Linköping University

Automated Planning

Heuristics for Forward State Space Search: Overview and Examples

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

jonas.kvarnstrom@liu.se – 2017

Heuristics in Forward State Space Search: Introduction

Heuristic Forward State Space Search

- General **Forward State Space Search Algorithm**
  - **forward-search**(A, s_0, g) {

    open ← { <s_0, ε> }

    while (open ≠ emptyset) {

    use a strategy to select and remove one n=<s,path> from open

    if goal g satisfied in state s **then**

    foreach $a \in A$ such that $\gamma(s, a)$

    {s'} ← $\gamma(s, a)$

    path' ← **append**(path, a)

    add n'=<s', path'> to open

    }

    }

    }

- A **heuristic strategy** bases its decisions on:
  - **Heuristic value h(*n*)**
  - Often other factors, such as **g(n) = cost of reaching *n***

Requires a heuristic function

How do we calculate $h(n)$?
$h_1(n), h_2(n), h_{add}(n)$,
landmarks,
pattern databases, …

Requires a heuristic strategy

How do we use $h(n)$?
A*, IDA*, D*, simulated annealing,
hill-climbing, (various forms of)
best first search, …

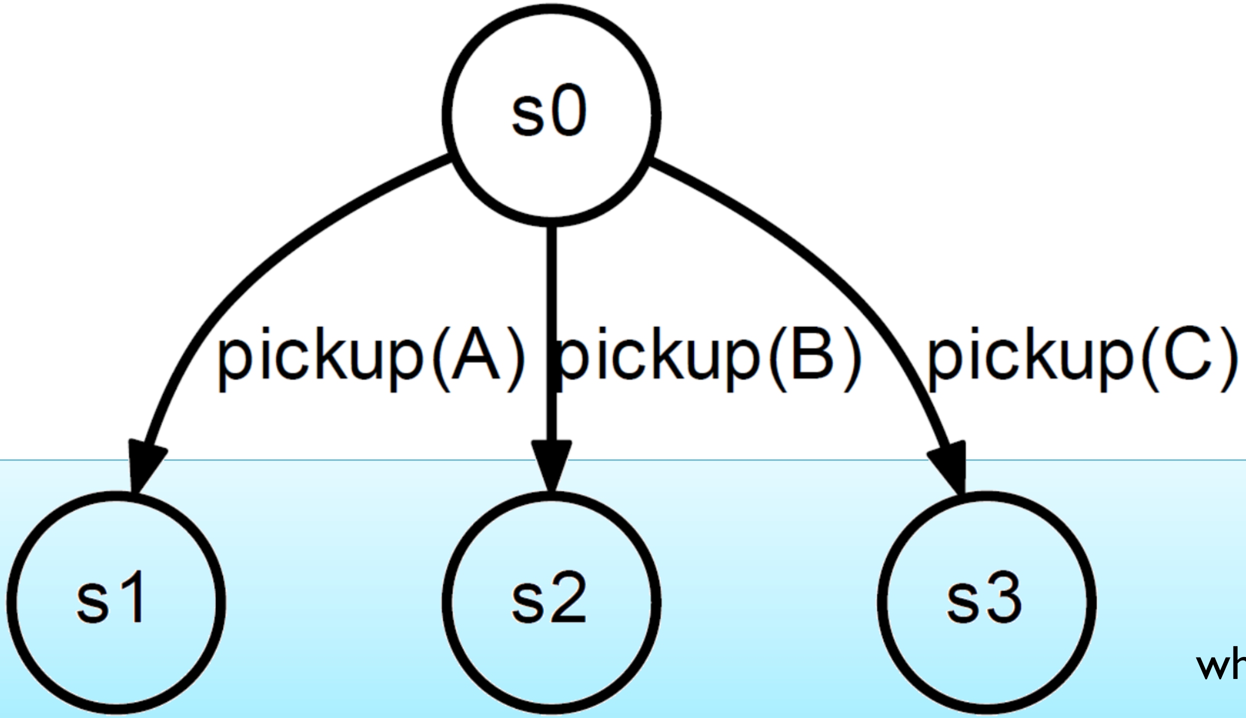
Example (1)

Example: 3 blocks, all on the table in s0

We now have
1 *open node*,
which is *unexpanded*

Example (2)

We visit s_0 and expand it

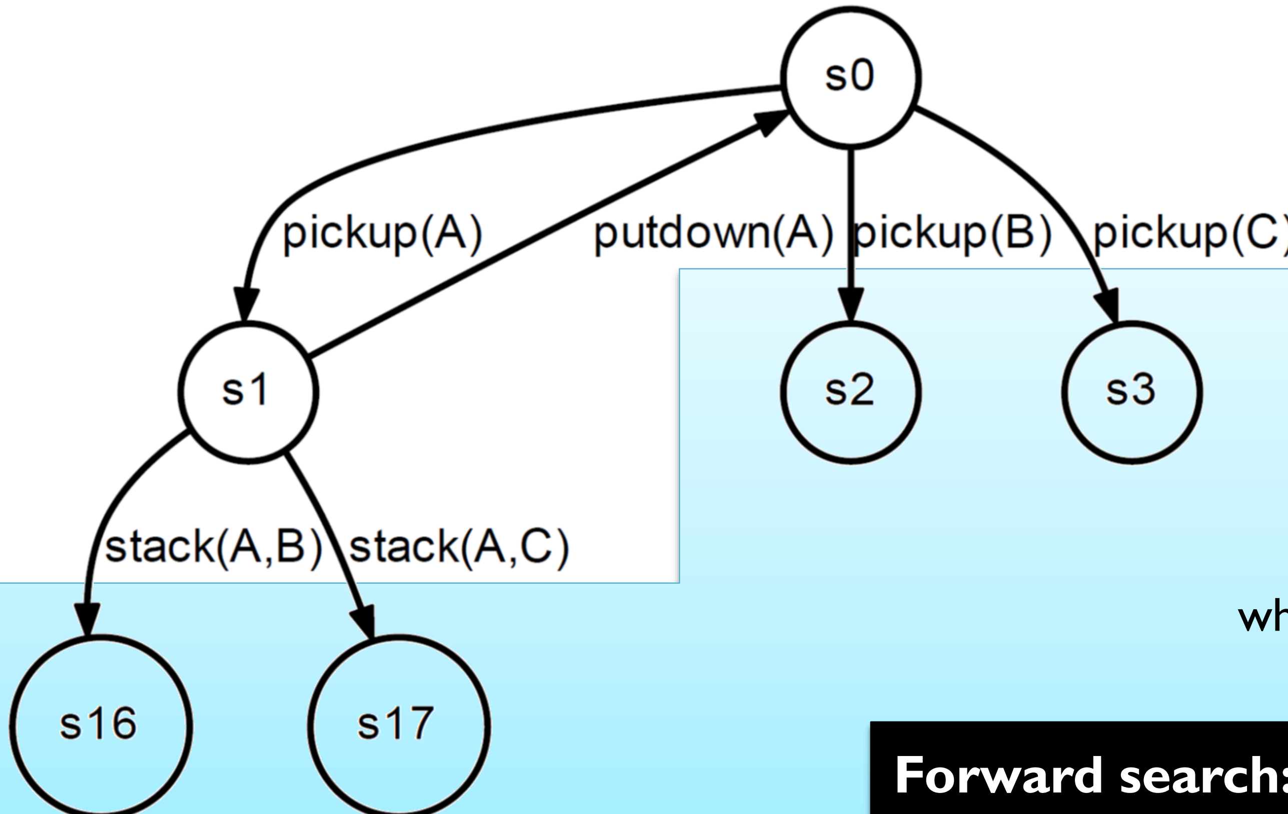


We now have
3 *open nodes*,
which are *unexpanded*

A ***heuristic function*** estimates the distance from each open node to the goal:
We calculate $h(s_1), h(s_2), h(s_3)$
A ***heuristic strategy*** uses this value (***and*** other info) to *prioritize*

Example (3)

Suppose the strategy chooses to visit s_1:



We now have
4 *open nodes*,
which are *unexpanded*

Forward search: node ≈ state, so we may write $h(n)$ or $h(s)$

2 new heuristic values are calculated: $h(s_{16}), h(s_{17})$
The ***search strategy*** now has 4 nodes to prioritize

Heuristic Functions: What to Measure?

What to Measure?

Question 1A: **What** should a heuristic function **measure**?

- A **heuristic strategy** bases its decisions on:
  - **Heuristic value h(*s*)**
  - Often other factors, such as **g(s) = cost of reaching *s***

Very general definition
→ **could** measure **anything** that **some** strategy might find useful!

Often: h(s) *tries to* measure the **cost** of achieving the goal from *s*

Useful for finding **cheap plans** –
and often, as a **side effect**, for finding **plans cheaply**

→ **Question 1B: What is "cost"?**

Plan Quality and Action Costs

- Could say: **Long** plan = **expensive** plan
  - $c(\pi) = |\pi|$ (number of actions)
    - Reasonable in Towers of Hanoi
    - But: How to make sure your car is clean?

go to car wash	get supplies	wash car
go to car dealer	buy new car	shortest plan is best?

- Would prefer to model different **action costs**
  - Supported by most current planners
    - Each action $a \in A$ associated with a cost $c(a)$
  - **Total cost**: $c(\pi) = \sum_{a \in \pi} c(a)$

Action Costs in PDDL

- PDDL: Specify requirements
  - (:requirements :action-costs)
- *Numeric state variable* for the total cost, called **(total-cost)**
  - And possibly numeric state variables to *calculate* action costs
  - (:functions (total-cost) - number
    (travel-slow-cost ?f1 - count ?f2 - count) - number
    (travel-fast-cost ?f1 - count ?f2 - count) - number)

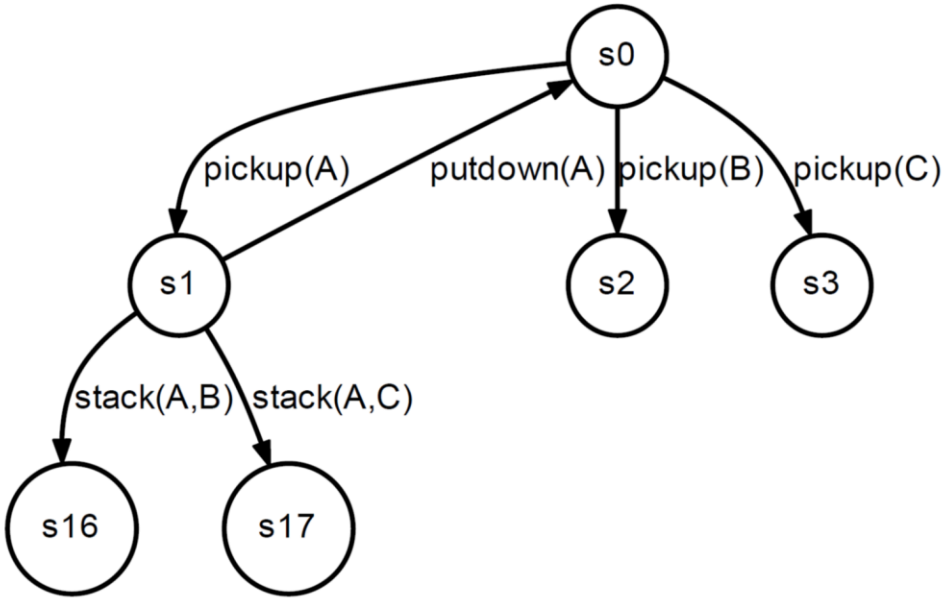
Built-in type supported by cost-based planners

- **Initial state**
  - (:init (= (total-cost) 0)
    (= (travel-slow-cost n0 n1) 6) (= (travel-slow-cost n0 n2) 7)
    (= (travel-slow-cost n0 n3) 8) (= (travel-slow-cost n0 n4) 9)
    …)
- Special **increase effects** to increase total cost
  - (:action move-up-slow
    :parameters (?lift - slow-elevator ?f1 - count ?f2 - count)
    :precondition (and (lift-at ?lift ?f1) (above ?f1 ?f2) (reachable-floor ?lift ?f2))
    :effect (and (lift-at ?lift ?f2) (not (lift-at ?lift ?f1))
    (increase (total-cost) (travel-slow-cost ?f1 ?f2))))

Remaining Costs

- The **remaining cost** in a search state *s*:
  - The cost of a **cheapest (optimal) solution** starting in *s*
  - Denoted by h*(s)

- The cost of an **optimal solution** to (Σ, s_0, S_g):
  - $h^*(s_0)$

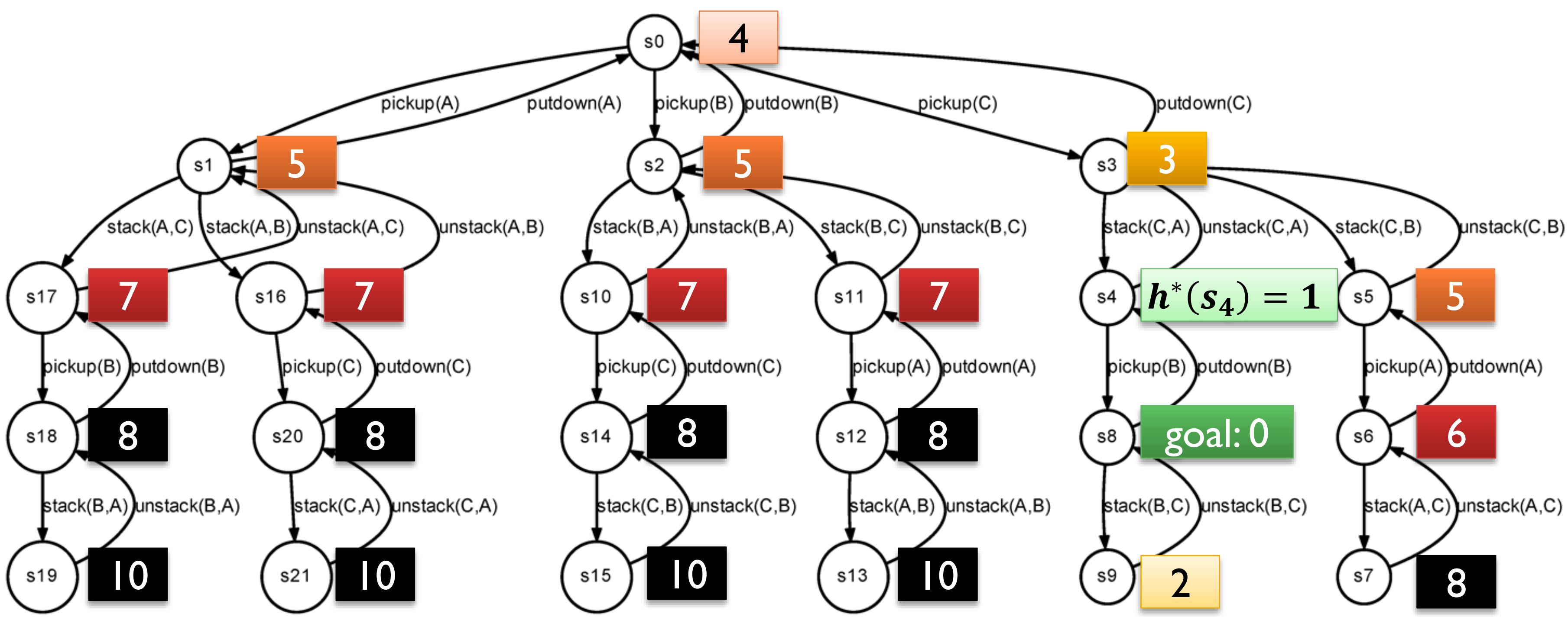


True Remaining Costs (1)

jonkv@ida

True Cost of Reaching a Goal: h*(n)

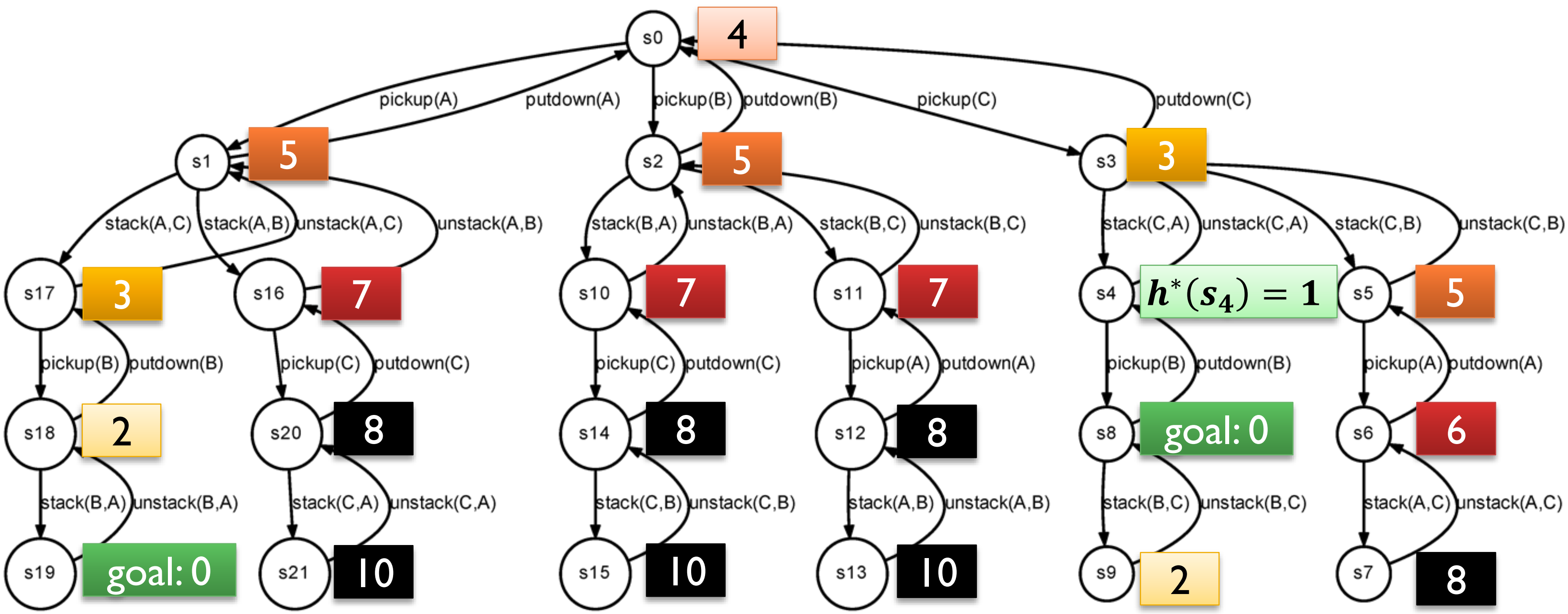
Initially: A,B,C on the table
pickup, putdown cost 1
stack, unstack cost 2 (must be more careful)



True Remaining Costs (2)

True Cost of Reaching a Goal: h*(n)

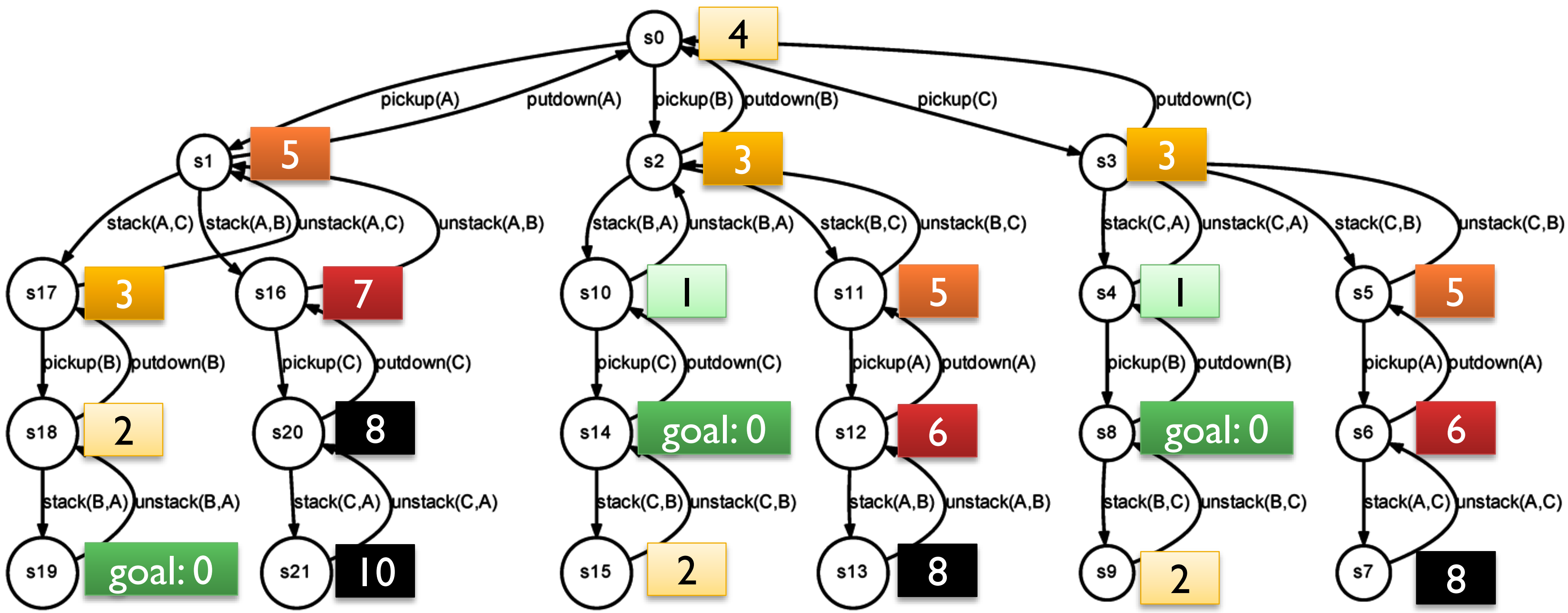
Two reachable goal states



True Remaining Costs (3)

True Cost of Reaching a Goal: h*(n)

Three reachable goal states
(there can be many)

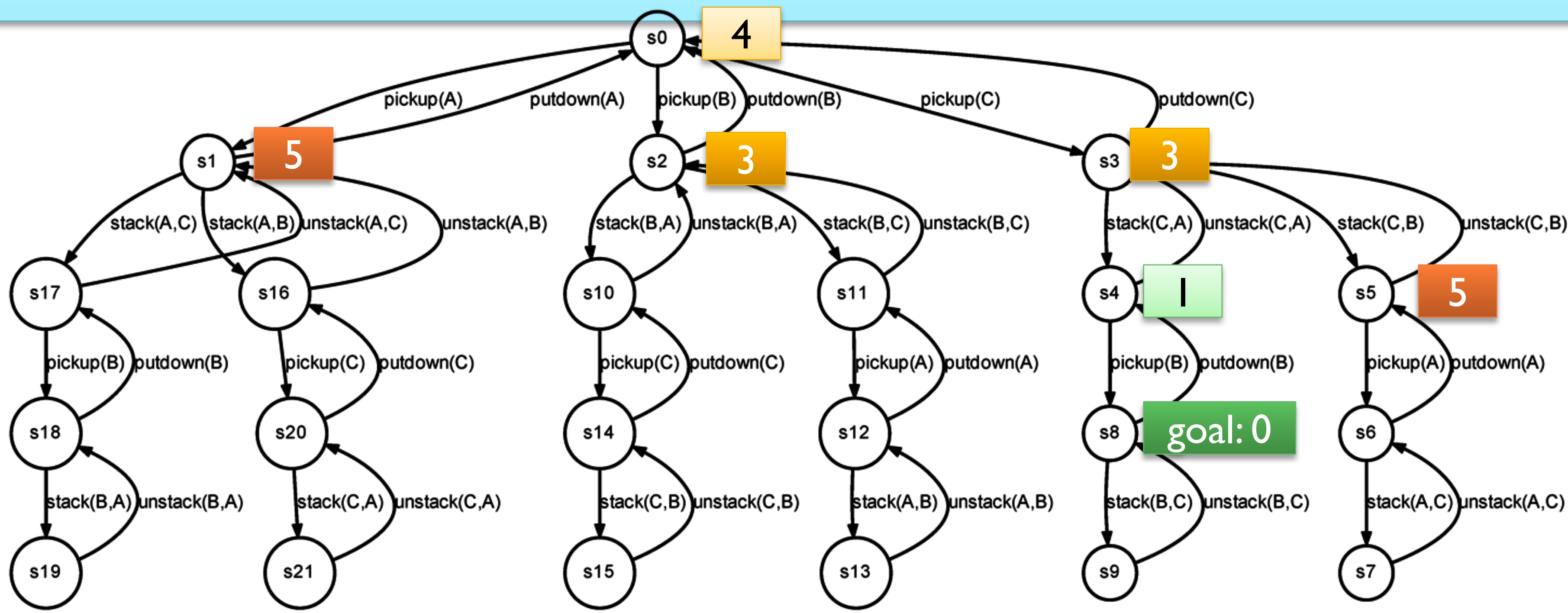


True Remaining Costs (4)

If we *knew* the true remaining cost h*(n) for every node:

Algorithm simplePlan:
node ← initstate
while (not reached goal) {
  node ← a successor of *node* with minimal h*(n)
}

Trivial straight-line path minimizing h* values gives an *optimal* solution!



Reflections

- What does this **mean**?
  - Calculating h*(n) is a **good idea**,
    because then we can easily find optimal plans?

- No – because we can prove that finding optimal plans is **hard**!
  - So calculating h*(n) *must* be hard as well…

1. We can always quickly compute h*(n)

2. Given h*(n),
we can quickly find optimal solutions

3. We can quickly find optimal solutions
for any classical planning problem

4. Known to be false!
(PSPACE-complete)

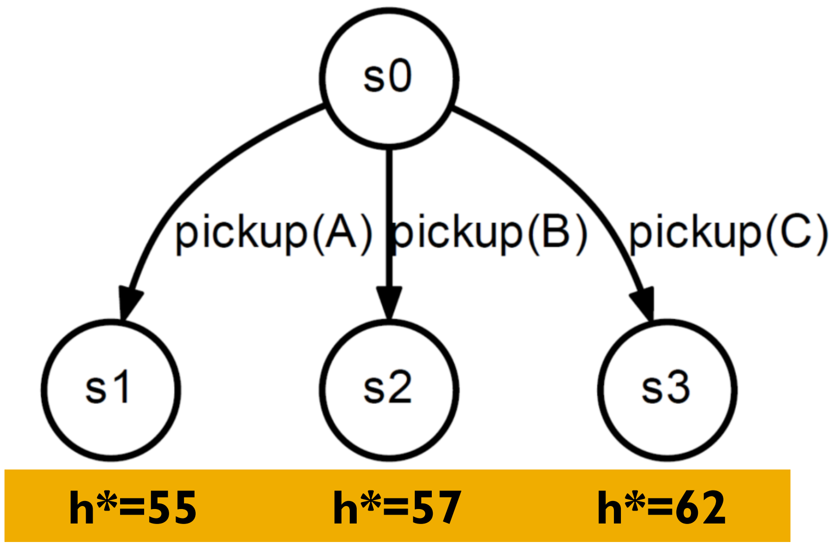
5. …so one of these premises
must be false

Must settle for an **estimate** that helps us **search less** than otherwise

Heuristic Functions:
What properties should an estimate have?

Minimization: Intro

Strategy: Depth first search; select a child with **minimal** $h(s)$

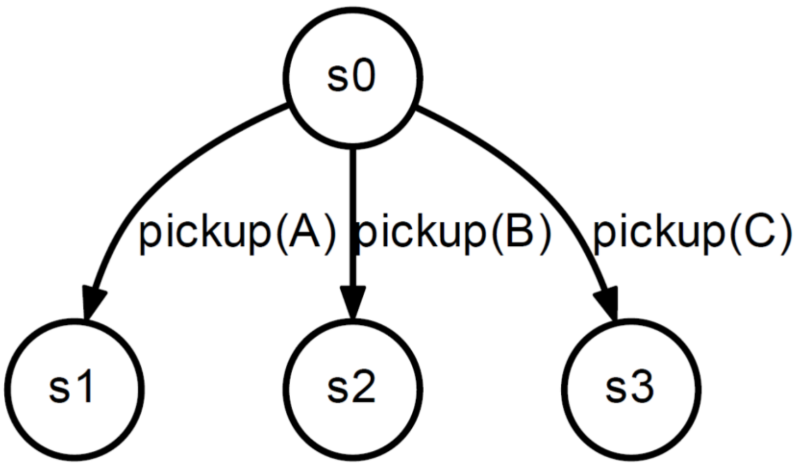


If I start with pickup(A),
then make optimal choices:
Plan cost = 55

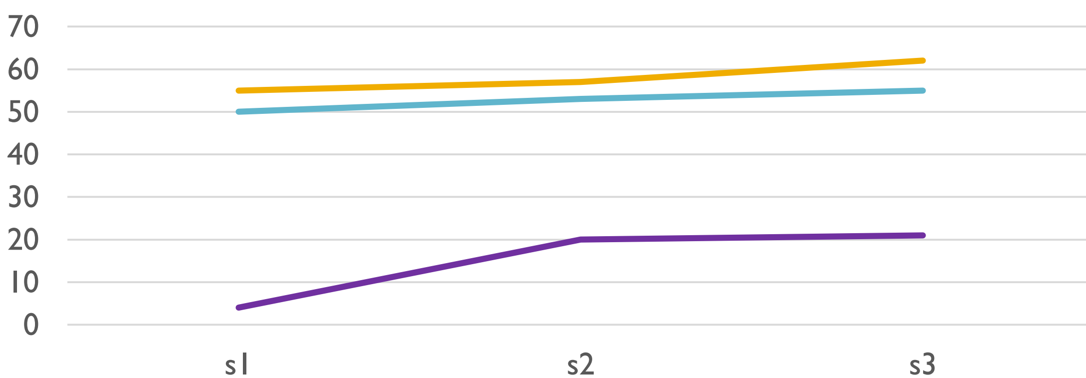
If I start with pickup(C),
then make optimal choices:
Plan cost = 62

Minimization, case 1

Strategy: Depth first search; select a child with **minimal** $h(s)$



h*=55	h*=57	h*=62	
hA=50	hA=53	hA=55	Close!
hB=4	hB=20	hB=21	Far from the truth…



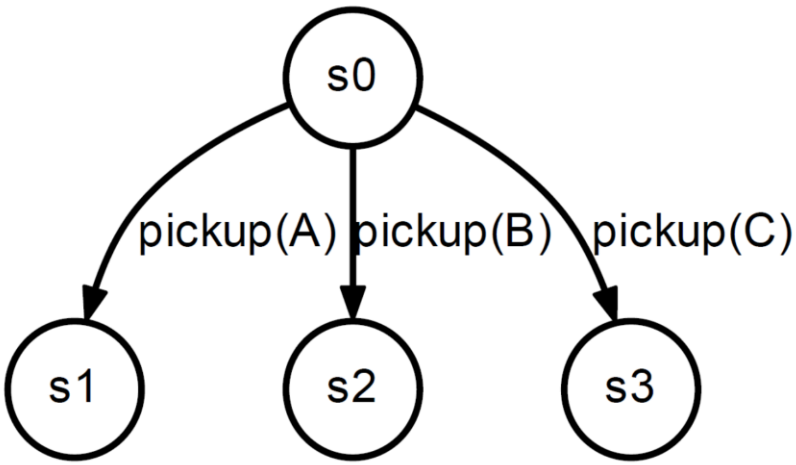
Which is best?

The strategy only cares about relative values

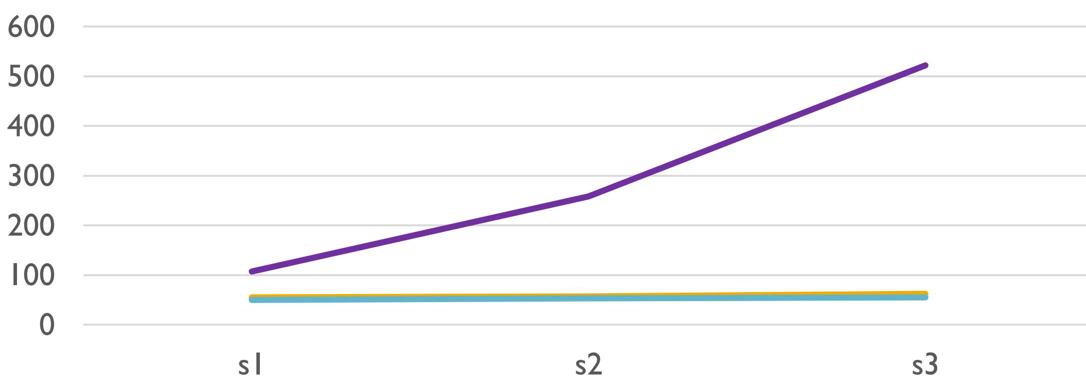
h*, hA, hB result in identical choices: s_1 first!

Minimization, case 2

Strategy: Depth first search; select a child with **minimal** $h(s)$



h*=55	**h*=57**	**h*=62**	
hA=50	**hA=53**	**hA=55**	Close!
hB=107	**hB=258**	**hB=522**	Large overestimate!



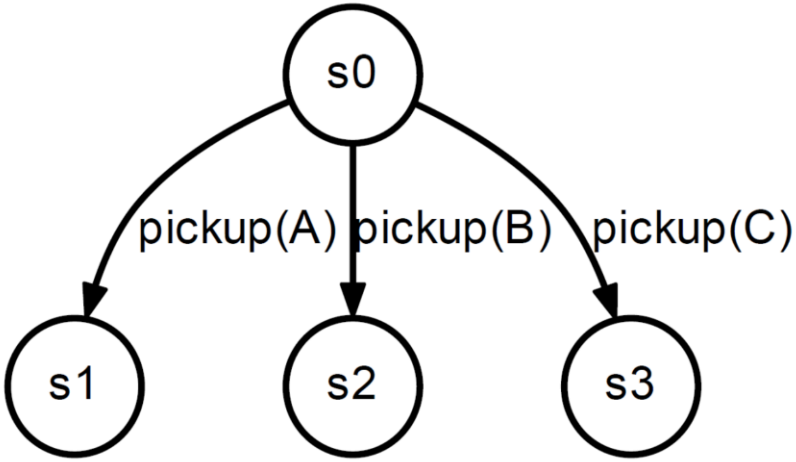
Which is best?

The strategy only cares about relative values

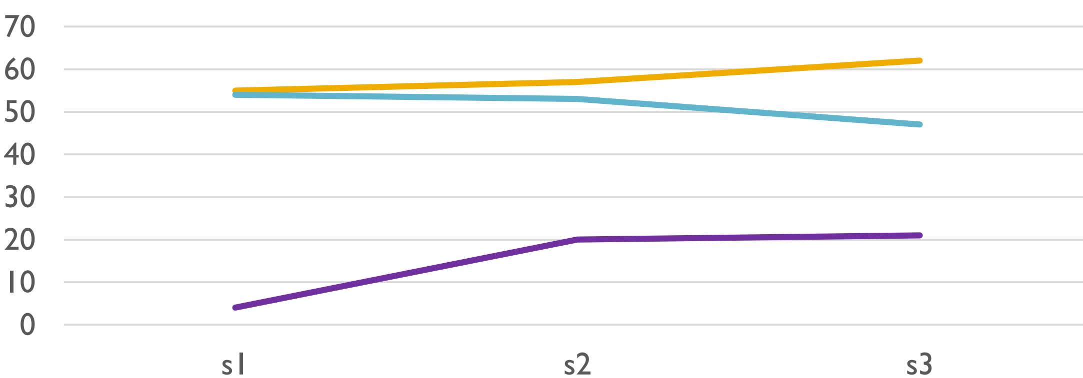
h*, hA, hB result in identical choices: s_1 first!

Minimization, case 3

Strategy: Depth first search; select a child with **minimal** $h(s)$



h*=55	h*=57	h*=62
hA=54	hA=53	hA=47
hB=4	hB=20	hB=21



Which is best?

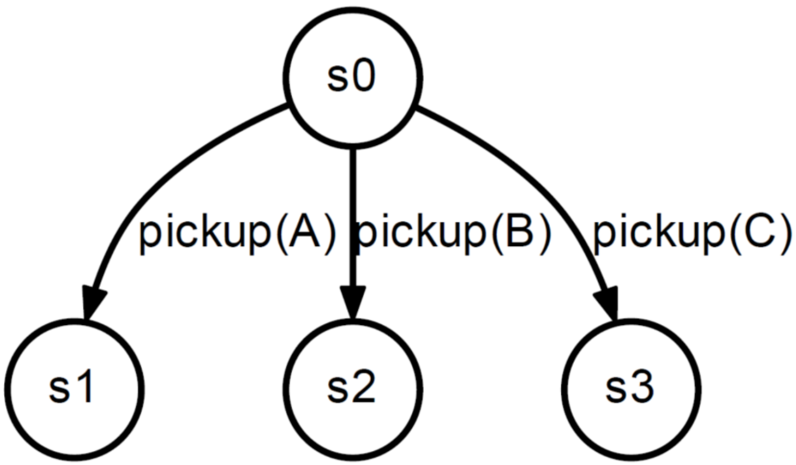
h* and hB result in identical choices

hA is worse, despite being closer to h*: Results in s_3 first

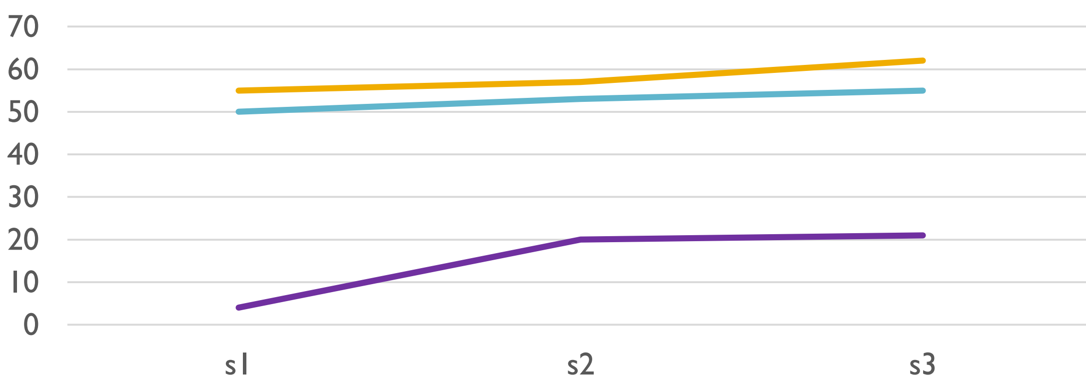
Even if we continue optimally, cost $\geq$ 62!

A*, case 1

Back to case 1 – but suppose the **strategy** is A*



h*=55	h*=57	h*=62
hA=50	hA=53	hA=55
hB=4	hB=20	hB=21



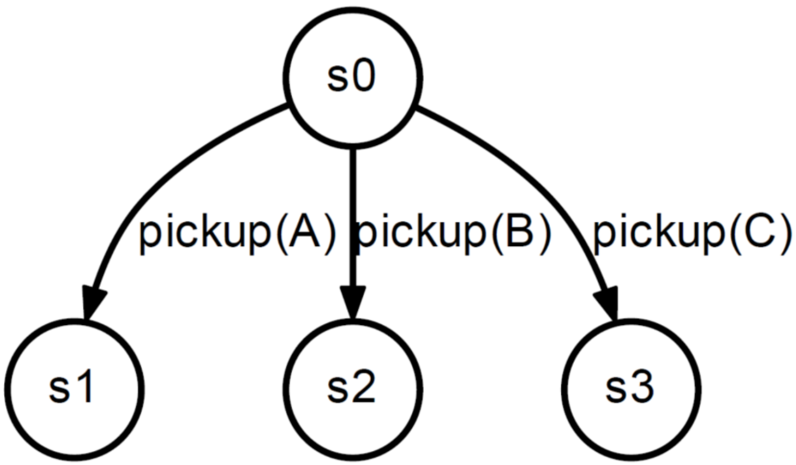
Which is best?

A* expands all nodes where $g(s) + h(s) \leq$ optcost

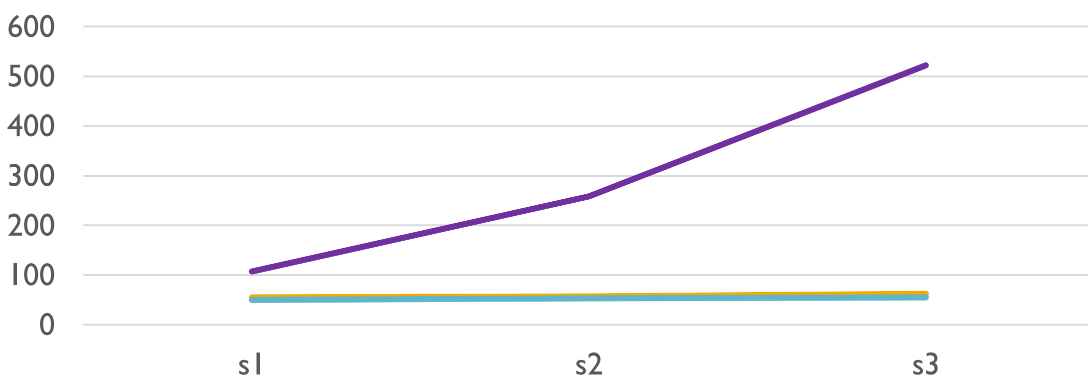
As long as h is admissible $[\forall s: h(s) \leq h^*(s)]$, increasing it is always better

A*, case 2

Case 2: Suppose the **strategy** is A*



h*=55	h*=57	h*=62
hA=50	hA=53	hA=55
hB=107	hB=258	hB=522



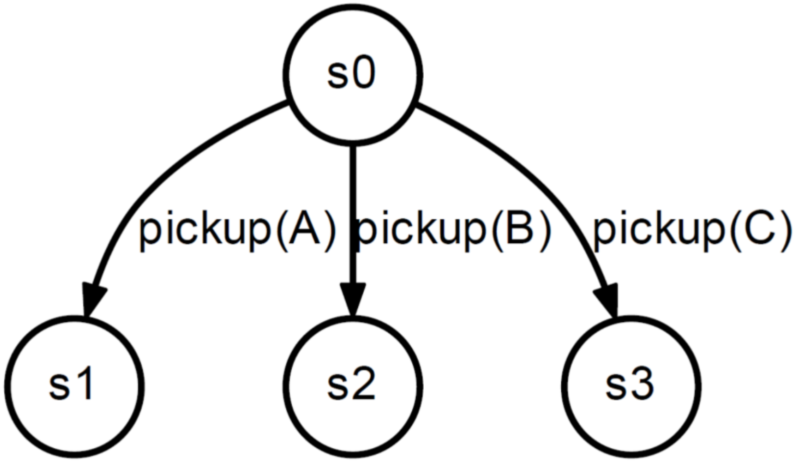
Which is best?

A* expands all nodes where $g(s) + h(s) \leq$ optcost

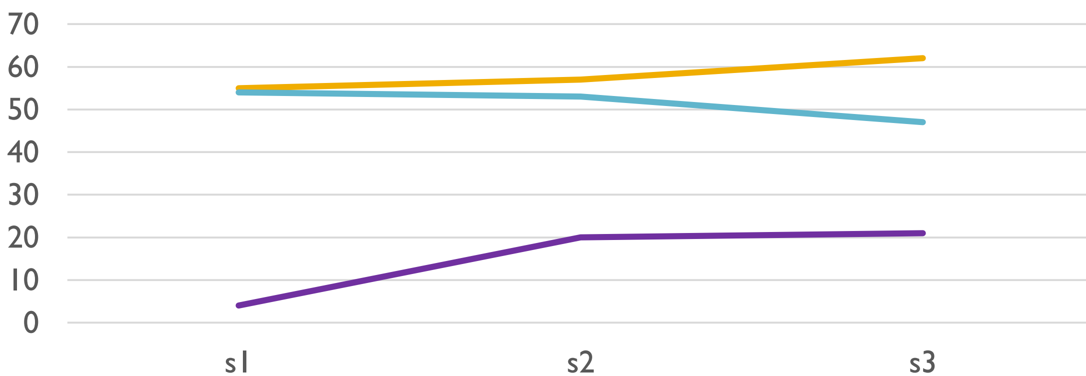
Because hB is not admissible, optimal solutions may be missed!

A*, case 3

Case 3: Suppose the **strategy** is A*



h*=55	**h*=57**	**h*=62**
hA=54	**hA=53**	**hA=47**
hB=4	**hB=20**	**hB=21**



Which is best?

