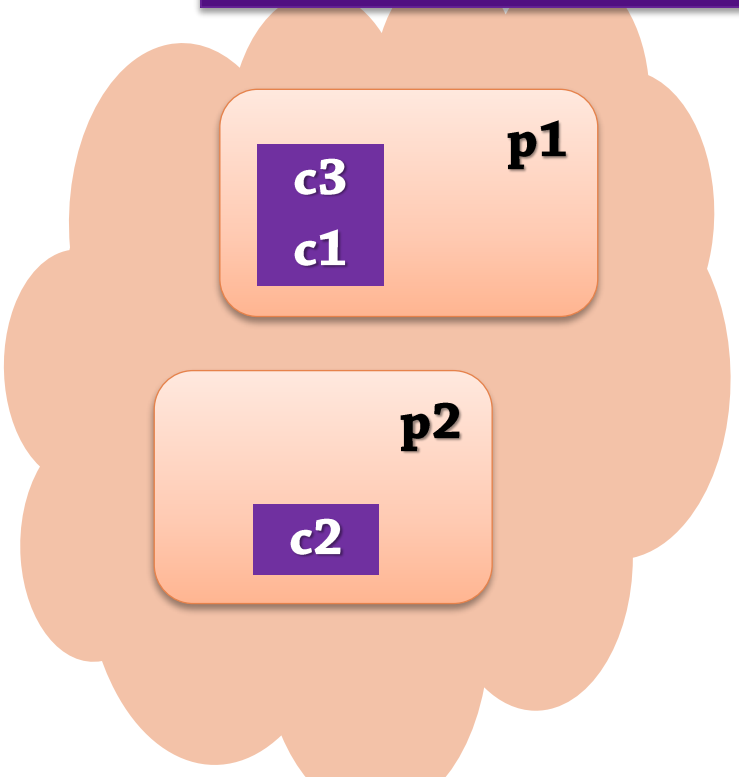
jonkv@ida

- Example: Dock Worker Robots
  - Suppose we ***only consider container locations***
    - *in(container, pile), top(container,pile), on(c1,c2), …*
  - **Ignore** robot locations, crane locations, …

Want to know *h(s)* for state *s* in *P*, where all facts are defined



Abstract state in P', information that remains after relaxation



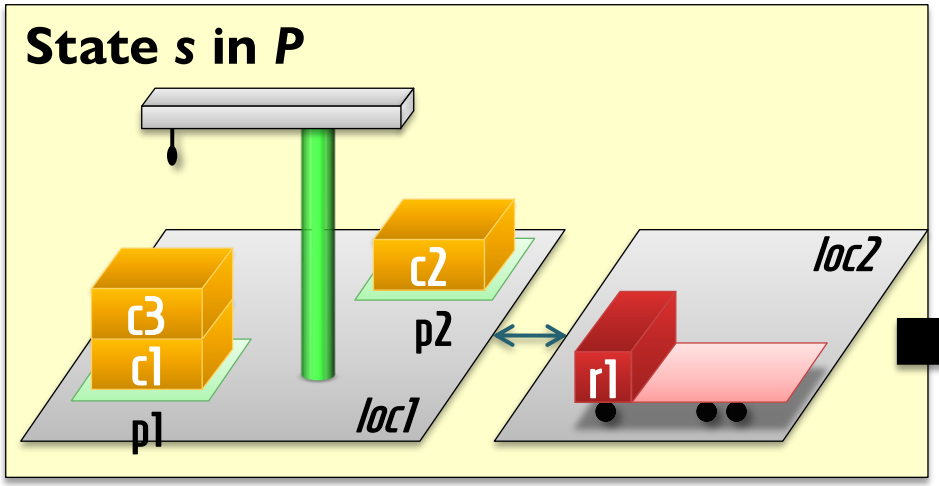
PDB 3: Planning in Patterns

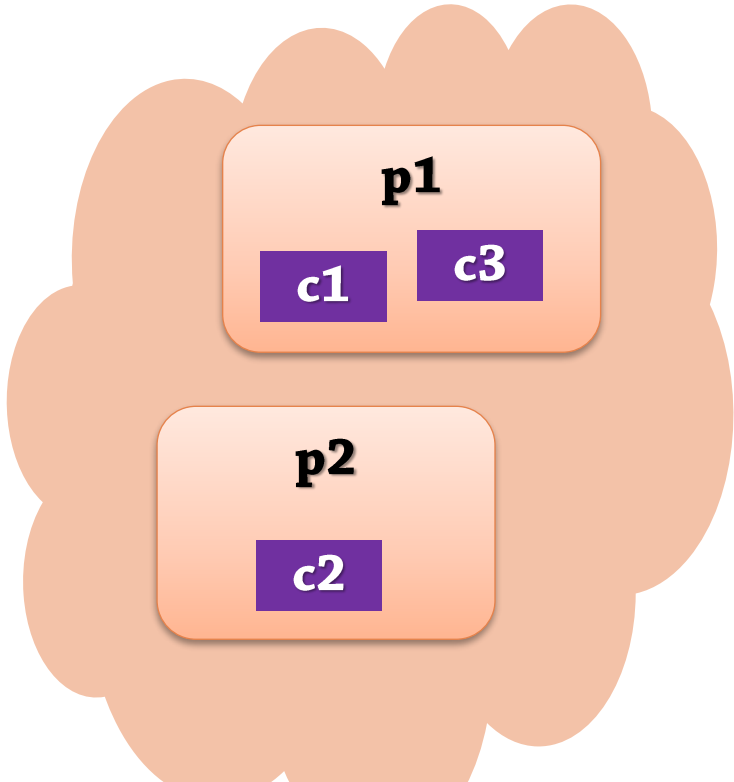
jonkv@ida

- 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 that precondition disappeared…)
  - **place five containers on one truck** (condition "truck is free" disappeared)

A tile can move from square A to square B if
A is horizontally or vertically adjacent to B
and B is blank

1. A tile can move from square A to square B if A is adjacent to B
2. A tile can move from square A to square B if B is blank
3. A tile can move from square A to square B

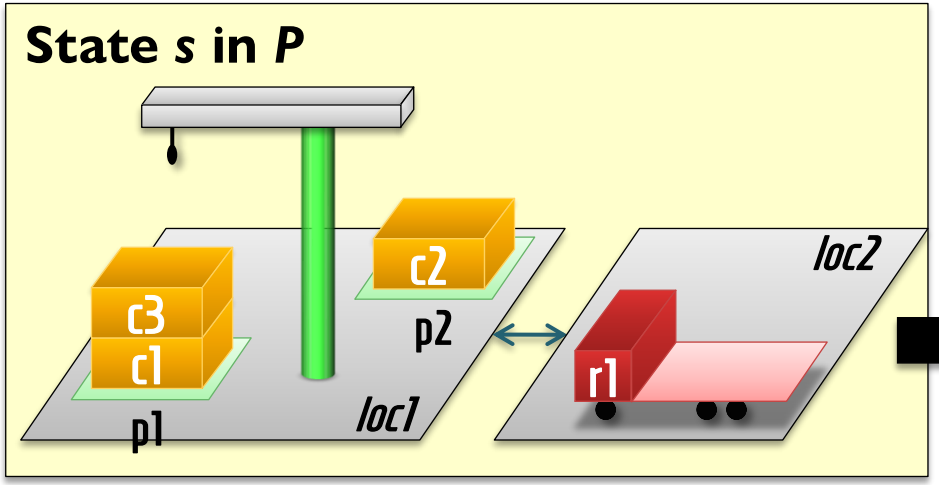


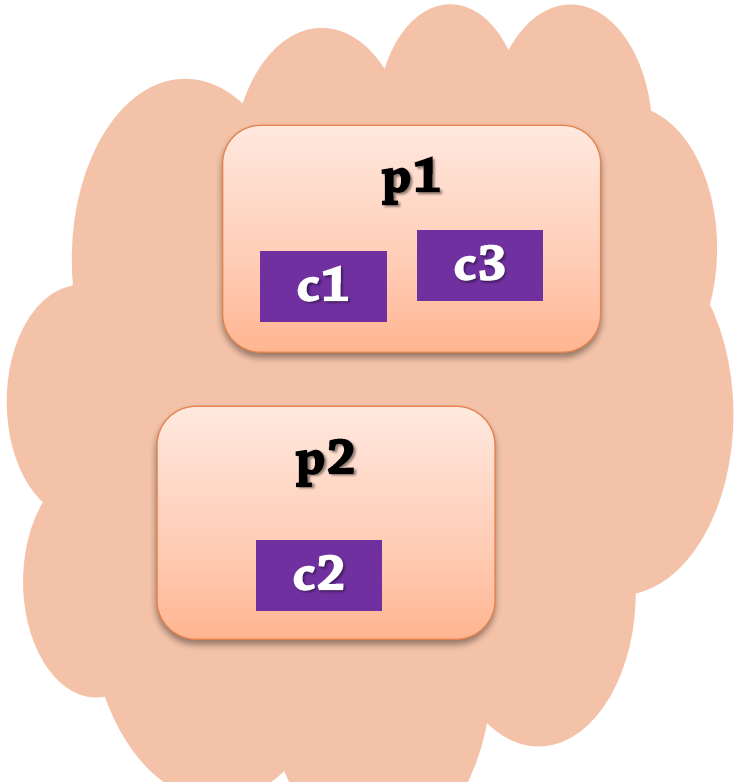


PDB 3b: Planning in Patterns

jonkv@ida

- In *P'* we still **can't**:
  - pick up c1 (preconditions about pile ordering are still there)
  - immediately place c1 below c2, …
  - ➔ **Still a planning problem P' left; need to find (cost of) optimal solution!**

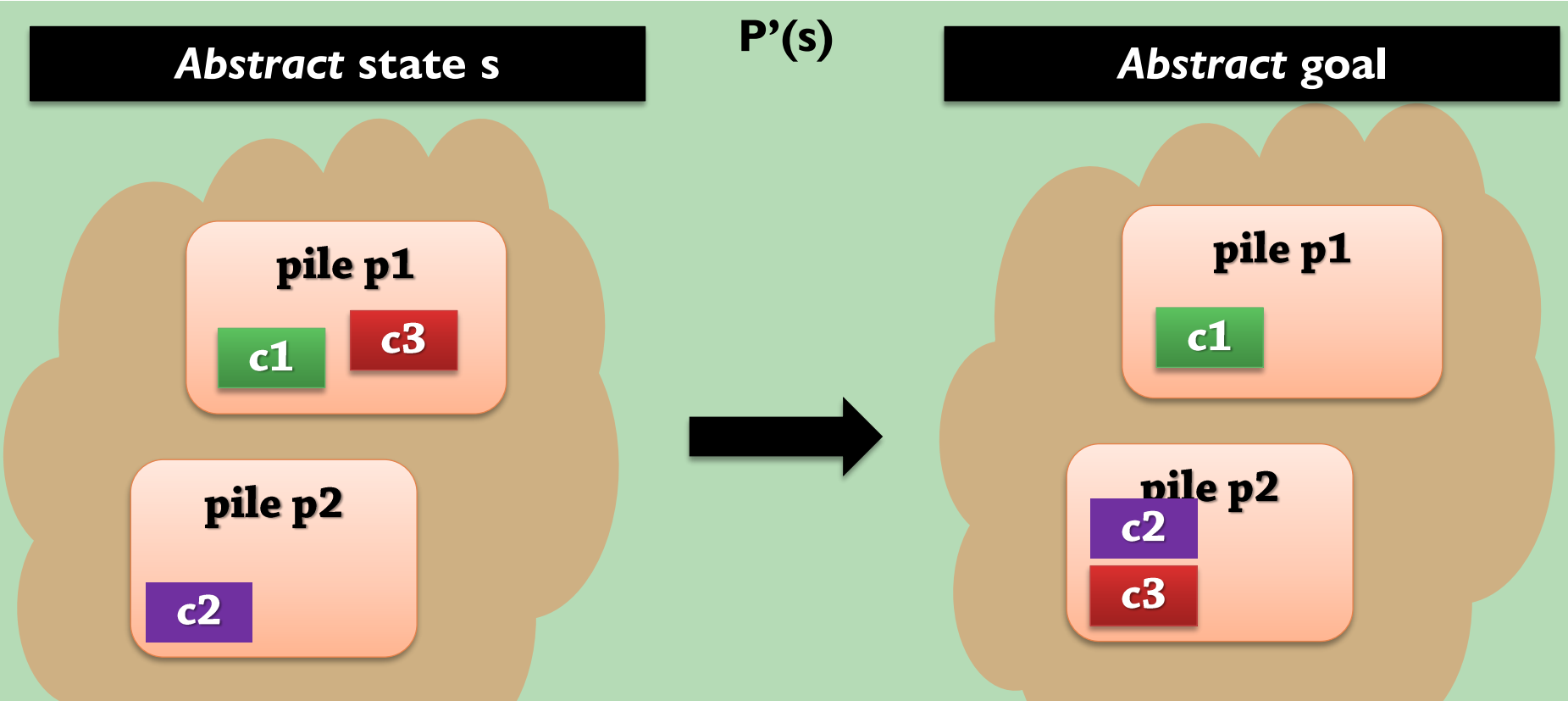




PDB 4: Computing a Heuristic Value

jonkv@ida

- **Solve P' optimally: 4 actions**
  - Take **c2** with the crane (it's in the way at the bottom of pile p2)
  - Take **c3** with the crane [relaxation – not checking if the crane is busy]
  - Place **c3** at the bottom of pile p2
  - Place **c2** on top