A* expands all nodes where $g(s) + h(s) \leq$ optcost

As long as $h(s)$ is admissible $[h(s) \leq h^*(s)]$, increasing it is always better

hA better than hB

Two Requirements for Heuristic Guidance

- Heuristic planners must consider **two** requirements

Define a **search strategy** able to take guidance into account	Find a **heuristic function** suitable for **the selected strategy**
Examples: A* uses a heuristic function. Hill-climbing uses a heuristic… differently!	**Example**: Find a heuristic function suitable specifically for A* or hill-climbing

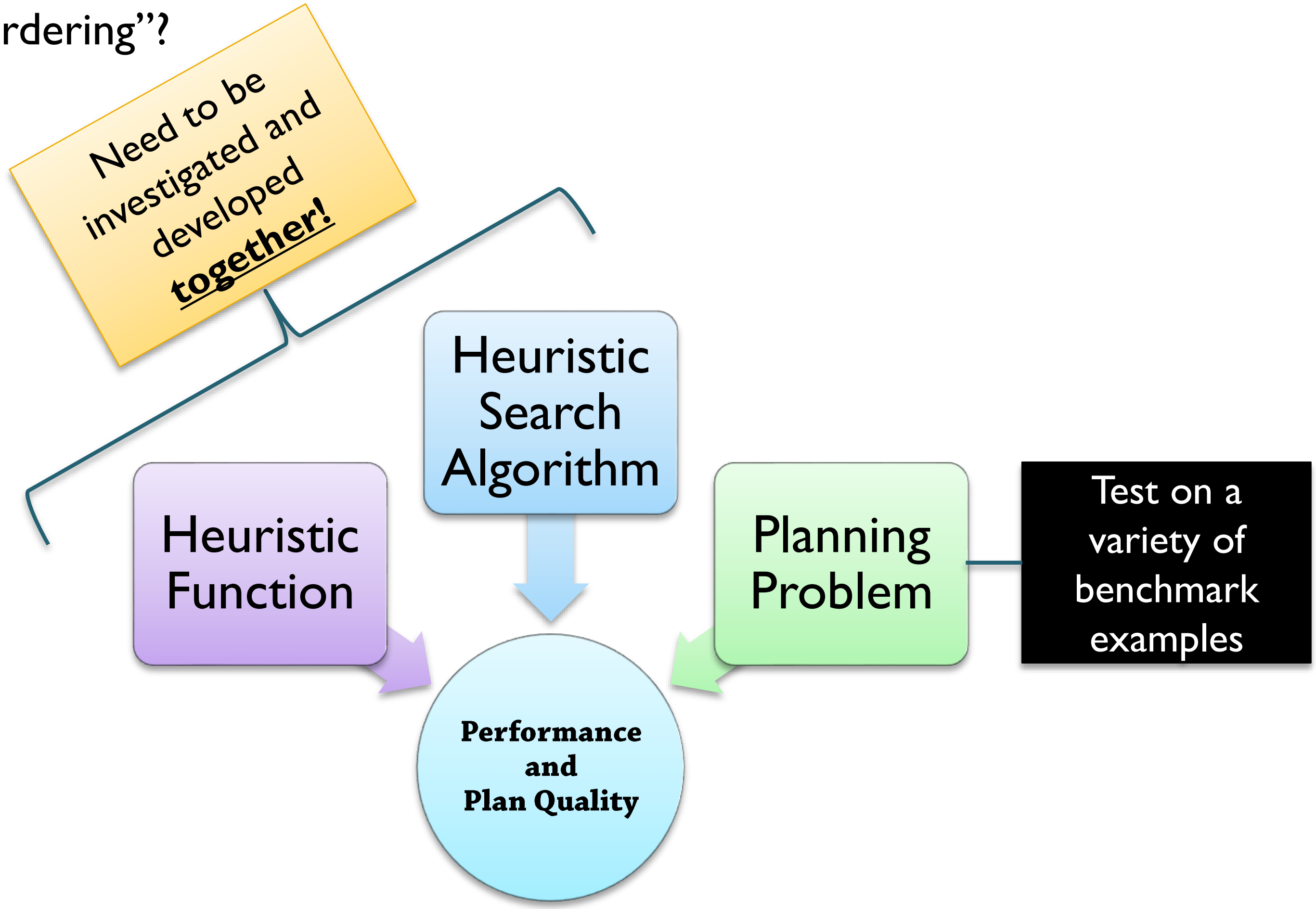
Can be **domain-specific**,
given as input in the planning problem

Can be **domain-independent**,
generated automatically by the planner
given the problem domain

We will consider both – heuristics more than strategies

Some Desired Properties (1)

- What properties do **good heuristic functions** have?
  - **Informative**: Provide good guidance to the specific search strategy we use
    - Close to $h^*(n)$?
    - Correct "ordering"?
    - …



Some Desired Properties (2)

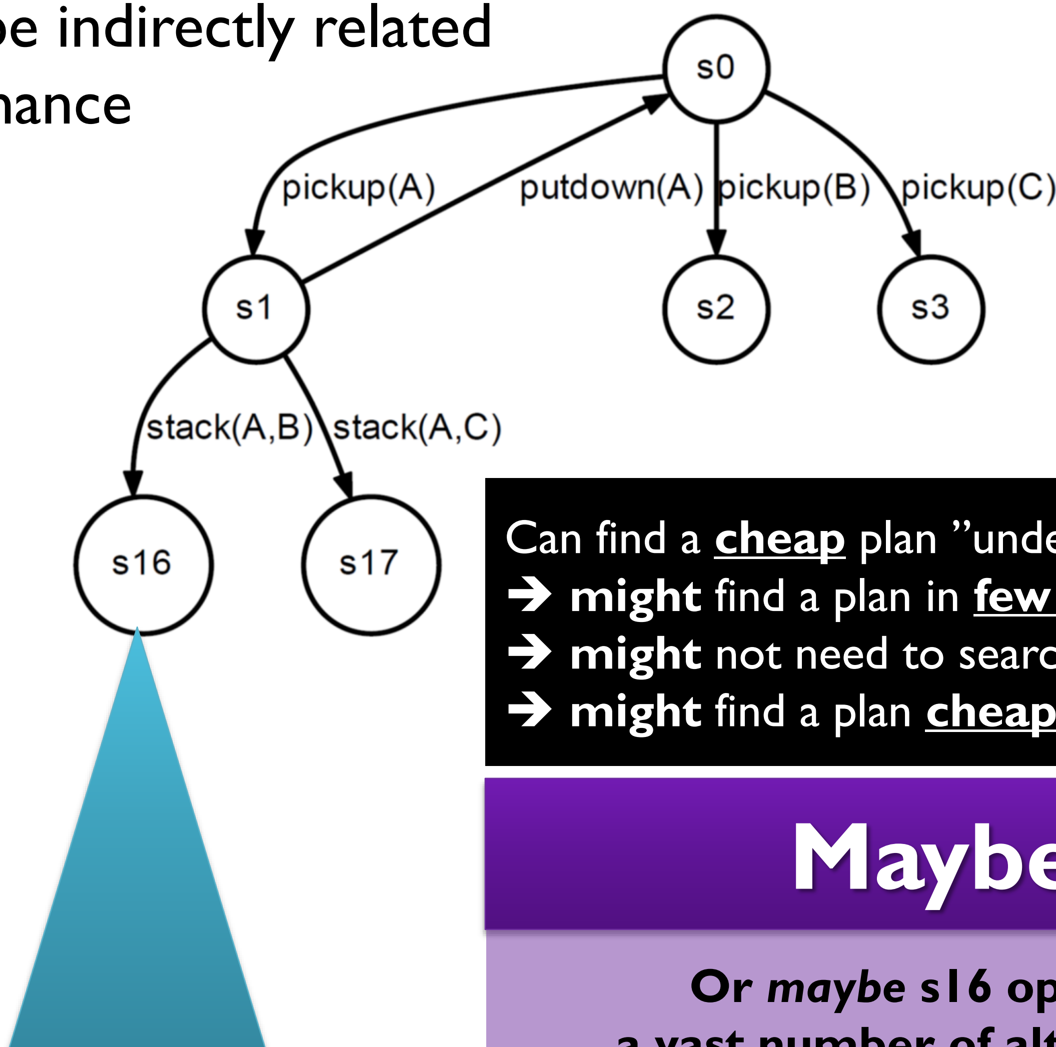
- What properties do **good heuristic functions** have?
  - **Efficiently computable**!
    - Spend as little time as possible deciding which nodes to expand
  - **Balanced**…
    - Many planners spend almost all their time calculating heuristics
    - But: Don't spend more time computing *h* than you gain by expanding fewer nodes!
    - Illustrative (made-up) example:

Heuristic quality	Nodes expanded	Expanding one node	Calculating h for one node	Total time
Worst	100000	100 μs	1 μs	10100 ms
Better	20000	100 μs	10 μs	2200 ms
…	5000	100 μs	100 μs	1000 ms
…	2000	100 μs	1000 μs	2200 ms
…	500	100 μs	10000 μs	5050 ms
Best	200	100 μs	100000 μs	20020 ms

Speed vs. Cost

Cheap Plans, Found Cheaply?

- Cost can be indirectly related to performance



Can find a **cheap** plan "under" s_{16}
➔ **might** find a plan in **few steps**
➔ **might** not need to search so many nodes
➔ **might** find a plan **cheaply**

Maybe!

Or *maybe* s16 opens up a vast number of alternatives, so finding a solution takes more time…

Prioritizing Speed or Plan Cost

Can design **strategies** to **prioritize** speed or plan cost

Find a solution **quickly**

Expand nodes where you think
you can **easily find a way**
to a goal node

Should prefer

Accumulated plan cost 50,
estimated "cost distance" to goal 10

Find a **good** solution

Expand nodes where you think
you **can** find a way
to a **good (high quality) solution**,
even if finding it will be difficult

Should prefer

Accumulated plan cost 5
estimated "cost distance" to goal 30

Often one strategy+heuristic can achieve *both* reasonably well,
but for optimum performance, the distinction can be important!

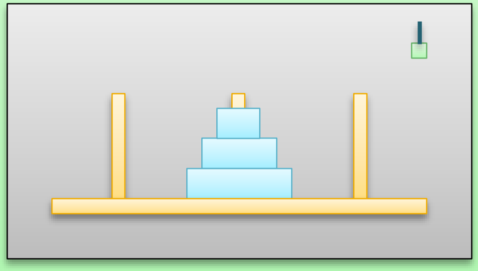
A Simple
Domain-Independent Heuristic
and Search Strategy

Heuristics given Structured States

- In planning, we often want **domain-independent** heuristics
  - Should work for **any** planning domain – how?
- Take advantage of **structured high-level representation**!

- **Plain state transition system**
  - We are in state 572,342,104,485,172,012
  - The goal is to be in one of the 10^47 states in S_g={ s[482,293], s[482,294], … }
  - Should we try action A297,295,283,291 leading to state 572,342,104,485,172,016?
  - Or maybe action A297,295,283,292 leading to state 572,342,104,485,175,201?

- **Classical representation**
  - We are in a state where **disk 1 is on top of disk 2**
  - The goal is for all disks to be on peg C
  - Should we try take(B), leading to a state where we are holding disk 1?
  - …



An Intuitive Heuristic

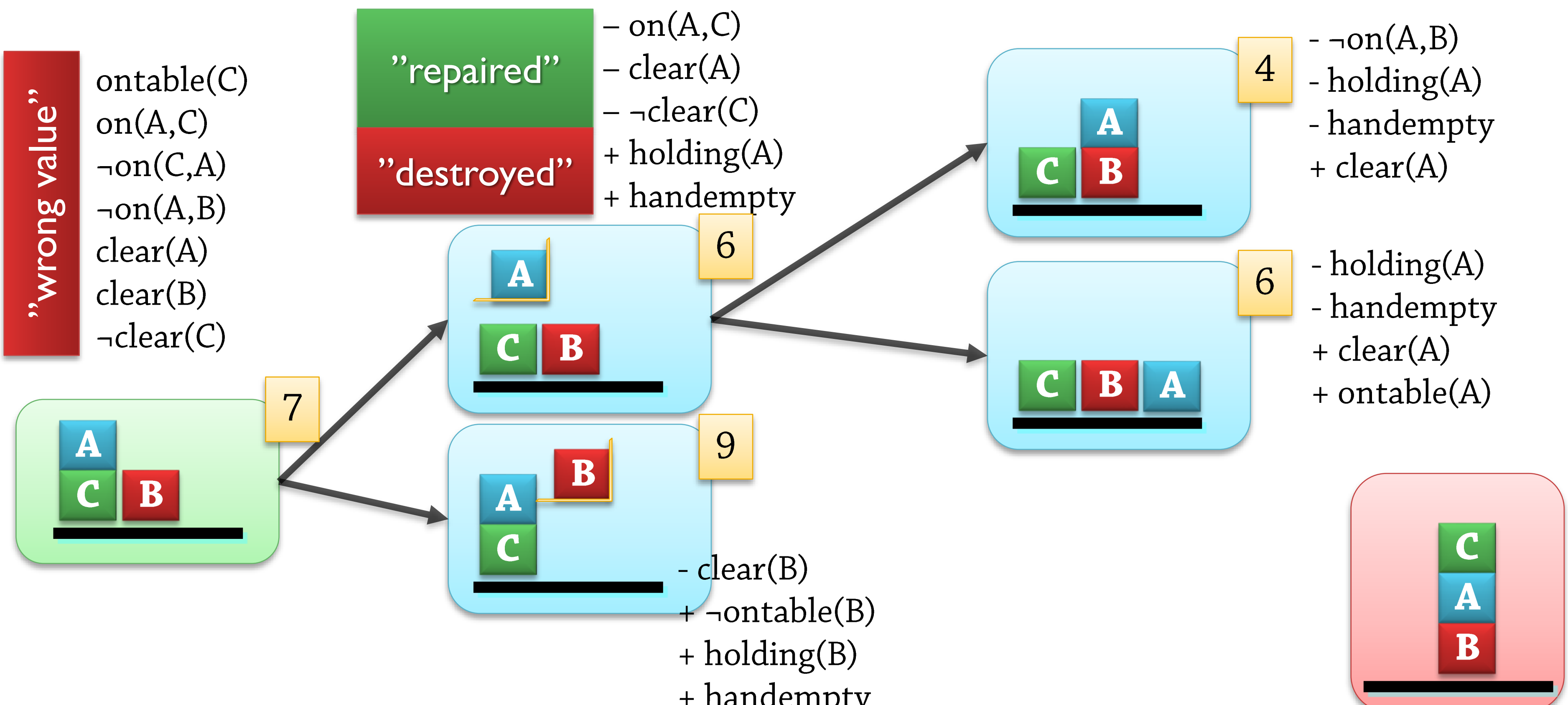
- An **intuitive** heuristic:
  - Number of steps required to reach the goal from *s*
    should be ***approximately proportional to***
    how many goal requirements are not yet achieved in *s*

- An associated **search strategy**:
  - Suppose we want to *minimize planning time*
  - Choose an open node
    with a *minimal* number of remaining goal facts to achieve

Counting Remaining Goals

- **Count** the number of facts that are "wrong"
  - Requires that *states and goals are sets of facts*
  - No

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



Counting Remaining Goals (2)

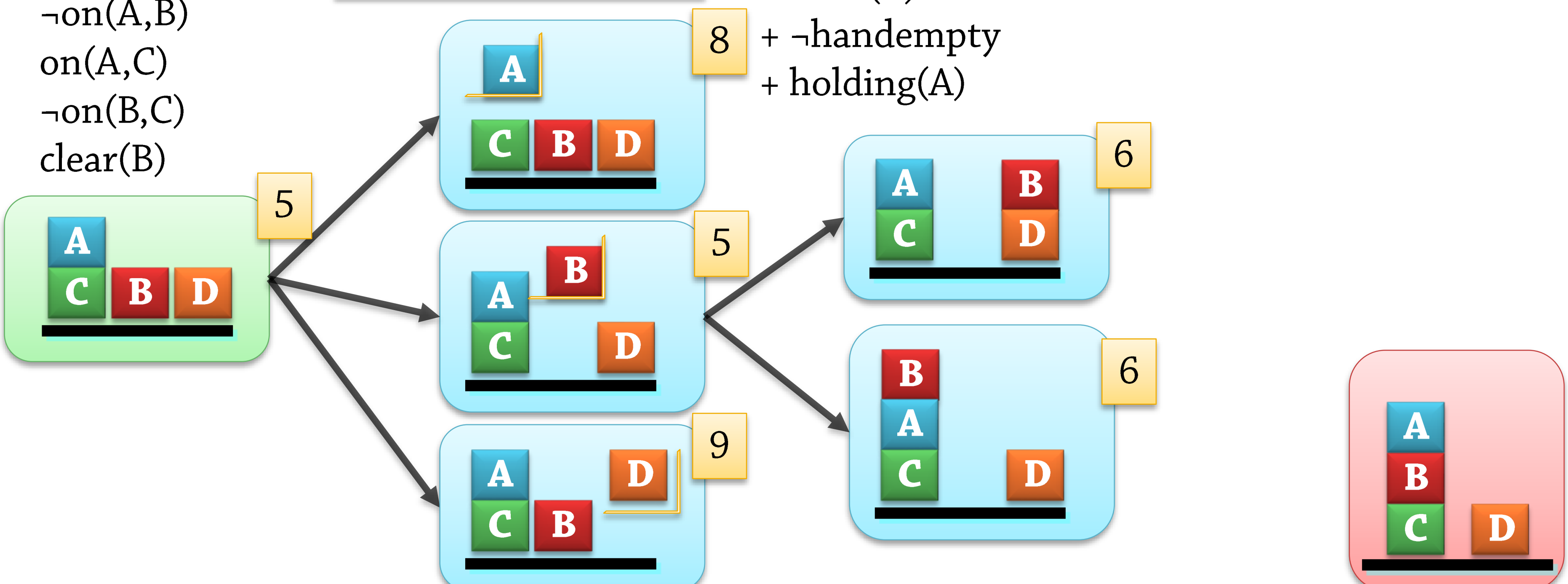
- A **perfect** solution? No!
  - We must often "**unachieve**" individual goal facts to get closer to a goal state!

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

unstack(A,C) best, but *looks* worse

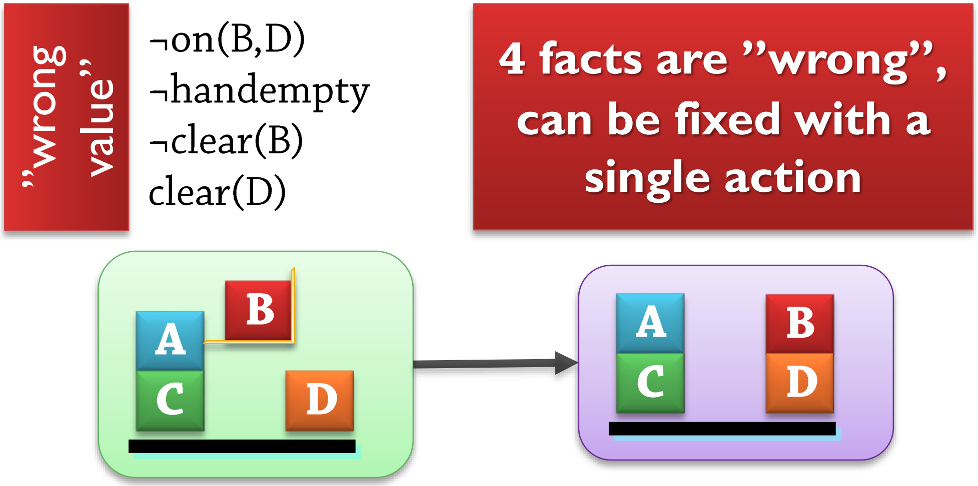
- on(A,C)
+ ¬clear(A)
+ clear(C)
+ ¬handempty
+ holding(A)

ontable(B)
¬on(A,B)
on(A,C)
¬on(B,C)
clear(B)



Counting Remaining Goals (3)

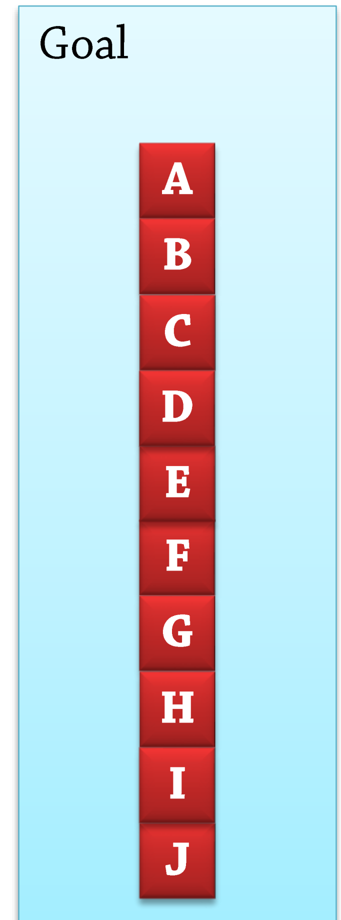
- **Admissible**?
  - No!
  - (Doesn't matter in our chosen search strategy)



- Can we **make** it admissible?
  - Yes: **Divide** by the maximum number of facts modified by any action

Counting Remaining Goals (4)

- **<u>Informative</u>**?
  - Facts to add: **on(I,J)**
  - Facts to remove: **ontable(I), clear(J)**
  - Heuristic value of **3** – but is it close to the goal?



Don't worry:
At least we know that heuristics *can* be *domain-independent*!

Counting Remaining Goals (5): Analysis

- What we see from this example…
  - Not very much: **All heuristics have weaknesses**!

Even the **best planners**
will make "strange" choices,
visit **tens**, **hundreds** or even
thousands of "unproductive" nodes
for every action in the final plan

The heuristic should make sure
we don't need to
visit **millions**, **billions** or even
trillions of " unproductive" nodes
for every action in the final plan!

- But a thorough empirical analysis *would* tell us:
  - **This** heuristic is **far** from sufficient!

Example Statistics

- Planning Competition 2011: Elevators domain, problem 1
  - A* with goal count heuristics
    - States: 108922864 generated, gave up
  - LAMA 2011 planner, good heuristics, other strategy:
    - Solution: 79 steps, 369 cost
    - States: 13236 generated, 425 evaluated/expanded
- Elevators, problem 5
  - LAMA 2011 planner:
    - Solution: 112 steps, 523 cost
    - States: 41811 generated, 1317 evaluated/expanded
- Elevators, problem 20
  - LAMA 2011 planner:
    - Solution: 354 steps, 2182 cost
    - States: 1364657 generated, 14985 evaluated/expande

Important insight:

Even a state-of-the-art planner can't go *directly* to a goal state!

Generates *many* more states than those actually on the path to the goal…

Search Strategies and Heuristics for Optimal Forward State Space Planning

jonas.kvarnstrom@liu.se – 2017

A Well Known Heuristic Search Algorithm: A*

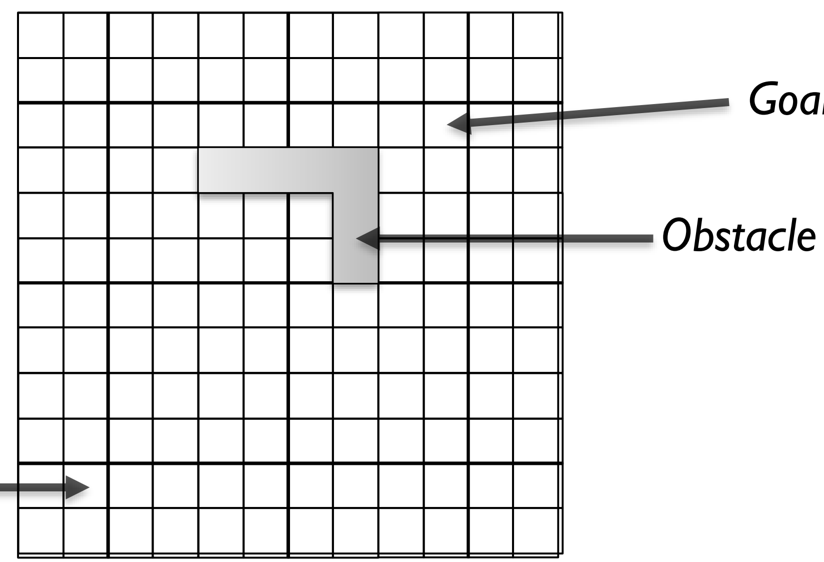
Used in many **optimal** planners

A* (1)

- Dijstra vs. A*: The essential difference

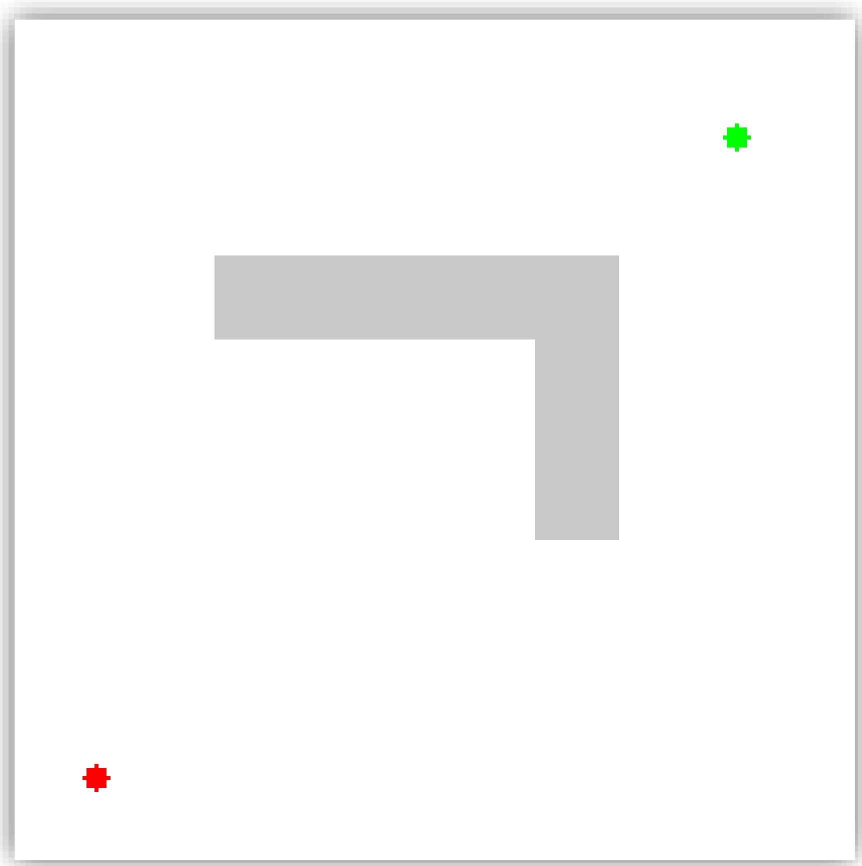
Dijkstra	A*
Selects from *open* a node *n* with minimal g(*n*) – Cost of reaching *n* from initial node	Selects from *open* a node *n* with minimal g(*n*) **<u>+ h(*n*)</u>** – + **<u>underestimated cost of reaching a goal from *n*</u>**
Uninformed (blind)	**Informed**

- Example:
  - **<u>Hand-coded</u>** heuristic function
  - Can move diagonally ➔ h(*n*) = **<u>Chebyshev distance</u>** from *n* to goal = <u>max</u>(abs(n.x-goal.x), abs(n.y-goal.y))
  - Related to **<u>Manhattan Distance</u>** = <u>sum</u>(abs(n.x-goal.x), abs(n.y-goal.y))



A* (2)

- A* Search:



Here:
A single
physical obstacle

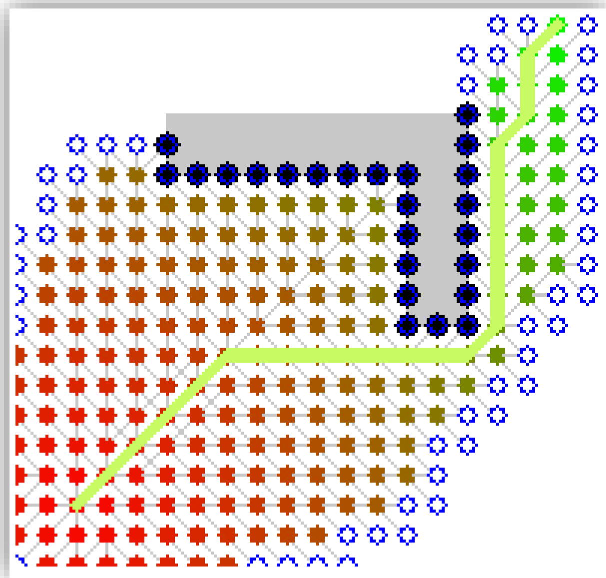
In general:
Many states where
all available actions
will increase g+h
(cost + heuristic)

Investigate *all* states
where g+h=15,
then all states
where g+h=16, …

A* (3)

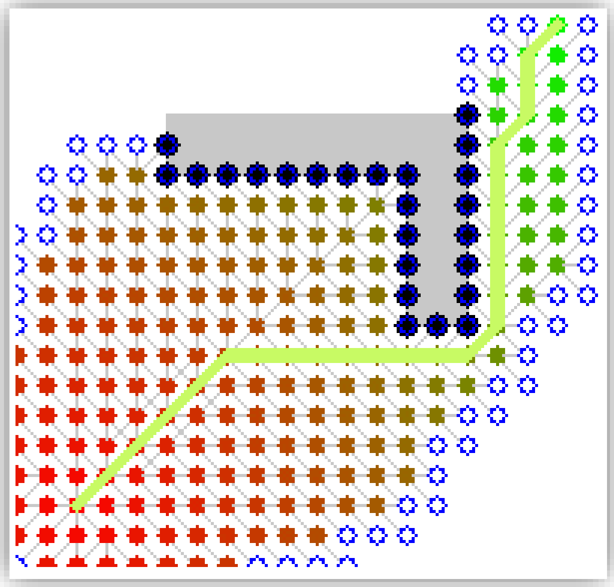
- Given an admissible heuristic *h*, A* is **<u>optimal in two ways</u>**
  - Guarantees an **<u>optimal</u>** plan
  - Expands the **<u>minimum number of nodes</u>** required to *guarantee optimality* with the given heuristic

- Still expands many "unproductive" nodes in the example
  - Because the heuristic is **<u>not perfectly informative</u>**
    - Even though it is hand-coded
    - Does not take **<u>obstacles</u>** into account
  - If we knew h*(n):
    - Expand optimal path to the goal



A* (4)

- What is an **informative** heuristic for A*?
  - Basic requirement: **Must be admissible** ➔ $\forall n. h(n) \leq h^*(n)$
  - As always, $h(n) = h^*(n)$ would be perfect – but not attainable…
  - As indicated before: The *closer* h(n) is to h*(*n*), the *better*
    - Suppose **hA** and **hB** are both **admissible**
    - Suppose **∀n. hA(n) ≥ hB(n)**: hA is at least close to true costs as hB
    - Then A* with hA *cannot* expand more nodes than A* with hB



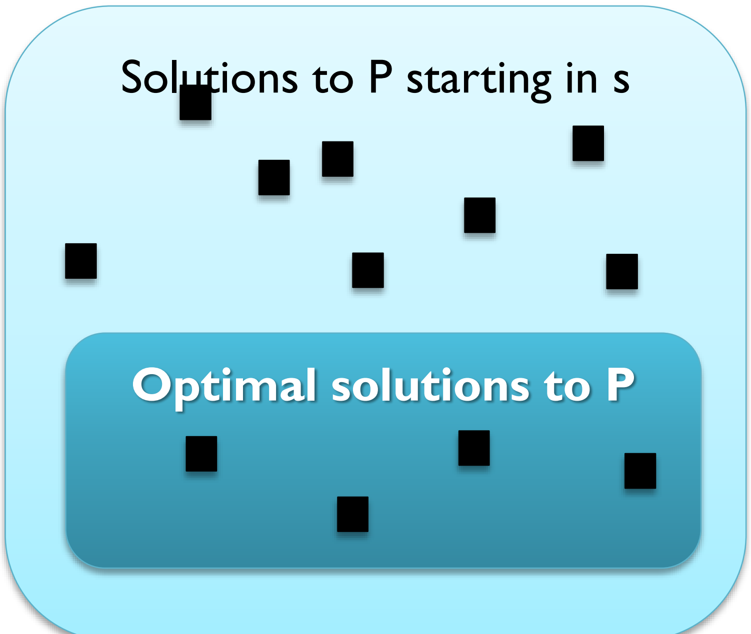
Problem

Given an **arbitrary** planning problem
$P = (\Sigma, s_0, g)$,
find an admissible heuristic function $h(s)$

Creating Admissible Heuristic Functions: The General Relaxation Principle

Fundamental ideas

- For an **arbitrary** problem P and a state s, **compute** an admissible heuristic value $h(s)$



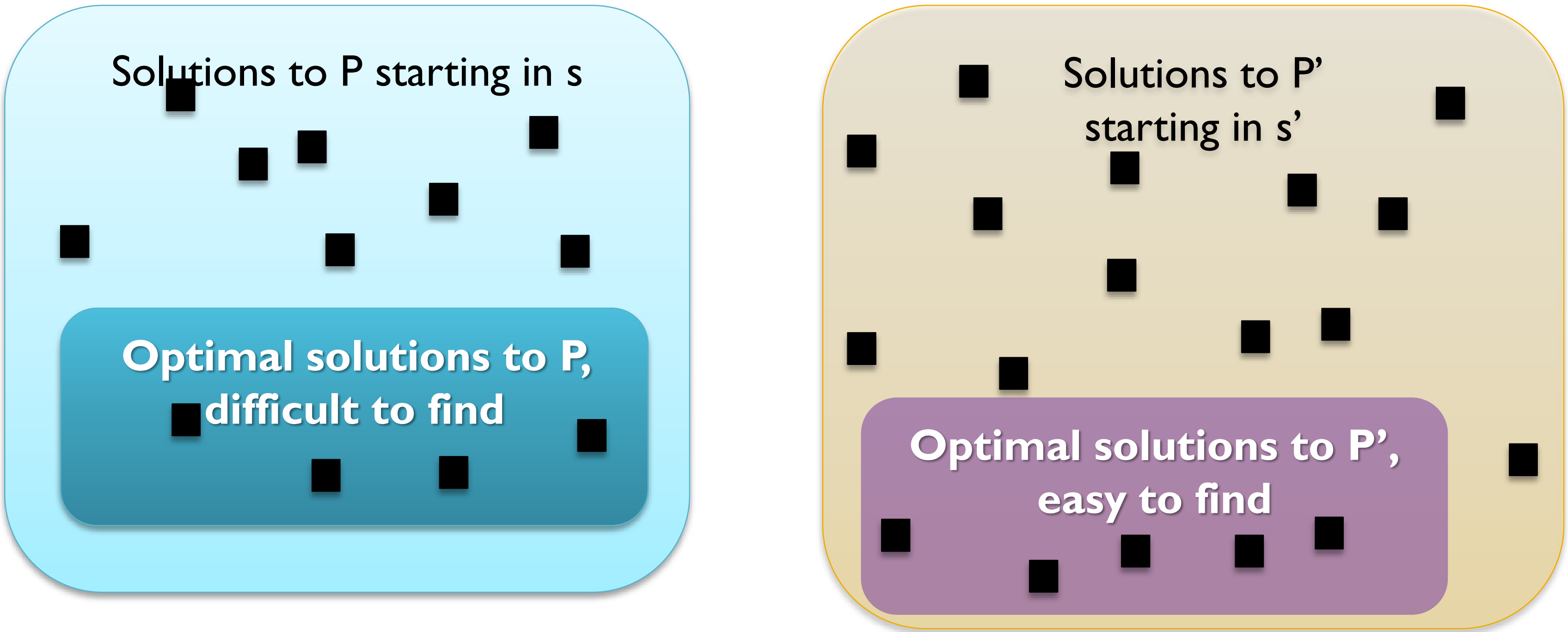
$h(s) \leq h^*(s)$ = cost of optimal solution starting in state s

Find optimal solution π
Return $h(s) = h^*(s) = \text{cost}(\pi)$
➔ Correct but not practical

Fundamental ideas (2)

- For an **<u>arbitrary</u>** problem P and a state s, **<u>compute</u>** an admissible heuristic value $h(s)$

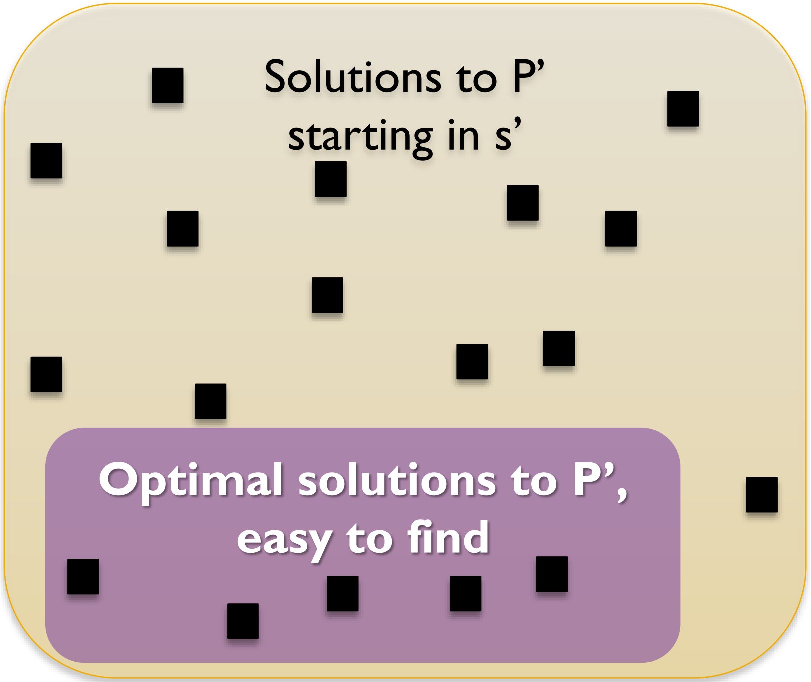
Transform <P,s> into some problem/state <P',s'> that we can **<u>easily</u>** solve optimally



…ensuring cost(optimal-solution(P',s')) ≤ cost(optimal-solution(P,s))

Fundamental ideas (3)

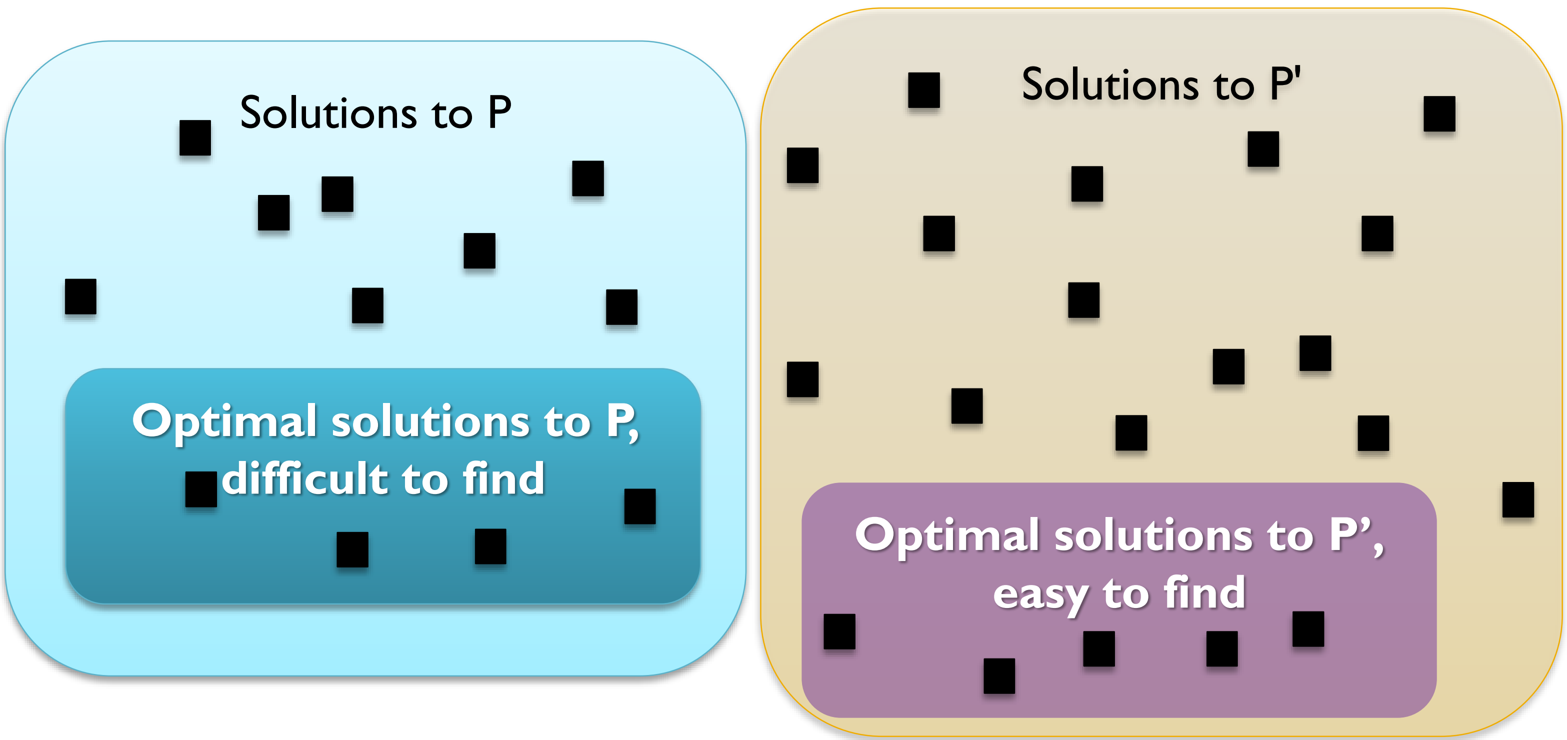
- For an **arbitrary** problem P and a state s, **compute** an admissible heuristic value $h(s)$
  - Solve problem <P',s'> **optimally** resulting in solution π
  - Let $h(s)$ = cost(π)



- We note:
  - $h(s)$ = cost(π) = cost(optimal-solution(P')) $\leq$ cost(optimal-solution(P)) = h*(s)
  - **$h(s)$ is admissible**

Fundamental ideas (4)

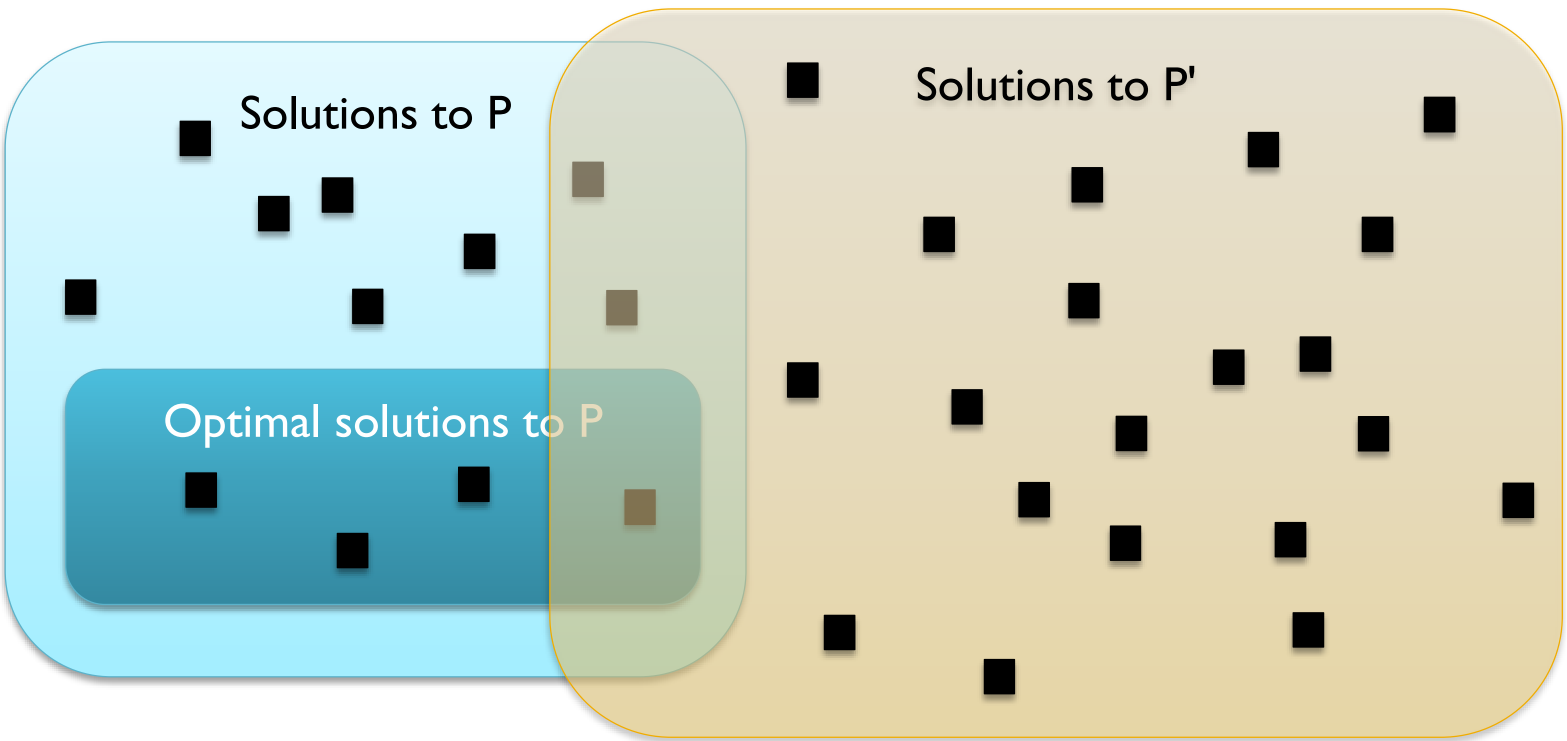
- Important:
  - What we **<u>need</u>**: cost(optimal-solution(P')) $\leq$ cost(optimal-solution(P))
  - **<u>Could</u>** be achieved using *completely disjoint* solution sets
    + a proof that solutions to P' are cheaper



Fundamental ideas (5)

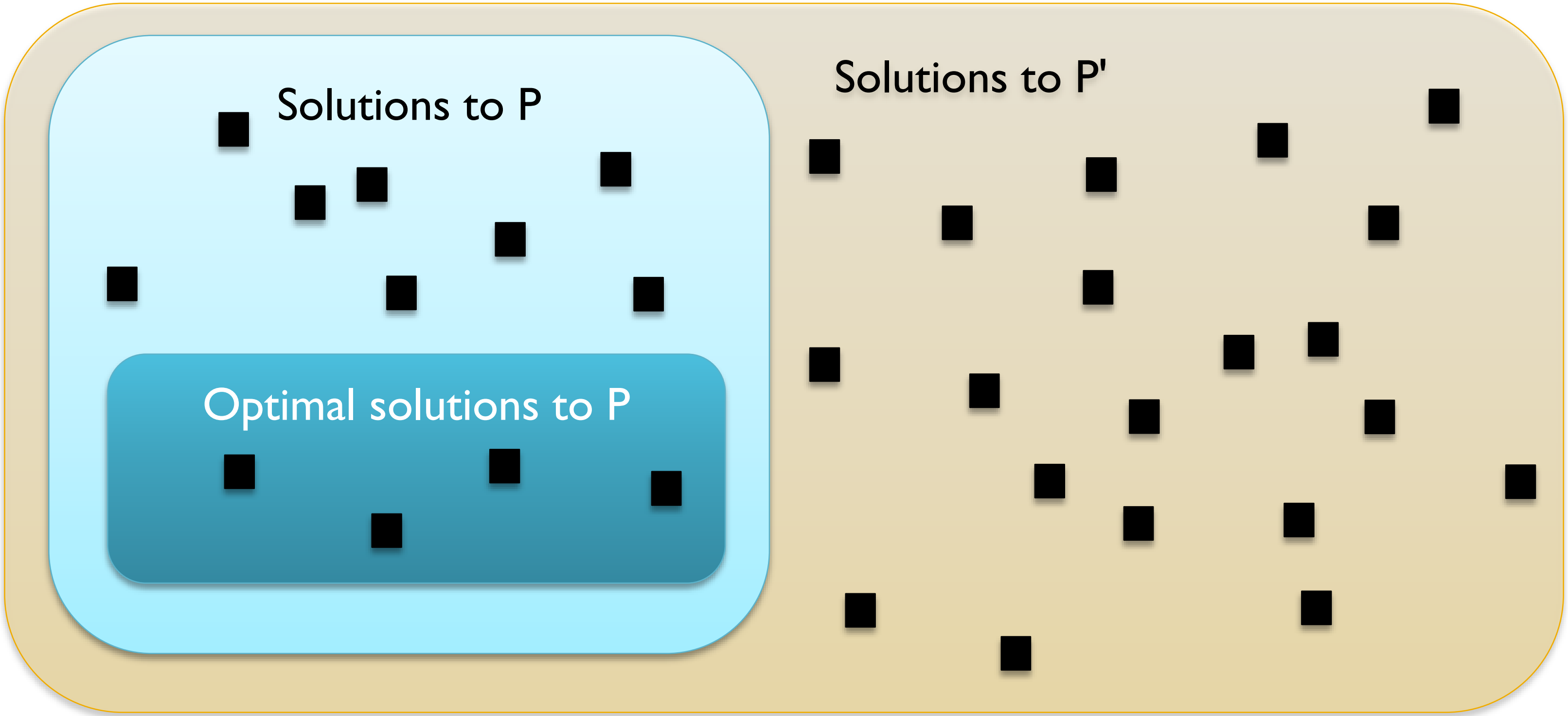
- How to prove cost(optimal-solution(P')) ≤ cost(optimal-solution(P))?
  - Sufficient criterion: **<u>One optimal solution</u>** to P **<u>remains</u>** a solution for P'
    - cost(optimal-solution(P')) = min { cost(π) | π is any solution to P' } <= cost(optimal-solution(P))

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



Relaxation

- A stronger criterion: **<u>All solutions</u>** to P **<u>remain</u>** solutions for P'
  - **<u>This</u>** is called **<u>relaxation</u>**: P' is a relaxed version of P
  - **<u>Relaxes</u>** the constraint on what is accepted as a solution:
    The **is-solution(plan)?** test is "expanded, relaxed" to cover additional plans



Relaxation for Planning Problems

- A classical planning problem $\mathcal{P} = (\Sigma, s_0, S_g)$ has a **set of solutions**
  - *Solutions*($\mathcal{P}$) = { π : π is an executable action sequence leading from s_0 to a state in S_g }
- Suppose that:
  - $\mathcal{P} = (\Sigma, s_0, S_g)$ is a classical planning problem
  - $\mathcal{P}' = (\Sigma', s_0', S_g')$ is another classical planning problem
  - *Solutions*($\mathcal{P}$) $\subseteq$ *Solutions*($\mathcal{P}'$)
- Then (and only then): $\mathcal{P}'$ is a relaxation of $\mathcal{P}$

Solutions for P:

Sol1, cost 10
Sol2, cost 12
Sol3, cost 27

Optimal in P

Solutions for P':

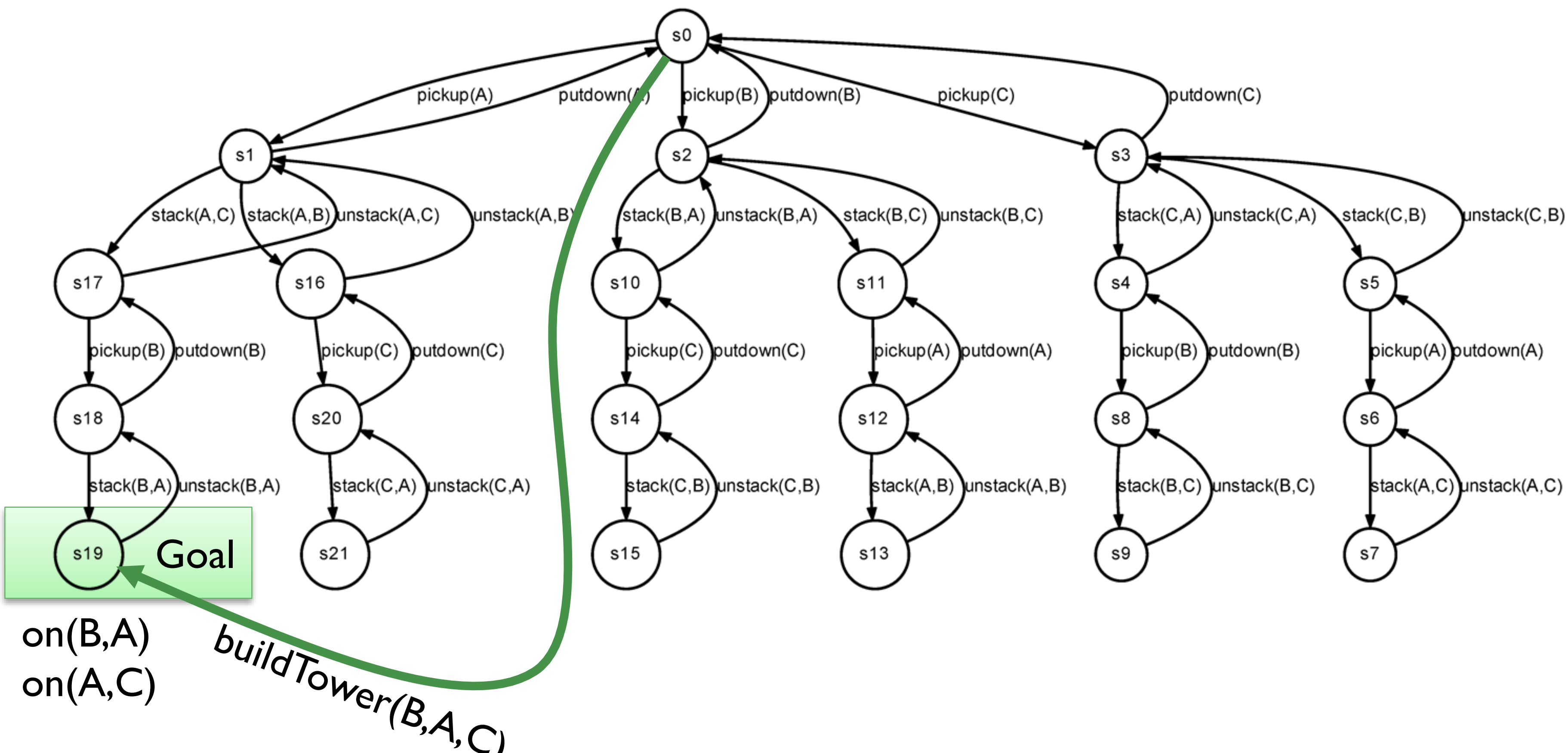
Sol1, cost 10
Sol2, cost 12
Sol3, cost 27
Sol4, cost 8
Sol5, cost 42

All old solutions remain solutions!

Now **sol4** is optimal

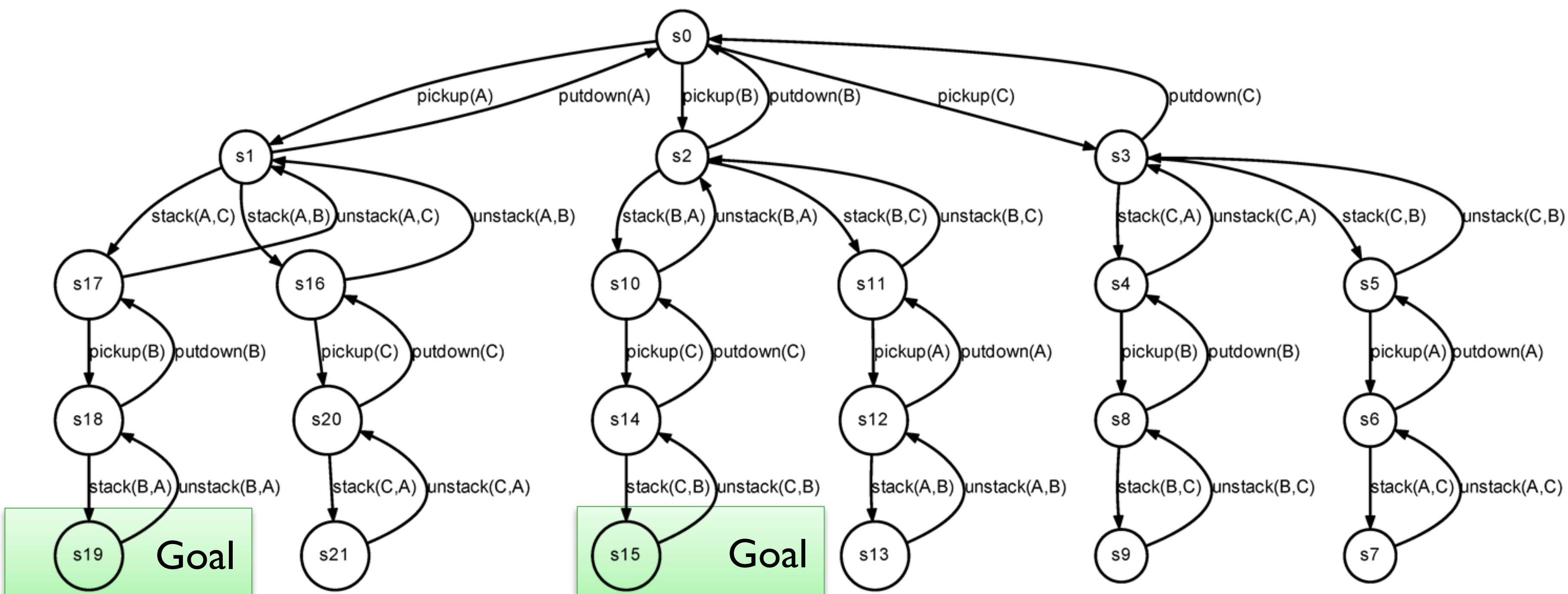
Relaxation Example 1

- Example 1: **Adding new actions**
  - All old solutions still valid, but new solutions may exist
  - Modifies the STS by **adding new edges / transitions**
  - This particular example: *shorter* solution exists



Relaxation Example 2

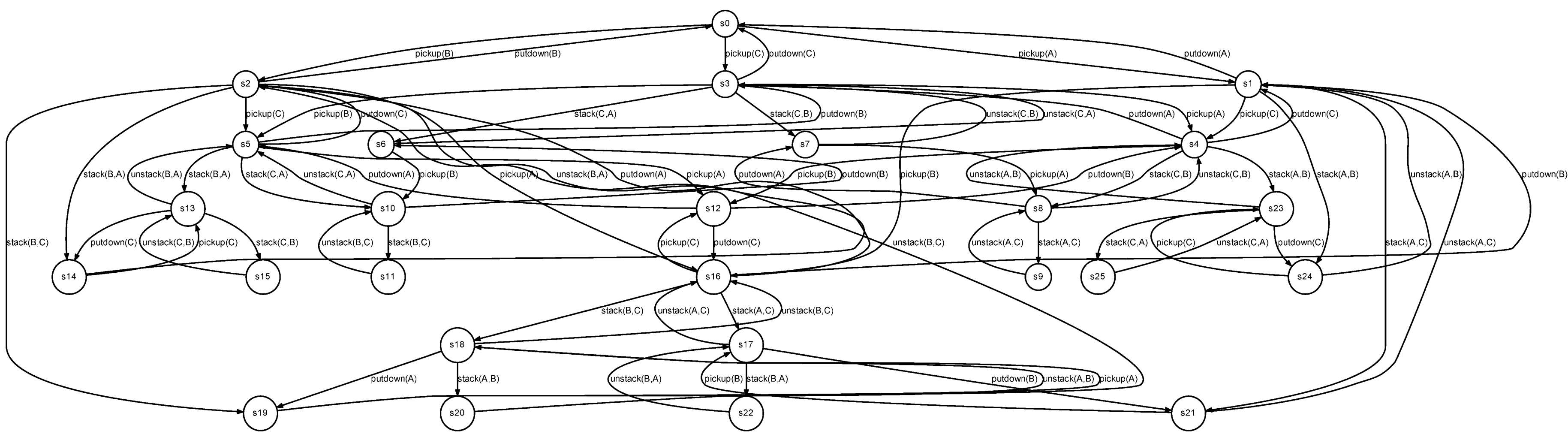
- Example 2: **Adding goal states**
  - All old solutions still valid, but new solutions may exist
  - Retains the **same STS**
  - This particular example: Optimal solution **from s_0** retains the same length



on(B,A)
on(A,C) or on(C,B)

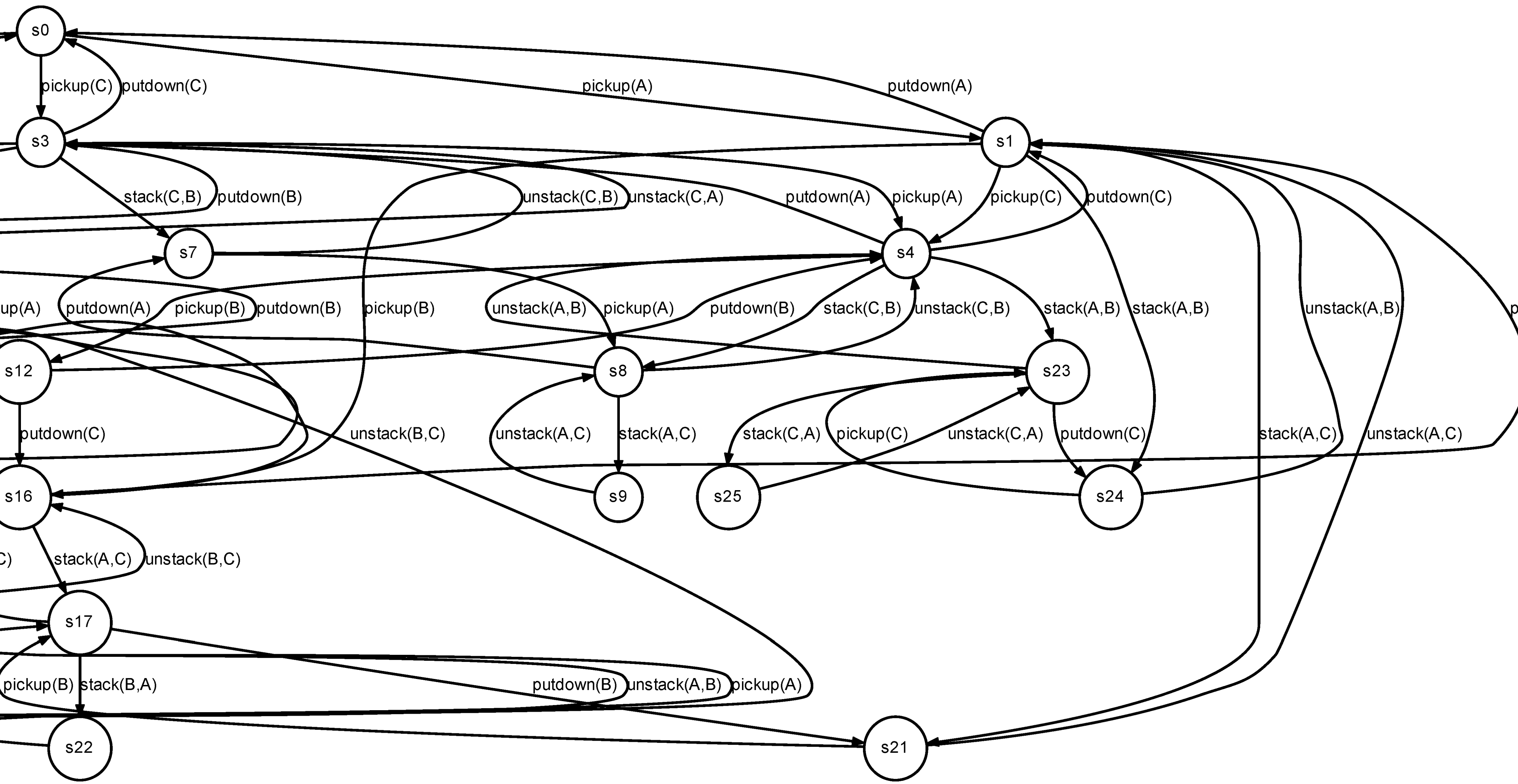
on(B,A)
on(A,C) or on(C,B)

- Example 3: **Ignoring** state variables
  - Ignore the *handempty* predicate in preconditions and effects
  - **Different** state space, no simple addition or removal,
    but all the old solutions (paths) still lead to goal states!
    - 22 reachable states ➔ 26
    - 42 transitions ➔ 72



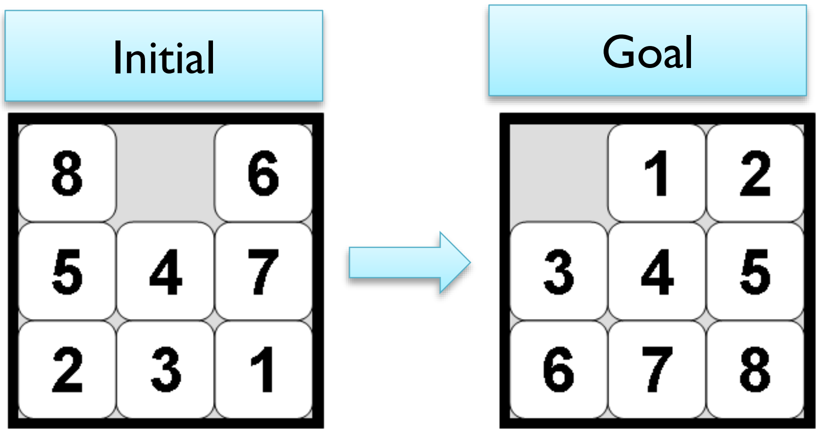
Relaxation Example 3b

- Example 3, enlarged



Relaxation Example 4

- Example 4: **Weakening preconditions** of existing actions



Possible first moves:
Move 8 right
Move 4 up
Move 6 left

  - Precondition relaxation: **Tiles can be moved across each other**
    - Now we have 21 possible first moves: **New transitions** added to the STS
  - All **old solutions are still valid**, but new ones are added
    - To move "8" into place:
    - Two steps to the right, two steps down, ends up in the same place as "1"

Can still be **solved** through **search**
The **optimal** solution for the *relaxed 8-puzzle*
can **never** be more expensive than the optimal solution for *original 8-puzzle*

Relaxation Heuristics: Summary

- **Relaxation**: **One general principle** for designing **admissible** heuristics for **optimal** planning
  - Find a way of transforming planning problems, so that given a problem instance P:
    - **Computing its transformation** P' is easy (polynomial)
    - **Finding an optimal solution** to P' is easier than for P
    - **All solutions to P are solutions to P'**, but the new problem can have additional solutions as well
  - Then the cost of an optimal solution to P' is an admissible heuristic for the original problem P

This is only *one* principle!
There are others, *not* based on relaxation

Relaxation:
Search or Direct Computation ?

Search or Direct Computation (1)

- As stated:
  - Compute an actual solution π for the relaxed problem P'
  - Compute cost(π)

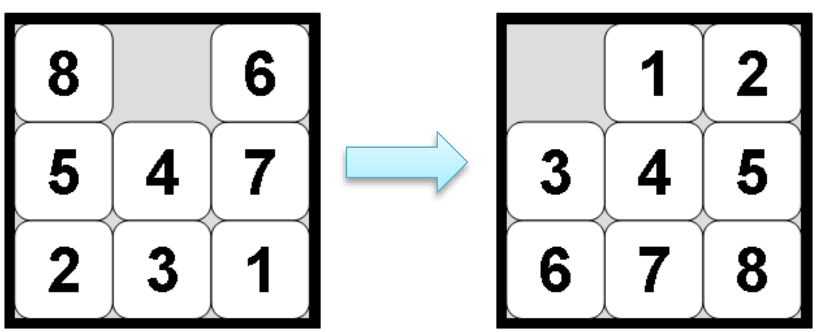
- Example: The **8-puzzle**…
  - Ignore **blank(x,y)** in preconditions and effects
  - Run the problem through an optimal planner
  - Compute the cost of the resulting plan π

```
(:action move-up
    :parameters (?t ?px ?py ?by)
    :precondition (and
                    (tile ?t) (position ?px) (position ?py) (position ?by)
                    (dec ?by ?py) (blank ?px ?by) (at ?t ?px ?py))
    :effect (and (not (blank ?px ?by)) (not (at ?t ?px ?py))
                 (blank ?px ?py) (at ?t ?px ?by)))
```

Search or Direct Computation (2)

- But we never use π!
  - Let's **analyze** the problem…
    - Each piece has to be moved to the intended row
    - Each piece has to be moved to the intended column
    - These are **exactly** the required actions given the relaxation!
  - → **optimal cost** for relaxed problem = sum of Manhattan distances
  - → **admissible heuristic** for *original* problem = sum of Manhattan distances
  - → **Cost** of optimal solution π can be computed efficiently:

$$\sum_{p \in pieces} xdistance(p) + ydistance(p)$$



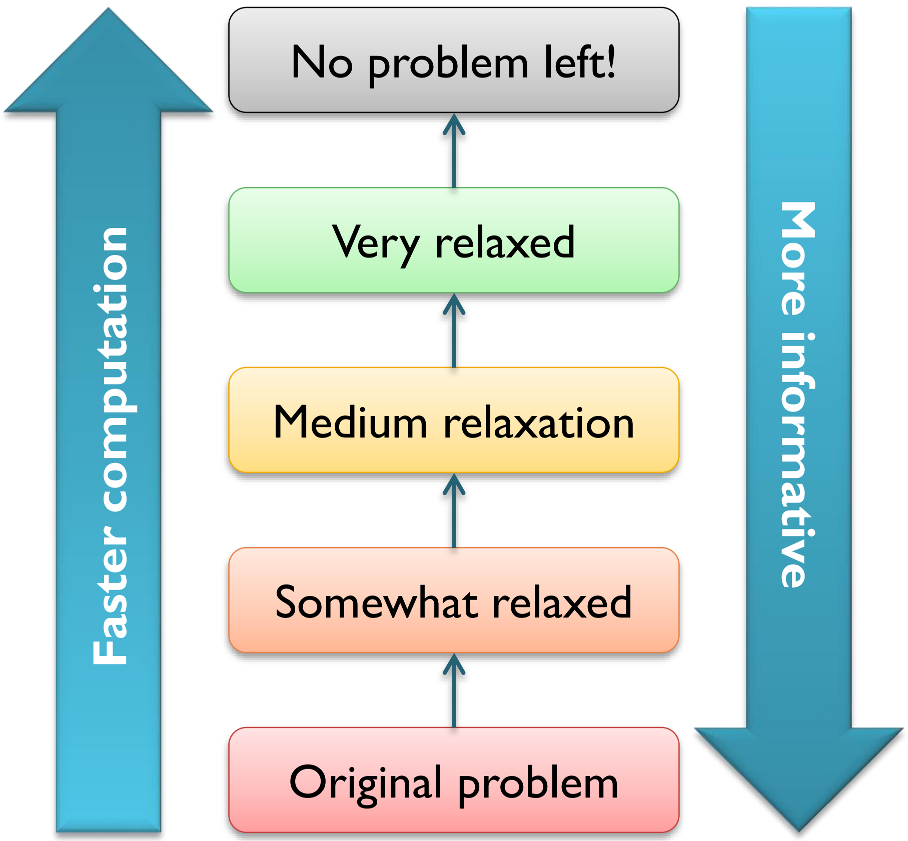
But now we had to **analyze** the problem:
(1) Decide to ignore "blank"
(2) Find "sum of manhattan distances"

Soon: How do we *automatically* find good relaxations + computation methods?

Relaxation:
Essential Facts

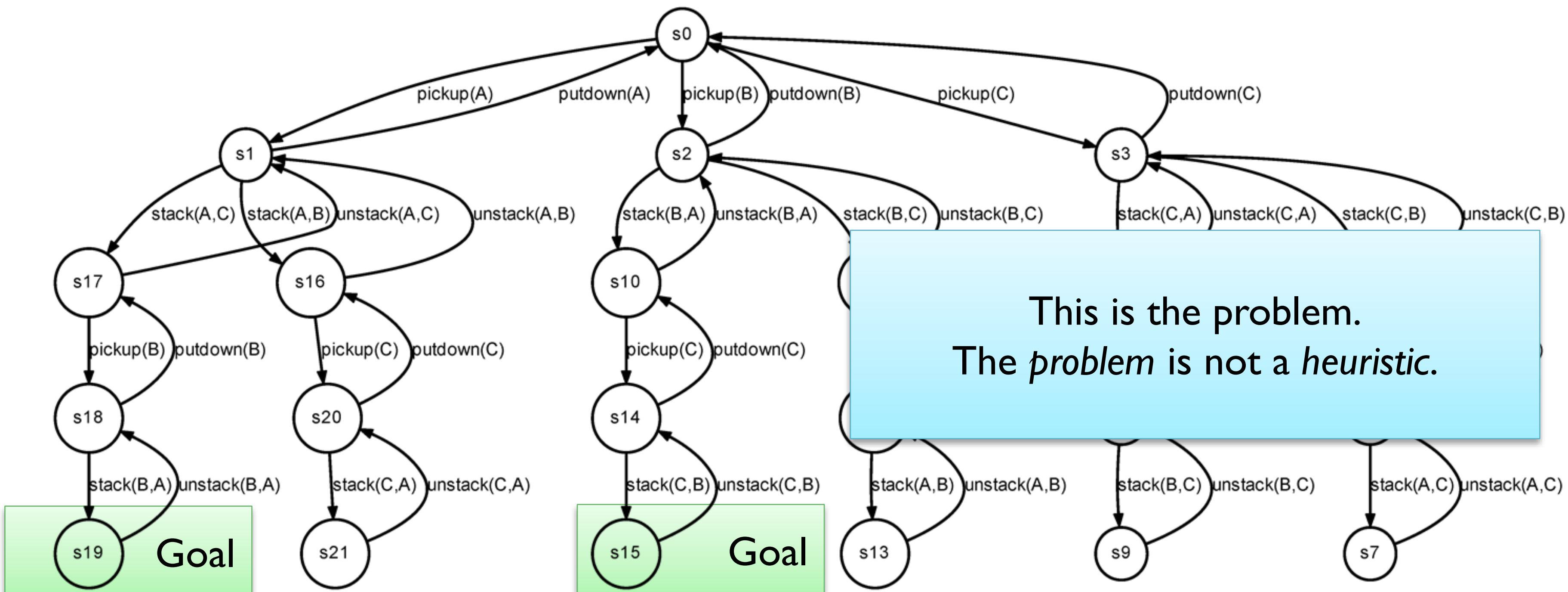
Relaxation Heuristics: Balance

- The **reason** for relaxation is **rapid calculation**
  - **Shorter solutions** are an *unfortunate side effect*: Leads to less informative heuristics
  - Relax too much ➔ not informative
    - Example: Any piece can teleport into the desired position ➔ h(n) = *number* of pieces left to move



Relaxation Heuristics: Important Issues!

You **cannot** "use a relaxed problem as a heuristic".
What would that mean?
You use the **cost** of an **optimal** **solution** to the relaxed problem as a heuristic.



This is the problem.
The *problem* is not a *heuristic*.

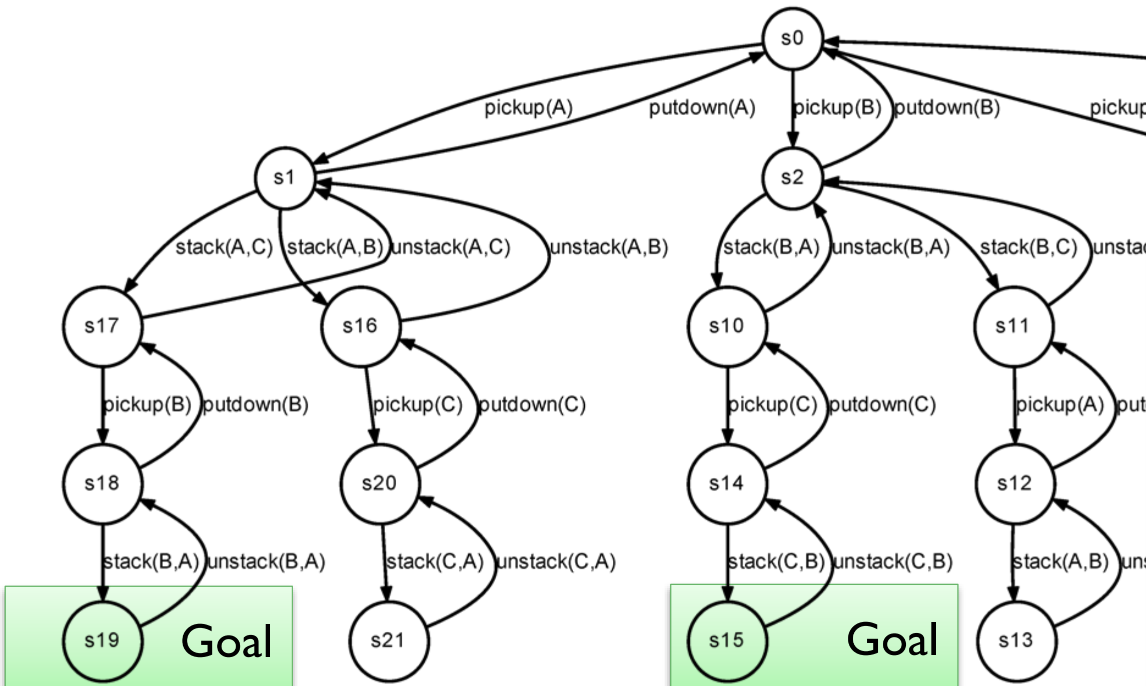
on(B,A)
on(A,C) or on(C,B)

on(B,A)
on(A,C) or on(C,B)

Relaxation Heuristics: Important Issues!

<u>Solving</u> the relaxed problem
<u>can</u> result in a more expensive solution
➔ inadmissible!

You have to solve it <u>optimally</u> to get the admissibility guarantee.



on(B,A)
on(A,C) or on(C,B)

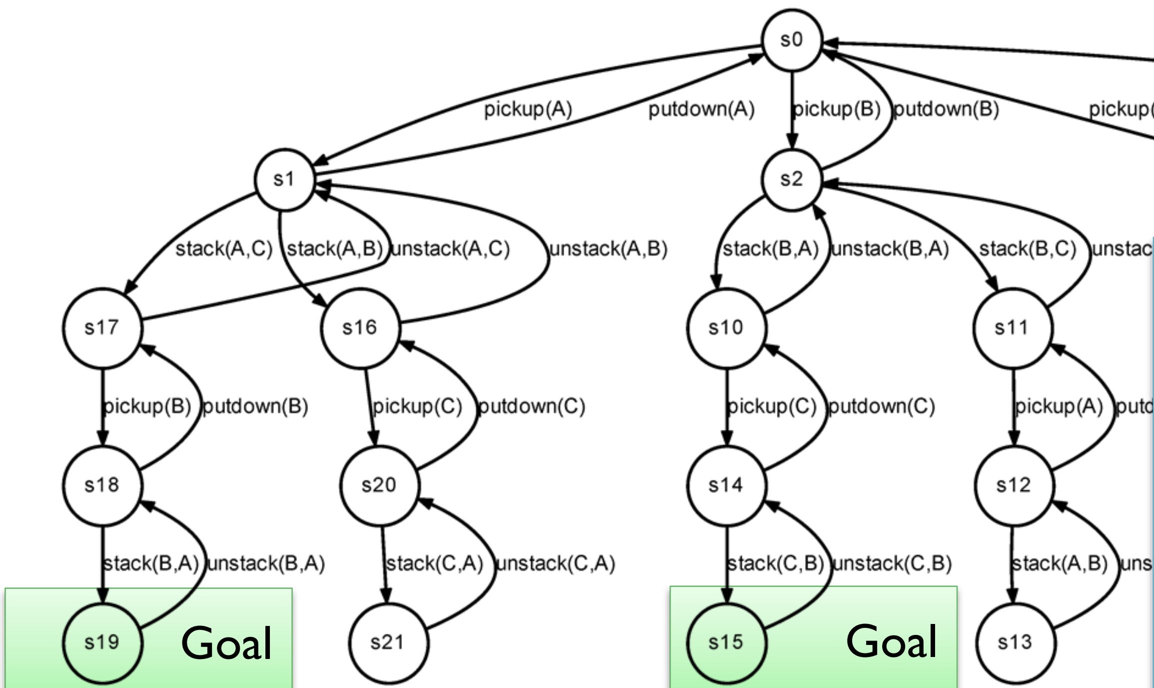
on(B,A)
on(A,C) or on(C,B)

One solution to relaxed problem:

pickup(C)
putdown(C)
pickup(B)
stack(B,A)
pickup(C)
stack(C,B)

Relaxation Heuristics: Important Issues!

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



on(B,A)
on(A,C) or on(C,B)

on(B,A)
on(A,C) or on(C,B)

Calculate:

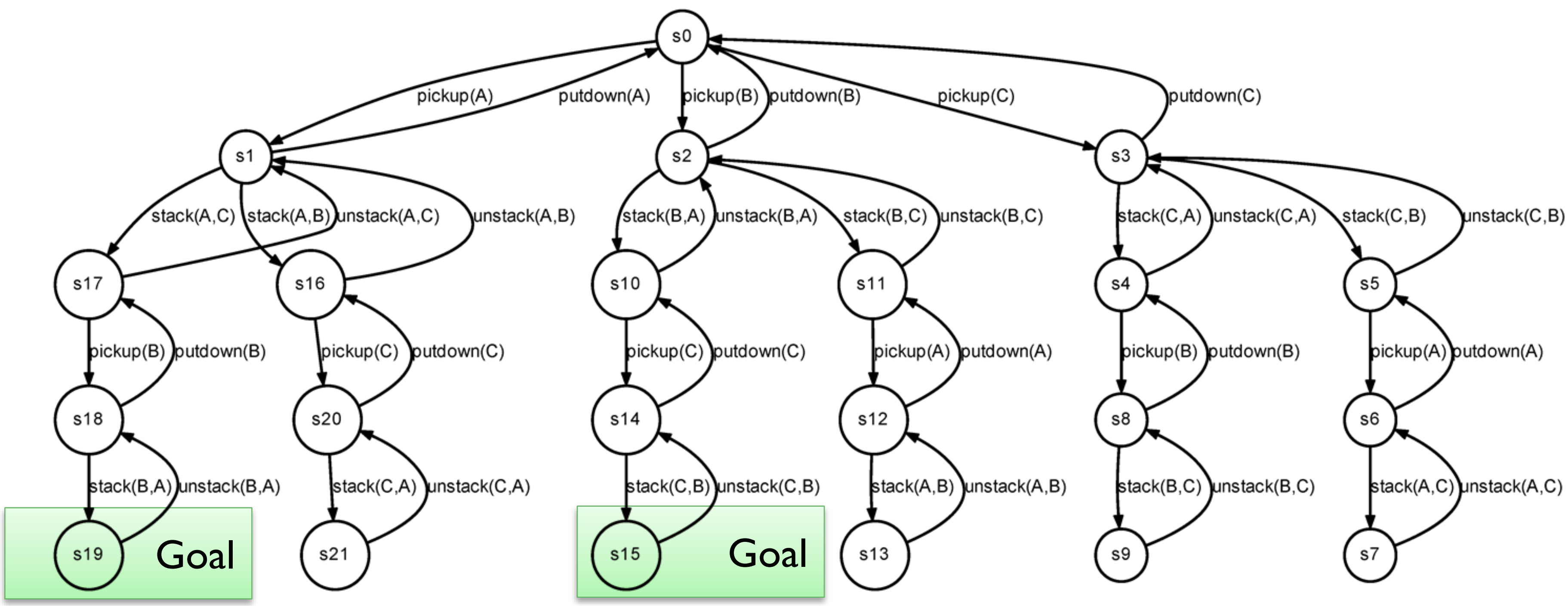
h(s0)
h(s1), h(s2), h(s3)

…then for every node you create, depending on the strategy

jonkv@ida

Relaxation Heuristics: Important Issues!

Relaxation does **not** always mean "**removing constraints**"
in the sense of *weakening preconditions* (moving across tiles, removing walls, …)
Sometimes we get new *goals*. Sometimes the entire *state space* is transformed.
Sometimes action *effects* are modified, or some other change is made.
What defines relaxation: **All old solutions are valid, new solutions may exist**.



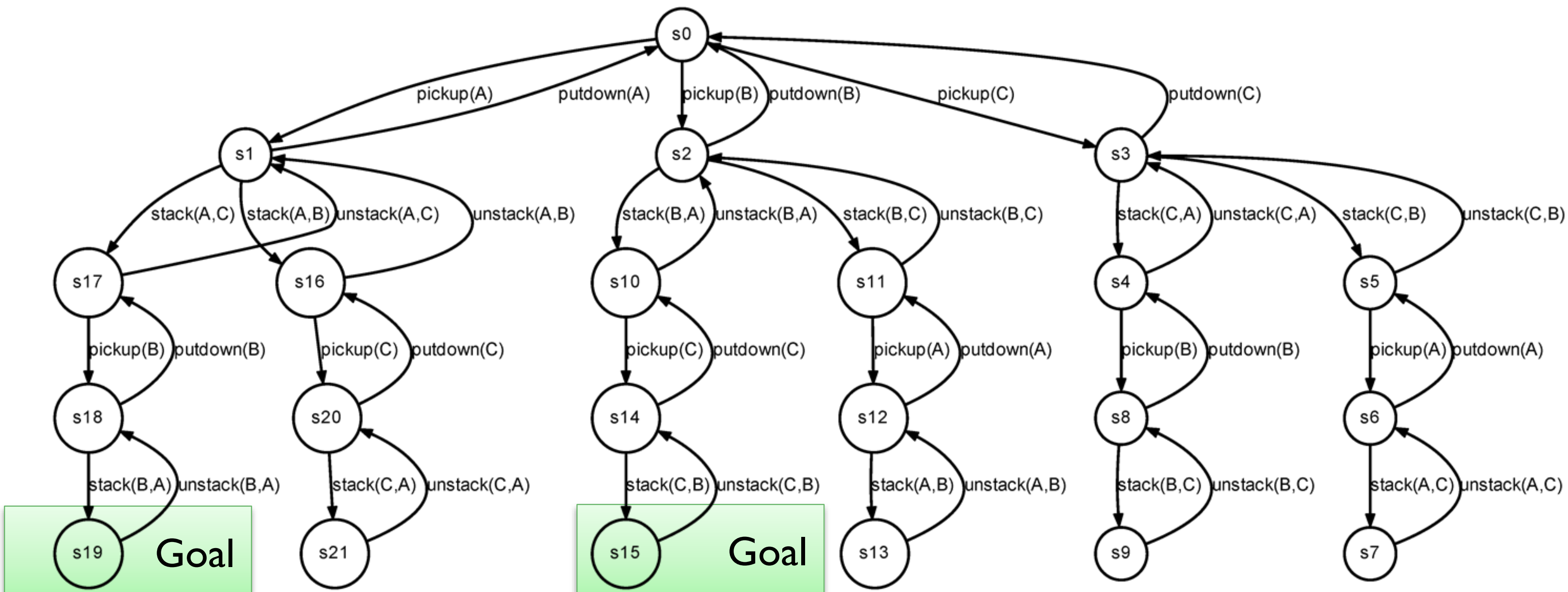
on(B,A)
on(A,C) or on(C,B)

on(B,A)
on(A,C) or on(C,B)

Admissibility: Important Issues!

Relaxation is useful for finding **admissible heuristics**.

A heuristic cannot be **admissible for some states**.
Admissible == does not overestimate costs for *any* state!



on(B,A)
on(A,C) or on(C,B)

on(B,A)
on(A,C) or on(C,B)

Admissibility: Important Issues!

If you are asked "why is a relaxation heuristic admissible?", don't answer "because it cannot overestimate costs". This is the *definition* of admissibility!

"Why is it admissible?" == "*Why* can't it overestimate costs?"

Admissible heuristics *can* "lead you astray" and you *can* "visit" suboptimal solutions.

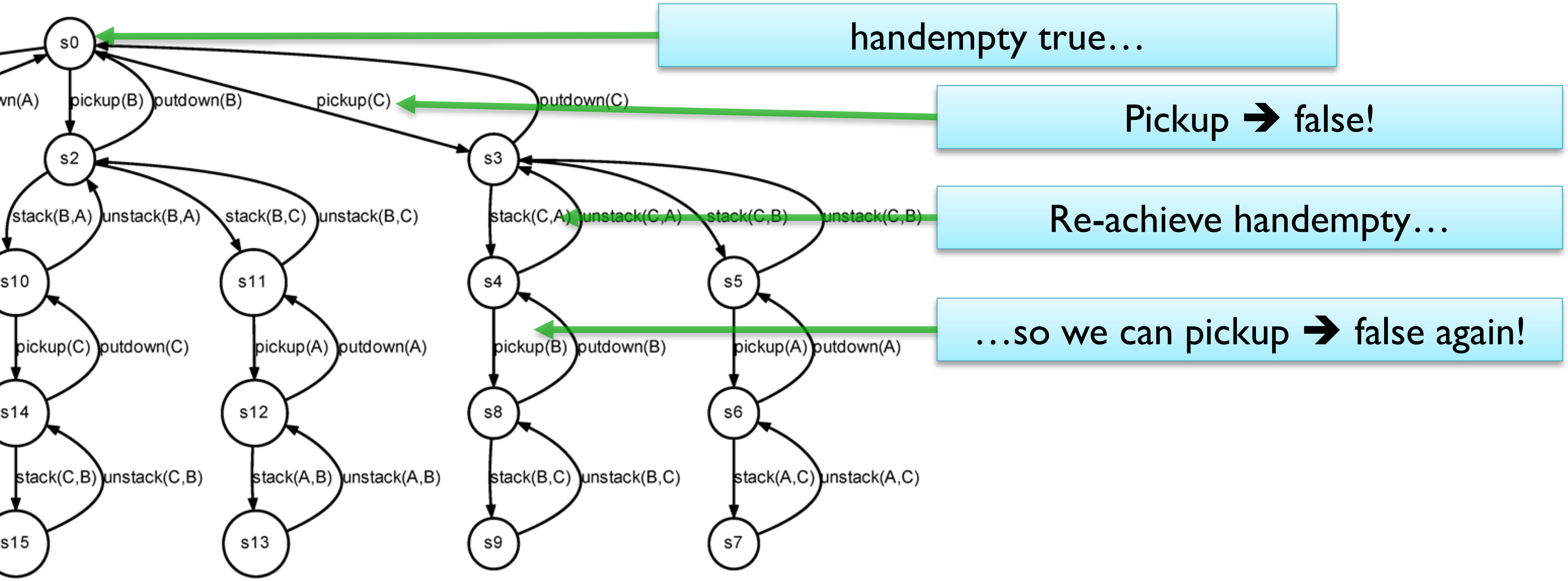
But with the right search strategy, such as A*,
the planner will eventually get around to finding an optimal solution.
This is not the case with A* + non-admissible heuristics.

Delete Relaxation

Delete Relaxation (1)

- In classical planning:
  - **Negative effects** can "un-achieve" goals or preconditions
  - A plan may have to achieve the same fact many times

- Example: If **handempty** is a goal



Delete Relaxation (2)

- Suppose we **remove all negative effects**
  - **Example**: (unstack ?x ?y)
    - **Before transformation:**
      :precondition (and (handempty) (clear ?x) (on ?x ?y))
      :effect (and (not (handempty)) (holding ?x) (not (clear ?x)) (clear ?y)
      (not (on ?x ?y))
    - **After transformation:**
      :precondition (and (handempty) (clear ?x) (on ?x ?y))
      :effect (and (holding ?x) (clear ?y))
  - A fact that is achieved **stays** achieved

Is this a relaxation?

Delete Relaxation (3)

- Suppose we use the book's **classical representation**:
  - Precondition = set of **literals** that must be true
  - Goal = set of **literals** that must be true
  - Effects = set of **literals** (making **atoms** true or false)

  - Suppose we have a solution <**A1,A2**>:
    - Initially handempty
    - Action A1 → handempty := false
    - Action A2 → requires handempty

  - Remove all negative effects:
    - Initially handempty
    - Action A1 → no effect
    - Action A2 → requires handempty, **not executable**

  - <**A1,A2**> is no longer a solution; **can't be a relaxation**

Delete Relaxation (4)

- Suppose we use PDDL's plain **:strips** level
  - **Forbids negative preconditions / goals**
    - Precondition = set of **atoms** (no negations!)
    - Goal = set of **atoms** (no negations!)
    - Effects = set of **literals** (making **atoms** true or false)
  - No solution can *depend on* a fact being false in a visited state
  - No solution can *disappear* because we stop making facts false

This is a **relaxation** if **the problem lacks negative preconditions / goals**!

Delete Relaxation (5): Example

STS for the original problem

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

unstack(A,C)

ontable(B)
ontable(C)
clear(B)
clear(C)
holding(A)

pickup(B)

on(A,C)
ontable(C)
clear(A)
holding(B)

Delete-relaxed STRIPS problem

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

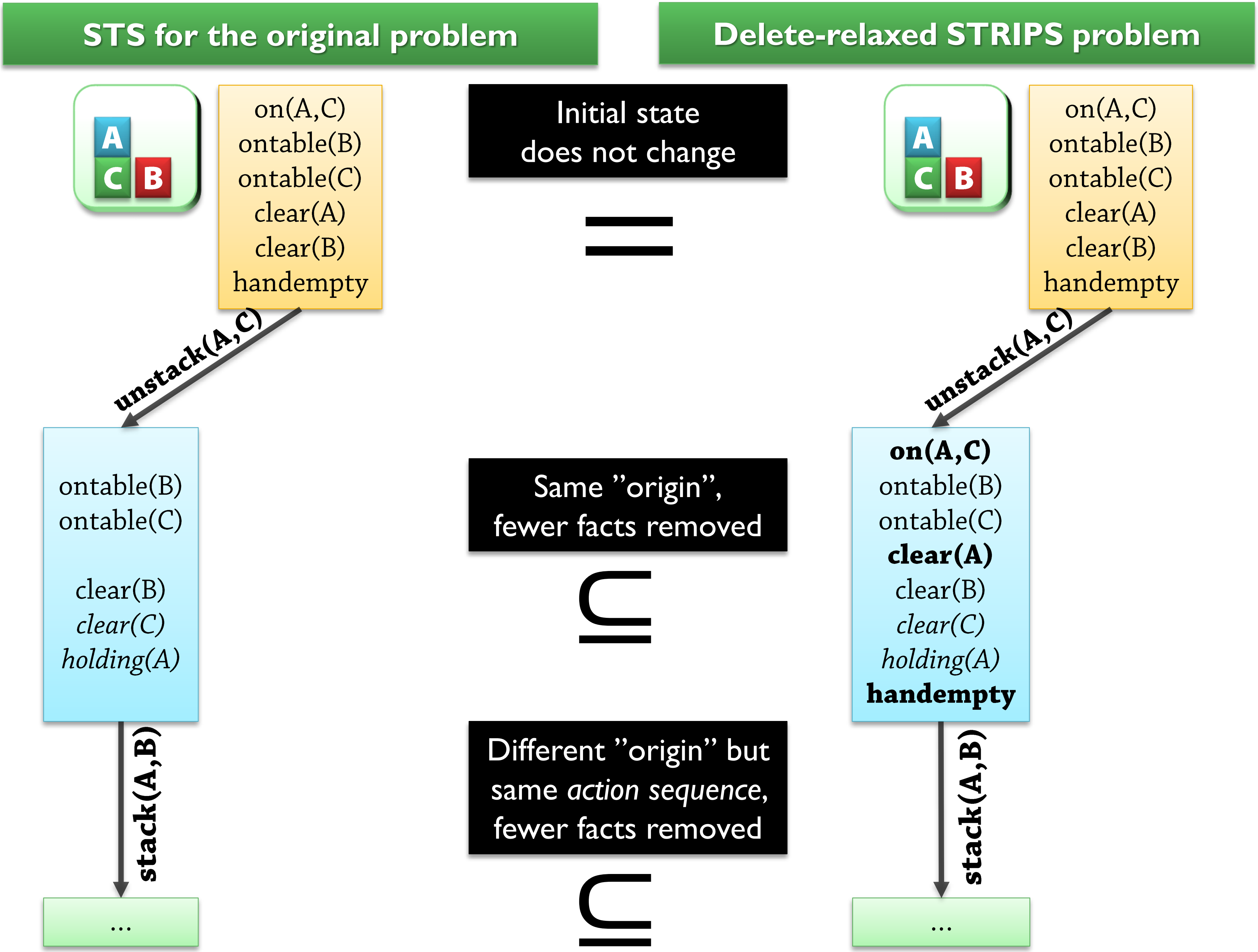
unstack(A,C)

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

No physical correspondence!