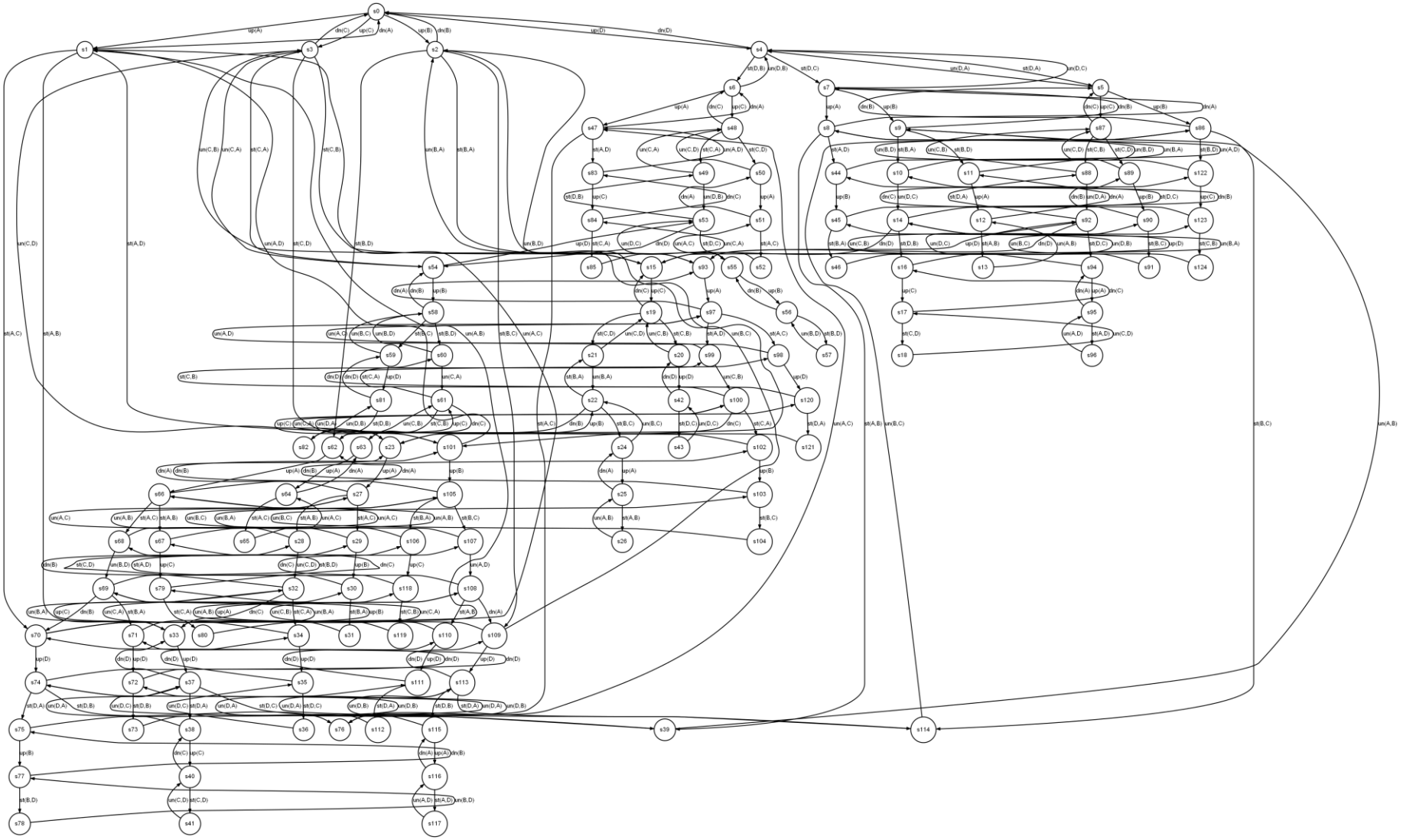


Let's formalize!

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: 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 <u>few</u> facts – for performance!

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)**}
  - **State:**

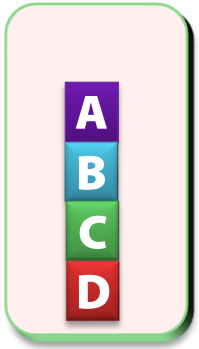
ontable(A)
ontable(B)
ontable(C)
ontable(D)
clear(A)
clear(B)
clear(C)
clear(D)
handempty

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) – **an action (instance), not an operator!**
    - **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)

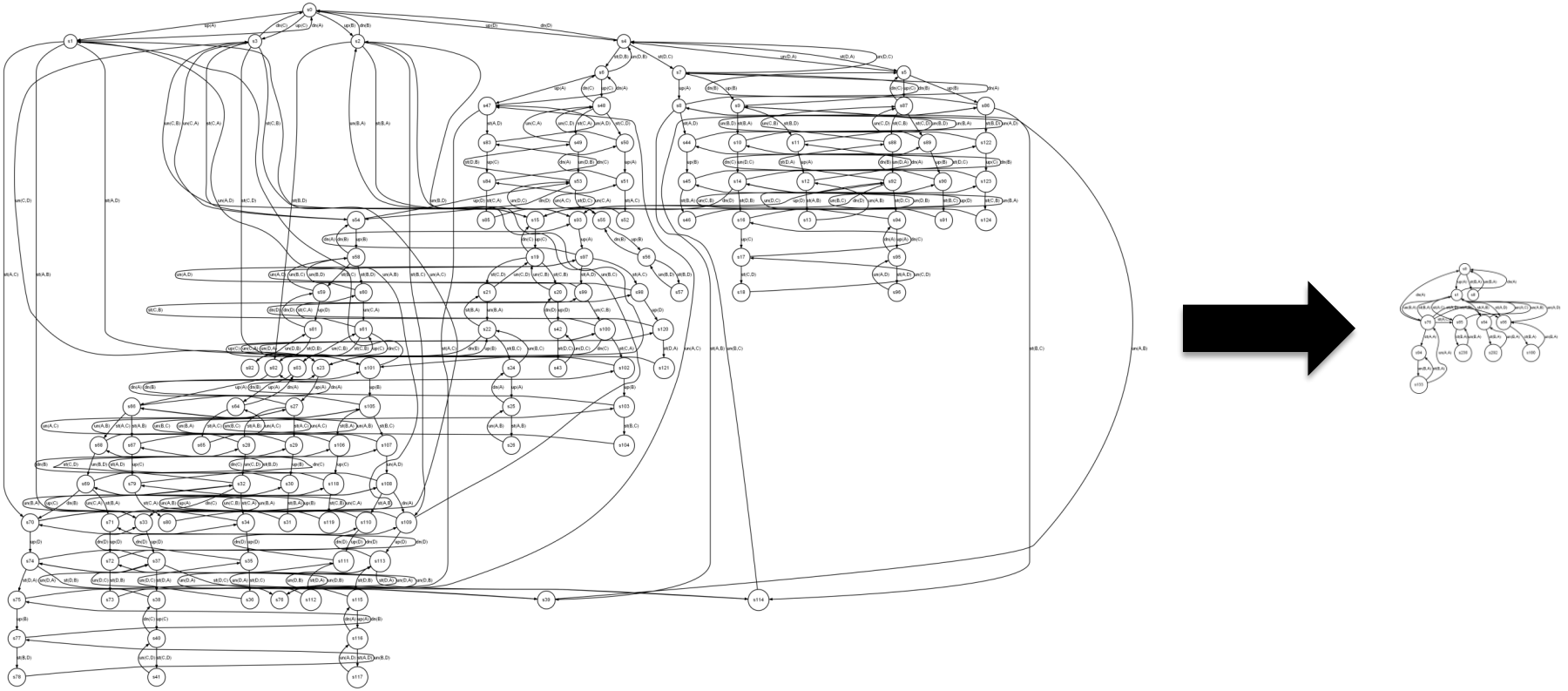
  - **Example action**: (unstack C D)
    - **Before transformation:**
      :precondition (and (handempty) (clear C) (on C D))
      :effect (and (not (handempty)) (holding C) (not (clear C)) (clear D) (not (on C D)))
    - **After transformation:**
      :precondition (and)
      :effect (and)