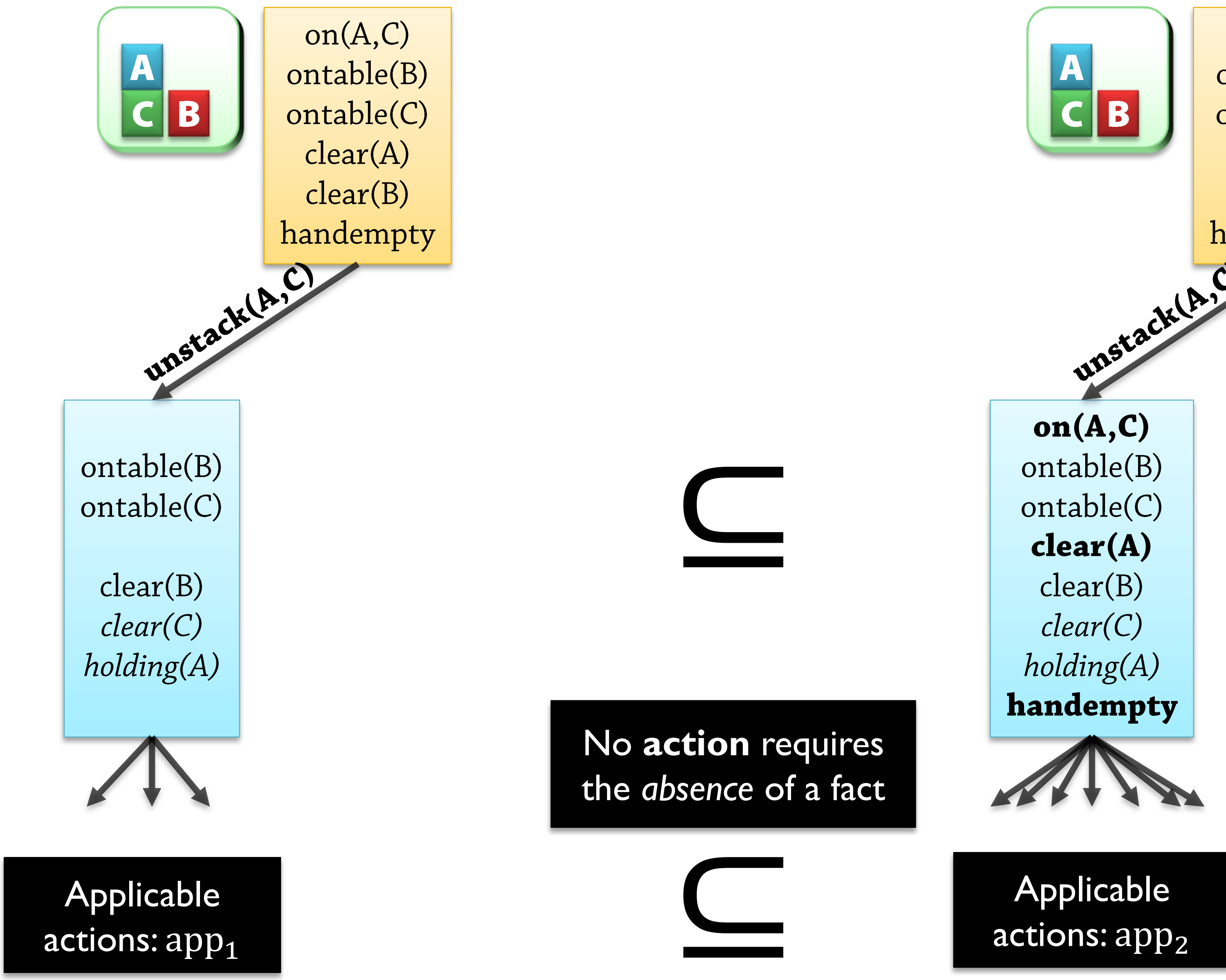
pickup(B)

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


STS for the original problem
Delete-relaxed STRIPS problem
on(A,C)
ontable(B)
ontable(C)
clear(A)
clear(B)
handempty
Initial state
does not change
=
on(A,C)
ontable(B)
ontable(C)
clear(A)
clear(B)
handempty
unstack(A,C)
unstack(A,C)
ontable(B)
ontable(C)
clear(B)
clear(C)
holding(A)
Same "origin",
fewer facts removed
⊆
on(A,C)
ontable(B)
ontable(C)
clear(A)
clear(B)
clear(C)
holding(A)
handempty
stack(A,B)
Different "origin" but
same *action sequence*,
fewer facts removed
⊆
stack(A,B)
...
...

STS for the original problem

Delete-relaxed STRIPS problem



Delete Relaxation (8): Example

STS for the original problem

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

unstack(A,C)

ontable(B)
ontable(C)
clear(B)
clear(C)
holding(A)

Satisfies the goal?

Delete-relaxed STRIPS problem

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

unstack(A,C)

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

No **goal** requires the *absence* of a fact

$\Rightarrow$

Also satisfies the goal

Delete Relaxation (9)

- **Negative effects** are also called "**delete effects**"
  - They delete facts from the state

- So this is called **delete relaxation**
  - "*Relaxing* the problem by getting rid of the *delete effects*"

Delete relaxation does not mean that we "delete the relaxation" (anti-relax)!

Delete relaxation is only a relaxation if preconditions and goals are positive!

Delete Relaxation (10)

- Since **solutions are preserved** when **facts are added**:

A state where additional facts are true can never be "worse"!
(Given positive preconds/goals)

$$h^*\left(\begin{array}{l}\text{ontable(B)}\\\text{ontable(C)}\\\text{clear(B)}\\\text{clear(C)}\\\text{holding(A)}\\\textbf{handempty}\end{array}\right) \leq h^*\left(\begin{array}{l}\text{ontable(B)}\\\text{ontable(C)}\\\text{clear(B)}\\\text{clear(C)}\\\text{holding(A)}\end{array}\right)$$

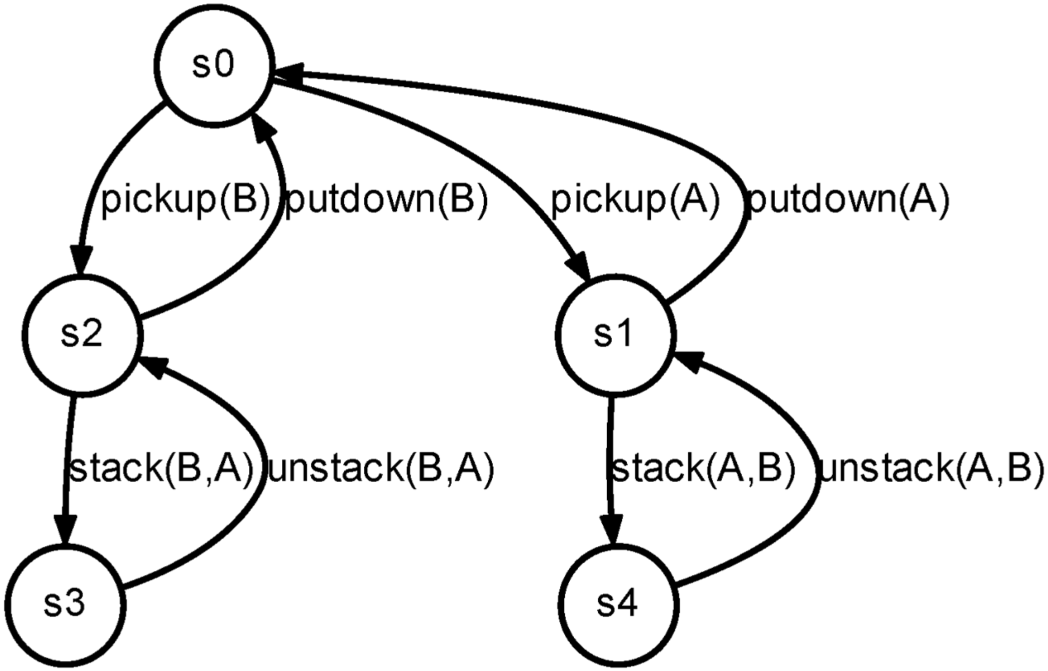
Given two states (sets of true atoms) s,s':
s ⊃ s' → h*(s) <= h*(s')

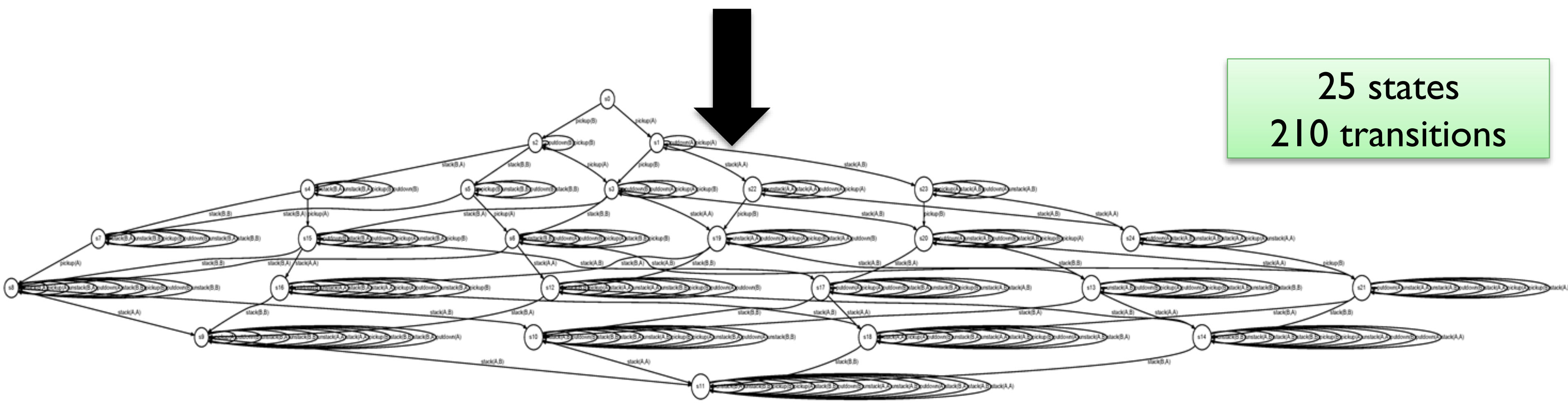
Delete Relaxation:

State Space Examples

Reachable State Space: BW size 2

5 states
8 transitions

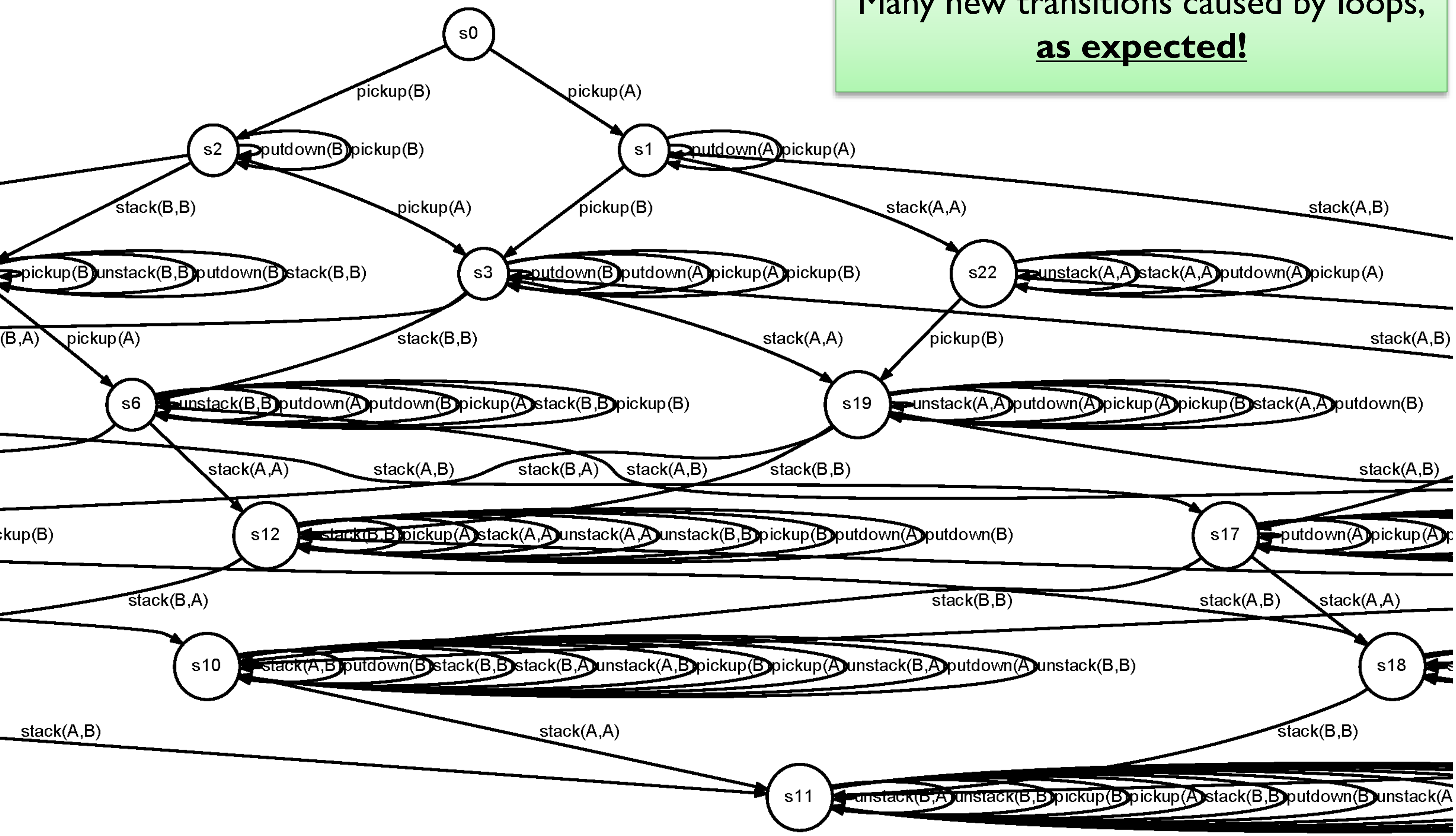




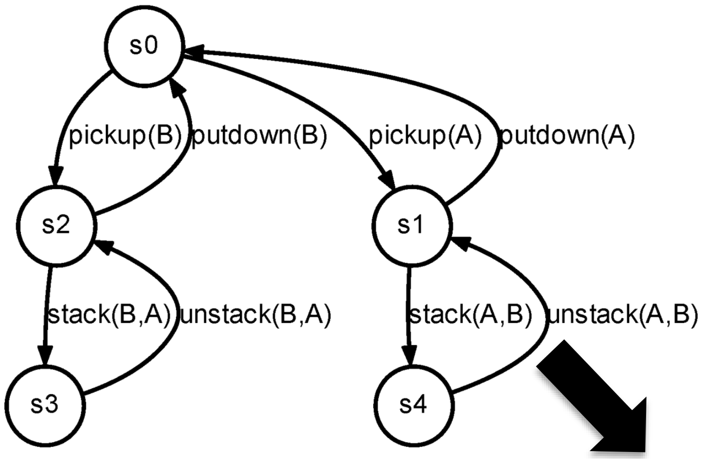
25 states
210 transitions

Delete-Relaxed BW size 2: Detail View

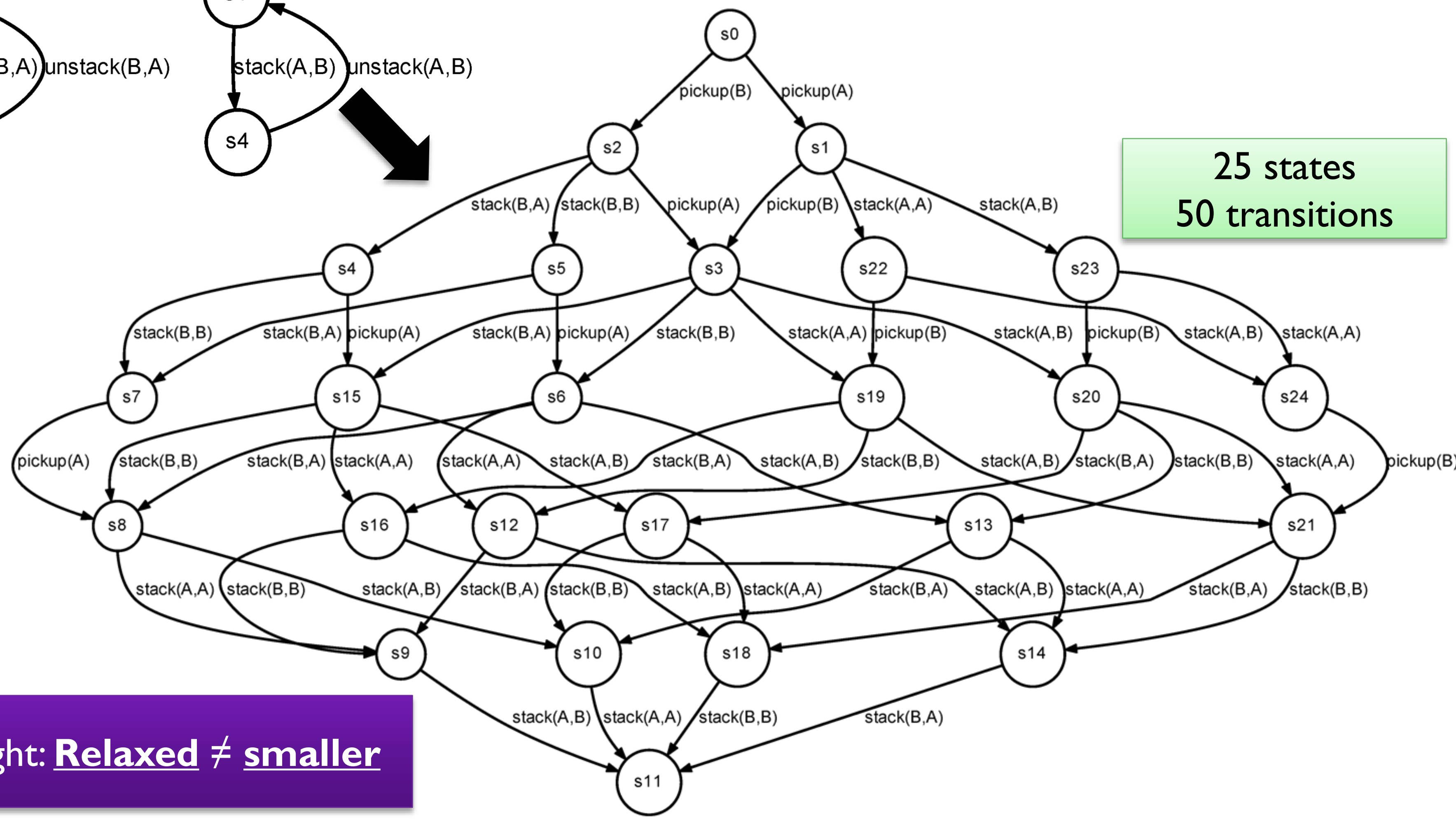
Many new transitions caused by loops, **as expected!**



Delete-Relaxed: "Loops" Removed



5 states
8 transitions



25 states
50 transitions

Insight: **Relaxed** ≠ **smaller**

The Optimal Delete Relaxation Heuristic

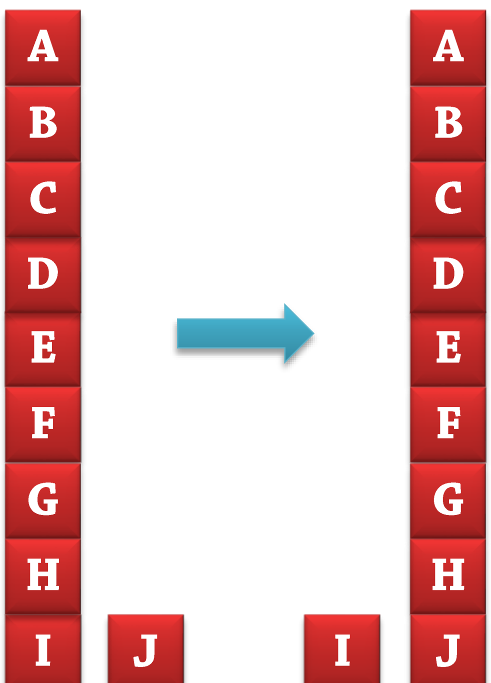
Optimal Delete Relaxation Heuristic

- If **only** delete relaxation is applied:
  - We can calculate the **optimal delete relaxation heuristic**, h+(*n*)
  - h+(*n*) = the cost of an **optimal solution**
    to a **delete-relaxed** problem
    starting in node *n*

Accuracy of h+ in Selected Domains

- **How close** is $h^+(n)$ to the true goal distance $h^*(n)$?
  - **Worst case asymptotic accuracy** as problem size approaches infinity:
    - Blocks world: 1/4 ➔ h+(n) ≥ 1/4 h*(n)

Optimal plans in delete-relaxed Blocks World
can be down to 25% of the length of optimal plans in "real" Blocks World



Standard:		**Relaxed**:
unstack(A,B)	pickup(G)	unstack(A,B)
putdown(A)	stack(G,H)	unstack(B,C)
unstack(B,C)	pickup(F)	unstack(C,D)
putdown(B)	stack(F,G)	unstack(D,E)
unstack(C,D)	pickup(E)	unstack(E,F)
putdown(C)	stack(E,F)	unstack(F,G)
…	pickup(D)	unstack(G,H)
unstack(H,I)	stack(D,E)	unstack(H,I)
stack(H,J)	…	stack(H,J)
		DONE!

Accuracy of h+ in Selected Domains (2)

- **<u>How close</u>** is $h^{+}(n)$ to the true goal distance $h^{*}(n)$?
  - **<u>Worst case asymptotic accuracy</u>** as problem size approaches infinity:
    - Blocks world: 1/4 ➔ h+(n) ≥ 1/4 h*(n)
    - Gripper domain: 2/3 (single robot moving balls)
    - Logistics domain: 3/4 (move packages using trucks, airplanes)
    - Miconic-STRIPS: 6/7 (elevators)
    - Miconic-Simple-ADL: 3/4 (elevators)
    - Schedule: 1/4 (job shop scheduling)
    - Satellite: 1/2 (satellite observations)
  - Details:
    - Malte Helmert and Robert Mattmüller
      Accuracy of Admissible Heuristic Functions in Selected Planning Domains

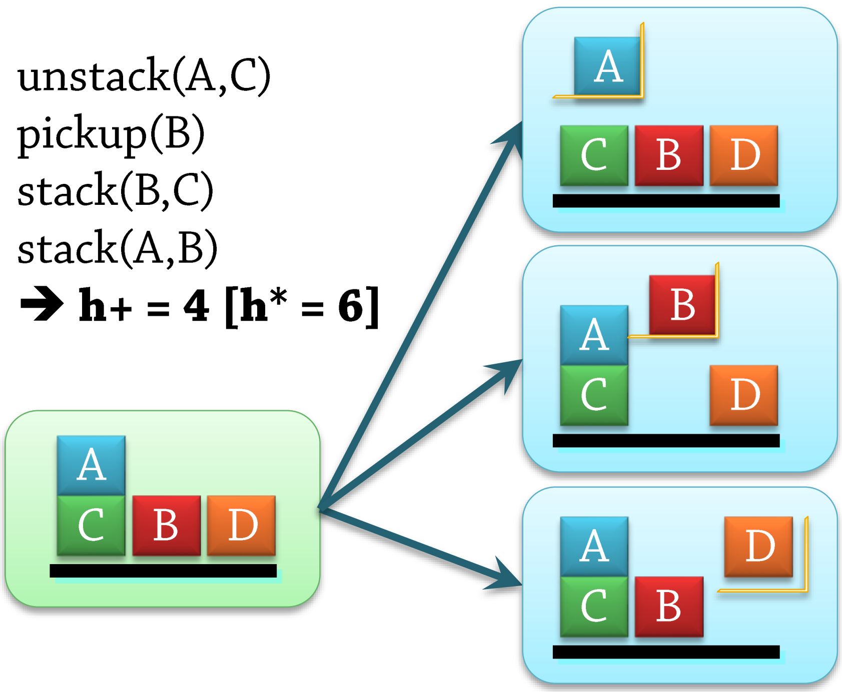
Accuracy of Admissible Heuristic Functions in Selected Planning Domains

Malte Helmert and Robert Mattmüller
Albert-Ludwigs-Universität Freiburg, Germany
{helmert,mattmuel}@informatik.uni-freiburg.de

Example of Accuracy

- **<u>How close</u>** is $h^+(n)$ to the true goal distance $h^*(n)$?
  - In practice: Also depends on the **<u>problem instance</u>**!

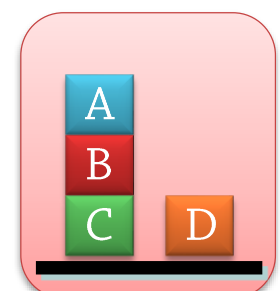
unstack(A,C)
pickup(B)
stack(B,C)
stack(A,B)
➔ **h+ = 4 [h* = 6]**



pickup(B); stack(B,C); stack(A,B)
➔ **h+ = 3 [h* = 5]**
Good action!

unstack(A,C); stack(B,C); stack(A,B)
➔ **h+ = 3 [h* = 7]**
Seems equally good as unstack, but *is* worse

unstack(A,C); pickup(B);
stack(B,C); stack(A,B)
➔ **h+ = 4 [h* = 7]**



- **<u>Performance</u>** also depends on the search strategy
  - How sensitive it is to specific types of inaccuracy

Computing the
Optimal Delete Relaxation Heuristic

Computing h+

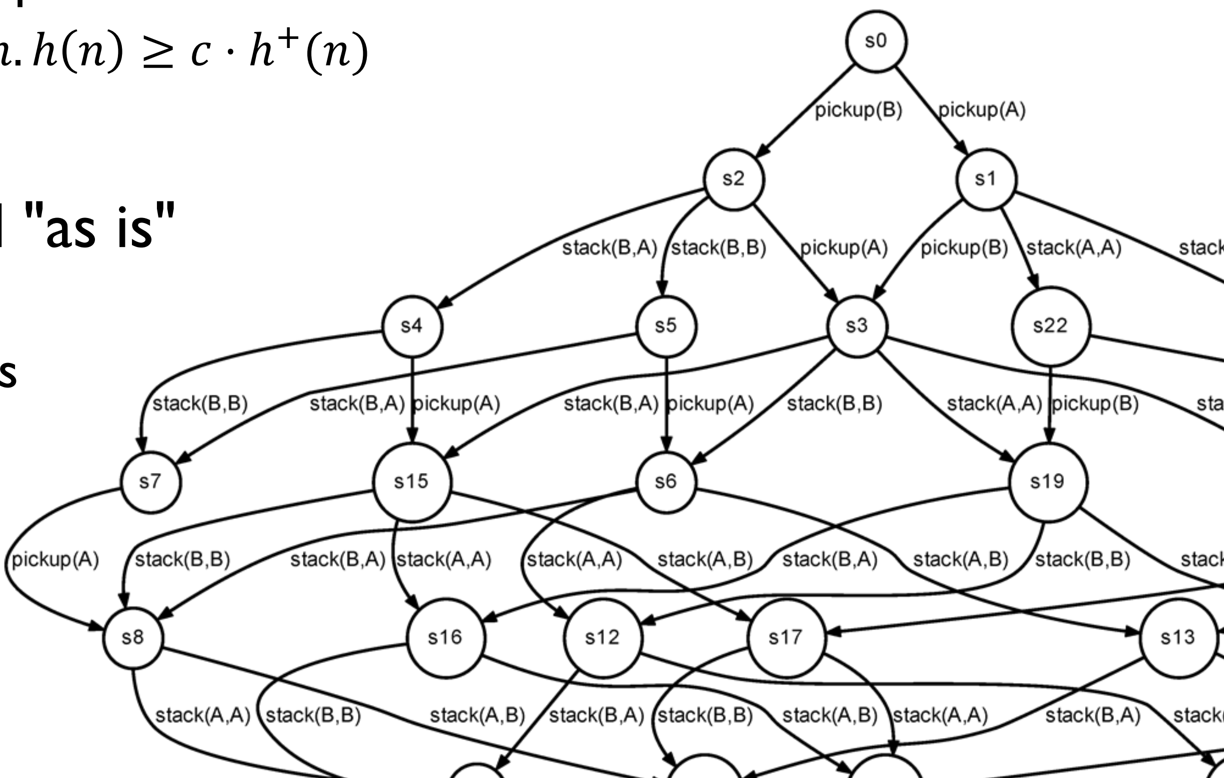
- Is $h^+(n)$ **easier to compute** than $h^*(n)$?
  - $h^*(n)$ = length of optimal plan for arbitrary planning problem
    - Supports negative effects
    - If we can execute either a1;a2 or a2;a1:
      - a1 removes p, a2 adds p ➔ net result: add p
      - a2 adds p, a1 removes p ➔ net result: remove p
      - *Both* orders must be considered
  - $h^+(n)$ = length of optimal plan after removing negative effects
    - If we can execute either a1;a2 or a2;a1:
      - Must lead to the same state (add p1 before p2, or p2 before p1)
      - Sufficient to consider *one* order
  - Incomplete analysis – but $h^+(n)$ **is** easier to compute, **in the worst case**

Calculating h+

- Still **difficult** to calculate in general!
  - NP-equivalent (reduced from PSPACE-equivalent)
    - Since you must find **optimal** solutions to the relaxed problem

  - Even a constant-factor approximation is NP-equivalent to compute!
    - Finding $h(n)$ so that $\forall n.\, h(n) \geq c \cdot h^{+}(n)$

- Therefore, rarely used "as is"
  - But forms the **basis** of many other heuristics

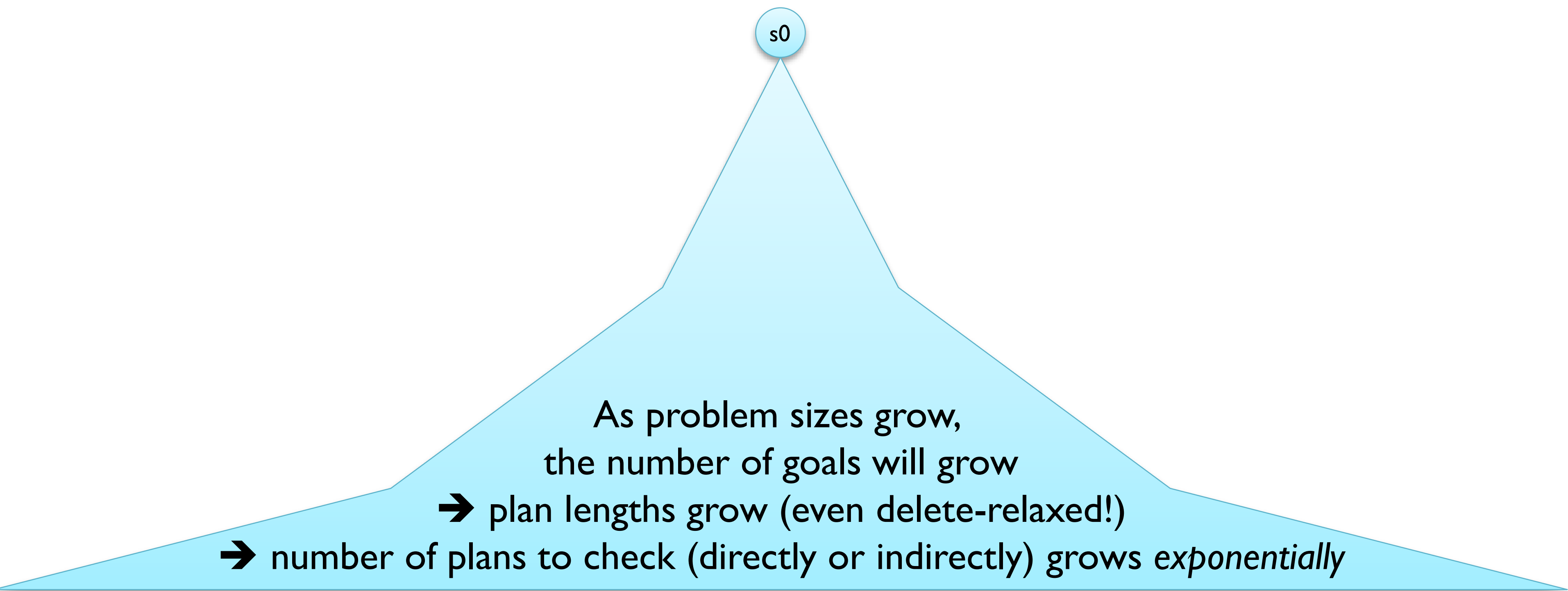


Optimal Classical Planning: The Admissible h_1 Heuristic

Intuitions (1)

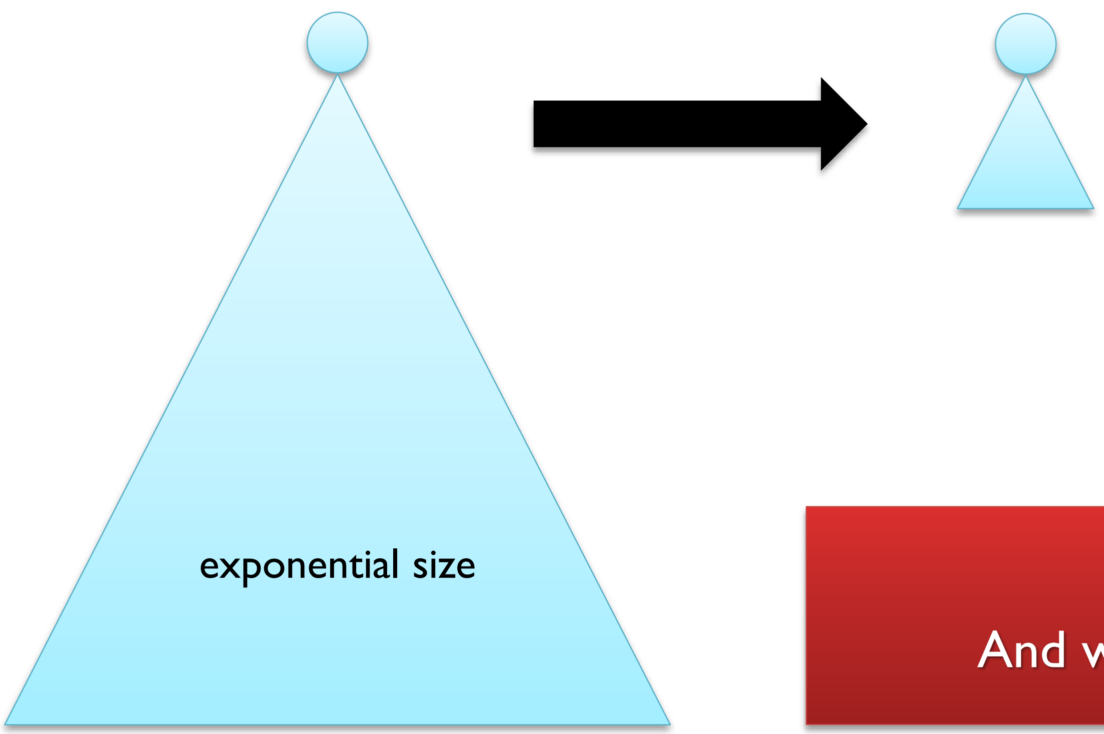
- Why is $h^+(n)$ so "slow"?

Must compute the **exact cost**
of an **optimal plan**
achieving **all goals**



Intuitions (2)

- Suppose we delete-relax, then only consider **<u>one goal fact</u>**
  - Remove **<u>goal requirements</u>** ➔ add new **<u>goal states</u>** in S_g
- Relaxation!
  - ”Old” plans achieving *all* goals are still valid solutions
  - **<u>Also has much shorter solutions</u>,** much faster to compute



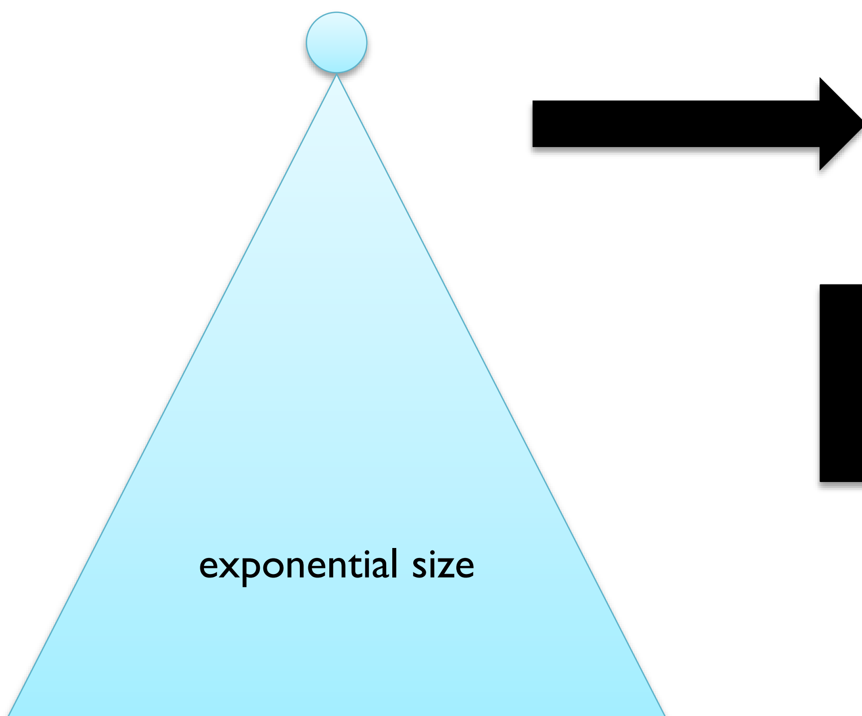
<u>Too</u> relaxed!
And which goal to choose?

Intuitions (3)

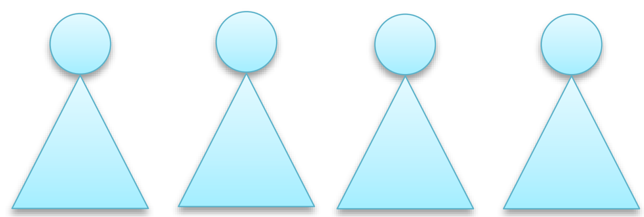
- Given **two admissible heuristics** $h_A(n)$ and $h_B(n)$:
  - $h_{AB}(n) = \max\big(h_A(n), h_B(n)\big)$ is admissible
  - If neither heuristic overestimates, their maximum cannot overestimate

The h_1 Heuristic

- Idea (from HSPr*): Consider **one** goal atom **at a time**



Treat **each** goal atom **separately**
Take the **maximum** of the costs

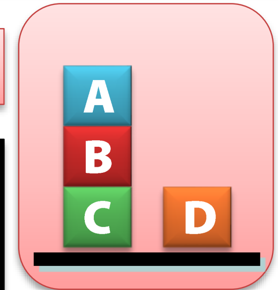


Uses a set of relaxations!

Computing $h_1(n)$

The h_1 Heuristic: Example (action cost = 1)

Goal:	clear(A)	on(A,B)	on(B,C)	ontable(C)	clear(D)	ontable(D)



Don't find the best way to achieve *all goal atoms*:
{ clear(A), on(A,B), on(B,C), on(B,C), ontable(C), clear(D), ontable(D) }

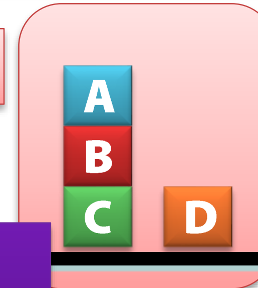
Avoid interactions:
Find the best way to achieve **clear(A)**
Then find the best way to achieve **on(A,B)**
…

Use **<u>backward search</u>**, starting with the goals

s_0: clear(A), on(A,C), ontable(C), clear(B), ontable(B), clear(D), ontable(D), handempty

The h_1 Heuristic: Example (action cost = 1)

Goal:	*clear(A)*	on(A,B)	on(B,C)	*ontable(C)*	*clear(D)*	*ontable(D)*
	cost 0					



First goal atom:
clear(A)

Already achieved,
cost 0

stack(A,B)

holding(A)	clear(B)

How to achieve on(A,B)?
Not true in the initial state.
Check *all actions* having on(A,B) as an effect…
Here: Only stack(A,B)!

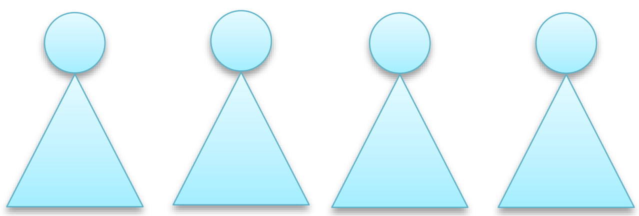
We have **two preconditions** to achieve.

Reduce interactions even more:
Consider each of *these* as a separate "subgoal"!
First holding(A), then clear(B).

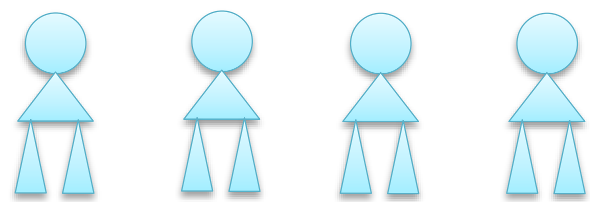
s_0: clear(A), on(A,C), ontable(C), clear(B), ontable(B), clear(D), ontable(D), handempty

The h_1 Heuristic: Intuitions (2)

Idea: Treat each **goal atom** separately
Take the **maximum** of the costs

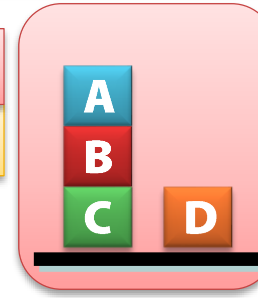


$h_1(n)$: Split the problem even further;
consider *individual subgoals* at every "level"



The h_1 Heuristic: Example (continued)

Goal:	*clear(A)*	on(A,B)	on(B,C)	*ontable(C)*	*clear(D)*	*ontable(D)*
	cost 0	cost 2	cost 2	cost 0	cost 0	cost 0



$h_1(s_0)$ = max(2,2) = 2

stack(A,B)

holding(A)	*clear(B)*
cost 1	cost 0

unstack(A,C)

handempty	*clear(A)*	*on(A,C)*
cost 0	cost 0	cost 0

Search continues: This is cheaper!

unstack(A,D)

handempty	*clear(A)*	on(A,D)

More calculations show:
This is expensive…

stack(B,C)

holding(B)	clear(C)
cost 1	cost 1

pickup(B)

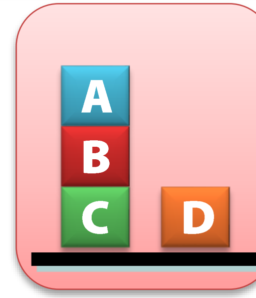
handempty	*clear(B)*
cost 0	cost 0

unstack(A,C)

handempty	*clear(A)*	*on(A,C)*

s_0: clear(A), on(A,C), ontable(C), clear(B), ontable(B), clear(D), ontable(D), handempty

The h_1 Heuristic: Important Property 1



on(B,C)
cost 2

Each goal considered separately!

We don't search for a **valid plan** achieving on(B,C)!

Then we would need putdown(A)…

The heuristic considers individual subgoals *at all levels*, misses interactions *at all levels*

stack(B,C)

holding(B)	clear(C)
cost 1	cost 1

Each precondition considered separately!

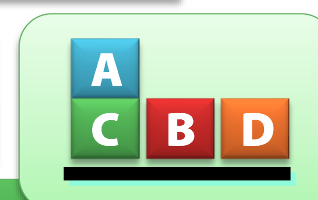
pickup(B)

handempty	*clear(B)*
cost 0	cost 0

Each precondition considered separately!

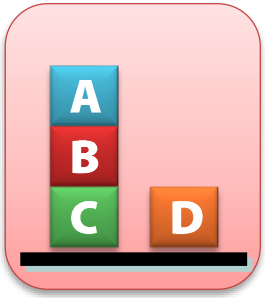
unstack(A,C)

handempty	*clear(A)*	*on(A,C)*



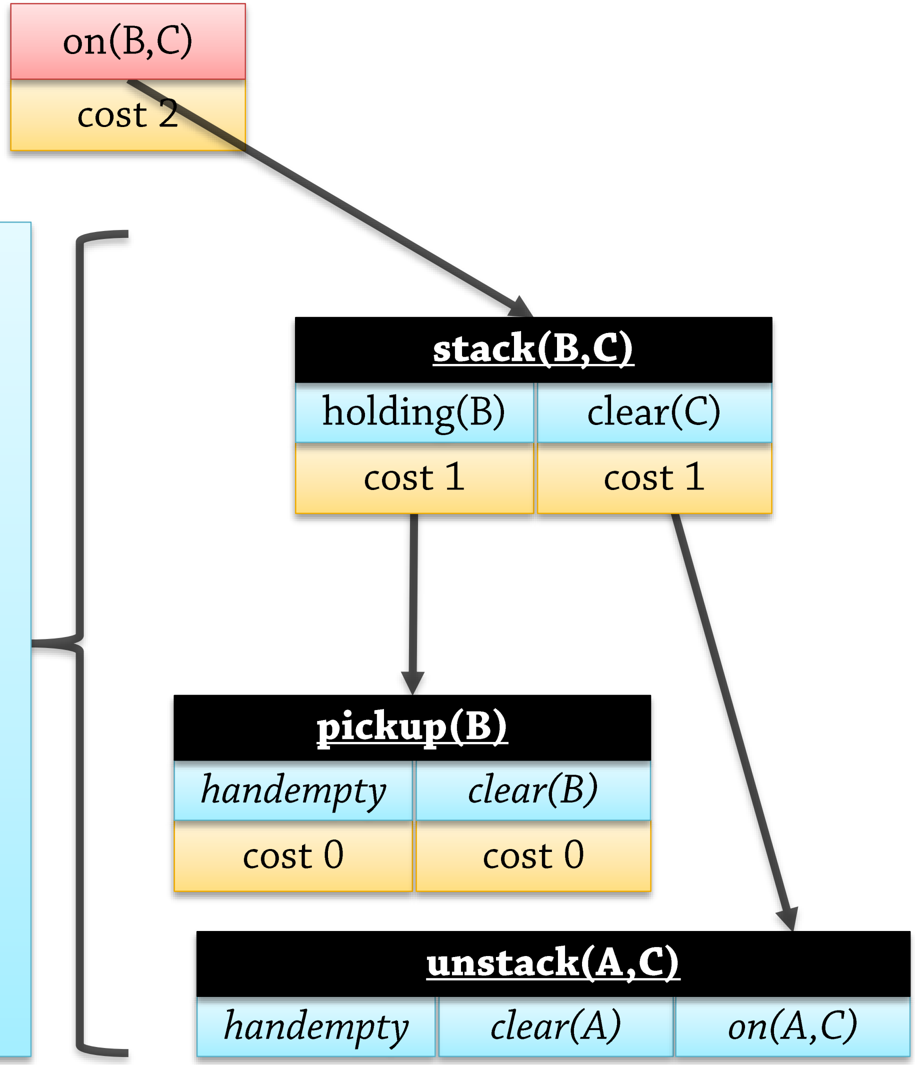
This is why it is fast! No need to consider interactions → **no combinatorial explosion**

The h_1 Heuristic: Important Property 2



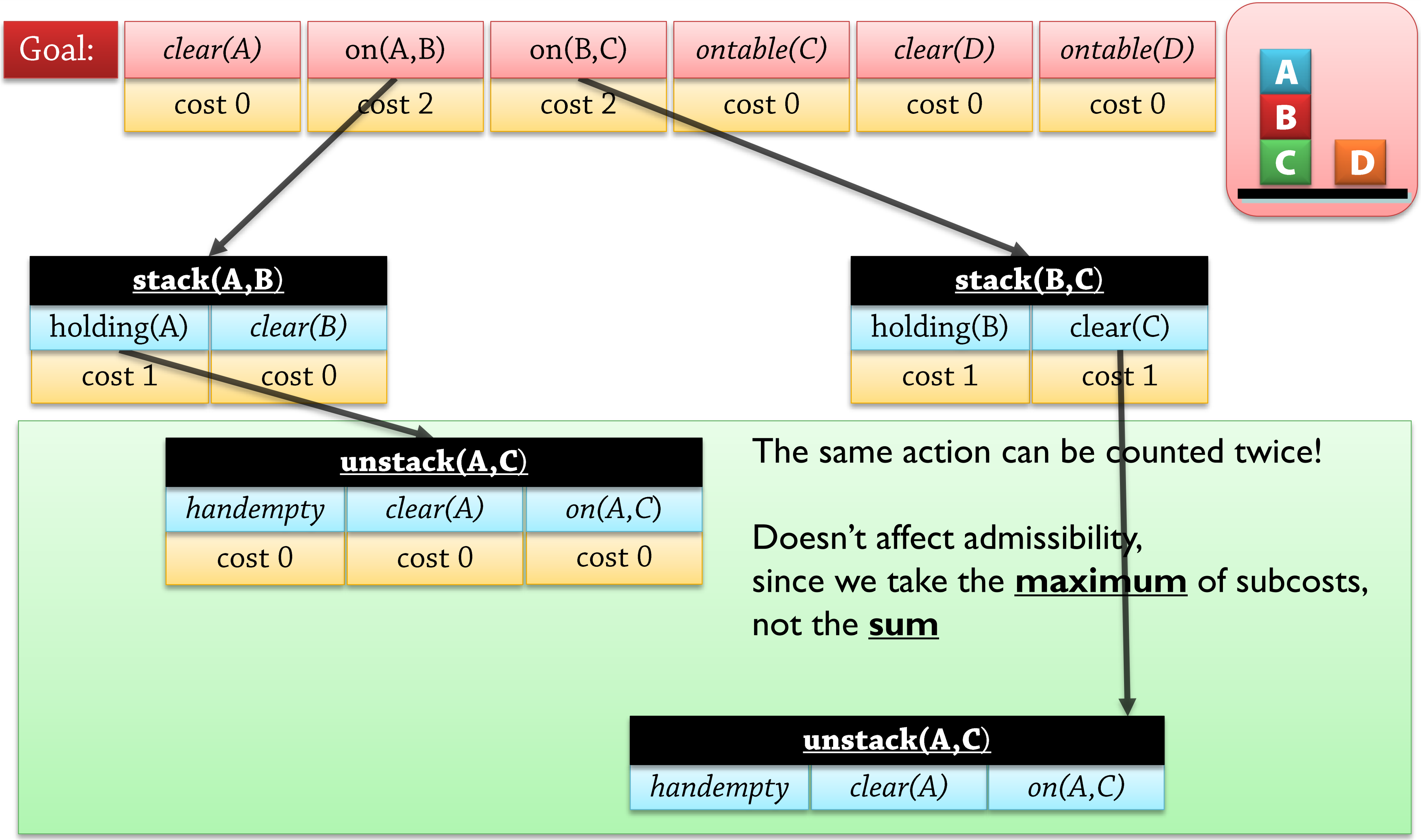
Given a problem
using :**strips** expressivity,
we ignore negative effects!

(Given a goal atom,
find an action achieving it,
without considering
any other effects)



h_1 takes the delete relaxation heuristic, relaxes it further

The h_1 Heuristic: Important Property 3



The same action can be counted twice!

Doesn't affect admissibility,
since we take the **maximum** of subcosts,
not the **sum**

The h_1 Heuristic: Formal Definition

$h_1(s) = \Delta_1(s, g)$ – the heuristic depends on the goal g

- For a **goal**, a **set** g of facts to achieve:
  - $\Delta_1(s, g)$ = the cost of achieving the **most expensive** proposition in g
    - $\Delta_1(s, g) = 0$ (zero) if $g \subseteq s$ *// Already achieved entire goal*
    - $\Delta_1(s, g) = \max \{ \Delta_1(s, p) \mid p \in g \}$ otherwise *// Part of the goal not achieved*

The cost of each atom in goal g

Max: The **entire** goal must be at least as expensive as the most expensive **subgoal**

Implicit delete relaxation: Cheapest way of achieving $p1 \in g$ may actually delete $p2 \in g$

So how expensive is it to achieve a single proposition?

The h_1 Heuristic: Formal Definition

$h_1(s) = \Delta_1(s, g)$ – the heuristic depends on the goal g

- For a **single proposition** p to be achieved:
  - $\Delta_1(s, p)$ = the cost of **achieving p from s**
    - $\Delta_1(s, p) = 0$ if $p \in s$ *// Already achieved p*
    - $\Delta_1(s, p) = \infty$ if $\forall a \in A.\ p \notin effects^+(a)$ *// Unachievable*
    - Otherwise:

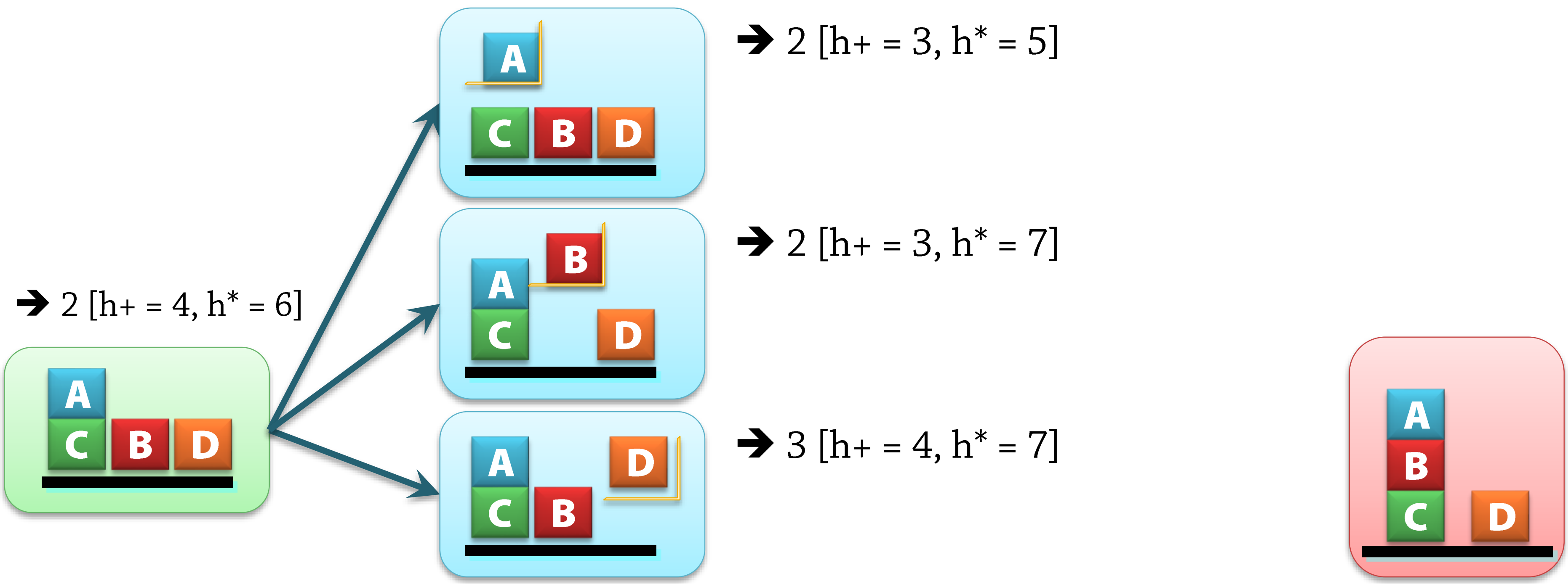
      $\Delta_1(s, p) = \min \{ cost(a) + \Delta_1(s, precond(a)) \mid a \in A \text{ and } p \in effects^+(a) \}$

Must **execute** an action a∈A that achieves p, and before that, *acheive its preconditions*

Min: Choose the action that lets you achieve the proposition *p* as cheaply as possible

The h_1 Heuristic: Examples

- In the problem below:
  - g = { ontable(C), ontable(D), clear(A), clear(D), on(A,B), on(B,C) }
- So for any state s:
  - $\Delta_1(s, g) = \max\{$ Δ_1(s, ontable(C)), Δ_1(s, ontable(D)), Δ_1(s, clear(A)), Δ_1(s, clear(D)), Δ_1(s, on(A,B)), Δ_1(s, on(B,C)) $\}$
- With unit action costs:



The h_1 Heuristic: Properties

- $h_1(s)$ is:
  - **<u>Easier</u>** to calculate than the optimal delete relaxation heuristic h+
  - Somewhat **<u>useful</u>** for this simple BW problem instance
  - **<u>Not sufficiently informative</u>** in general
- Example:
  - Forward search in Blocks World using Fast Downward planner, A*

Blocks	nodes blind	nodes h1
5	1438	476
6	6140	963
7	120375	24038
8	1624405	392065
9	25565656	14863802
10	>84 million (out of mem)	208691676

Optimal Classical Planning: The Admissible h_m Heuristics

The h_m Heuristics

- Next idea: Why only consider single atoms?
  - $h_1(s)=\Delta_1(s,g)$: The most expensive atom
  - $h_2(s)=\Delta_2(s,g)$: The most expensive pair of atoms
  - $h_3(s)=\Delta_3(s,g)$: The most expensive triple of atoms
  - …
  - ➔ A **<u>family</u>** of **<u>admissible</u>** heuristics $h_m = h_1, h_2, \ldots$ for **<u>optimal</u>** classical planning

The h_2 Heuristic

- $h_2(s) = \Delta_2(s, g)$: The most expensive **pair** of goal propositions

Goal (set)

- $\Delta_2(s, g) = 0$ if $g \subseteq s$ // Already achieved
- $\Delta_2(s, g) = \mathbf{max} \{ \Delta_2(s,p,q) \mid p,q \in g \}$ otherwise // Can have p=q!

Pair of propositions (maybe p=q)

- $\Delta_2(s, p, q) = 0$ if $p,q \in s$ // Already achieved
- $\Delta_2(s, p, q) = \infty$ if $\forall a \in A.\ p \notin effects^+(a)$ or $\forall a \in A.\ q \notin effects^+(a)$
- $\Delta_2(s, p, q) = \mathbf{min} \{$
  - $\min \{ cost(a) + \Delta_2(s, precond(a)) \mid a \in A \text{ and } p,q \in effects^+(a) \}$,
  - $\min \{ cost(a) + \Delta_2(s, precond(a) \cup \{q\}) \mid a \in A,\ p \in effects^+(a),\ q \notin effects^-(a) \}$,
  - $\min \{ cost(a) + \Delta_2(s, precond(a) \cup \{p\}) \mid a \in A,\ q \in effects^+(a),\ p \notin effects^-(a) \}$

  $\}$

- $h_2(s)$ is more informative than $h_1(s)$, requires non-trivial time
- $m > 2$ rarely useful

The h_2 Heuristic and Delete Effects

- In this definition of h_2:
  - $\Delta_2(s, p, q) = \underline{\textbf{min}}\ \{$
    $\text{cost}(a) + \min \{ \Delta_2(s, \text{precond}(a)) \mid a \in A \text{ and } p,q \in \text{effects}^+(a) \},$
    $\text{cost}(a) + \min \{ \Delta_2(s, \text{precond}(a) \cup \{q\}) \mid a \in A,\ p \in \text{effects}^+(a),\ q \notin \text{effects}^-(a) \},$
    $\text{cost}(a) + \min \{ \Delta_2(s, \text{precond}(a) \cup \{p\}) \mid a \in A,\ q \in \text{effects}^+(a),\ p \notin \text{effects}^-(a) \}$
    $\}$

Takes into account **<u>some</u>** delete effects

So h_2 is **<u>not</u>** a *delete* relaxation heuristic (but it **<u>is</u>** admissible)!

- Misses other delete effects
  - Goal: $\{p, q, r\}$
  - A1: Adds $\{p,q\}$ Deletes $\{r\}$
  - A2: Adds $\{p,r\}$ Deletes $\{q\}$
  - A3: Adds $\{q,r\}$ Deletes $\{p\}$
  - $\Delta_2(s, p,q), \Delta_2(s, q,r), \Delta_2(s, p,r) = 1$: Any pair can be achieved with a single action
  - $\Delta_2(s, g) = \max(\Delta_2(s, p,q), \Delta_2(s, q,r), \Delta_2(s, p,r)) = \max(1, 1, 1) = 1$,
    but the problem is unsolvable!

The h_2 Heuristic and Pairwise Mutexes

- If $\Delta_2(s_0, p, q) = \infty$:
  - Starting in s_0, can't reach a state where p and q are true
  - Starting in s_0, p and q are *mutually exclusive (mutex)*
- One-way implication!
  - Can be used to find *some* mutex relations, not necessarily *all*

The h_2 Heuristic and Delete Relaxation

- In the book:
  - $\Delta_2(s, p, q) = \underline{\mathbf{min}}$ {
    1 + min { Δ_2(s, precond(a)) | a∈A and p,q ∈ $effects^+$(a) },
    1 + min { Δ_2(s, precond(a) ∪ {q}) | a∈A, p ∈ $effects^+$(a) },
    1 + min { Δ_2(s, precond(a) ∪ {p}) | a∈A, q ∈ $effects^+$(a) }
    }
- This is **not** how the heuristic is normally presented!
  - Corresponds to applying (full) delete relaxation
  - Uses constant action costs (1)

The h_m Heuristics: Calculating

- Calculating $h_m(s)$ **in practice**:
  - Characterized by Bellman equation over a specific search space
  - Solvable using variation of Generalized Bellman-Ford (GBF)
  - (Not part of the course)

$$h^m(s) = \begin{cases} 0 & \text{if } s \subseteq I \\ \min_{s' \in succ(s)} h^m(s') + \delta(s, s') & \text{if } |s| \leqslant m \\ \max_{s' \subseteq s, |s'| \leqslant m} h^m(s') & \end{cases}$$

Cost of cheapest action taking you from s to s'

Accuracy of h_m in Selected Domains

- **How close** is $h_m(n)$ to the true goal distance $h^*(n)$?
  - **Asymptotic** accuracy as problem size approaches infinity:
    - Blocks world: 0 ➔ $h_m(n) \geq 0\, h^*(n)$
    - For any constant m!

Accuracy of h_m in Selected Domains (2)

- Consider a constructed **family of problem instances**:
  - $10n$ blocks, all on the table
  - Goal: n specific towers of 10 blocks each
- What is the **true cost** of a solution from the initial state?
  - For each tower, 1 block in place + 9 blocks to move
  - 2 actions per move
  - $9 * 2 * n = 18n$ actions
- h_1(initial-state) = 2 – regardless of n!
  - All instances of clear, ontable, handempty already achieved
  - Achieving a single on(…) proposition requires two actions
- h_2(initial-state) = 4
  - Achieving two on(…) propositions
- h_3(initial-state) = 6
- …

As problem sizes grow,
the number of goals can grow
and plan lengths can grow indefinitely

But $h_m(n)$ only considers a constant number of goal facts!
Each individual *set* of size m does not necessarily become harder to achieve,
and we only calculate *max*, not *sum*…

Accuracy of h_m in Selected Domains (3)

- **<u>How close</u>** is $h_m(n)$ to the true goal distance $h^*(n)$?
  - **<u>Asymptotic</u>** accuracy as problem size approaches infinity:
    - Blocks world: 0 ➔ $h_m(n) \geq 0\, h^*(n)$
    - Gripper domain: 0
    - Logistics domain: 0
    - Miconic-STRIPS: 0
    - Miconic-Simple-ADL: 0
    - Schedule: 0
    - Satellite: 0
  - For any constant m!

Still **<u>useful</u>** – this is a **<u>worst-case</u>** analysis as **<u>sizes approach infinity</u>**!
+ Variations such as additive h_m exist

  - Details:
    - Malte Helmert, Robert Mattmüller
      Accuracy of Admissible Heuristic Functions in Selected Planning Domains

The h_2 Heuristic: Accuracy

- **Experimental** accuracy of h2 in a few classical problems:

Instance	Opt.	$h(root)$
blocks-9	6	5
blocks-11	9	7
blocks-15	14	11
eight-1	31	15
eight-2	31	15
eight-3	20	12
grid-1	14	14
gripper-1	3	3
gripper-2	9	4
gripper-3	15	4

Seems to work well for the blocks world…

Less informative for the gripper domain!

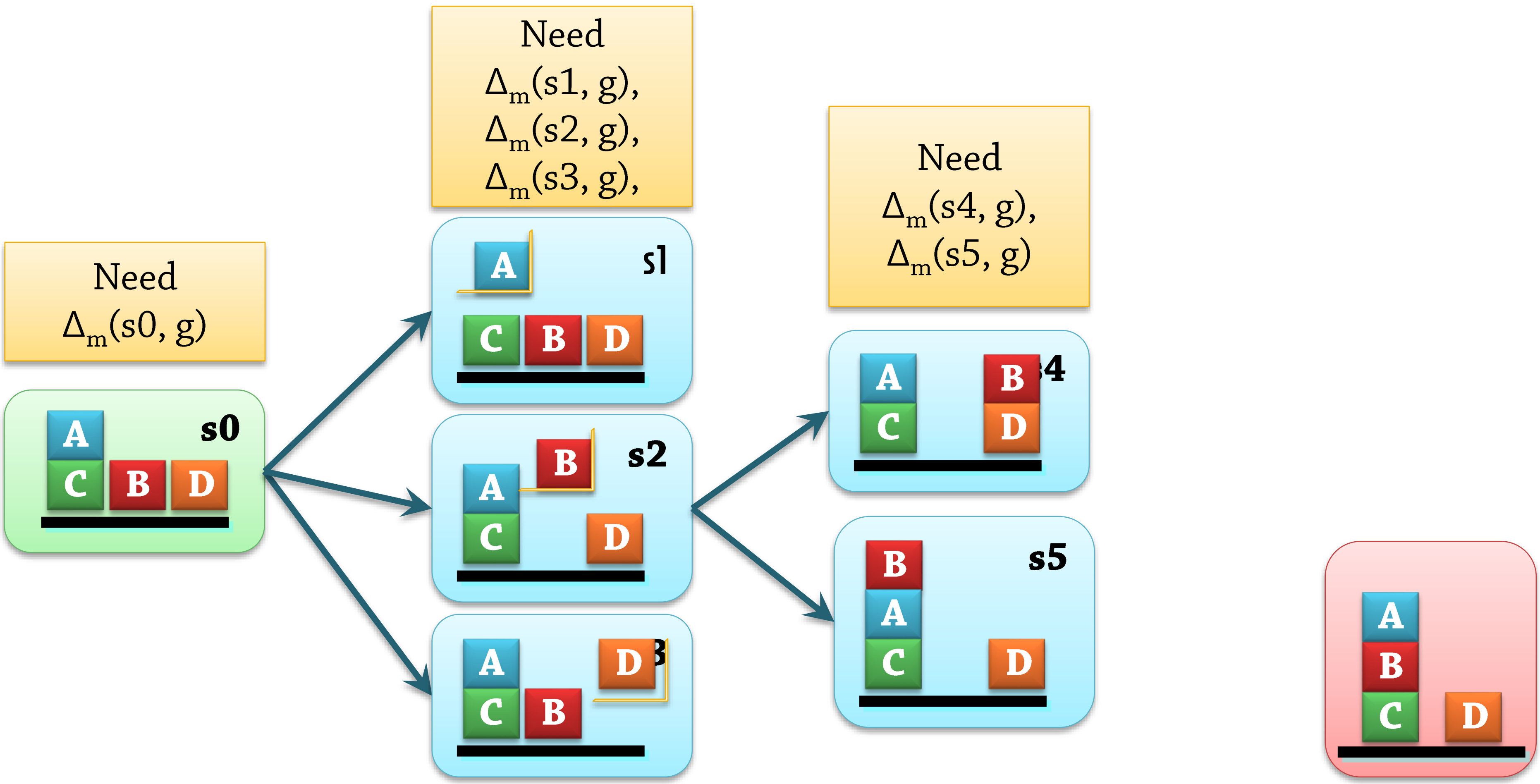
The h_m Heuristic: Nodes Expanded

Blocks/length	nodes blind	nodes h1	nodes h2	nodes h3	nodes h4
5	1438	476	112	18	13
6	6140	963	78	23	
7	120375	24038	1662	36	
8	1624405	392065	35971		
9	25565656 (25.2s)	14863802			
10	>84 million (out of mem)	208691676			

Backward Search and h_m Heuristics

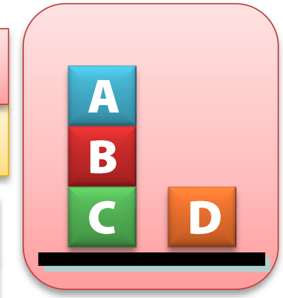
Forward Search with h_m

- Consider $\mathbf{h_m}$ heuristics using forward search:



Forward Search with h_m: Illustration

Goal: *clear(A)* cost 0 | on(A,B) cost 2 | on(B,C) cost 2 | *ontable(C)* cost 0 | *clear(D)* cost 0 | *ontable(D)* cost 0



stack(A,B): holding(A) cost 1 | *clear(B)* cost 0

unstack(A,C): *handempty* cost 0 | *clear(A)* cost 0 | *on(A,C)* cost 0

Search continues: This is cheaper!

unstack(A,D): *handempty* | *clear(A)* | on(A,D)

More calculations show:
This is expensive…

stack(B,C): holding(B) cost 1 | clear(C) cost 1

pickup(B): *handempty* cost 0 | *clear(B)* cost 0

unstack(A,C): *handempty* | *clear(A)* | *on(A,C)*

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

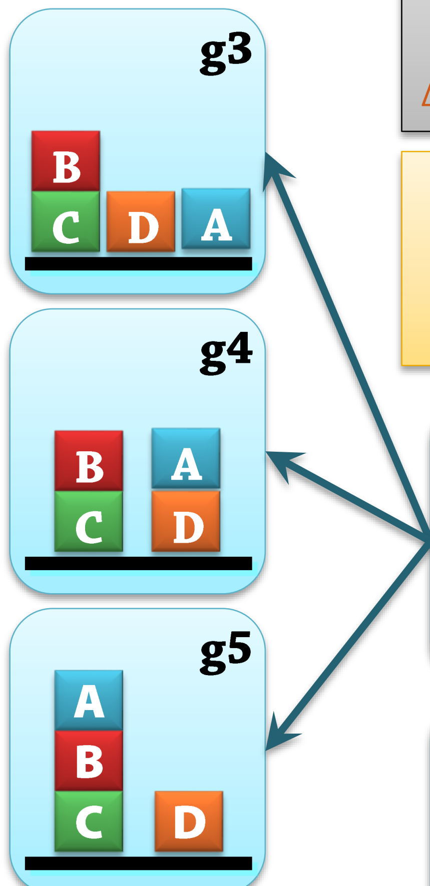
Calculations depend very much on the entire current state!
New search node → new current state → recalculate Δ_m from scratch

Backward Search with h_m

- In **backward** search:

Need
$\Delta_m(s0, g3)$,
$\Delta_m(s0, g4)$,
$\Delta_m(s0, g5)$

New search node → <u>same</u> starting state → use the <u>old</u> Δ_m values for previously encountered goal subsets

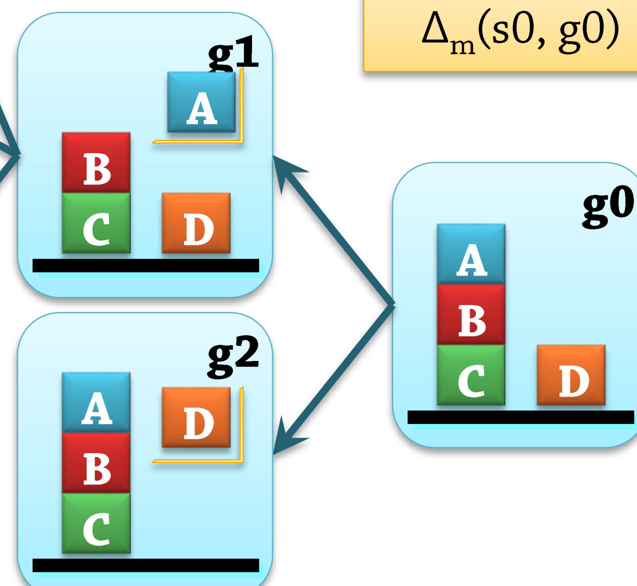


$\Delta_1(s0, g1)$
is the max of
$\Delta_1(s0, ontable(C))$,
$\Delta_1(s0, ontable(D))$,
$\Delta_1(s0, clear(A))$,
…,
$\Delta_1(s0, holding(A))$

Need
$\Delta_m(s0, g1)$,
$\Delta_m(s0, g2)$

$\Delta_1(s0, g0)$
is the max of
$\Delta_1(s0, ontable(C))$,
$\Delta_1(s0, ontable(D))$,
$\Delta_1(s0, clear(A))$,
$\Delta_1(s0, clear(D))$,
$\Delta_1(s0, on(A,B))$,
…

Need
$\Delta_m(s0, g0)$



HSPr, HSPr*

- Results:
  - Faster calculation of heuristics
  - **Not applicable for *all* heuristics**!
    - Many other heuristics work better with forward planning

Heuristics for Satisficing Forward State Space Planning

Optimal and Satisficing Planning

- Optimal planning often uses admissible heuristics + A*
  - Are there **worthwhile alternatives**?

- If we need **optimality**:
  - Can't use non-admissible heuristics
  - Can't expand fewer nodes than A*

- But we are not limited to optimal plans!
  - High-quality non-optimal plans can be quite useful as well
  - **Satisficing** planning
    - Find a plan that is sufficiently good, sufficiently quickly
    - Handles larger problems

Investigate many **different points** on the efficiency/quality spectrum!

The h_{add} Heuristic Function

Also called h_0

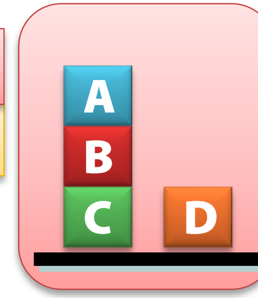
Background

- h_m heuristics are **admissible**, but not very **informative**
  - Only measure the **most expensive** goal subsets

- For satisficing planning, we do not need admissibility
  - What if we use the **sum** of individual plan lengths for each atom!
  - Result: h_{add}, also called h_0

The h_{add} Heuristic: Example

Goal:	*clear(A)*	on(A,B)	on(B,C)	*ontable(C)*	*clear(D)*	*ontable(D)*
	cost 0	cost 2	cost 3	cost 0	cost 0	cost 0



$h_{add}(s_0)$ = sum(2,3) = 5

stack(A,B)

holding(A)	*clear(B)*
cost 1	cost 0

unstack(A,C)

handempty	*clear(A)*	*on(A,C)*
cost 0	cost 0	cost 0

Cheaper!

unstack(A,D)

handempty	*clear(A)*	on(A,D)

More calculations → expensive…

stack(B,C)

holding(B)	clear(C)
cost 1	cost 1

pickup(B)

handempty	*clear(B)*
cost 0	cost 0

unstack(A,C)

handempty	*clear(A)*	*on(A,C)*

s_0: clear(A), on(A,C), ontable(C), clear(B), ontable(B), clear(D), ontable(D), handempty

The h_{add} Heuristic: Formal Definition

$h_{add}(s) = h_0(s) = \Delta_0(s, g)$ – the heuristic depends on the goal g

- For a **goal**, a **set** g of facts to achieve:
  - $\Delta_0(s, g)$ = the cost of achieving the **most expensive** proposition in g
    - $\Delta_0(s, g) = 0$ if $g \subseteq s$ // *Already achieved entire goal*
    - $\Delta_0(s, g) = \text{sum} \{ \Delta_0(s, p) \mid p \in g \}$ otherwise // *Part of the goal not achieved*

The cost of each atom p in goal g

Sum: We assume we have to achieve every subgoal separately

So how expensive is it to achieve a single proposition?

The h_{add} Heuristic: Formal Definition

$h_{add}(s) = h_0(s) = \Delta_0(s, g)$ – the heuristic depends on the goal g

- For a **single proposition** p to be achieved:
  - $\Delta_0(s, p)$ = the cost of **achieving p from s**
    - $\Delta_0(s, p) = 0$ if $p \in s$ *// Already achieved p*
    - $\Delta_0(s, p) = \infty$ if $\forall a \in A.\ p \notin effects^+(a)$ *// Unachievable*
    - Otherwise:

$$\Delta_0(s, p) = \min \{ cost(a) + \Delta_1(s, precond(a)) \mid a \in A \text{ and } p \in effects^+(a) \}$$

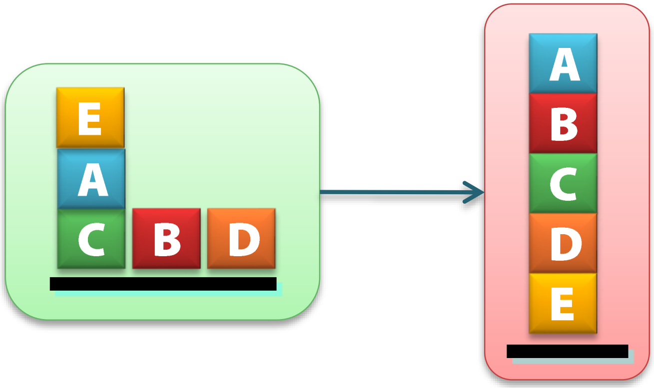
Must execute an action a∈A that achieves p,
and before that, *acheive its preconditions*

Min: Choose the action
that lets you achieve *p* as cheaply as possible

The h_{add} Heuristic: Example

jonkv@ida

- $h_{add}(s) = \Delta_0(s, g)$
  - For another example:
    - **ontable(E)**: unstack(E,A), putdown(E) ➔ 2
    - **clear(A)**: unstack(E,A) ➔ 1
    - **on(A,B)**: unstack(E,A), unstack(A,C), stack(A,B) ➔ 3
    - **on(B,C)**: unstack(E,A), unstack(A,C), pickup(B), stack(B,C) ➔ 4
    - **on(C,D)**: unstack(E,A), unstack(A,C), pickup(C), stack(C,D) ➔ 4
    - **on(D,E)**: pickup(D), stack(D,E) ➔ 2
    - ➔ sum is 16 [h+ = 10, h* = 12]



Can underestimate but also **overestimate**, not admissible!

The h_{add} Heuristic: Admissibility

- Why not admissible?
  - Does not take into account **interactions between goals**
  - Simple case: Same action used
    - **on(A,B)**: unstack(E,A); unstack(A,C); stack(A,B) ➔ 3
    - **on(B,C)**: unstack(E,A); unstack(A,C); pickup(B); stack(B,C) ➔ 4
  - More complicated to detect:
    - Goal: p and q
    - A1: effect p
    - A2: effect q
    - A3: effect p and q
    - To achieve p: Use A1
    - To achieve q: Use A2

– No specific action used twice
– Still misses interactions

Hill Climbing
in HSP (Heuristic Search Planner)

Satisficing planning, in a nutshell:

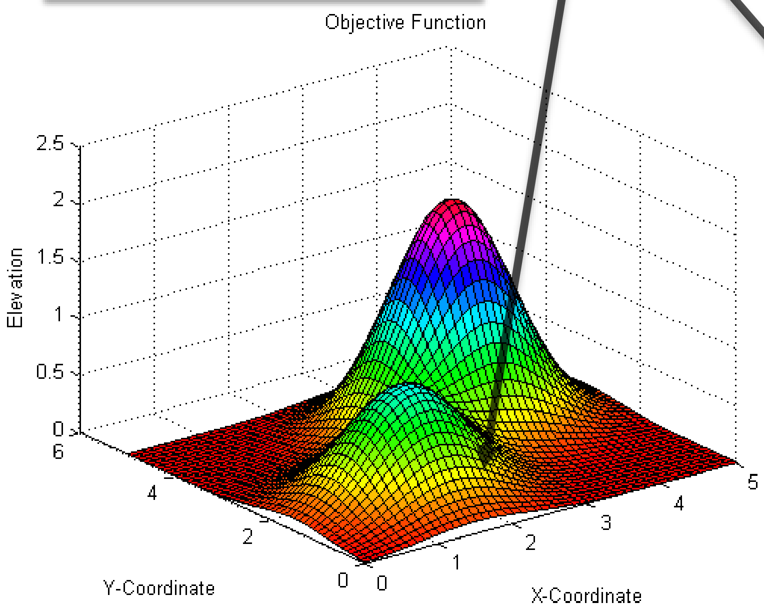
Try to move **quickly** towards a **reasonably good solution**

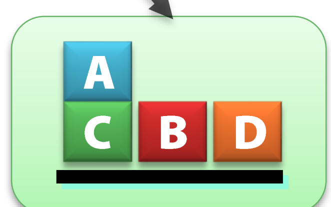
Hill Climbing (1)

- What about **Steepest Ascent Hill Climbing**?
  - **Greedy local search** algorithm for **optimization problems**
  - (1) Start in some **current location**

s=(x,y)

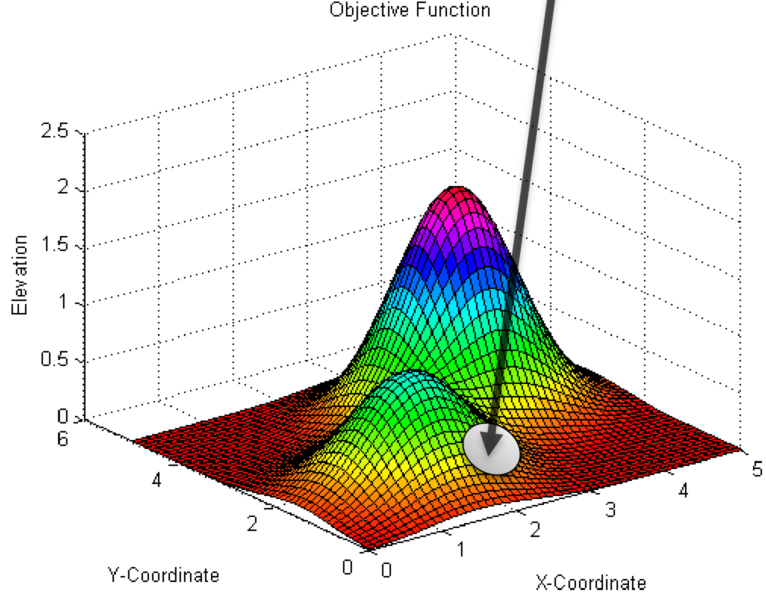
s={on(A,C),…}



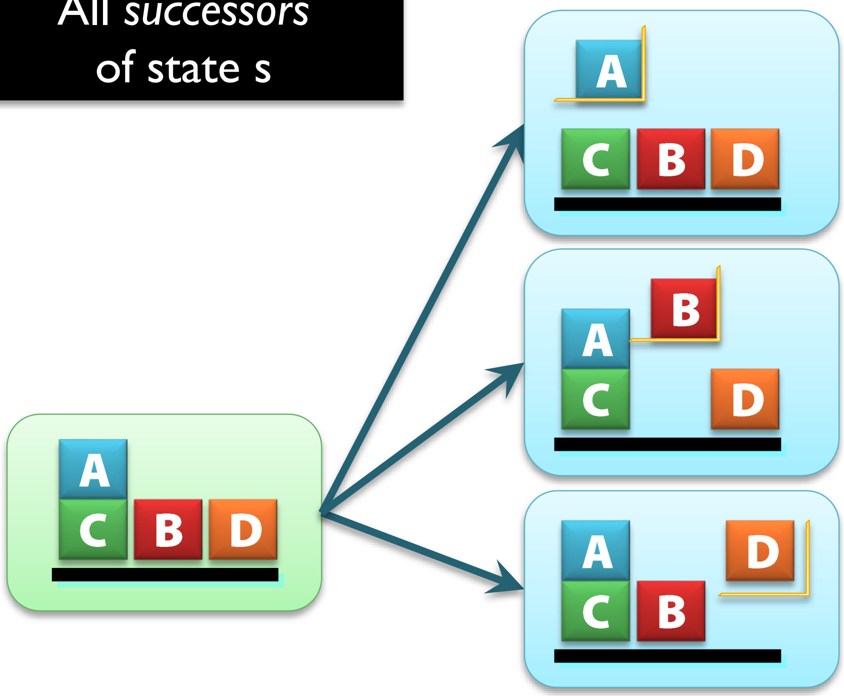


- (2) Find the **local neighborhood**, which can easily be reached

Example: Points (x,y) at a distance of 0.1

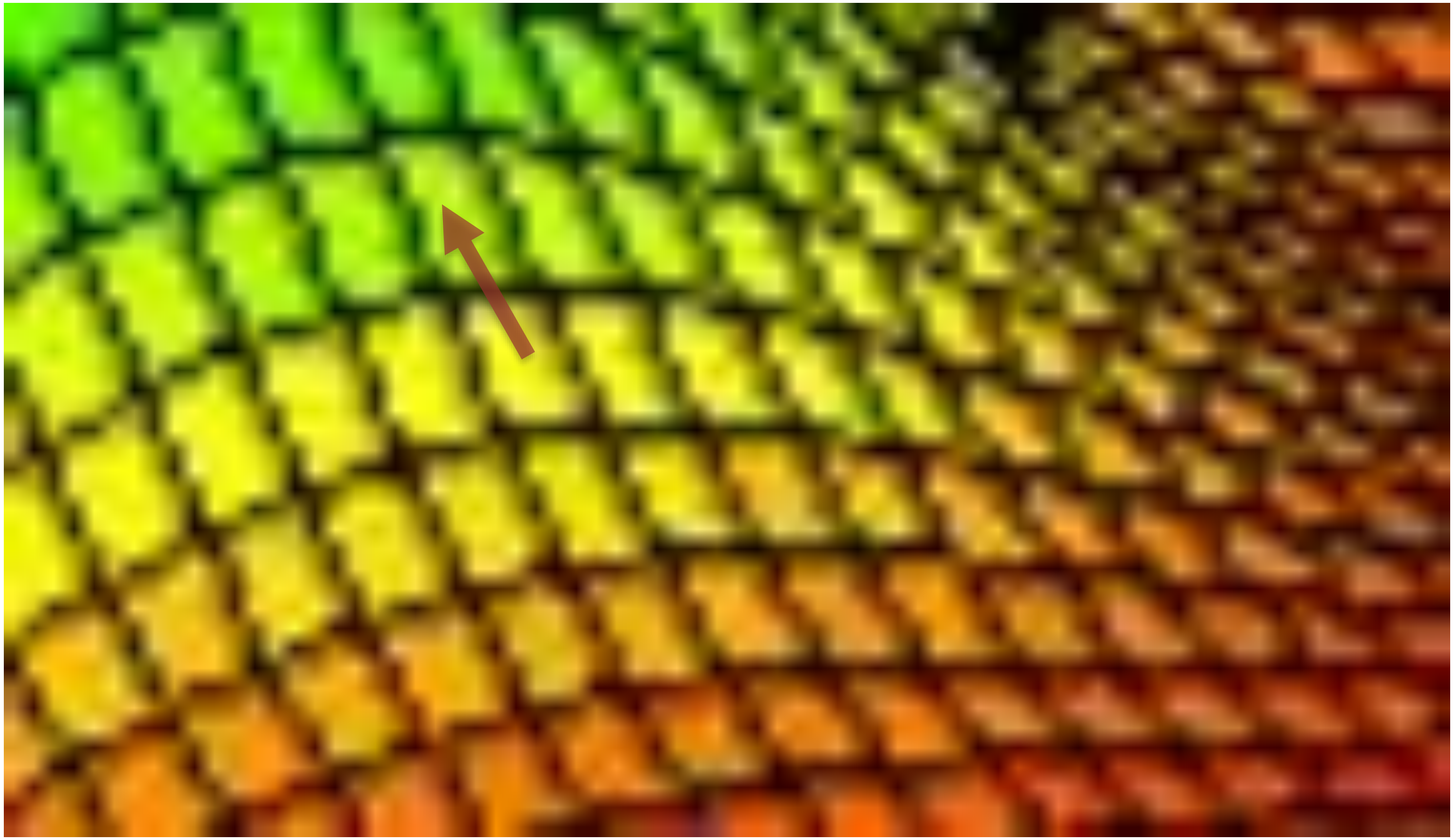


All *successors* of state s



Hill Climbing (3)

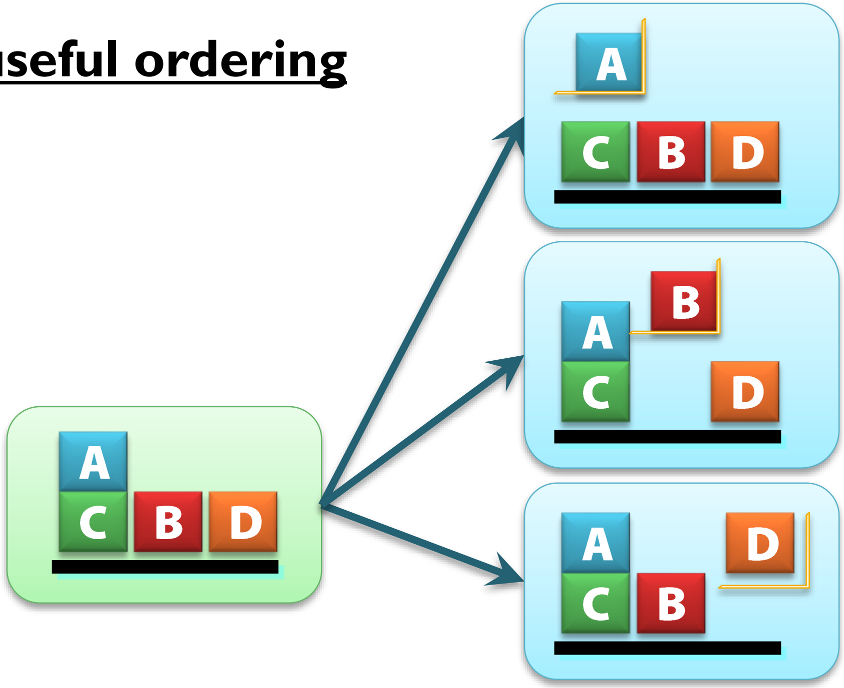
- (3) Make a **locally optimal** choice at each step:
  Chooses the *best* successor/neighbor



Hill Climbing (4)

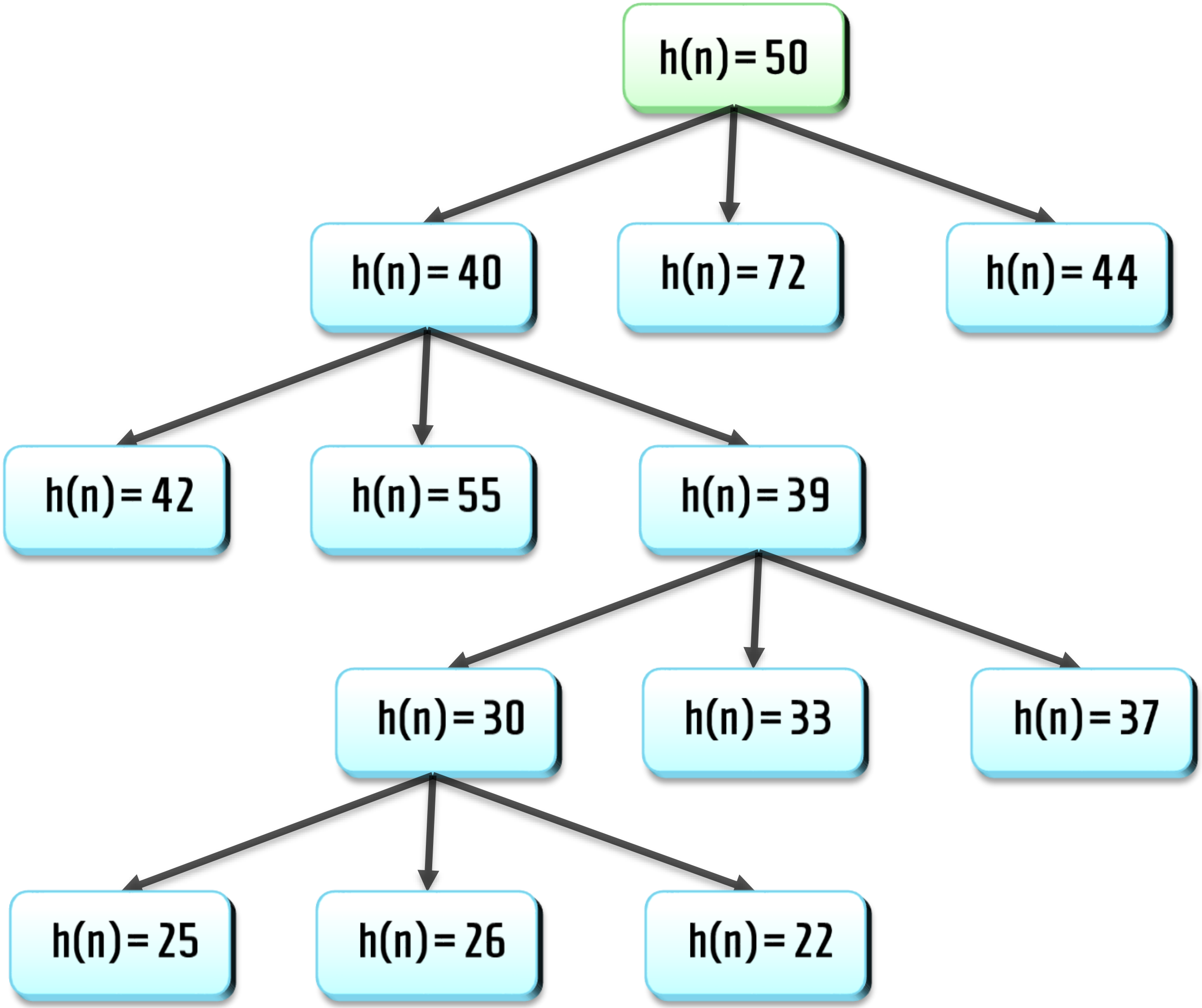
- We don't have a strict *state quality* measure!
  - Goal states are *perfect*, other states are *not solutions*

- But **minimizing heuristic value** might lead to a goal state…
  - (Minimize $h(n)$ = maximize $-h(n)$)
  - ➔ A good heuristic should provide **useful ordering**



Hill Climbing (5)

- Example of hill climbing search:



Hill Climbing (6)

Be stubborn:
Only consider children of this node, don't even keep track of other nodes to return to

<u>A* search:</u>

n ← initial state
open ← ∅
loop
  if *n* is a solution **then return** *n*
  <u>expand</u> children of *n*
  <u>calculate</u> *h* for children

  <u>add</u> children to *open*
  n ← node in *open*
    minimizing f(n) = g(n) + h(n)
end loop

<u>Steepest Ascent Hill-climbing</u>

n ← initial state

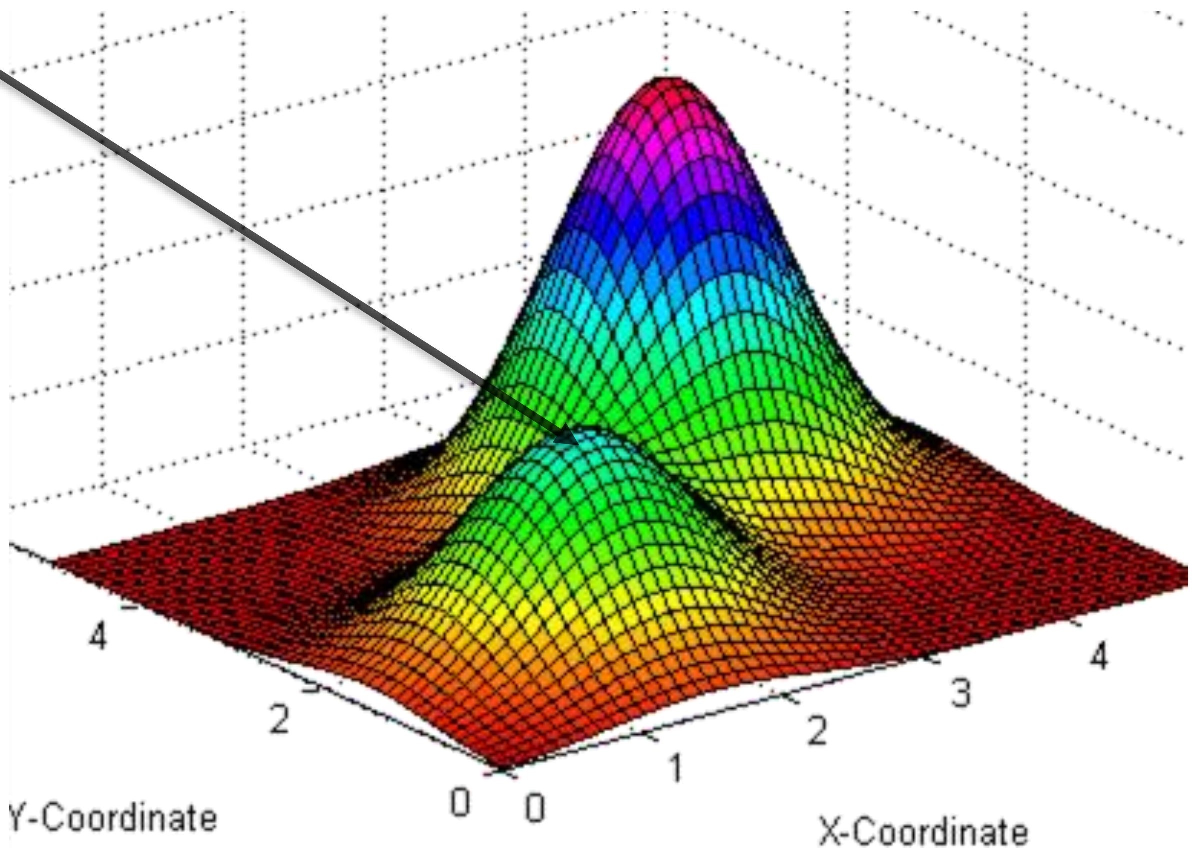
loop
  if *n* is a solution **then return** *n*
  <u>expand</u> children of *n*
  <u>calculate</u> *h* for children

  if (some <u>child</u> decreases h(*n*)):
    n ← child with minimal h(*n*)
  else stop // local optimum
end loop

Ignore g(n): prioritize **<u>finding a plan quickly</u>** over **<u>finding a good plan</u>**

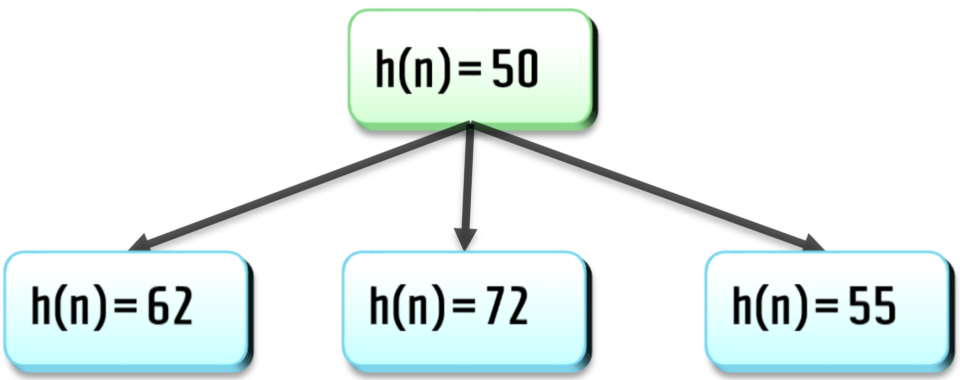
Local Optima (1)

- (4) When there is **nothing better** nearby: Stop!
  - HC is used for *optimization*
    - Any point is a *solution*,
      we search for the *best* one
  - Might find a *local optimum*:
    The top of a hill



Local Optima (2)

- Classical planning ➔ *absolute goals*
  - Even if we can't decrease h(n), we can't simply *stop*



<u>Steepest Ascent Hill-climbing</u>

n ← initial state
loop
  if *n* is a solution **then return** *n*
  <u>expand</u> children of *n*
  <u>calculate</u> *h* for children

  if (some <u>child</u> decreases h(*n*)):
    n ← child with minimal h(*n*)
  else stop // local optimum
end loop

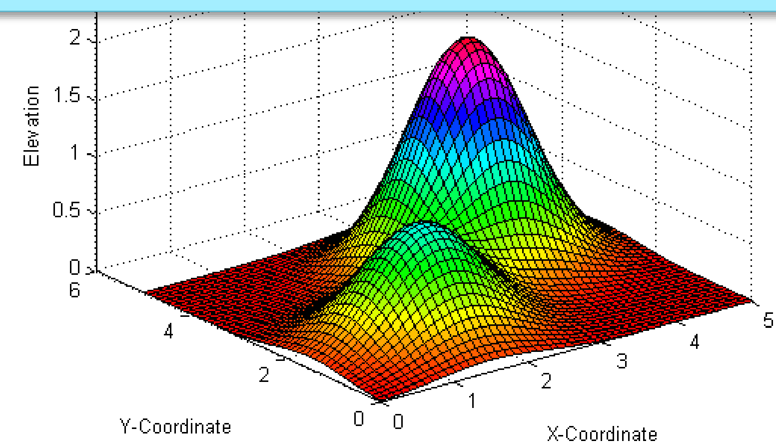
Local Optima (3)

- Standard solution to local optima: **<u>Random restart</u>**
  - Randomly choose another node/state
  - Continue searching from there
  - Hope you find a global optimum eventually
- Can *planners* choose *arbitrary* random states?