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

PDB: New State Space

jonkv@ida

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

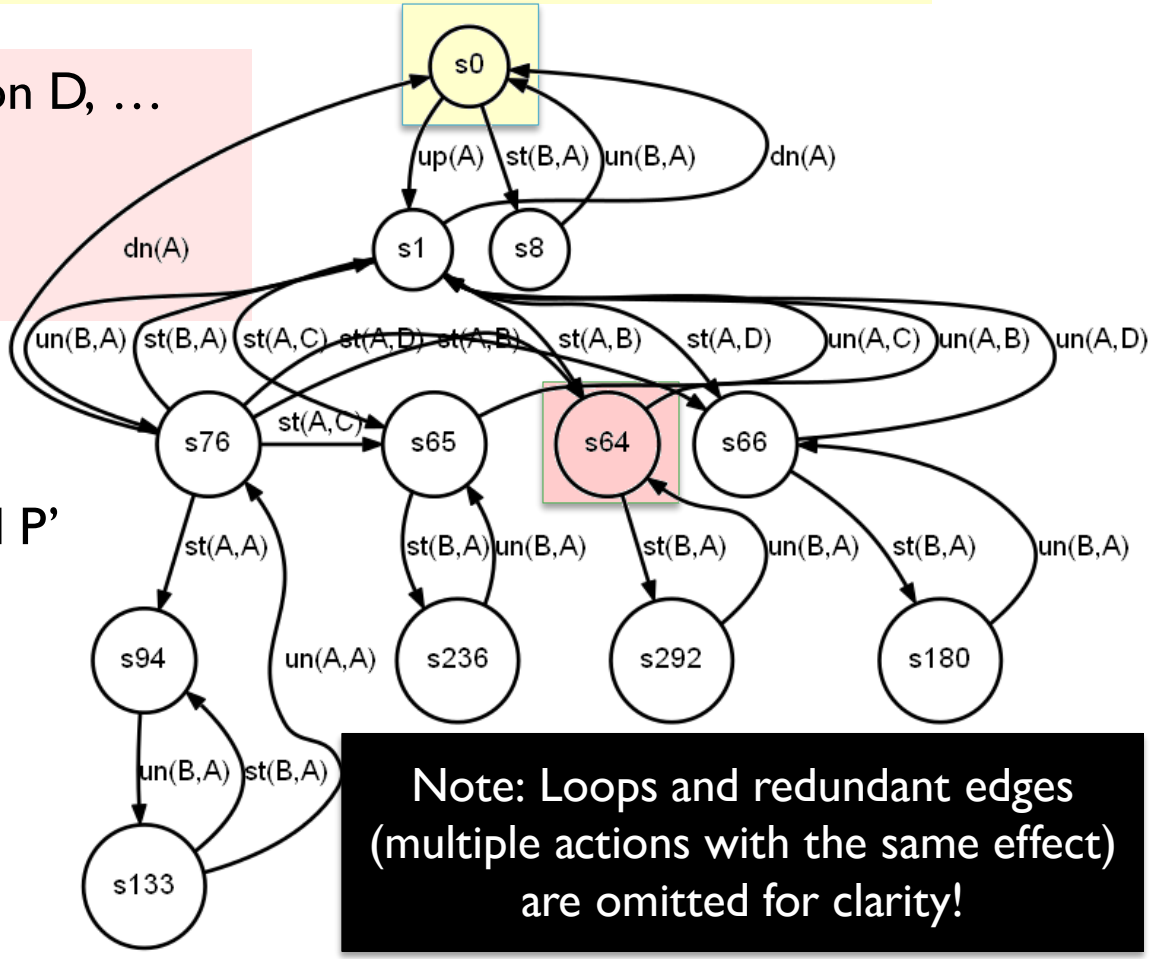


We lose information – and the size of the search space shrinks

PDB: State Transition Graph

jonkv@ida

- New **reachable state transition graph**:
  - Real state s: Everything on the table, hand empty, all blocks clear
    - ➔ **Abstract state**: s0 = { (ontable A), (clear A) }
  - Real goal g: A on B on C on D, …
    - ➔ **Abstract goal**: { (on A B), (clear A) } satisfied only in s64
  - Few abstract states ➔ **quickly** compute **optimal** costs for relaxed P'
    - Relaxed cost is exactly 2: Shortest path s0 ➔ s64
    - So *real* cost is *at least 2*; h(s)=2 is admissible

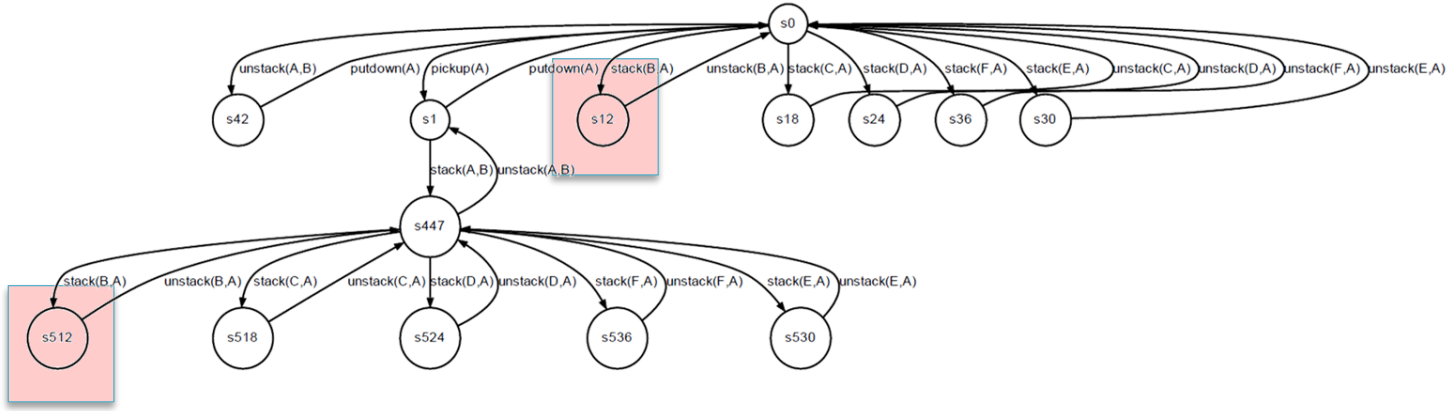


Note: Loops and redundant edges (multiple actions with the same effect) are omitted for clarity!