<u>Steepest Ascent</u>
<u>Hill-climbing with Restarts</u>

n ← initial state
loop
    if *n* is a solution **then return** *n*
    <u>expand</u> children of *n*
    <u>calculate</u> *h* for children

    if (some <u>child</u> decreases h(*n*)):
        n ← child with minimal h(*n*)
    else *n* ← some random state
end loop



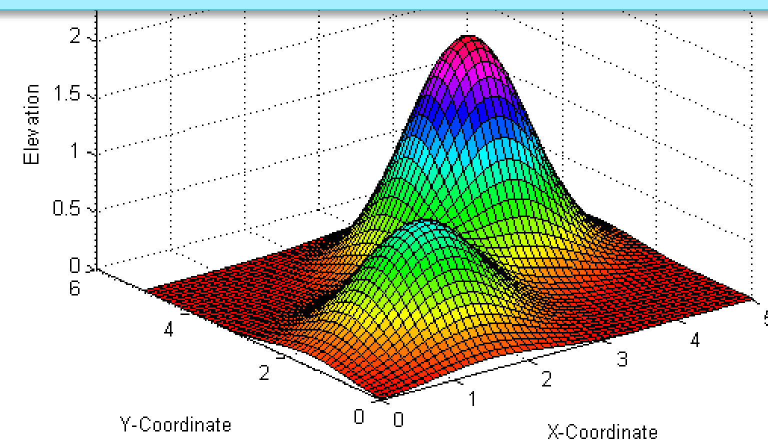
Local Optima (4)

- In planning:
  - The solution is not a *state* but the *path to the state*
  - Random states may not be reachable from the initial state
- So:
  - Randomly choose another *already visited* node/state
  - This node *is* reachable!

<u>Steepest Ascent</u>
<u>Hill-climbing with Restarts (2)</u>

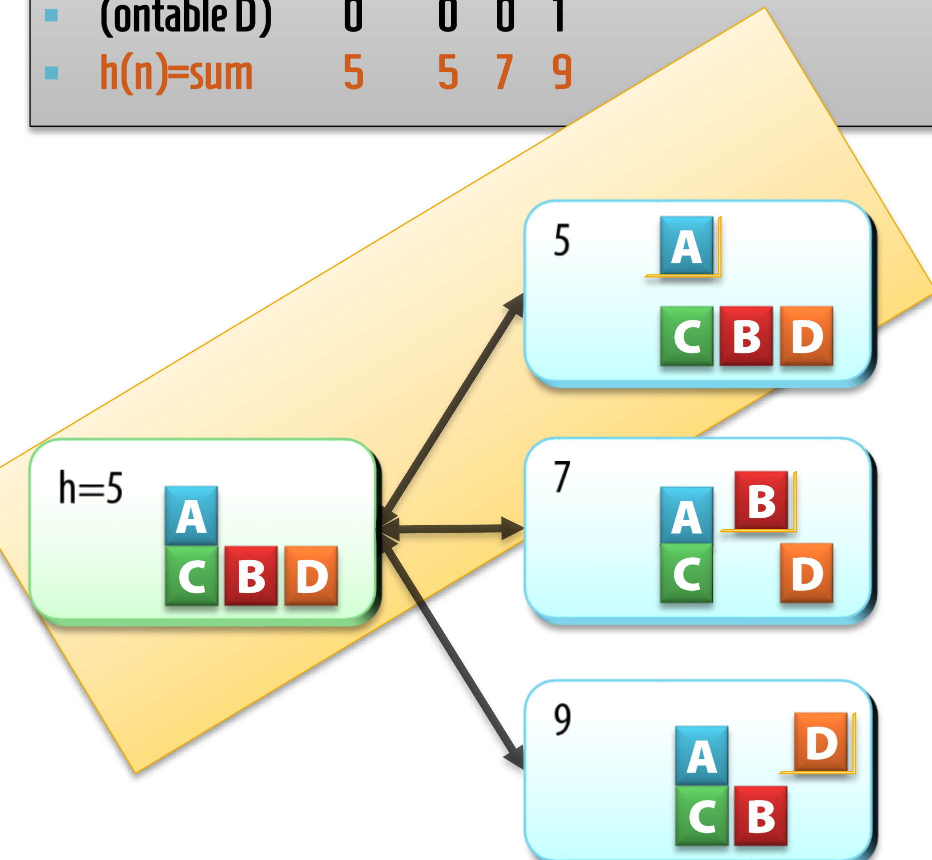
n ← initial state
loop
    if *n* is a solution **then return** *n*
    <u>expand</u> children of *n*
    <u>calculate</u> *h* for children

    if (some <u>child</u> decreases h(*n*)):
        n ← child with minimal h(*n*)
    else *n* ← some rnd. *visited* state
end loop



Hill Climbing with h_{add}: Plateaus

(on A B)	2	1	3	3
(on B C)	3	3	4	4
(clear A)	0	1	0	0
(clear D)	0	0	0	1
(ontable C)	0	0	0	0
(ontable D)	0	0	0	1
h(n)=sum	5	5	7	9



No successor <u>improves</u> the heuristic value; some are equal!

We have a **<u>plateau</u>**…

Jump to a random state immediately?

No: the heuristic is not so accurate – maybe some child *is* closer to the goal even though h(n) isn't lower!

➔ Let's keep exploring:
Allow a small number of consecutive **<u>moves across plateaus</u>**

Plateaus

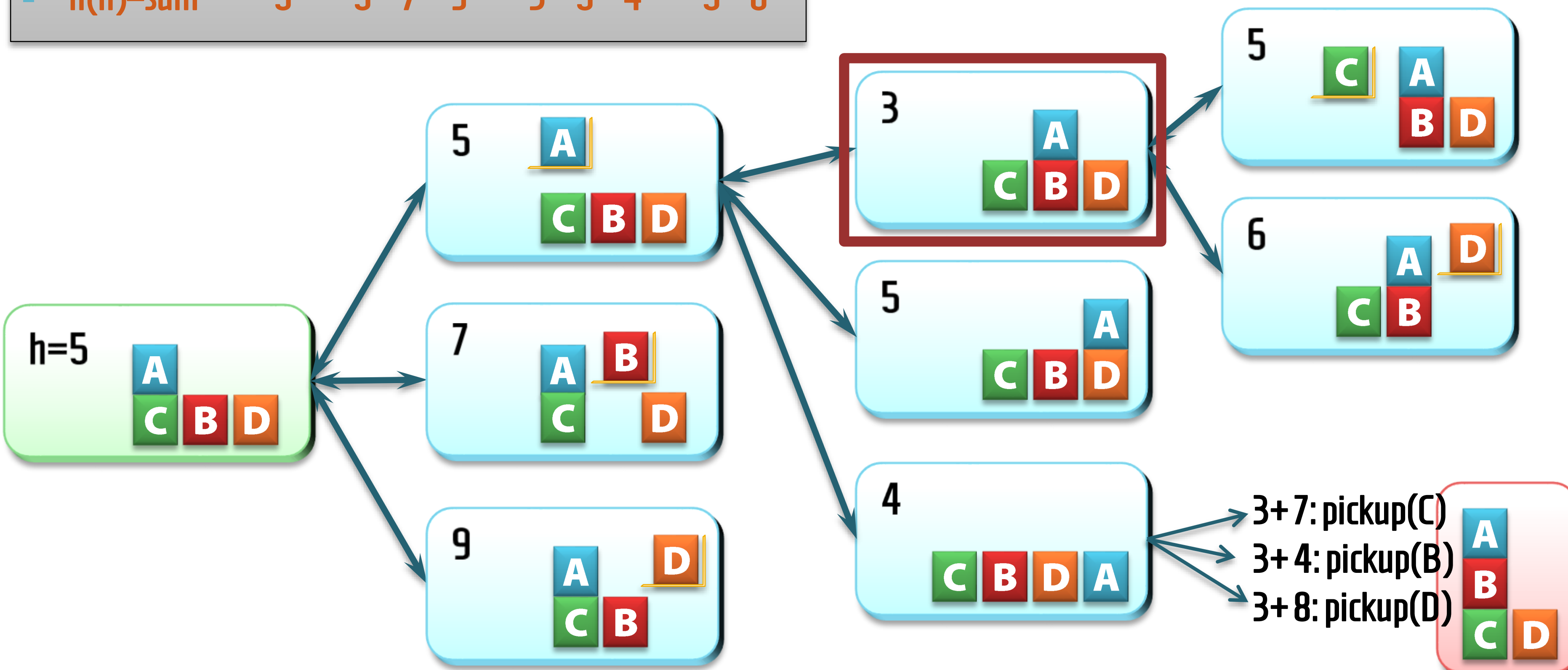
- A plateau…

Hill Climbing with h_{add}: Local Optima

(on A B)	2	1	3	3	0	2	2	0	0
(on B C)	3	3	4	4	3	2	2	4	4
(clear A)	0	1	0	0	0	0	0	0	0
(clear D)	0	0	0	1	0	1	0	0	1
(ontable C)	0	0	0	0	0	0	0	1	0
(ontable D)	0	0	0	1	0	0	0	0	1
h(n)=sum	5	5	7	9	3	5	4	5	6

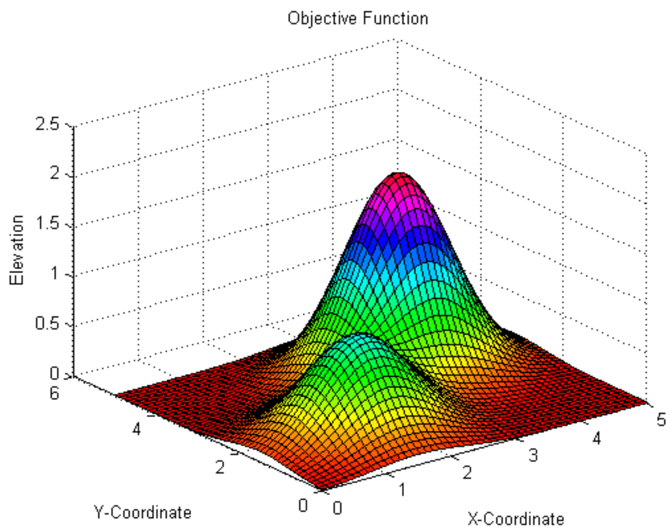
If we continue, all successors have higher heuristic values!

We have a **local optimum**…
Impasse = optimum or plateau
Some impasses allowed



Local Optima

- Local optimum: You can't improve the *heuristic function* in *one step*
  - But maybe you can still *get closer to the goal*:
    The heuristic only *approximates* our real objectives



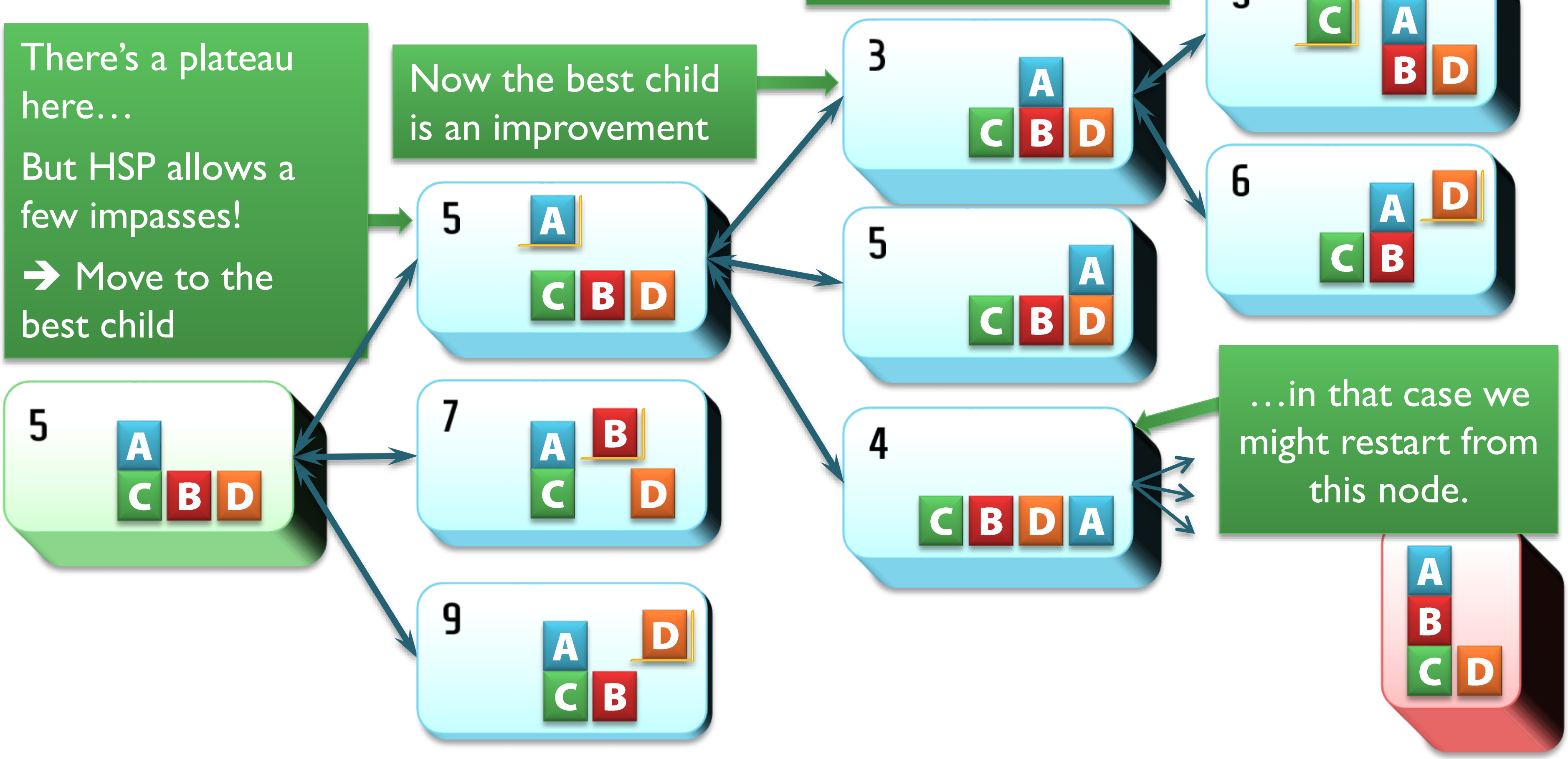
Impasses and Restarts

- What if there are **many** impasses?
  - Maybe we *are* in the wrong part of the search space after all…
    - Misguided by h_{add} at some earlier step
  - ➔ Select another *promising* expanded node where search continues

HSP Example

- Example from HSP 1.x:
  - Hill Climbing with h_{add} allowing some impasses (plus some other tweaks)

Its children seem to be worse. If we have reached the impasse threshold:



HSP 1: Heuristic Search Planner

- HSP 1.x: h_{add} heuristic + hill climbing + modifications
  - Works **approximately** like this (some intricacies omitted):
    - ```
      impasses = 0;
      unexpanded = { };
      current = initialNode;
      while (not yet reached the goal) {
          children ← expand(current);          // Apply all applicable actions
          if (children = Ø) {
              current = pop(unexpanded);
          } else {
              bestChild ← best(children);      // Child with the lowest heuristic value
              add other children to unexpanded in order of h(n); // Keep for restarts!
              if (h(bestChild) ≥ h(current)) {
                  impasses++;
                  if (impasses == threshold) {
                      current = pop(unexpanded);   // Restart from another node
                      impasses = 0;
                  }
              }
          }
      }
      ```

Dead end → restart

Essentially hill-climbing, but not all steps have to move "up"

Too many downhill/plateau moves → escape

Simple structure,
but highly competitive at its introduction
(using h_{add} as a heuristic)
      ```

Heuristics part III

Pattern Database Heuristics

Pattern Database Heuristics: Intro

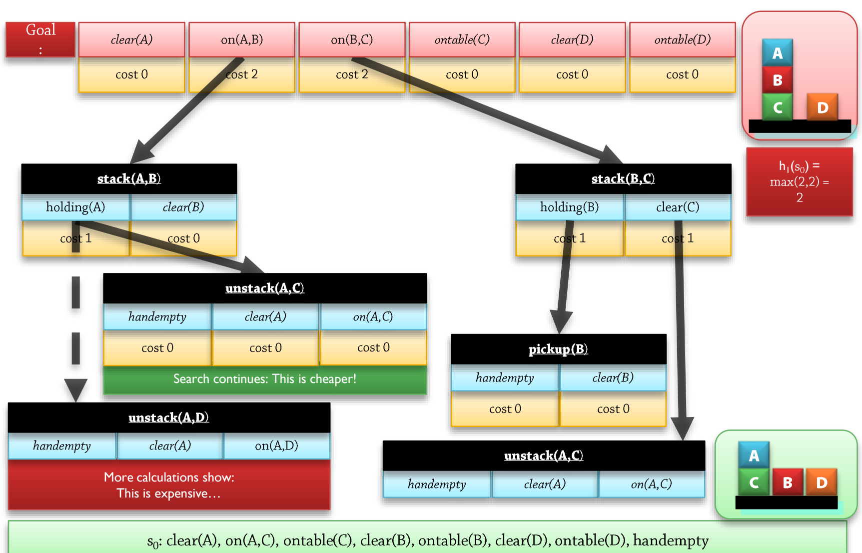
Many heuristics solve **subproblems**, combine their cost

In each subproblem for the h_m heuristics:

Pick m **goal literals** at a time
Ignore the others
Solve a subproblem optimally

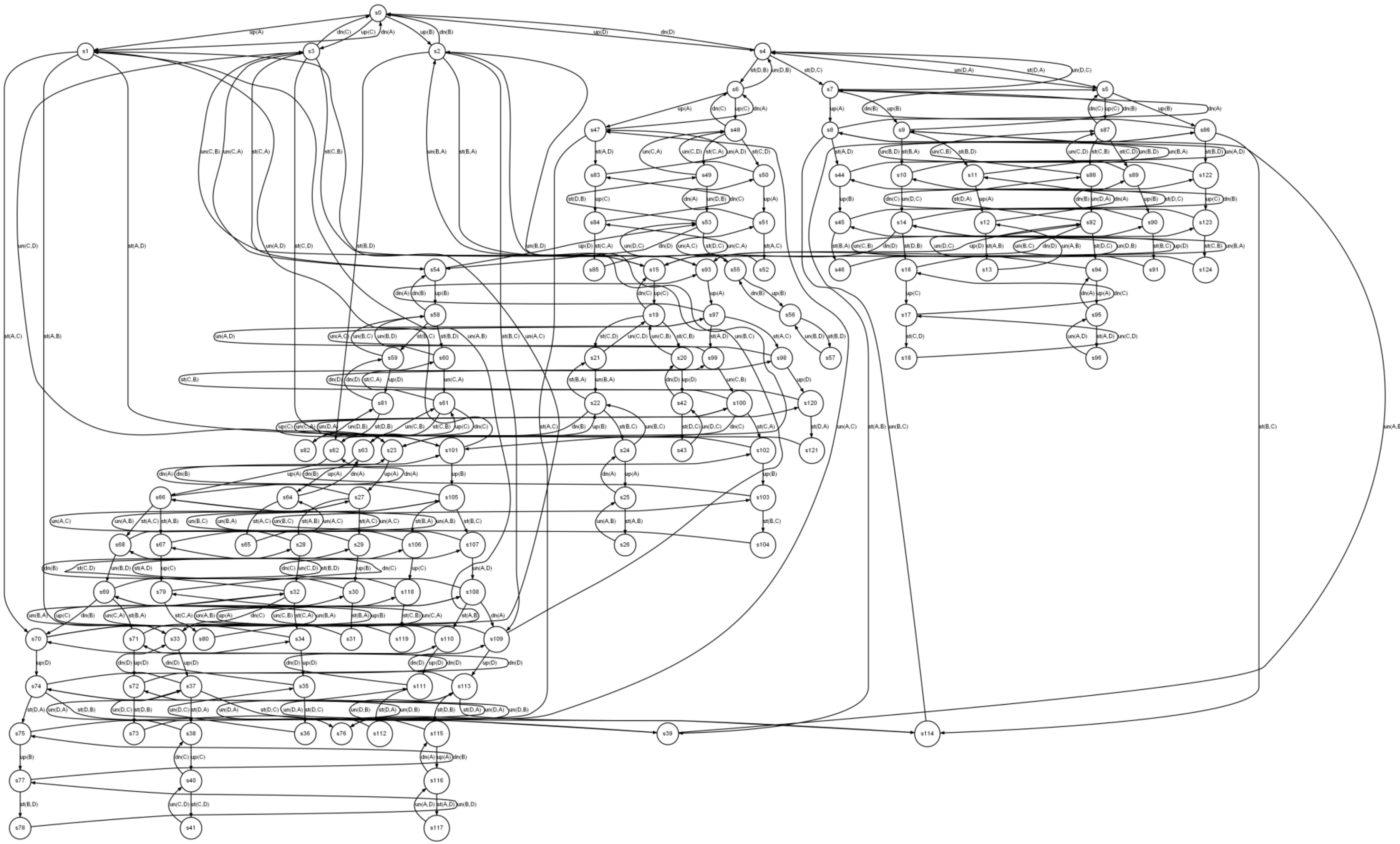
In each subproblem for Pattern Database (PDB) Heuristics

Pick some **ground facts** from the problem
Ignore the others
Solve a subproblem optimally



BW4: Achievable States

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



BW4: Ground Facts

- All **ground 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)

BW4: Potential Subproblem

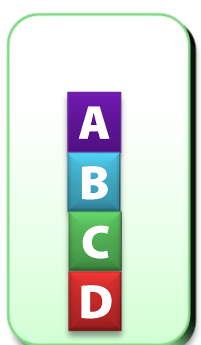
- Example: **only** consider some **ground facts** related to **block A**
  - **(on A B), (on A C), (on A D), (clear A), (ontable A)**
- **Initial state:**

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

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

An "**abstract state**"

- **Goal:**



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

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

BW4: Potential Subproblem (2)

- Including **(on A B), (on A C), (on A D), (clear A), (ontable A)**
  - **Example action**: (unstack A B)
    - **<u>Before transformation:</u>**
      :precondition (and (handempty) (clear A) (on A B))
      :effect (and (not (handempty)) (holding A) (not (clear A)) (clear B) (not (on A B)))
    - **<u>After transformation:</u>**
      :precondition (and (clear A) (on A B))
      :effect (and (not (clear A)) (not (on A B)))

Loses **some** preconditions and effects

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

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

PDB Heuristics: Patterns

- The **set of ground facts** is called a **pattern**
  - Many states **match** the pattern,
    are represented by a single **abstract state**
  - Such states are considered *equivalent*

aboveA=clear, aboveB=clear, aboveC=clear, aboveD=clear, posA=on-table, posB=on-table, posC=on-table, posD=on-table, hand=empty	≈	aboveA=C, aboveB=clear, aboveC=A, aboveD=clear, posA=on-table, posB=on-table, posC=on-table, posD=on-table, hand=full	≈	aboveA=B, aboveB=clear, aboveC=D, aboveD=clear, posA=on-table, posB=on-table posC=on-table, posD=on-table, hand=empty	represented by a single abstract state	aboveB=clear, aboveD=clear, posB=on-table, posD=on-table

A pattern generally contains few variables/facts – sometimes only one!

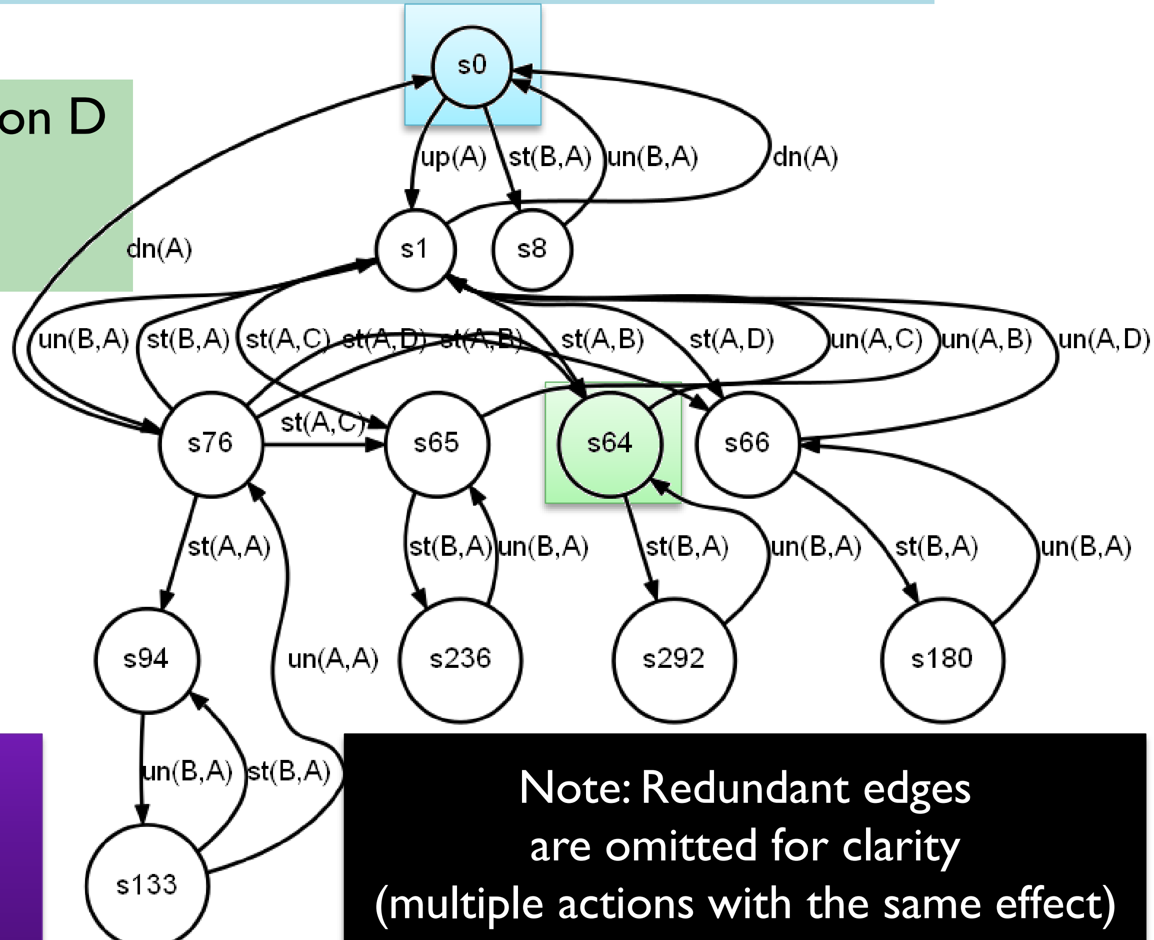
Relaxation?

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

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

BW4: State Transition Graph

- 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



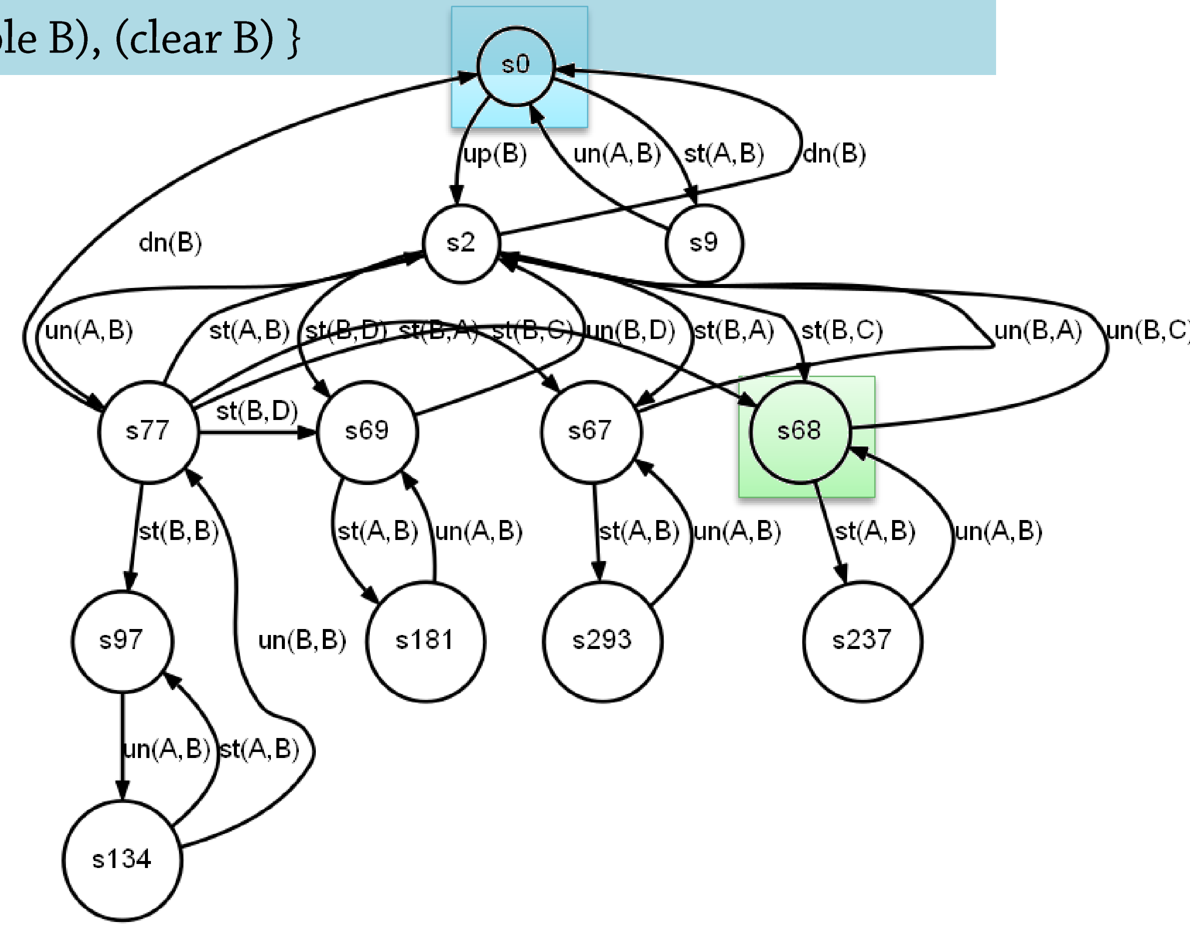
Optimal cost of a relaxation
➔
admissible heuristic

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

BW4: Subproblem 2

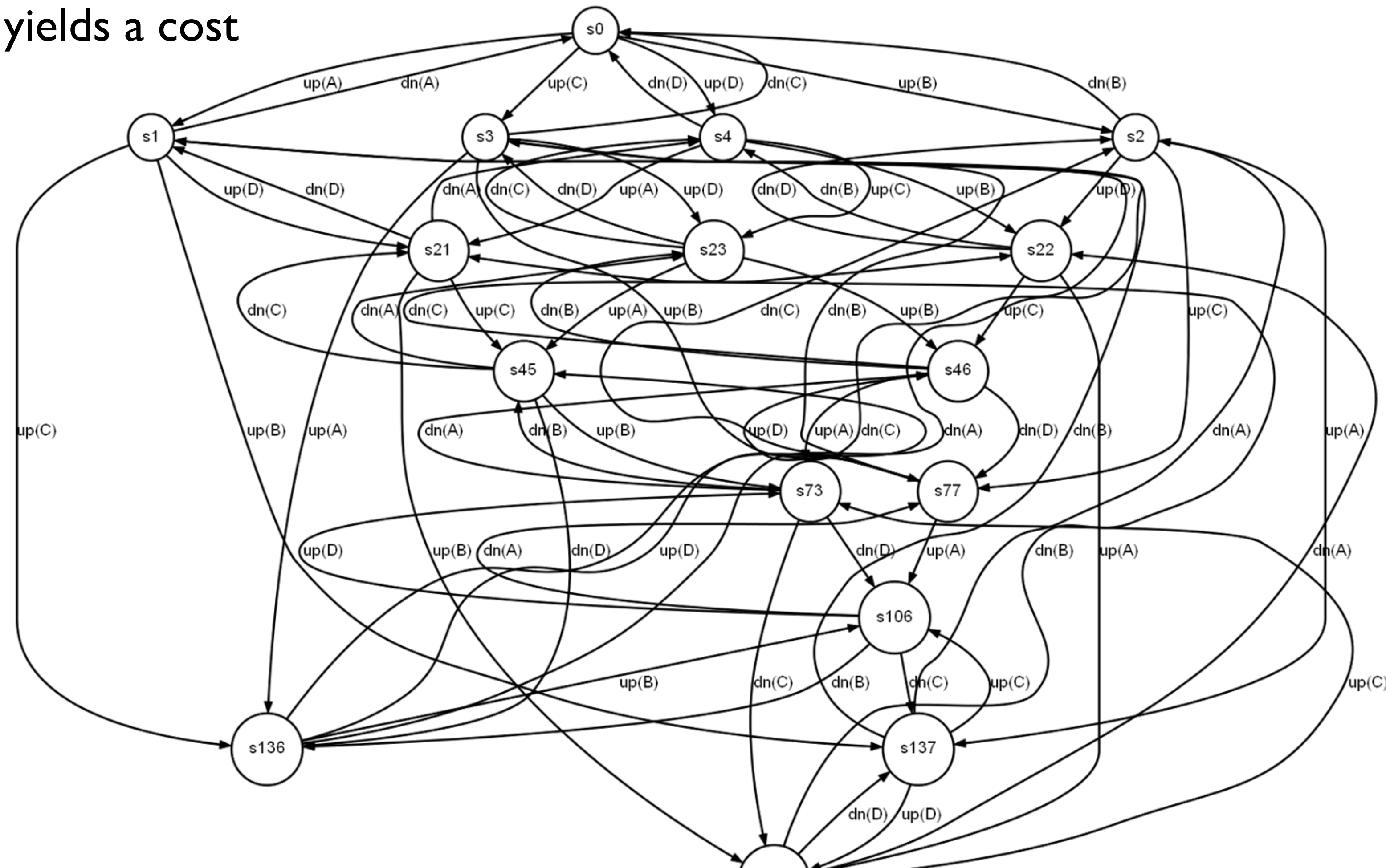
As in h_m, use multiple subproblems!

- Subproblem 2: Some facts related to B
  - **Current state**: Everything on the table, hand empty, all blocks clear
    - Abstract state: { (ontable B), (clear B) }
  - **Goal state**:
    A on B on C on D
    - Abstract goal:
      { (on B C) }
  - Find a path,
    compute its cost



BW4: Subproblem 3

- Subproblem 3: Only consider (holding ?x) facts…
  - Also yields a cost



As in h_m, take the maximum of these costs → admissible heuristic

Pattern Database Heuristics:

State Representation

PDB Heuristics: State Variables

- For PDB heuristics, a **state variable representation** is useful
  - Typically:
    - Reduces the number of facts
    - Provides more information about which states are actually reachable!
  - **Model** problems using the state variable representation,
    or let planners **convert** automatically from predicate representation

PDB Heuristics: State Variables (2)

- Example: Blocks world with 4 blocks
  - **536,870,912 states** (reachable and unreachable)
    in the standard predicate representation
  - But in **all states reachable** from "all-on-table" (all "normal" states):
    - Block A is:
      - Held in the gripper
      - Clear – at the top of a tower (possibly a tower of one block)
      - Below B
      - Below C, or
      - Below D
    - Equivalently: Exactly one of these facts is true *in every reachable state* (mutex!)
      - **(holding A), (clear A), (on B A), (on C A), (on D A)**
    - ➔ Remove those facts,
      introduce state variable **aboveA** ∈ **{ clear, B, C, D, holding }**

PDB Heuristics: State Variables (3)

- Example, continued
  - 536,870,912 states (reachable and unreachable) in predicate representation
  - 20,000 states (reachable and unreachable) in state variable representation:
    - aboveA ∈ { clear, B, C, D, holding }
    - aboveB ∈ { clear, A, C, D, holding }
    - aboveC ∈ { clear, A, B, D, holding }
    - aboveD ∈ { clear, A, B, C, holding }
    - posA ∈ { on-table, other }
    - posB ∈ { on-table, other }
    - posC ∈ { on-table, other }
    - posD ∈ { on-table, other }
    - hand ∈ { empty, full }

The state variable *translation*
is not part of the PDB heuristic!

Using state variables is *useful*
because PDBs work better
with fewer "irrelevant states"
in the state space…

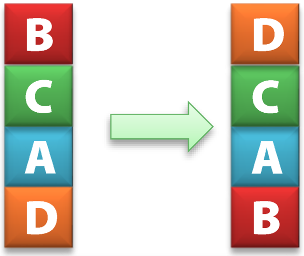
…so we can model using state variables,
or let the planner rewrite the problem
from PDDL predicates.

Provides more structure: Obvious that A can't be under B <u>and</u> under C

Useful when ignoring facts: Ignore <u>where A is</u>, care about <u>where B is</u>

PDB Heuristics: Rewriting the Problem

- **Rewriting** works as before
  - Suppose the pattern is **{ aboveB, aboveD, posB, posD }**
  - **Rewrite** the goal
    - Suppose that the original goal is expressed as
      Original: { aboveB = A, aboveA = C, aboveC = D, aboveD = clear, hand = empty }
    - Abstract: { aboveB = A, aboveD = clear }
  - **Rewrite** actions, removing some preconds / effects
    - (unstack A D) no longer requires aboveA = clear
    - (unstack B C) still requires aboveB = clear
  - …



aboveA	∈ { clear, B, C, D, holding }
aboveB	∈ { clear, A, C, D, holding }
aboveC	∈ { clear, A, B, D, holding }
aboveD	∈ { clear, A, B, C, holding }
posA	∈ { on-table, other }
posB	∈ { on-table, other }
posC	∈ { on-table, other }
posD	∈ { on-table, other }
hand	∈ { empty, full }

PDB Heuristics: Gripper Example

- A common *restricted* gripper domain:
  - **One** robot with **two** grippers
  - **Two** rooms
  - All n balls originally in the first room
  - Objective: All balls in the second room

Compact state variable representation:

loc(ball$_k$) $\in$ { room1, room2, gripper1, gripper2 }

loc-robot $\in$ { room1, room2 }

$2 * 4^n$ states, some unreachable – which ones?

Standard predicate representation: $2^{4n+4} = 4^{2n+2}$

Possible patterns:

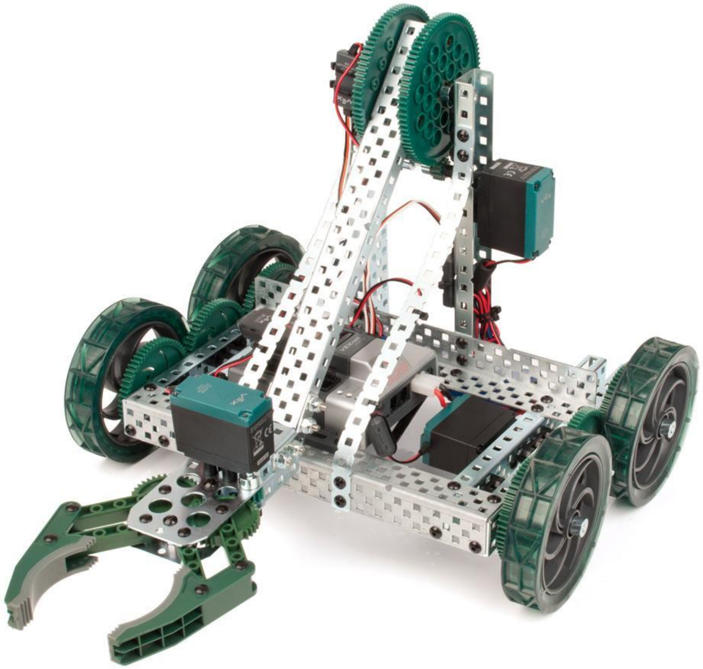
{ loc(ball$_1$) } → 4 abstract states

{ loc(ball$_1$), loc-robot } → 8 abstract states

{ loc(ball$_k$) | k ≤ n} → 4^n abstract states

{ loc(ball$_k$) | k ≤ log(n)} → $4^{\log(n)}$ abstract states

…

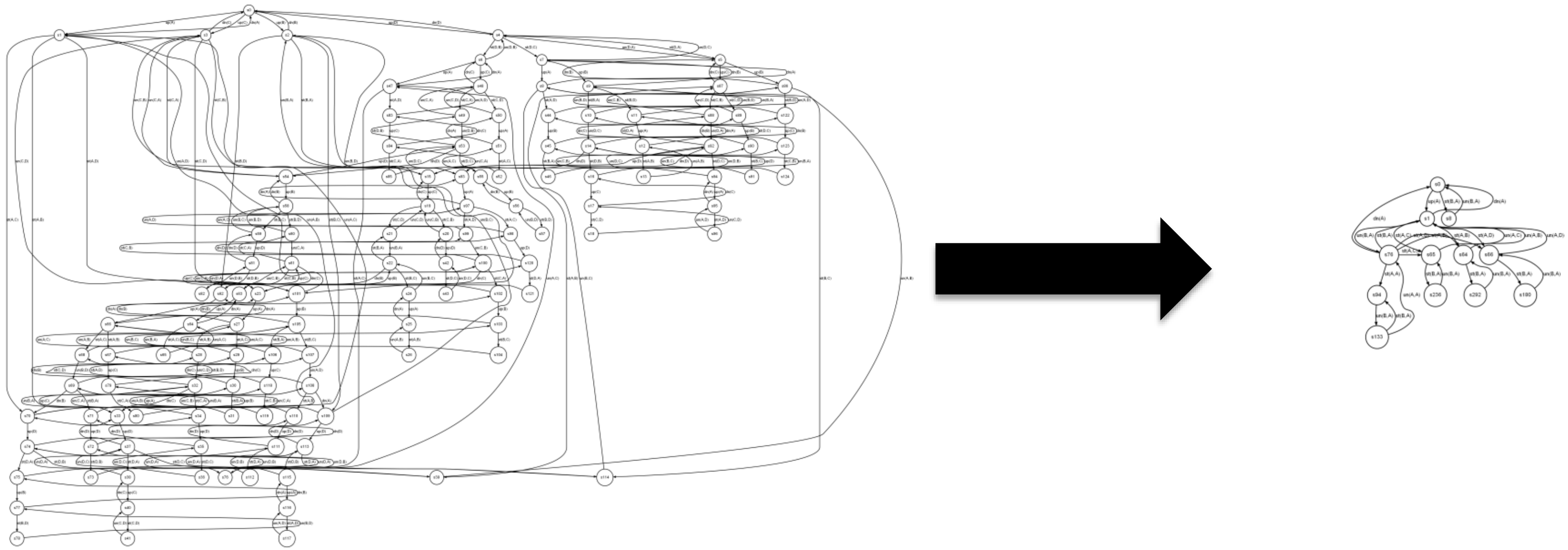


Pattern Database Heuristics:

Computation

PDB Heuristics: Databases!