PDB: Databases

jonkv@ida

- Where did the ***databases*** go?
  - During the main search,
    many visited *actual states* will correspond to the **same** *abstract states*
    ➔ need the same value over and over again
  - Given a pattern, we **precompute** a database for all abstract states
    - Improves performance; the principle remains



PDB: More information

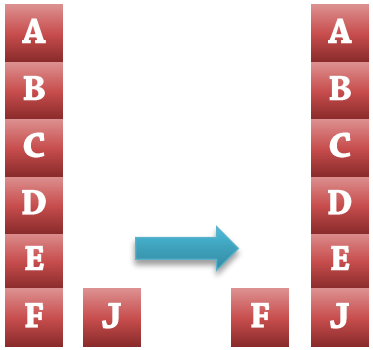
jonkv@ida

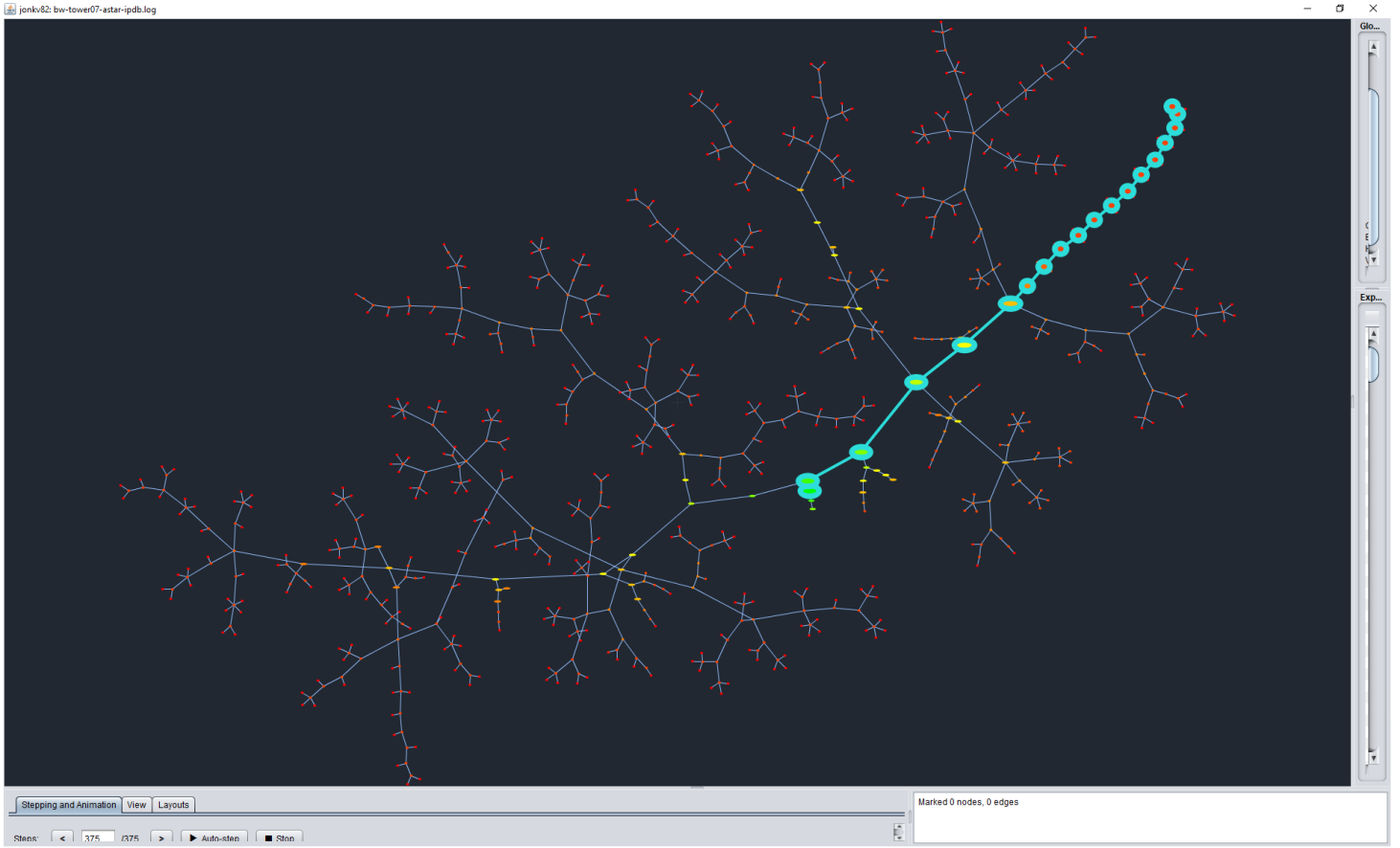
- To make PDB heuristics **more informative**:
  - Calculate costs for **several** patterns
    - Suppose we only care about **{clear(A), ontable(A)}**
    - Suppose we only care about **{on(A,B), on(C,D)}**
    - Suppose we…
  - Take the **maximum** of the computed heuristic values

All are below the limit (admissible)
➔ the largest is below the limit

- One difficulty:
  - Choosing which patterns to use…

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





- Blind A*, h(s)=0: 43150 states calculated, 33436 visited
- A* using 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)!

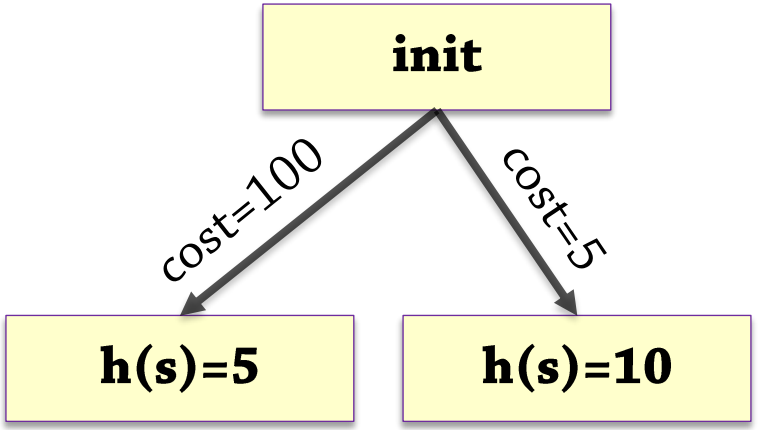
- **Satisficing** plan generation:
  - Find a plan that is sufficiently ***good***, sufficiently ***quickly***