- Because we keep *few* state variables:
  - *Many* real states map to the *same* abstract state
  - ➔ Every abstract state may be encountered many times during search
  - ➔ Cache calculated costs

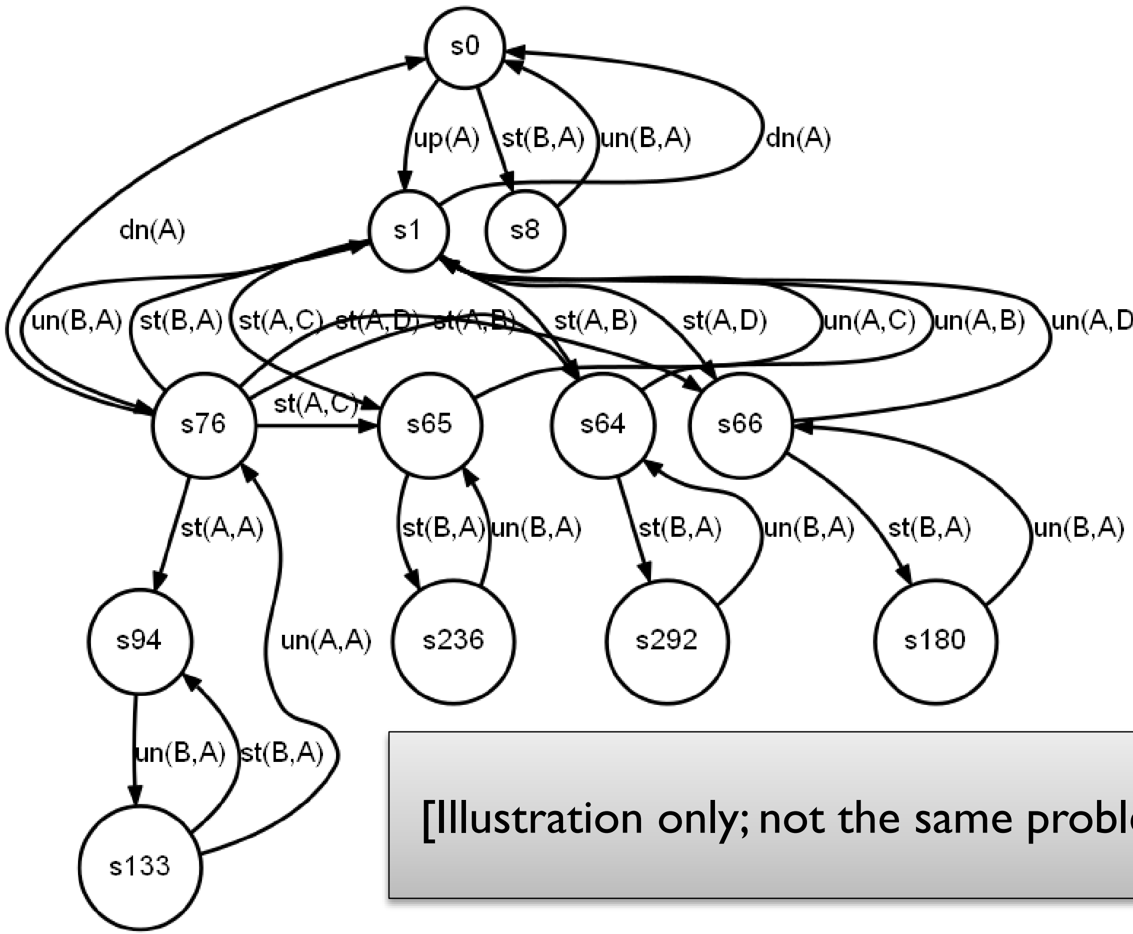


- Dijkstra efficiently finds optimal paths from *all* abstract states
  - ➔ Precalculate **all** heuristic values for each pattern
  - Store in a look-up table – a **database**

PDB Heuristics: Calculating (1)

- 1: Find all abstract states *reachable* from the abstract initial state

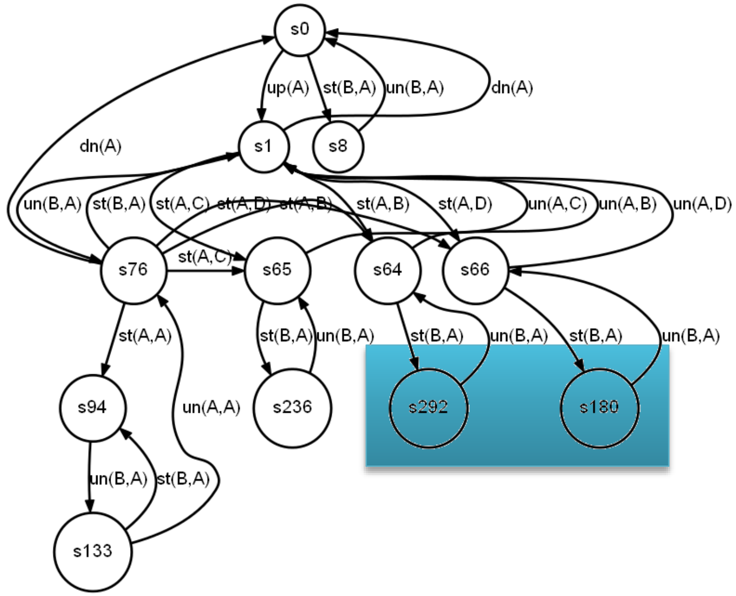
aboveA=clear,
aboveB=clear,
aboveC=clear,
aboveD=clear,
posA=on-table,
posB=on-table,
posC=on-table,
posD=on-table,
hand=empty



[Illustration only; not the same problem]

PDB Heuristics: Calculating (2)

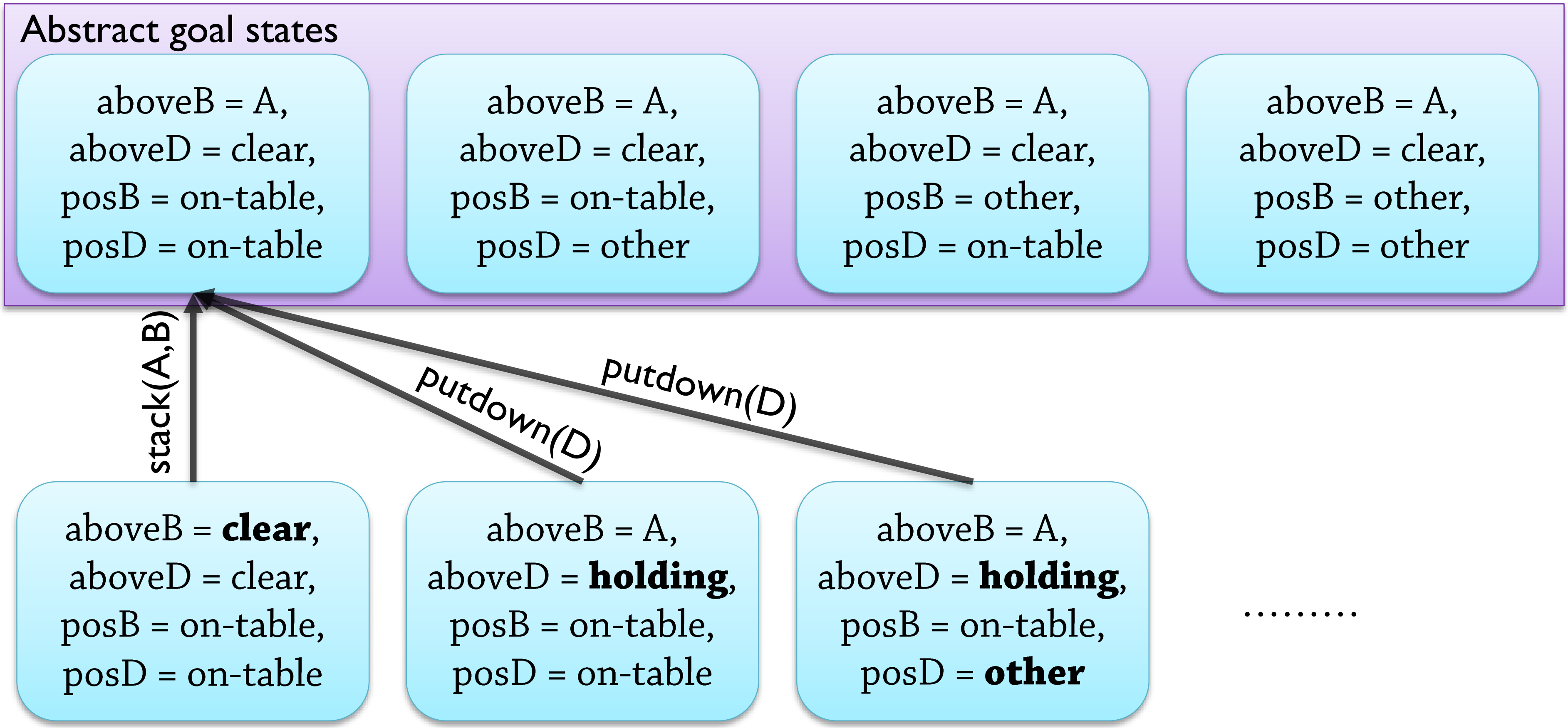
- 2: Find all *reachable abstract goal states*
  - Real goal = { aboveB = A, aboveA = C, aboveC = D, aboveD = clear, hand = empty }
  - Abs. goal = { aboveB = A, aboveD = clear }
  - Abs. goal states = { aboveB = A, aboveD = clear, posB = on-table, posD = on-table },
    { aboveB = A, aboveD = clear, posB = on-table, posD = other },
    { aboveB = A, aboveD = clear, posB = other, posD = on-table },
    { aboveB = A, aboveD = clear, posB = other, posD = other }



PDB Heuristics: Calculating (3)

- 3: Compute the database
  - For **every reachable abstract state**,
    find a cheapest path to **any abstract goal state**
  - Can be done with backward search
    from the set of reachable abstract goal states, using Dijkstra

PDB Heuristics: Calculating (4)



PDB Heuristics: Databases

jonkv@ida

Abstract goal states

aboveB = A, aboveD = clear, posB = on-table, posD = on-table **cost 0**	aboveB = A, aboveD = clear, posB = on-table, posD = other **cost 0**	aboveB = A, aboveD = clear, posB = other, posD = on-table **cost 0**	aboveB = A, aboveD = clear, posB = other, posD = other **cost 0**

aboveB = **clear**, aboveD = clear, posB = on-table, posD = on-table **cost 1**	aboveB = A, aboveD = **holding**, posB = on-table, posD = on-table **cost 1**	aboveB = A, aboveD = **holding**, posB = on-table, posD = other **cost 1**	…and so on for all reachable abstract states

This database represents an *admissible heuristic*!
Given a *real* state:
Find the unique abstract state that matches; **return** its precomputed cost

PDB Heuristics: Complexity

- Database:
  - Stores one cost for every **abstract state** s
    - Cost is **optimal** within the relaxed problem
    - Cost is **admissible** for the "real" problem

- For the database to be computable in polynomial time:
  - As **problem instances** grow,
    the **pattern** can (only) grow to include a *logarithmic* number of variables

  - Problem size n, maximum number of values for a state variable d ➔
    number of pattern variables: $O(\log n)$,
    number of abstract states for the pattern: $O\left(d^{\log n}\right) = O(n^{\log d})$

  - Dijkstra is polynomial in the number of states

How are PDBs used
when solving the original planning problem?

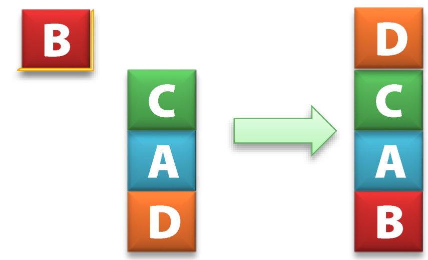
Step 1: Using a single pattern

PDB Heuristics in Forward Search (1)

- Step 1: **Automatically** generate a planning space abstraction
  - A pattern, a selection of state variables to consider
  - Choosing a **good** abstraction is a **difficult problem**!
    - Different approaches exist…
- Step 2: Calculate the pattern database
  - As already discussed

PDB Heuristics in Forward Search (2)

- Step 3: Forward search in the **original problem**
  - For each new successor state s_1, calculate heuristic value $h_{pdb}(s_1)$
    - Example: s_1 = { aboveD = A, aboveA = C, aboveC = clear, aboveB = holding,
      posA = other, posB = other,
      posC = other, posD = on-table,
      hand = full }



- Convert this to an *abstract* state
  - Example: s_1' = { aboveB = holding, aboveD = A, posB = other, posD = on-table }
- Use the database to quickly look up $h_{pdb}(s_1)$ =
  the cost of reaching the nearest abstract goal from s_1'

aboveB = holding, aboveD = A, posB = other, posD = on-table ➔ cost *n1*
aboveB = holding, aboveD = A, posB = other, posD = other ➔ cost *n2*
…

How can PDB heuristics become more informative?

Accuracy for a Single PDB Heuristic

- **<u>How close</u>** to h*(*n*) can an admissible PDB-based heuristic be?
  - Assuming polynomial computation:
    - Each abstraction can have at most O(log n) variables/groups
    - h(n) ≤ cost of reaching the *most expensive subgoal* of size O(log n)

Significant differences compared to h_m heuristics!

Subgoal size is not constant but *grows* with problem size

On the other hand, does not consider *all* subgoals of a particular size

Decides state variables *in advance* – for h_m, facts are chosen *on each level*

  - But still, log(n) grows much slower than n
    - ➔ For any given pattern, asymptotic accuracy is (often) 0
    - As before, *practical results* can be better!

Improving PDBs

- How to increase information?
  - Can't increase the **size** of a pattern beyond logarithmic growth…

- Can use **multiple** abstractions / patterns!
  - For each abstraction, compute a separate **pattern database**
  - Each such cost is an admissible heuristic
    - So the **maximum** over many different abstractions is also an admissible heuristic

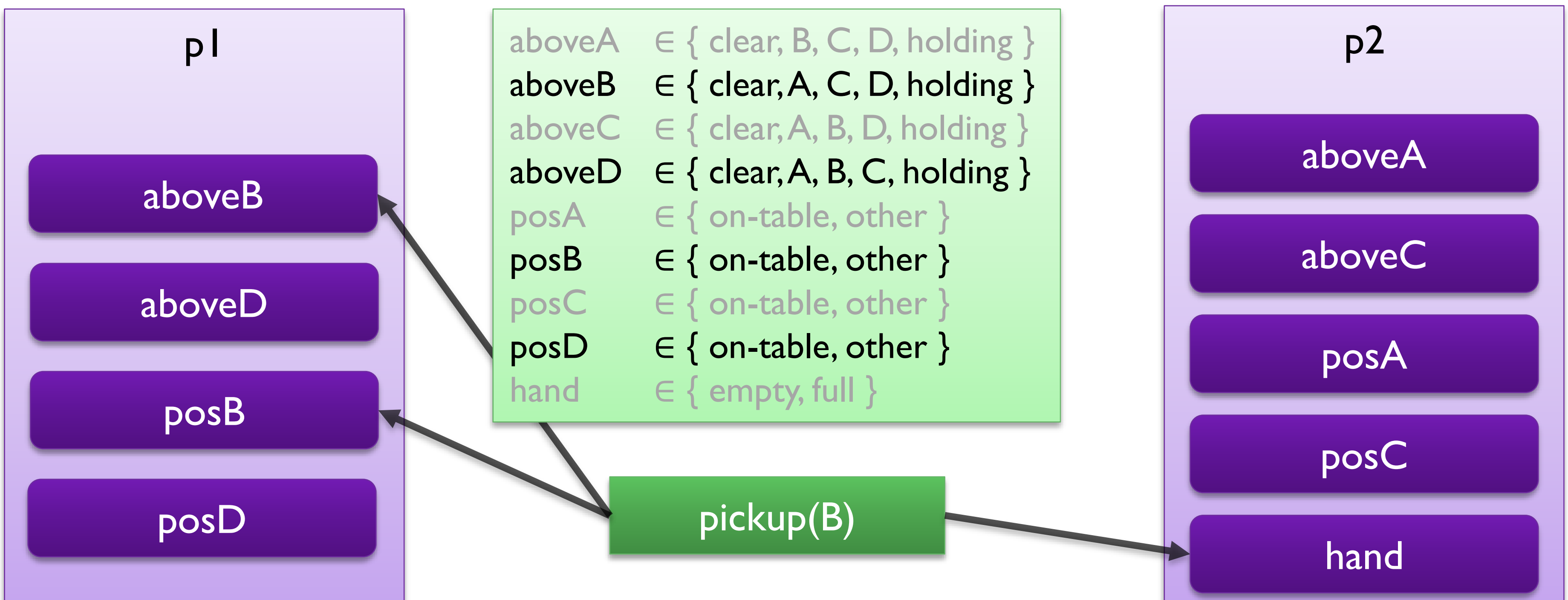
- What is the new level of accuracy?
  - Still 0… *asymptotically*
  - But this can still help *in practice*!

Additive PDB Heuristics (1)

- To improve further:
  - Define **multiple** patterns
  - **Sum** the heuristic values given by each pattern
- As in h_{add}, this **could** lead to **overestimation problems**
  - Some of the effort necessary to reach the goal is counted twice
- To avoid this and create an **admissible** heuristic:
  - Each fact should be in *at most* one pattern
  - Each action should affect facts in *at most* one pattern
  - → **Additive** pattern database heuristics

Additive PDB Heuristics (2)

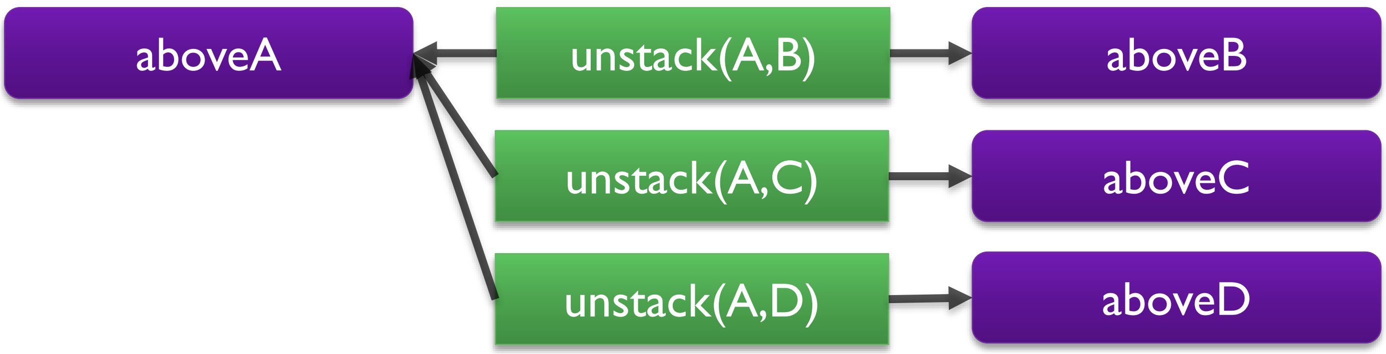
- BW: Is p1={facts in even rows}, p2={facts in odd rows} *additive*?
  - No: pickup(B) affects {aboveB,posB} in p1, {hand} in p2



One potential problem:
Both patterns could use pickup(B) in their optimal solutions
➔ sum counts this twice! This is what we're trying to avoid…

Additive PDB Heuristics (3)

- BW: Is p1={aboveA}, p2={aboveB} additive?
  - No: unstack(A,B) affects {aboveB} in p1, {aboveA} in p2
  - True for *all* combinations of above*X*

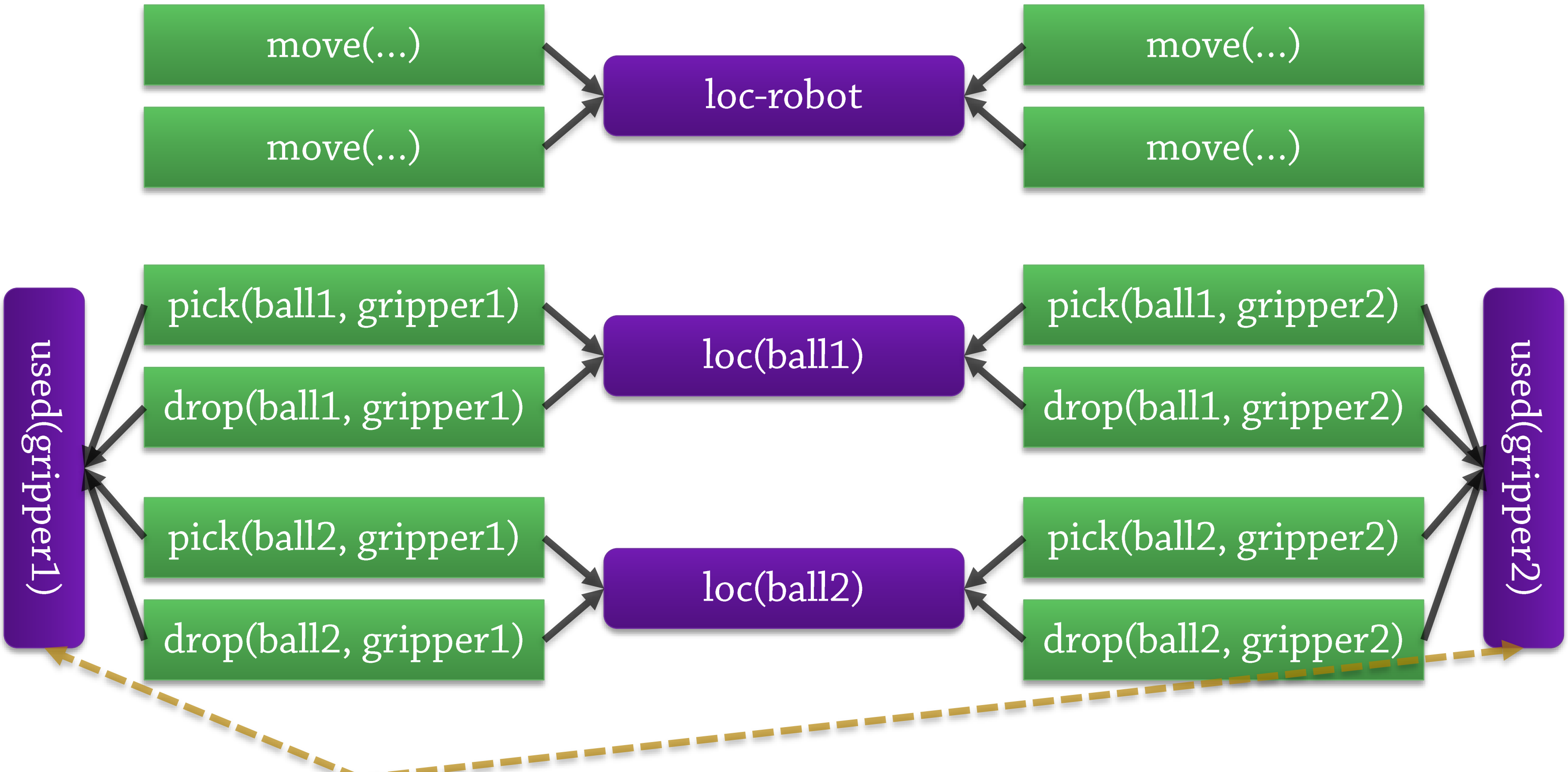


- An additive PDB heur. could use **one** of these:
  - p1 = { aboveA }
  - p1 = { aboveA, aboveC, aboveD }
  - …
- Can't have **two** separate patterns p1,p2 both of which include an above*X*
  - Those aboveX *will* be directly connected by some unstack action

This formulation of the Blocks World is "connected in the wrong way" for this approach to work well

Additive PDB Heuristics (4)

- "Separating" patterns in the Gripper domain:

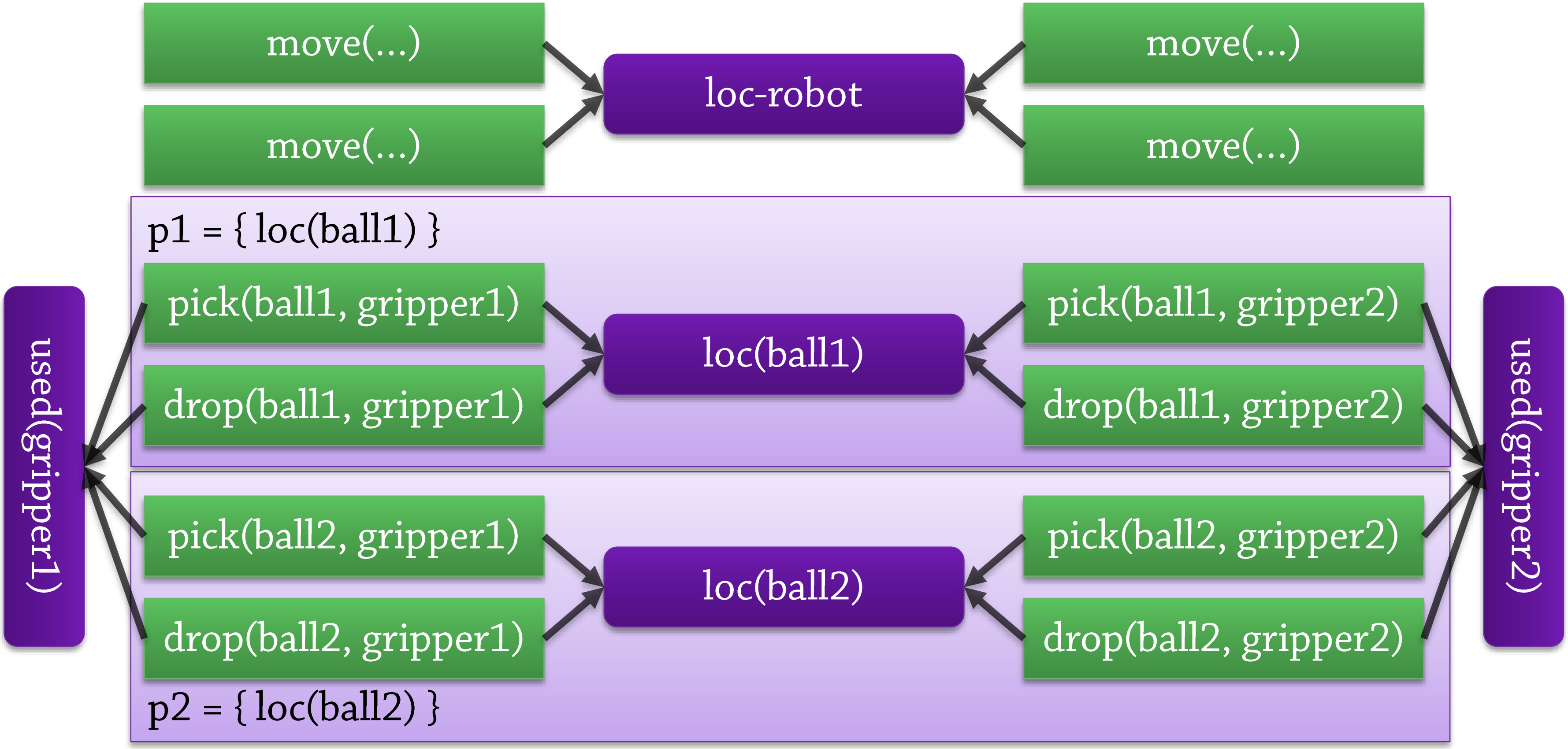


Are these a problem?

loc($ball_k$)	∈ { room1, room2, gripper1, gripper2 }
loc-robot	∈ { room1, room2 }
used($gripper_k$)	∈ { true, false }

Additive PDB Heuristics (5)

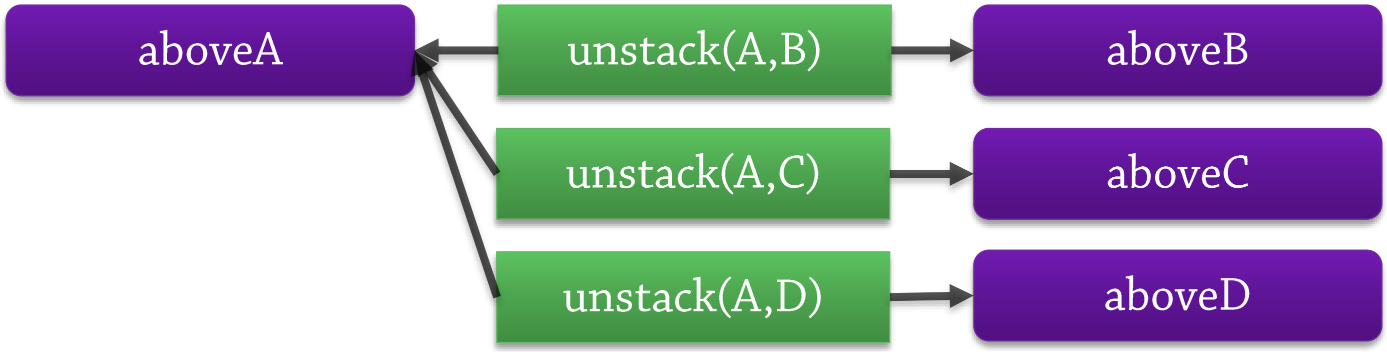
- No problem: We don't have to use **all** variables in patterns!



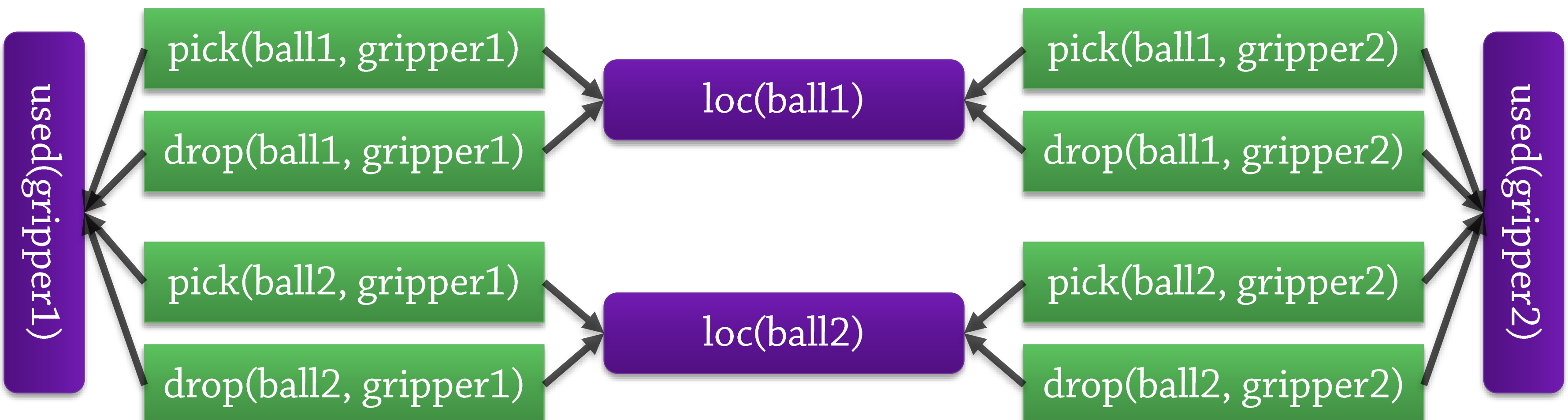
For each pattern we chose one **variable**
Then we have to include **all actions** affecting it
The other variables those actions affect [used()] don't have to be part of *any* pattern!

Additive PDB Heuristics (6)

- Notice the difference in structure!



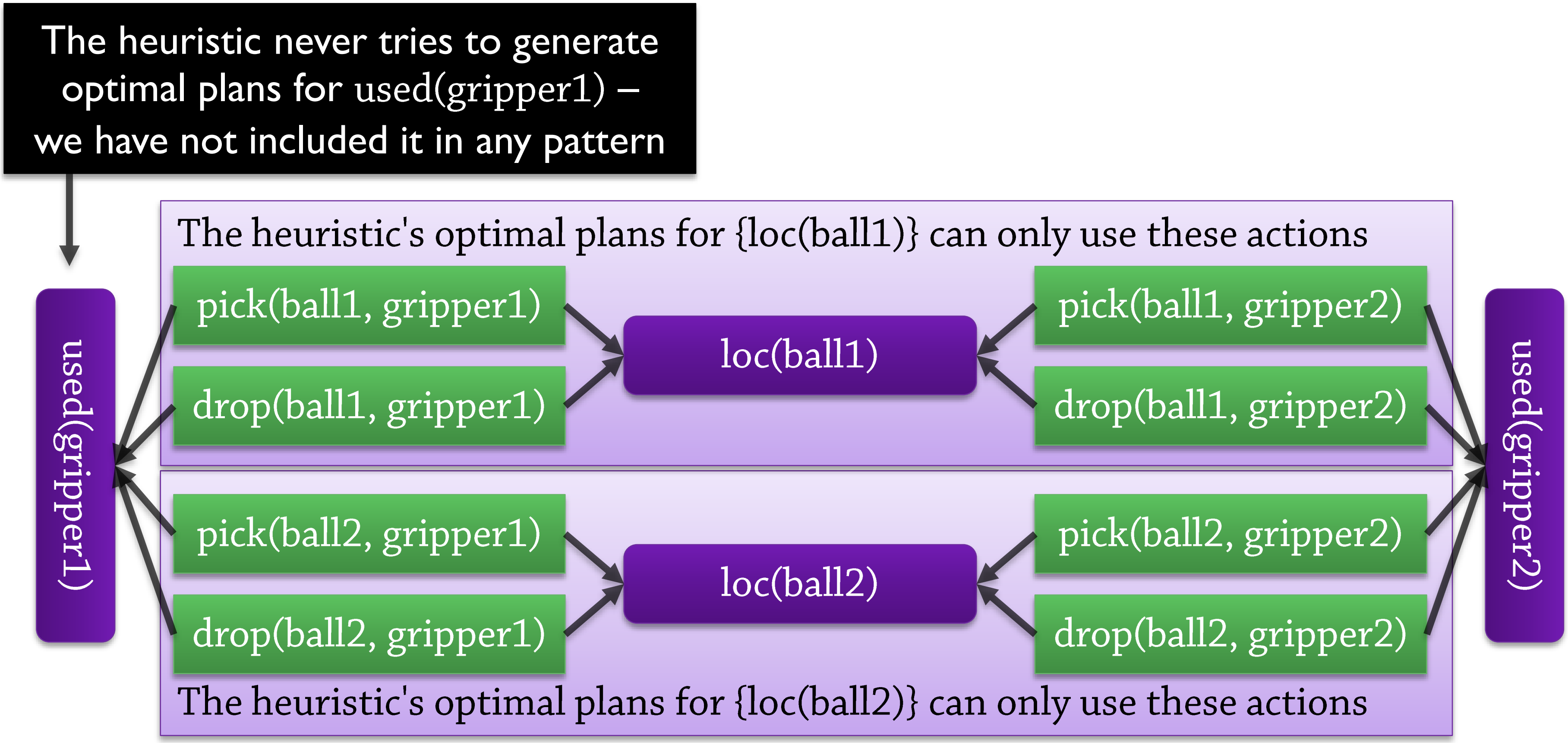
BW: ***Every*** **pair of aboveX facts has a *direct connection* through an action**



Gripper: ***No*** **pair of loc() facts has a *direct connection* through an action**

Additive PDB Heuristics (7)

- When every action affects facts in at most *one* pattern:
  - The subproblems we generated are completely *disjoint*
    - They achieve **<u>different aspects</u>** of the *goal*
    - Optimal solutions **<u>must</u>** use **<u>different actions</u>**



Additive PDB Heuristics (8)

- Avoids the overestimation problem we had with h_{add}

Problem earlier:

Goal: p and q
A1: effect p
A2: effect q
A3: effect p and q

To achieve p: Heuristic uses A1
To achieve q: Heuristic uses A2

Sum of costs is 2 – optimal cost is 1, using A3

This cannot happen
when every action affects facts in at most *one* pattern

→ The abstractions are additive
→ Adding costs from multiple heuristics yields an *admissible* heuristic!

Additive PDB Heuristics (9)

- Can be taken one step further…
  - Suppose we have several *sets* of additive abstractions:
    - Can calculate an admissible heuristic from **each** additive set, then take the maximum of the results as a **stronger** admissible heuristic

Max ➔
admissible heuristic $h^3_{pdb}(s) = \max(h^1_{pdb}(s), h^2_{pdb}(s))$

Sum ➔
admissible heuristic $h^1_{pdb}(s)$

Sum ➔
admissible heuristic $h^2_{pdb}(s)$

p1 p2 p3 p4

p5 p6 p7 p8 p9

4 patterns satisfying additive constraints

5 patterns satisfying additive constraints

Additive PDB Heuristics (10)

- **How close** to h*(*n*) can an **additive** PDB-based heuristic be?
  - For additive PDB heuristics with a single sum, **asymptotic accuracy** as problem size approaches infinity…

- **In Gripper**:
  - In state s_n there are *n* balls in room1, and no balls are carried
  - Additive PDB heuristic $h_{add}^{PDB}(s_n)$:
    - One singleton pattern for each ball location variable loc($ball_k$)
    - For each pattern, the optimal cost is 2
      - pick(ball,room1,gripper1): loc(ball)=room1 ➔ loc(ball)=gripper1
      - drop(ball,room2,gripper1): loc(ball)=gripper1 ➔ loc(ball)=room2
    - $h_{add}^{PDB}(s_n)$ = sum for *n* balls = 2*n*
  - Real cost:
    - Use both grippers: *pick, pick, move(room1,room2), drop, drop, move(room2,room1)*
    - Repeat n/2 times, total cost ≈ 6n/2 = 3n
  - ➔ Asymptotic accuracy 2n/3n = 2/3

Additive PDB Heuristics (11)

- **How close** to h*(*n*) can an **additive** PDB-based heuristic be?
  - For additive PDB heuristics with a single sum, **asymptotic accuracy** as problem size approaches infinity:

	h+ (too slow!)	**h2**	**Additive PDB**
Gripper	2/3	0	2/3
Logistics	3/4	0	1/2
Blocks world	1/4	0	0
Miconic-STRIPS	6/7	0	1/2
Miconic-Simple-ADL	3/4	0	0
Schedule	1/4	0	1/2
Satellite	1/2	0	1/6

  - **Only achieved** if the planner **finds** the best combination of abstractions!
  - This is a very difficult problem in itself!

Heuristics part IV

An Overview of 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)

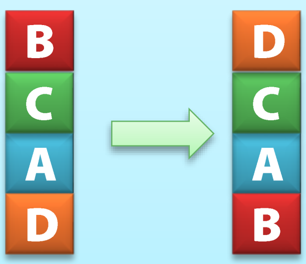
<u>Landmarks in planning</u>:

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

Assume we are currently in state *s*…

<u>Fact Landmark for *s*</u>:

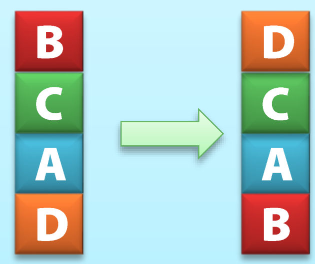
A <u>**fact**</u> that is <u>**not true**</u> in *s*,
but must be true at some point
in *every* solution starting in *s*



clear(A)
holding(C)
…

<u>Formula Landmark for *s*</u>:

A <u>**formula**</u> that is <u>**not true**</u> in *s*,
but must be true at some point
in *every* solution starting in *s*

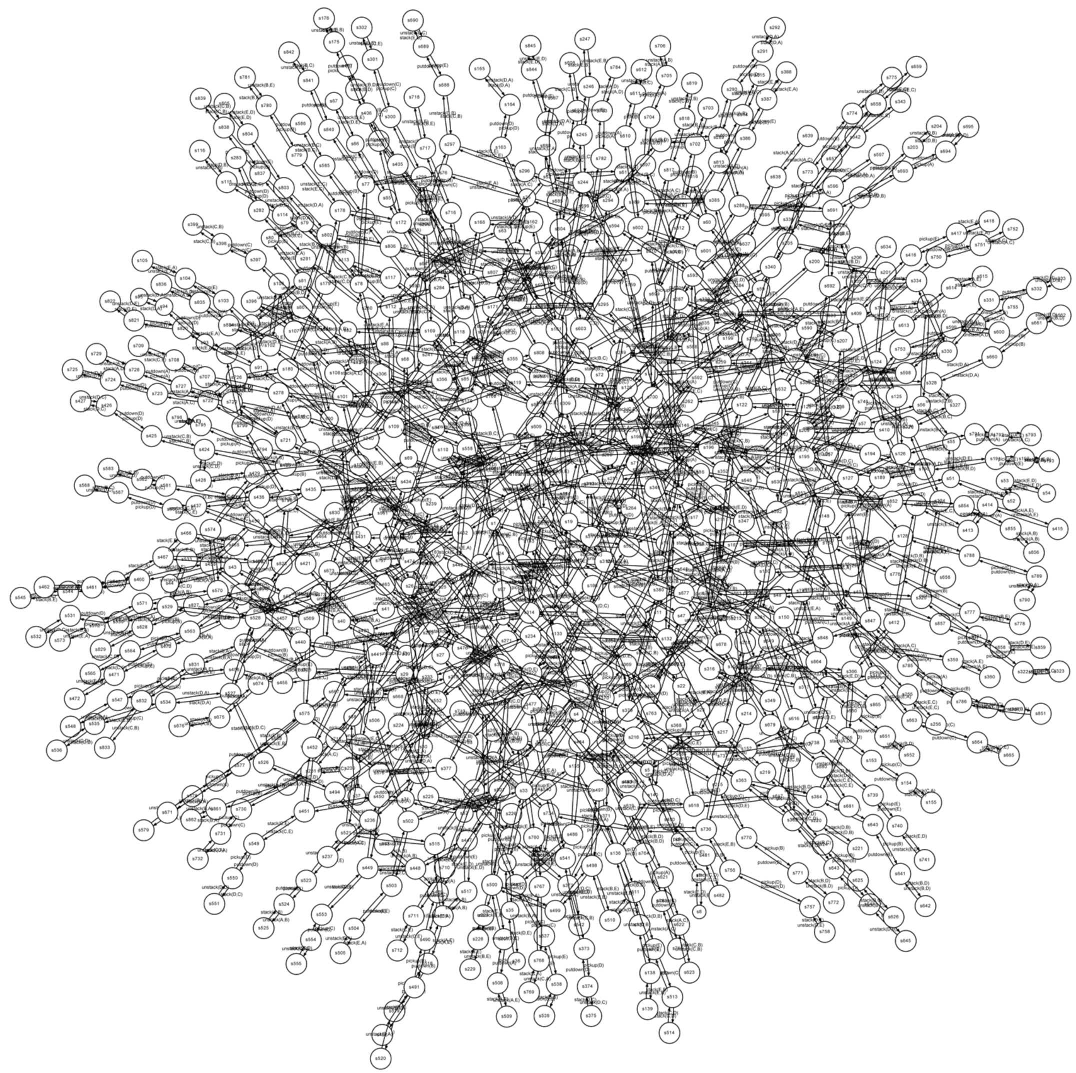


clear(A) $\wedge$ handempty
…

Landmark Heuristics (3)

Facts and formulas, not states! Why?

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



Not "we must reach the landmark state"!

Instead "we must reach *some state* that satisfies the fact/formula landmark"

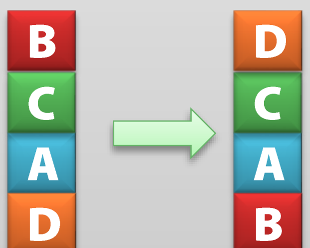
Landmark Heuristics (4)

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

Assume we are currently in state s…

<u>Fact Landmark for *s*</u>:

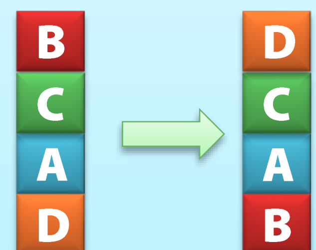
A **<u>fact</u>** that is not true in *s*,
but must be true at some point
in *every* solution starting in *s*



clear(A)
holding(C)
…

<u>Action Landmark for *s*</u>:

An **<u>action</u>** that must be used
in *every* solution starting in *s*



unstack(B,C)
putdown(B)
stack(D,C)
…but *not* putdown(C)! (Why?)

…so the effects of action landmarks are *fact landmarks*, and so are their *preconds*

(except those facts that are already true in *s*)

Landmark Heuristics (5)

- Generalization:
  - **<u>Disjunctive</u>** action landmark $\{a_1, a_2, a_3\}$ for state s
    - Every solution starting in state s and reaching a goal must use *at least one* of these actions
  - **<u>From action to fact</u>**:
    - Every fact in $(\bigcap\{\text{eff}(a) | a \in \text{landmark}\} - s)$ is a fact landmark for s
  - **<u>From fact to action</u>**:
    - If p is a fact landmark, then $\{a \in A \mid p \in \text{eff}(a)\}$ is a disjunctive action landmark for s
    - Not necessarily minimal: Some of the actions may not be required (removing an action can still result in a disjunctive A.L.)

Finding Landmarks:
A (Too) General Technique

Finding Landmarks: General Technique

- One general technique for **discovering landmarks**:

Current planning problem, P

Initial state does not include atom A

Modified planning problem, P'

Removed all actions that add atom A

If this problem (P') is **unsolvable**…

Test:
Delete relaxation of P' is unsolvable,
or $h_m(s_0) = \infty$, or …
➔ P' is unsolvable

…then **every** solution to P must use one of the removed actions

➔ **Action set is a disj. act. landmark**
➔ **Atom A is a fact landmark**

Unsolvable when removing a set of actions
➔ some action in the set must be used ➔ disjunctive action landmark!

Finding Landmarks: General Technique (2)

- This technique is very general
  - Applicable to *any* planning problem, *any* atom

- General techniques tend to be **widely applicable** but **slow**…

Verifying Landmarks (1)

- How difficult is it to **verify** that an action is an **action landmark**, in the **general** case?
  - Suppose we **can** verify this
  - Then given any STRIPS problem *P*, we can **determine if it has a solution**:
    - Add a new action:
      - **cheat**

        :precond true

        :effects goal-formula
    - If **cheat** is an action landmark, then it is *needed* in order to solve the problem

      ➔ the original problem was *unsolvable*
  - ➔ As difficult as solving the planning problem (PSPACE-complete)

Porteous et al (2001): On the Extraction, Ordering, and Usage of Landmarks in Planning

Verifying Landmarks (2)

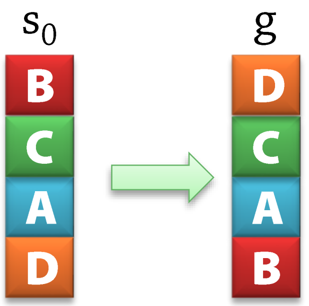
- How difficult is it to **verify** that a fact is a **fact landmark**, in the **general** case?
  - Suppose we **can** verify this
  - Then given any STRIPS problem *P*, we can **determine if it has a solution**:
    - Add a new fact:
      - **cheated** (false in the initial state)
    - Add new action:
      - **cheat**
        :precond true :effects
        (and **cheated** goal-formula)
    - If **cheated** is a fact landmark,
      then **cheat** was necessary ➔ the original problem was unsolvable
  - ➔ Again , as difficult as solving the planning problem

But of course there are special cases…

Finding Landmarks: Efficiently

Means-Ends Analysis (1)

- Discover landmarks using **means-ends analysis**



The goals are (obviously) fact landmarks,
except those true in the current state:
clear(D), on(D,C), on(A,B), ontable(B)

on(D,C) is a landmark,
on(D,C) is not true in the current state (s0)
➔ we must *cause* **on(D,C)** with an action

All actions causing on(D,C) require holding(D),
which is not true in the current state
➔ **holding(D) is a landmark**!