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

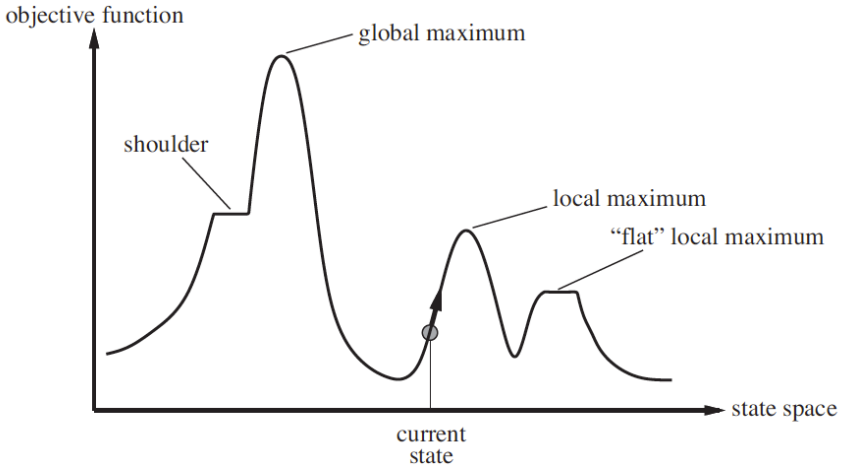
Speed: Strategies

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

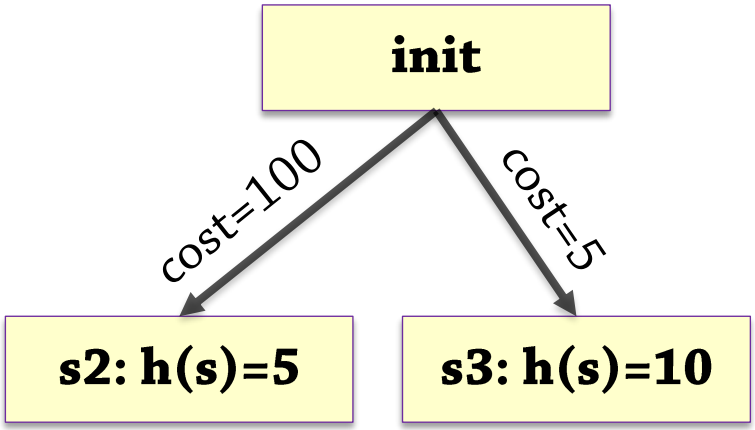


  - **Many others!**

Speed: Heuristics

jonkv@ida

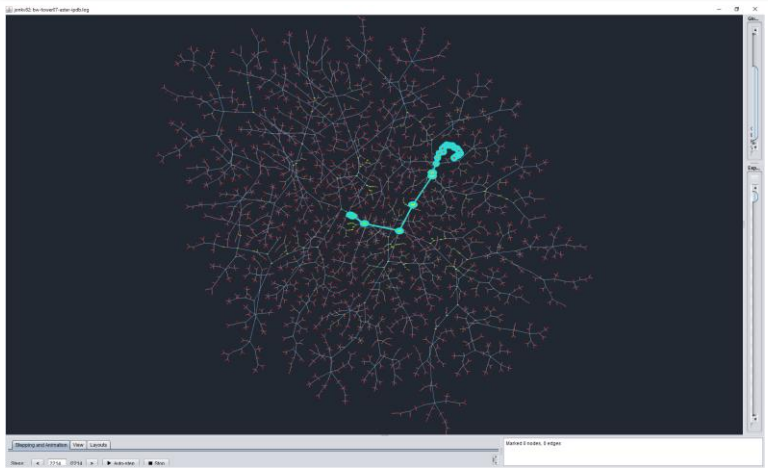
- One reason for speed: Often **more informative heuristics**
  - **Optimal planning**:
    - Often requires **admissibility**: "We *must never* overestimate, *ever*!"
    - Result: Usually *underestimates* (a lot!)
  - **Satisficing planning:**
    Extreme example – ***greedy best first***
    - *Only*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

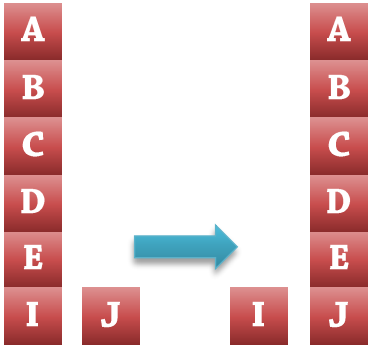


  - In many cases:
    - Admissibility is not required
    - Lack of admissibility is not "only slightly harmful"
    - Lack of admissibility is *irrelevant*

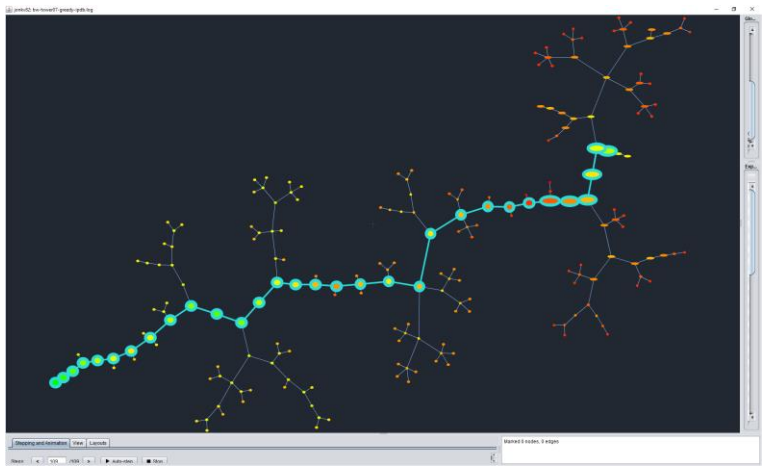
7 blocks (tiny problem)

A*, IPDB: Expand 1321 states, 18 actions in the solution





Greedy, IPDB: Expand 171 states, 22 actions in the solution



Larger problem instances → the difference increases

Example:
Landmark Heuristics

Landmark Heuristics (1)

jonkv@ida

Landmark:

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

Landmarks (2)

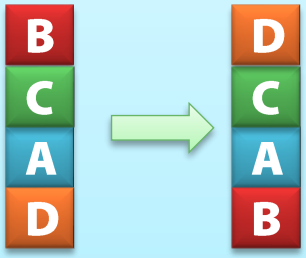
jonkv@ida

Landmarks in planning:
Something you must ***achieve*** *or* ***use*** in ***every solution*** to a problem instance

Assume we are considering a state *s*…

Fact Landmark for *s*:

A single **fact** (ground atom) that must be true
at *some* point
in *every* solution starting in *s*



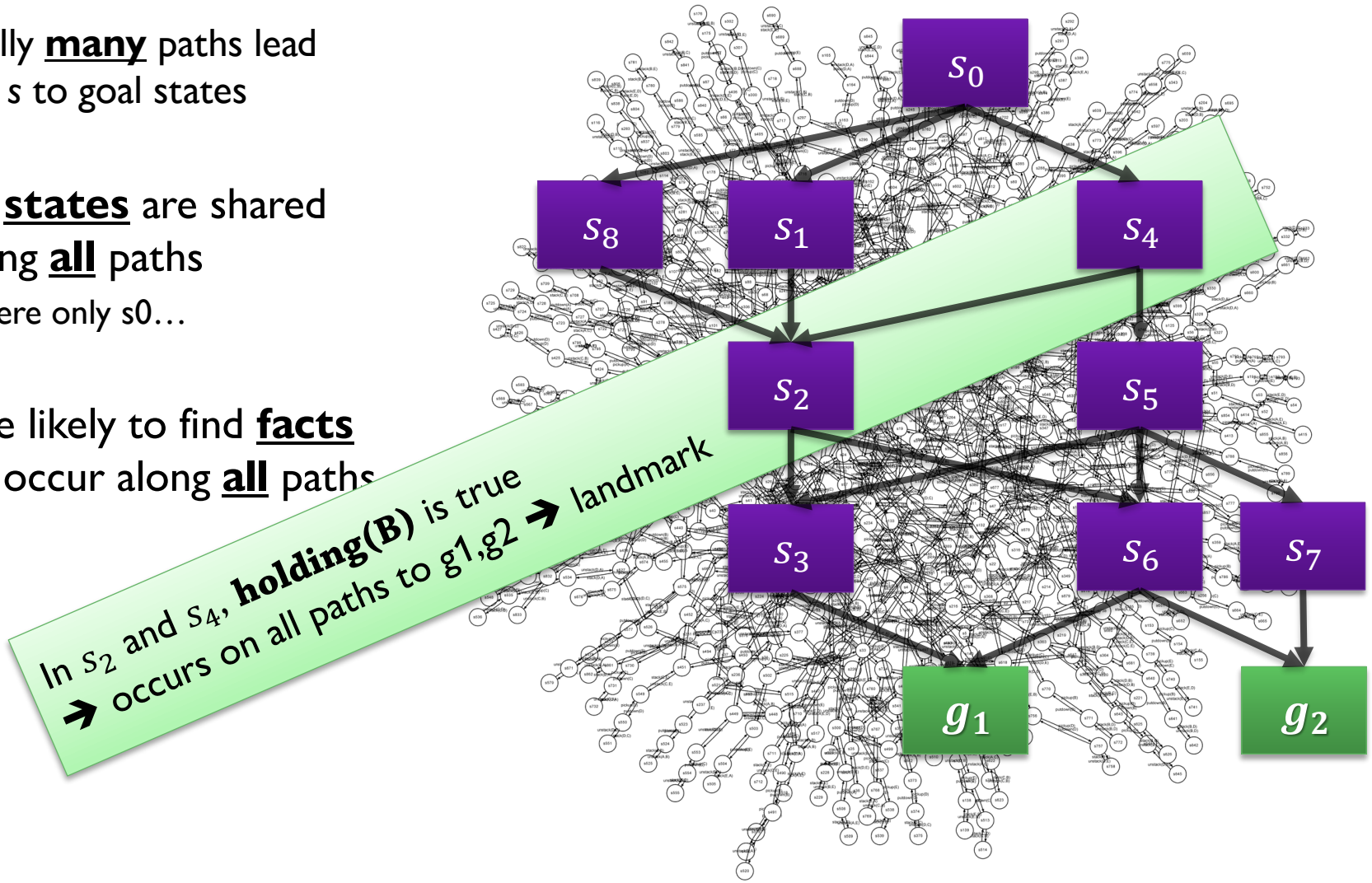
clear(A)
holding(C)
…

Landmarks (3)

jonkv@ida

Facts, not states! Why?

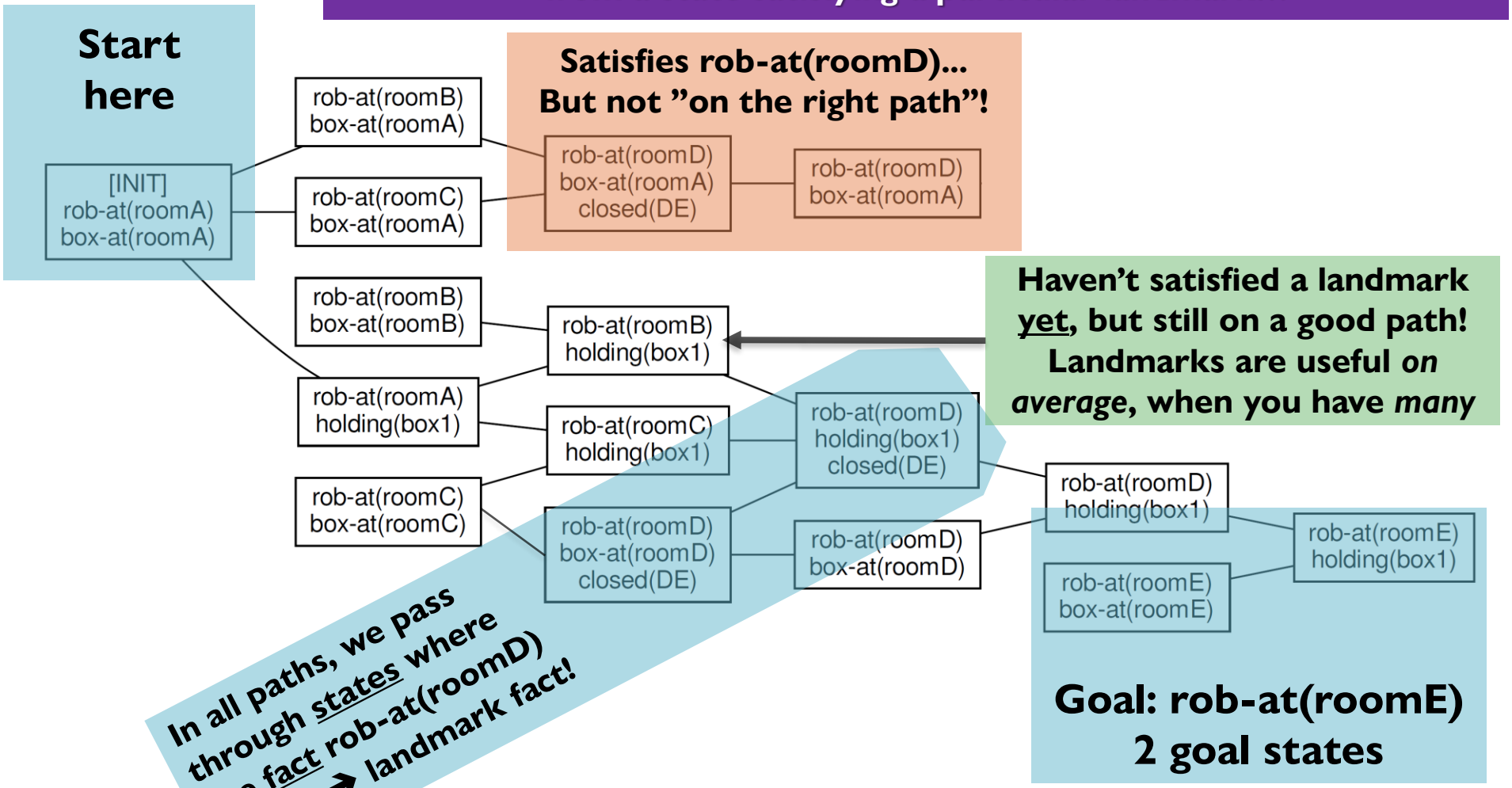
- Usually **many** paths lead from s to goal states
- Few **states** are shared among **all** paths
  - Here only s0…
- More likely to find **facts** that occur along **all** paths



Landmarks (4)

jonkv@ida

Notice: No Euclidean distance; you can't easily know *how far* a state is from a state satisfying a particular landmark…



Landmarks (5): Misunderstandings

jonkv@ida

Not ”we must reach (pass through) **the** landmark state”!

Instead ”we must reach ***some state*** that satisfies the landmark”

Not ”A landmark fact is a state that…”

A fact is not a state.
A state *consists* of *many* facts.

(”A word is a sentence that…”)

A landmark fact is **not**
”a fact that is true in every solution”

A solution is a plan.
Facts are true in *states*.

A landmark fact is
”a fact that is true in some state
along every path
from the initial state to any goal state”.

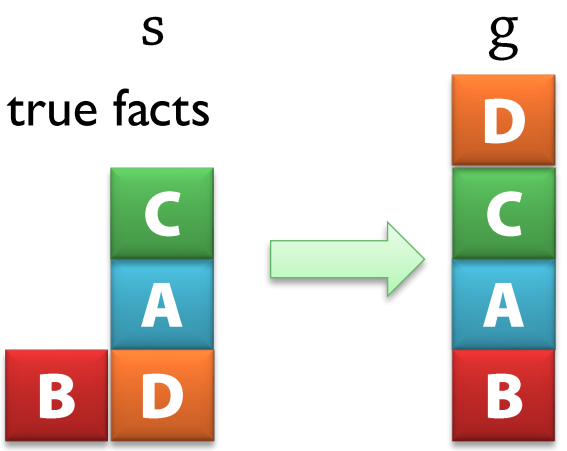
But isn't the state space graph too large to generate?

Let's try to find some of the landmarks,
more efficiently...

Means-Ends Analysis Algorithm

jonkv@ida

- **Problem setup**
  - s = the state whose heuristic value we want, a set of true facts
    - **{ clear(B), ontable(B), clear(C), on(C,A), on(A,D), ontable(D), handempty }**
  - g = the goal specification, a set of desired facts
    - **{ clear(D), on(D,C), on(C,A), on(A,B), ontable(B) }**
    - (This goal does not mention handempty!)



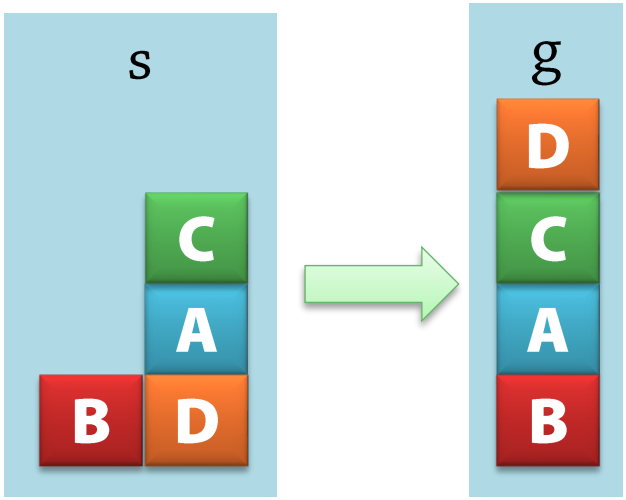
- One way of discovering landmarks:
  means-ends analysis ("backwards from the goal")
  - All facts in g must be landmarks – must occur *at the end of a solution path*!
  - **{ clear(D), on(D,C), on(C,A), on(A,B), ontable(B) }**

- But let's focus on the most interesting part
  - **4 "unachieved" landmarks, not already true in state *s***:
    $g - s$ = **{ clear(D), on(D,C), on(A,B), ontable(B) }**

Means-Ends Analysis Algorithm (2)

jonkv@ida

- Means-ends analysis, informally:
  - We start with $g - s$ = **{ clear(D), on(D,C), on(A,B), ontable(B) }**
  - Now let's consider **on(D,C)**
    - Must be **achieved** by some action:
      Is a landmark, but not already true in s
  - **How** can we achieve **on(D,C)**?
    - **Only** using **stack(D,C)**

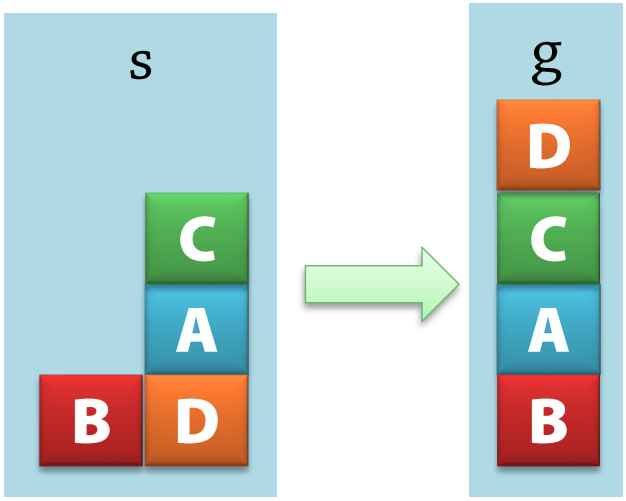


  - What do we **also** need, in order to actually **execute stack(D,C)**?
    - **All** of its preconditions
    - **{ holding(D), clear(C) }** must also be fact landmarks, but clear(C) is true *now*…
    - **{ holding(D) }** – another **unachieved** fact landmark!
  - Updated list of 5 distinct unachieved LM found:
    - **{ clear(D), on(D,C), on(A,B), ontable(B), holding(D) }**

Means-Ends Analysis Algorithm (3)

jonkv@ida

- Some more intuitions:
  - Current list:
    { clear(D), on(D,C), on(A,B), ontable(B), holding(D) }
  - Let's consider **achieving holding(D)**
    - Not true now (in s), but must be *made* true
  - **How** can we achieve **holding(D)**?
    - Remember we must consider **all possible paths** to a goal state
    - ➔ **All** actions having **holding(D)** as an effect
    - ➔ **{ pickup(D), unstack(D,A), unstack(D,B), unstack(D,C), unstack(D,D) }**
  - What do we **also** need, **regardless of which of these actions we use**?
    - The **intersection** of the **preconditions** of the 5 actions
    - Only **pickup** requires **ontable(D)**; only **unstack** requires **on(D,*something*)…**
    - But all require **{ clear(D), handempty }** – must also be fact landmarks
    - **{ clear(D) }** – another **unachieved** fact landmark!



So now we have 6…

Means-Ends Analysis Algorithm (4)

jonkv@ida

Unachieved goal facts:
clear(D), on(D,C), on(A,B), ontable(B)

fact-landmarks ← g – s

p=on(D,C) is an unachieved fact landmark
➔ all solutions must at some point *achieve* **on(D,C)** with an action effect
➔ compute *achievers* = { **stack(D,C)** }, the only action achieving **on(D,C)**

do {
  for each p in fact-landmarks {
    // Which actions could achieve p?
    achievers ← $\{a \in A \mid p \in \text{effects}(a)\}$

All achievers have some common requirements / preconditions:
{ ***holding(D), handempty, clear(C), …*** }

    // What would *all* the achievers need?
    common ← $\bigcap_{a \in achievers} preconds(a)$

handempty is already true, but *new* = { **holding(D), clear(C)** } are *unachieved* landmarks

    new ← common – s
    fact-landmarks ← fact-landmarks ∪ new

Maybe we can find more landmarks related to achieving *those*!

  }
} **until** no more fact-landmarks found

Learn to apply this algorithm! Test it on some problem instances!

Landmark Counts and Costs

jonkv@ida

- One simple form of landmark heuristic: **Counting** landmarks
  - $h(s)$ = the **number** of **unachieved** landmarks in state s

One action can actually achieve multiple landmarks at once (multiple effects)
➔ landmark count is not admissible

More complex (and stronger) forms of landmark heuristics also exist – pioneered by the LAMA planner

See, for example,
Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks.
Journal of Artificial Intelligence Research, 39:127–177, 2010.