Actions causing holding(D) require handempty,
but handempty is true in the current state
➔ **handempty is not necessarily a landmark**

Means-Ends Analysis (2)

- **Formally**:
  - Discovering landmarks through means-ends analysis
  - // All unachieved goal facts are fact landmarks
    fact-landmarks ← g - currentstate
    do {
      for each p in landmarks {
        achievers ← $\{a \in A \mid p \in \mathrm{eff}(a)\}$
        candidates ← $\bigcap_{a \in achievers} \mathrm{pre}(a)$
        fact-landmarks ← fact-landmarks ∪ (candidates – currentstate)
      }
    } **until** no more fact-landmarks found

Add those facts
that are preconditions of *all* actions
achieving the known landmark p
and that are not true
in the current state

Means-Ends Analysis (3)

- **Weakness** of means-ends analysis:
  - Suppose the goal is {A, B, C, D}, initial state is {}
  - Processing for landmark A:

Action A1 effect A precond X, Y, Z	Action A2 effect A precond V, W, X	$\{X, Y, Z\} \cap \{V, W, X\} = \{X\}$ ➔ **Discover landmark X**

  - Processing for landmark X:

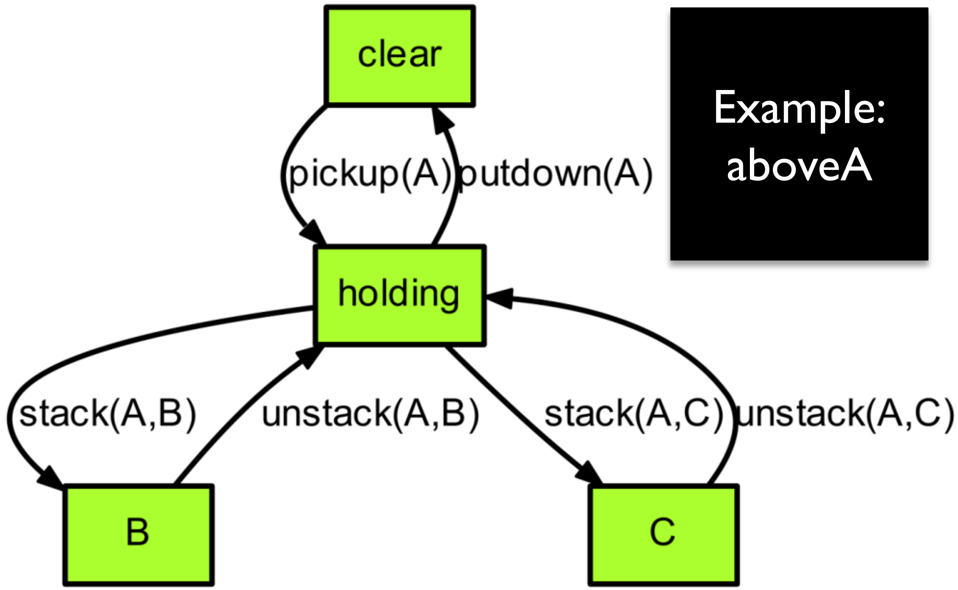
Action A3 effect X precond P, Q	Action A4 effect X precond R, S	$\{P, Q\} \cap \{R, S\} = \emptyset$ ➔ **No landmarks, "stop" processing**

Maybe all actions achieving P require Z,
and all actions achieving R also require Z
Weakness: We do **not** check this! Why?

Checking interactions across branches ➔ full backward-chaining
➔ complexity as in full plan generation…

Domain Transition Graphs (1)

- General concept: **domain transition graphs**
  - Assume we use a **state variable representation**
    - Each variable has a **domain**, a set of possible values
    - aboveA ∈ { clear, B, C, holding }
    - aboveB ∈ { clear, A, C, holding }
    - aboveC ∈ { clear, A, B, holding }
    - posA ∈ { on-table, other }
    - posB ∈ { on-table, other }
    - posC ∈ { on-table, other }
    - hand ∈ { empty, full }
  - For each state variable:
    - Add a node for each value
    - Add an edge
      for each action changing the value

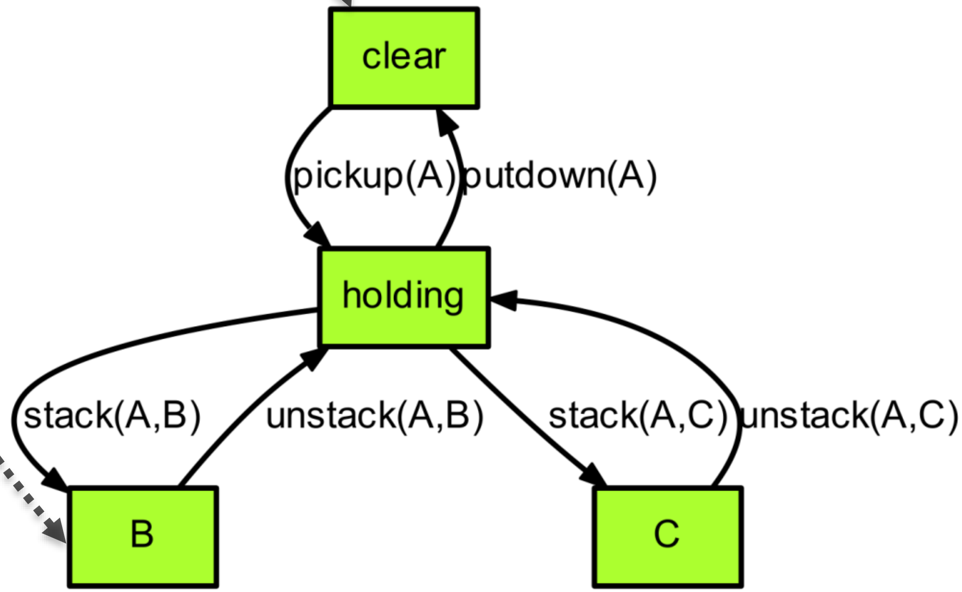


Example:
aboveA

Landmarks from DTGs

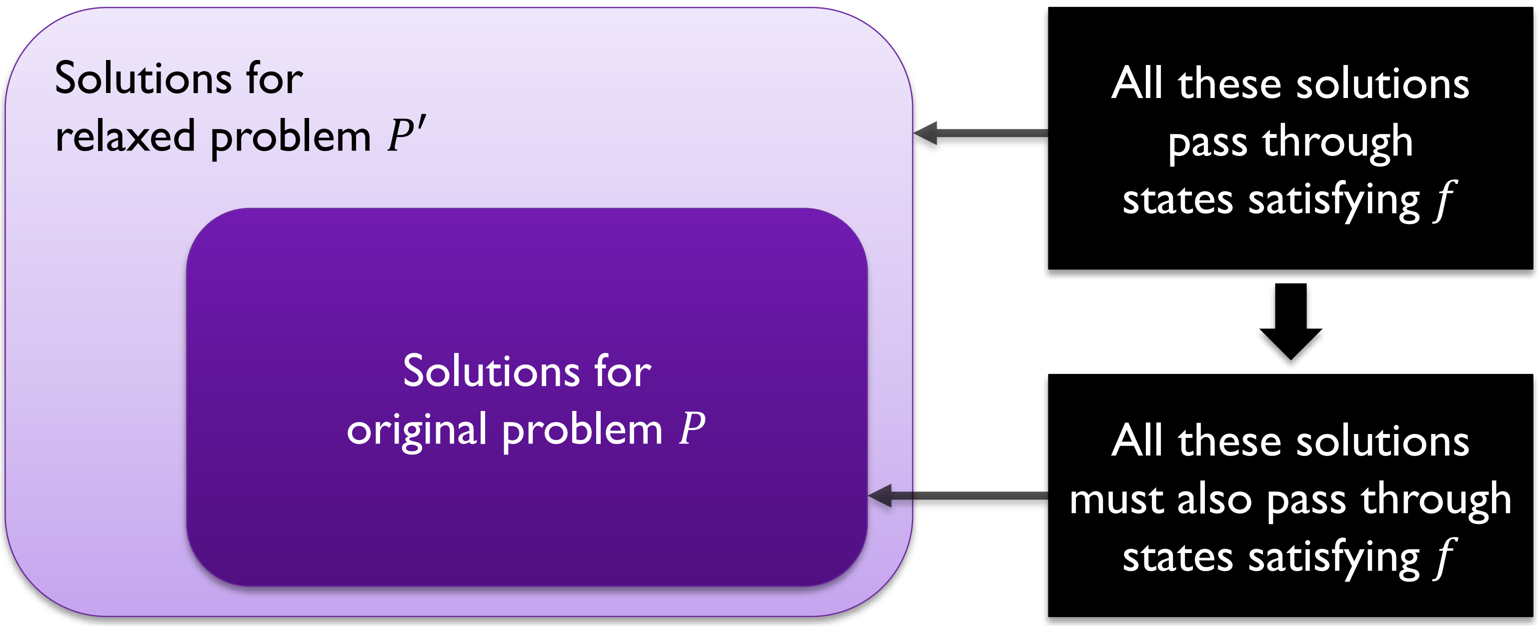
- Suppose:
  - In the current state, aboveA = clear
  - In the goal, aboveA = B

- Then aboveA=holding is a landmark



Landmarks and Relaxation

- Assume a problem P, and a **relaxed problem** P'
  - Suppose f is a fact landmark for a P'



- Then f is a fact landmark for the original problem as well!
- Similarly for action landmarks, etc.

Landmarks

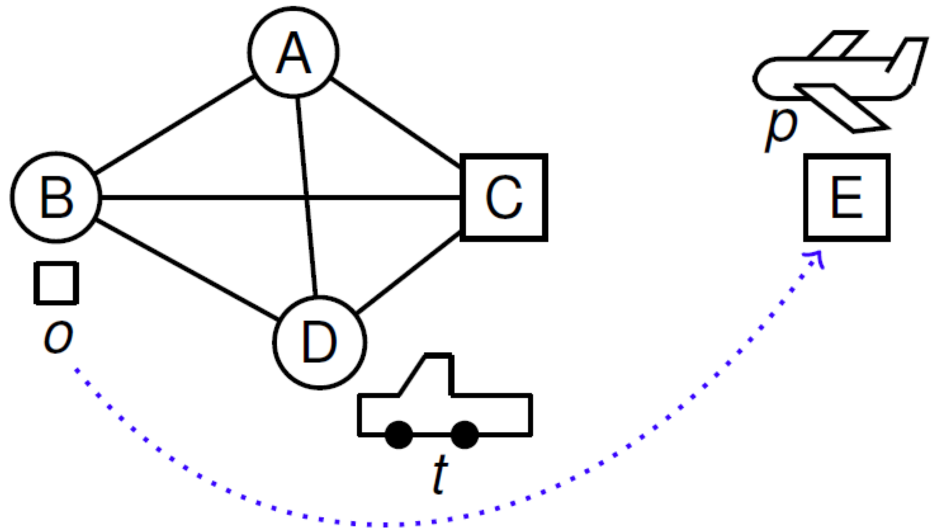
- Many other techniques exist…
  - Beyond the scope of the course

- Also, can sometimes find or approximate **necessary orderings**
  - We must achieve holding(A), *then* holding(B)

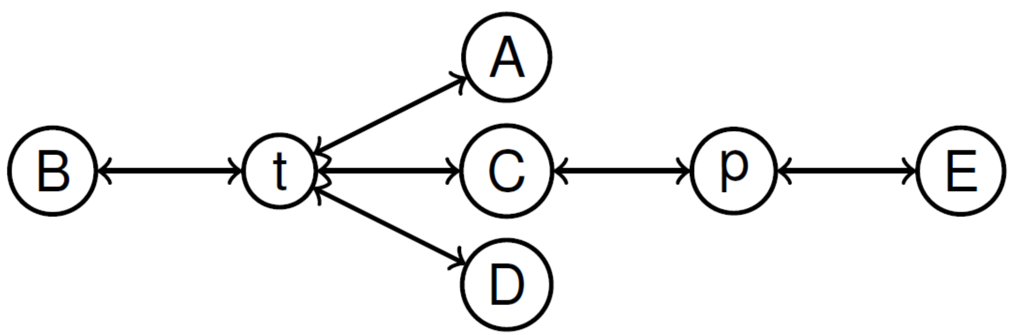
Using Landmarks as Subgoals

Example

- Example Problem:
  - Truck t transports object o within road network A/B/C/D
  - Airplane p transports object between airports C/E
  - Goal: Object at E

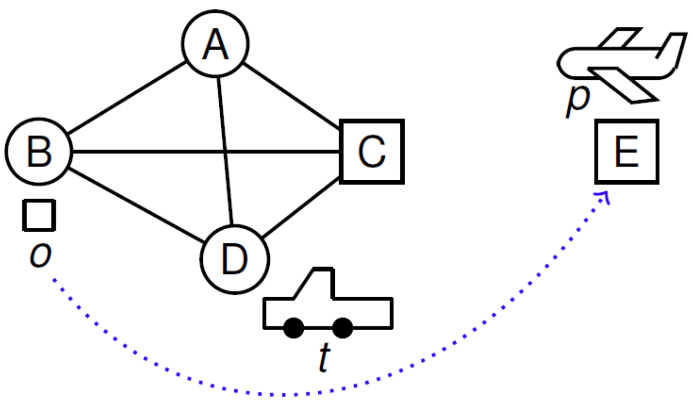


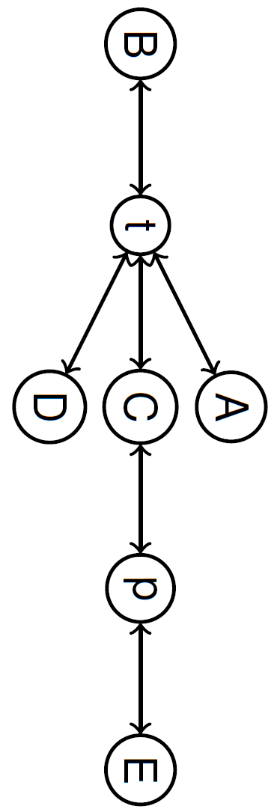
- Domain transition graph for location of object:

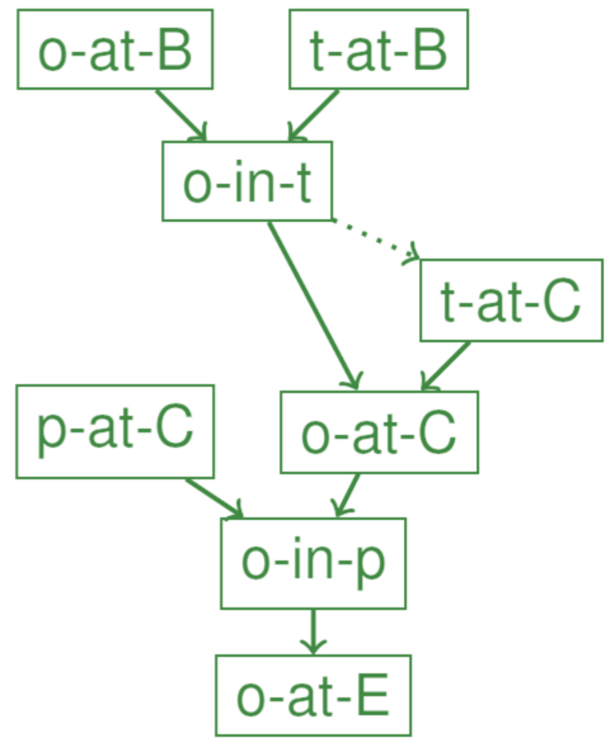


Landmarks as Subgoals (1)

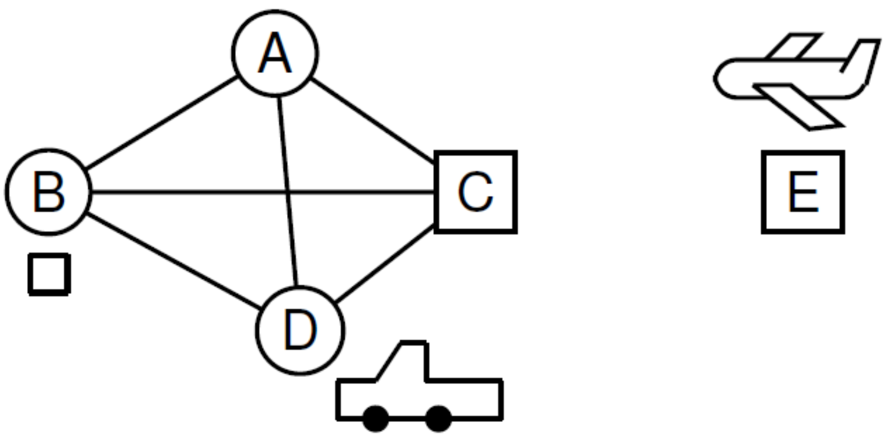
- Use of landmarks:
  - As **subgoals**: Try to achieve each landmark **in succession**, using inferred landmark **orderings**
    - Example from *Karpas & Richter: Landmarks – Definitions, Discovery Methods and Uses*



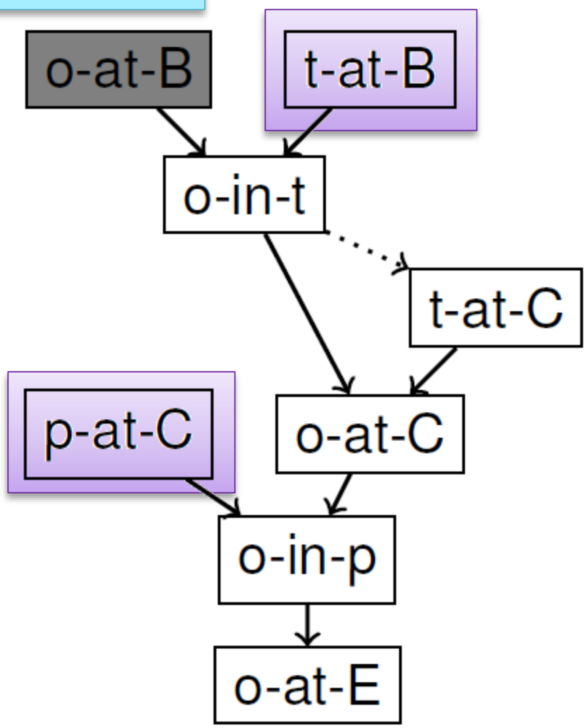




Landmarks as Subgoals (2)



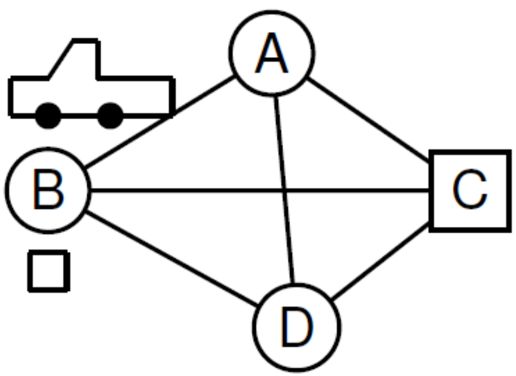
Already true when we start

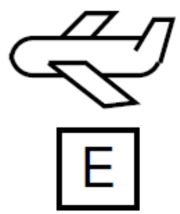


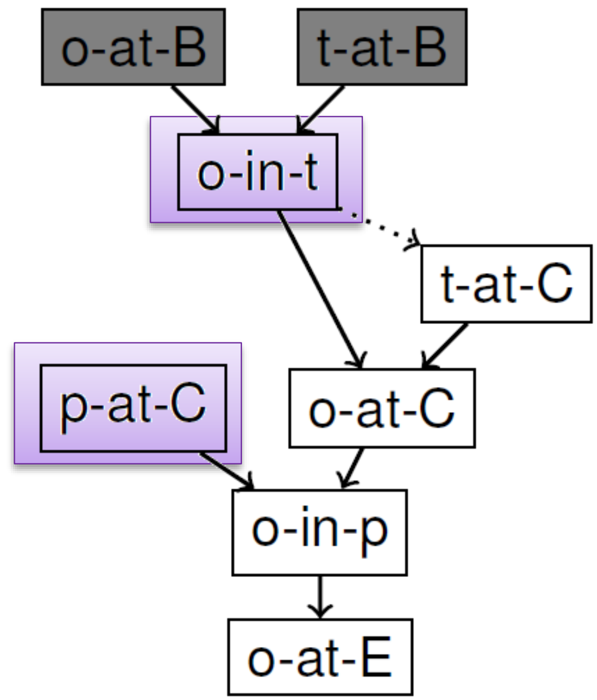
Two landmarks could be "first" (all predecessors achieved)
Current goal: t-at-B ∨ p-at-C (disjunctive!)

Landmarks as Subgoals (3)

Suppose we begin by achieving t-at-B:
Simple planning problem,
results in a single action -- drive(t, B)



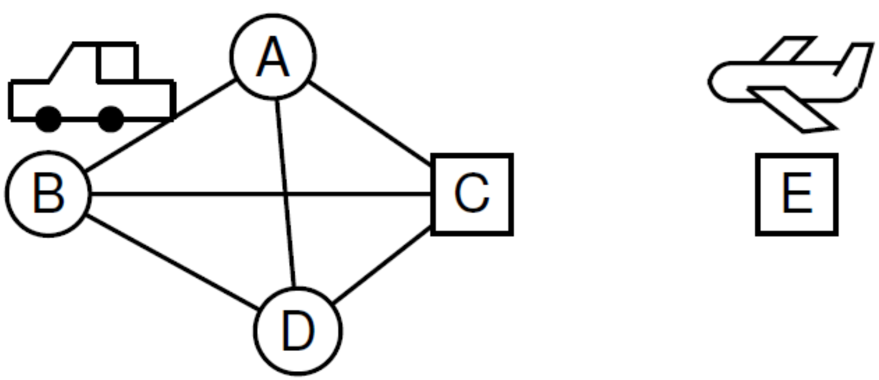


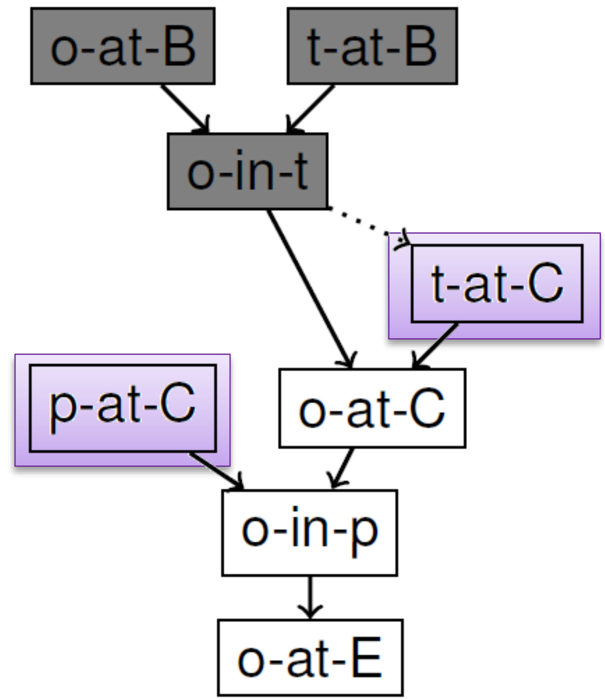


Current goal: o-in-T *or* p-at-C

Landmarks as Subgoals (4)

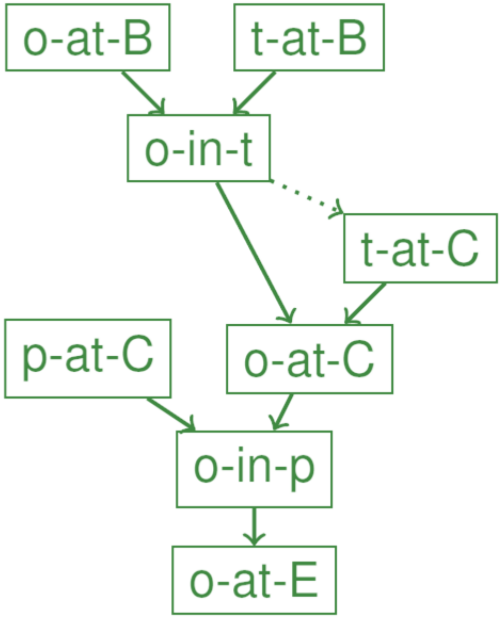
Suppose we continue by achieving o-in-T:
Simple planning problem,
results in a single action -- load-truck(o,t,B)





Landmarks as Subgoals (5)

- Sometimes very helpful, but:
  - There are still *choices* to be made – backtrack points!
  - Optimizing for one **part** of the overall goal at a time:
    - Can't see the whole picture
    - Can miss opportunities:
      Cheapest solution *here* → more expensive solution *later*
    - Can be incomplete:
      Cheapest solution *here* → impossible to solve *later*

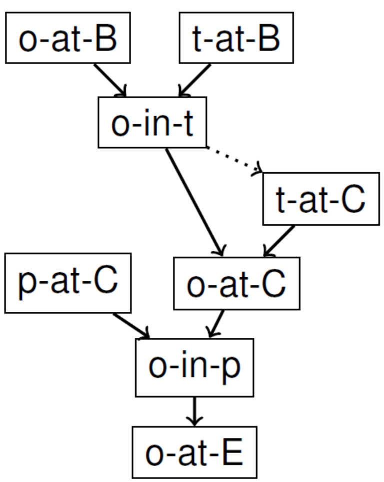


Landmark Counts and Costs

Landmarks for Heuristics: Intro

- Use of landmarks:
  - As a basis for **non-admissible heuristic estimates** in standard forward state space search
  - Pioneered by LAMA, which is:
    - The winner of the *sequential satisficing* track of the 2008/2011 competitions
  - If LAMA-2011 had participated in IPC-2014 (the latest competition):
    - Would have been 12th of 21 planners
  - But LAMA is *part* of the following planners from the 2014 competition:
    - IBaCoP2, 1st place in the sequential satisficing track
    - IBaCoP, 2nd place in the sequential satisficing track
    - ArvandHerd, 1st place in the sequential multi-core track
    - IBaCoP, 2nd place in the sequential multi-core track

- 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

Not admissible: One action may achieve multiple landmarks!

Landmark Counts and Costs (2)

- The **LAMA heuristic** combines:
  - The **number** of landmarks still to be achieved in a state
  - FF heuristics (relaxed planning graph)

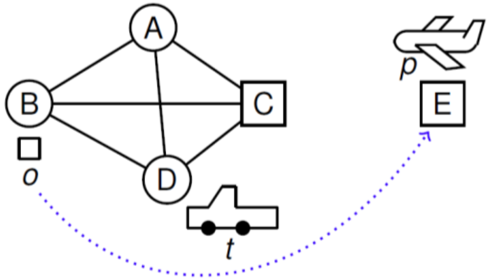
  - Searches for **low-cost plans**
    - But we also want to find plans quickly!
    - Heuristics estimate both:
      - Cost of *actions* required to reach the goal
      - Cost of the *search effort* required to reach the goal

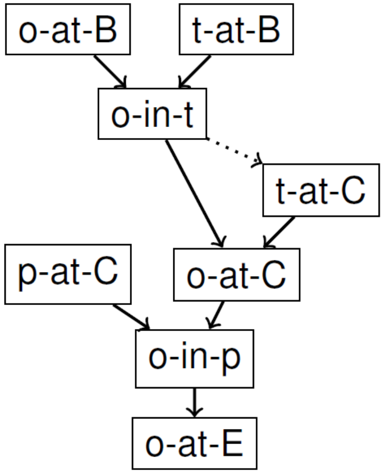
  - **Search strategy**:
    - First, **greedy best-first** (create a solution as quickly as possible)
    - Then, **repeated weighted A*** search with decreasing weights

  - Iteratively improve the plan – **anytime planning**!

Landmark Counts and Costs (3)

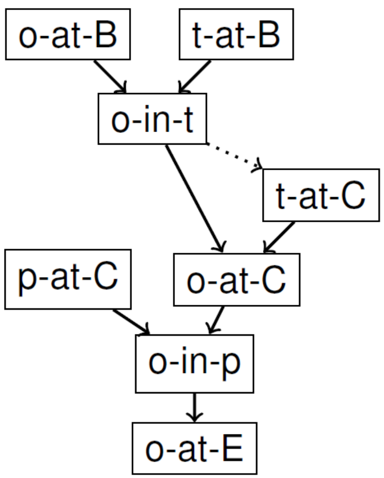
- Other uses of landmarks:
  - As a basis for **admissible heuristic estimates**
  - Idea: The cost of each action is *divided* across the landmarks it achieves
  - **Simplified example**:
    - Suppose there is a goto-and-pickup action of cost 10, that achieves both t-at-B and o-in-t
    - Suppose *no other action* can achieve these landmarks
    - One can then let (for example) cost(t-at-B)=3 and cost(o-in-t)=7
  - The sum of the cost of remaining landmarks is then an **admissible heuristic**
    - Must decide how to split costs across landmarks
    - Optimal split *can* be computed polynomially, but is still expensive

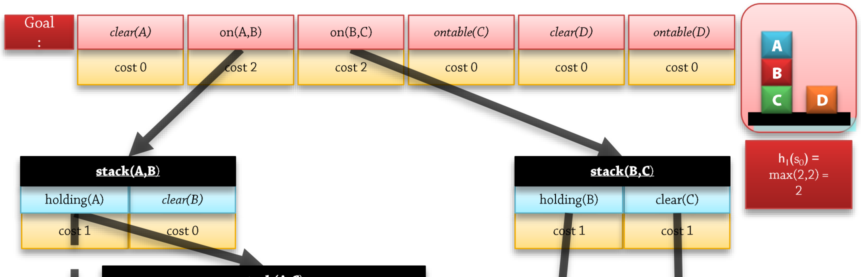




Landmarks: Modified Problem

- Landmarks as a basis for a **modified planning problem**
  - Add new predicates "achieved-landmark-*n*"
    - Concretely: *object-has-been-in-plane*
  - An action achieving a landmark makes the corresponding predicate true
    - (load object plane) ➔ object-has-been-in-plane := true
  - The goal requires all such predicates to be true
    - (**:goal** object-has-been-in-plane …)
  - ➔ Any *other* heuristic can be applied to the modified problem!
    - $h_1(s)$: What is the cost of achieving object-has-been-in-plane?



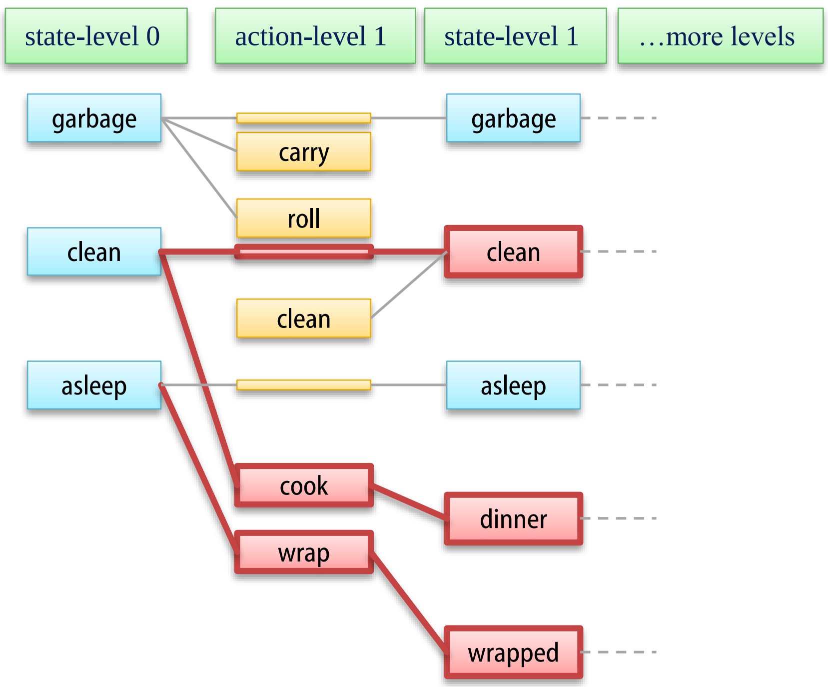


Search Techniques

Dual Queue Techniques

Helpful Actions and Completeness

- Recall FF's **helpful actions**
  - ≈ Actions chosen in the first level of the relaxed planning graph when computing the heuristic



- FF uses these to prune the tree in Enforced Hill Climbing
  - Leads to **incompleteness**
  - May search for a long time, exhaust the search space, **then** start over using complete search

- "Helpful actions" are **more likely** to be helpful
  - But skipping the other actions **completely** is too strict!
  - Fast Downward: **Prioritize** helpful actions ("preferred successors")

Dual Queues

- **Dual queues** (”open lists”):
  One for **ordinary** successors, one for **preferred** successors

  - In each expansion step:
    - Pick the **best** action from the **preferred** queue
      - Expand it (create successors); place each successor in the appropriate queue
    - Pick the **best** action from the **non-preferred** queue
      - Expand it (create successors); and place each successor in the appropriate queue
  - Fewer preferred successors than non-preferred
    - Takes less time to reach a node in the preferred queue ➔ we *prefer* these
  - If we ”misclassified” an action as non-helpful:
    - We don’t have to exhaust the ”preferred part” of the search space before we can ”recover”

Boosted Dual Queues

- **Boosted** Dual Queues:
  - Used in later versions of Fast Downward and LAMA
  - Whenever **progress** is made (better h-value reached):
    - Expand 1000 *preferred* successors
  - If progress is made again within these 1000 successors:
    - Add another 1000, accumulating
    - (Progress made after 300 ➔ keep expanding 1700 more)
  - After reaching the preferred successor limit:
    - Expand a node from the non-preferred queue
  - Still complete
    - More aggressive than ordinary dual queues
    - Less aggressive than pure pruning

Deferred Evaluation / Lazy Search

Deferred Evaluation

- Standard **best-first** search:
  - Remove the "best" (most promising) state from the priority queue
  - Check whether it satisfies the goal
  - Generate all successors
  - Calculate their heuristic values
  - Place in priority queue (”open list”)

Typically takes most of the time

- Potentially faster: **Deferred Evaluation** (Fast Downward, …)
  - Remove the "best" state from the priority queue
  - Check whether it satisfies the goal
  - Calculate **its** heuristic value (only one!)
  - Generate all successors
  - Place in priority queue using the **parent's** heuristic value

Takes less time, but less accurate heuristic – "one step behind"
Often faster but lower-quality plans

Parameter Optimization and Portfolio Planners

A general technique – not limited to state-space search!

Parameter Optimization (1)

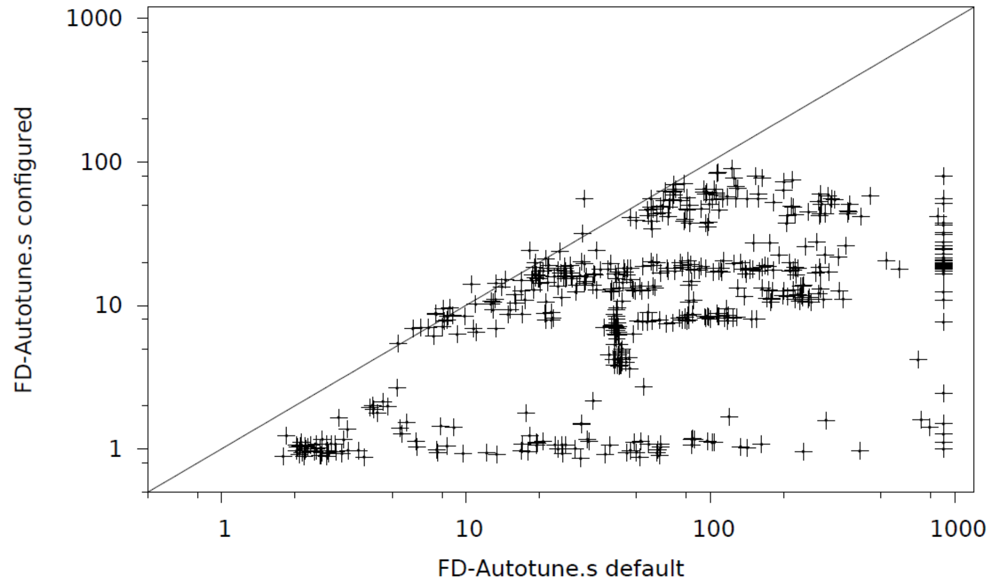
- Some planners have **<u>many parameters</u>** to tweak
  - In early planning competitions, domains were **<u>known in advance</u>**
    - Participants could manually adapt their "domain-independent" planners…
  - Somewhat **<u>exaggerated quote</u>** from IPC-2008 results:
    - if domain name begins with "PS" and part after first letter is "SR":
      use algorithm 100
    - else if there are 5 actions, all with 3 args, and 12 non-ground predicates:
      use algorithm −1000
    - else if all predicates ground and 10th/11th domain name letters "PA":
      use algorithm −1004
    - else if there are 11 actions and action name lengths range from 5 to 28:
      use algorithm 107
  - From 2008, this was no longer allowed
    - Planners were handed in
    - Then the **<u>organizers</u>** ran the planners – also on previously unknown domains

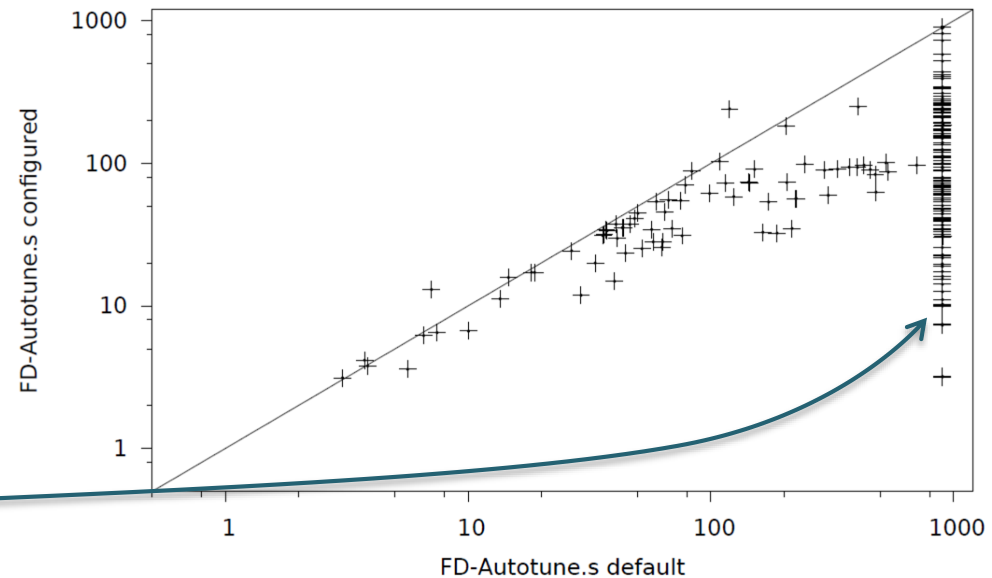
Parameter Optimization (2)

- How about *automatically* learning parameters?
  - One specific form of learning in planning – others exist
  - Experimental application to **Fast Downward**
    - Optimization for speed: 45 params, $2.99 * 10^{13}$ possible configurations
    - Optimization for quality: 77 params, $1.94 * 10^{26}$ possible configurations
  - Example parameters:
    - **Heuristics used**:
      $h_{max} = h_0$, h_m, h_{add}, h_{FF}, h_{LM} (landmarks), h_{LA} (admissible landmarks), goal count, …
    - Method used to **combine heuristics**: Max, sum, selective max (learns which heuristic to use per state), tie-breaking, Pareto-optimal, alternation
    - **Preferred operators** used or not, for each heuristic
      - Like FF's helpful actions, but used for *prioritization*, not pruning
    - **Search strategy** combinations: Eager best-first, lazy best-first, EHC
    - …
  - Parameter learning framework **ParamILS** used

Parameter Optimization (3): Results

- **Under** the diagonal = **faster** than default configuration
  - For 540 small **training instances**:
    - Very good results
    - To be expected – parameters tuned for these specific problems!
  - For 270 larger **test instances**:
    - From the same domains
    - Performance still improves

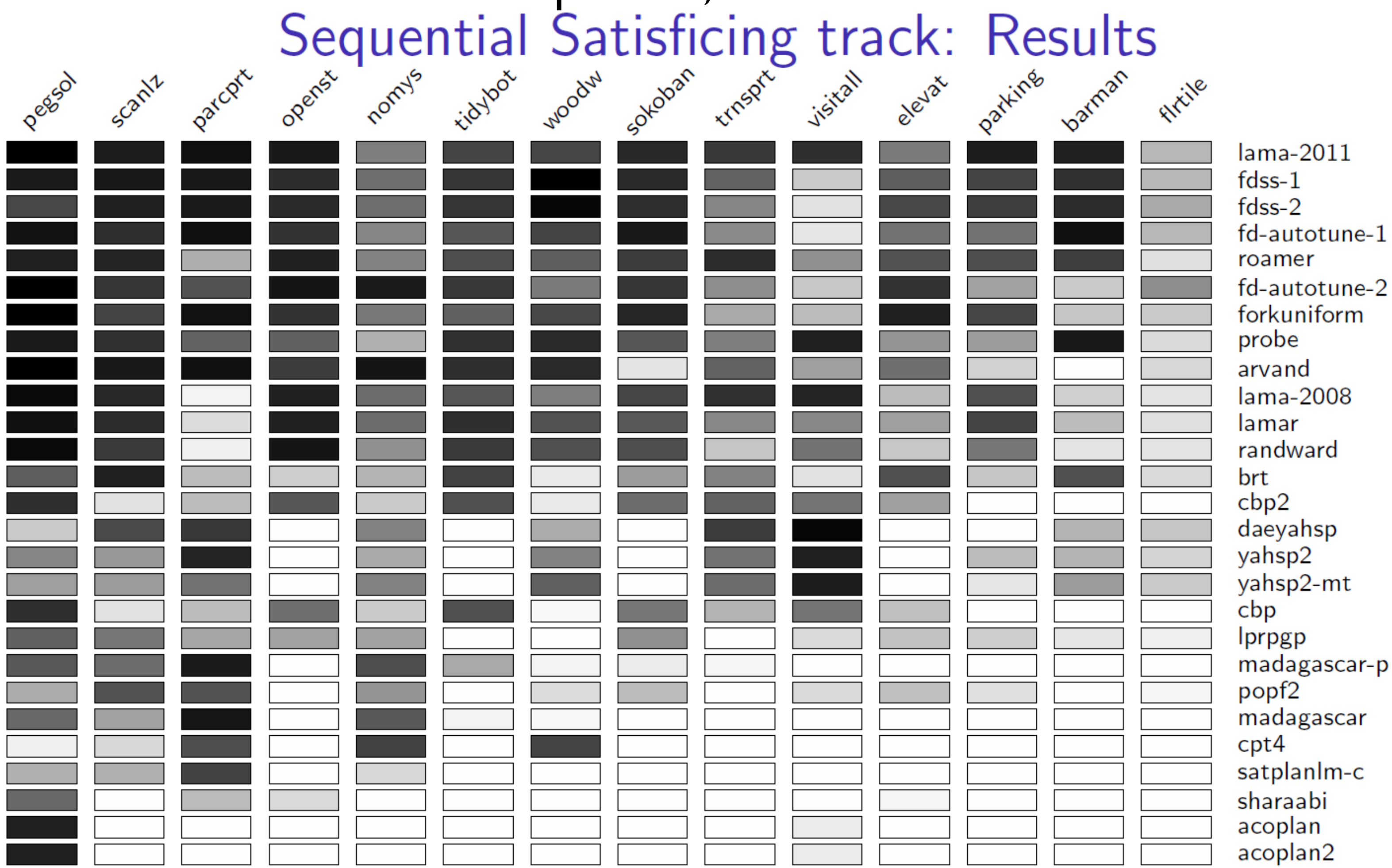




Unsolvable in 900 seconds by the default configuration

Parameter Optimization (4): Results

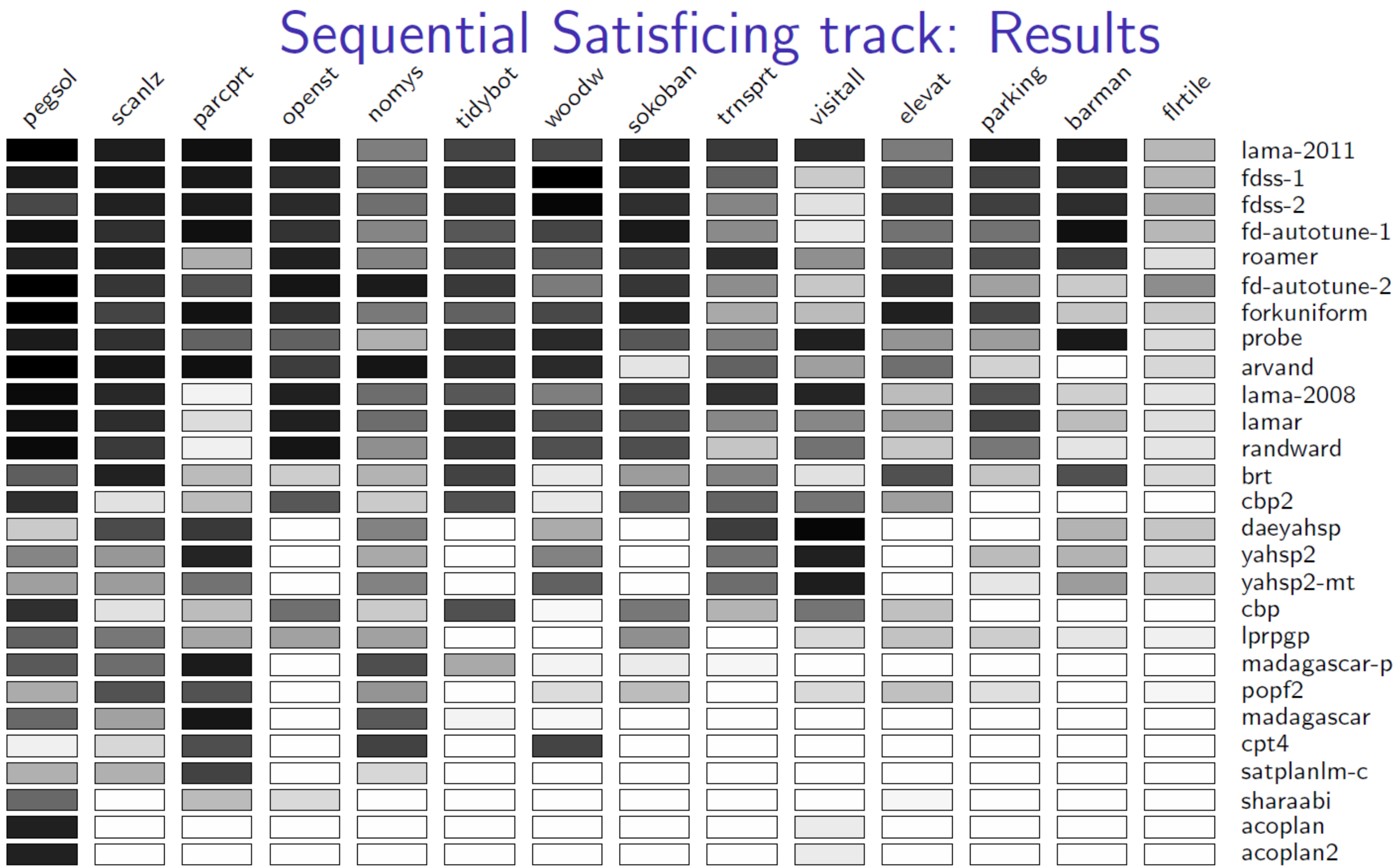
- Results from the **satisficing** track of IPC-2011
  - Two versions of FD-autotune competed, adapted to *older* domains
  - Some were reused in this competition, most were new



Darker = better!

Portfolio Planning (1)

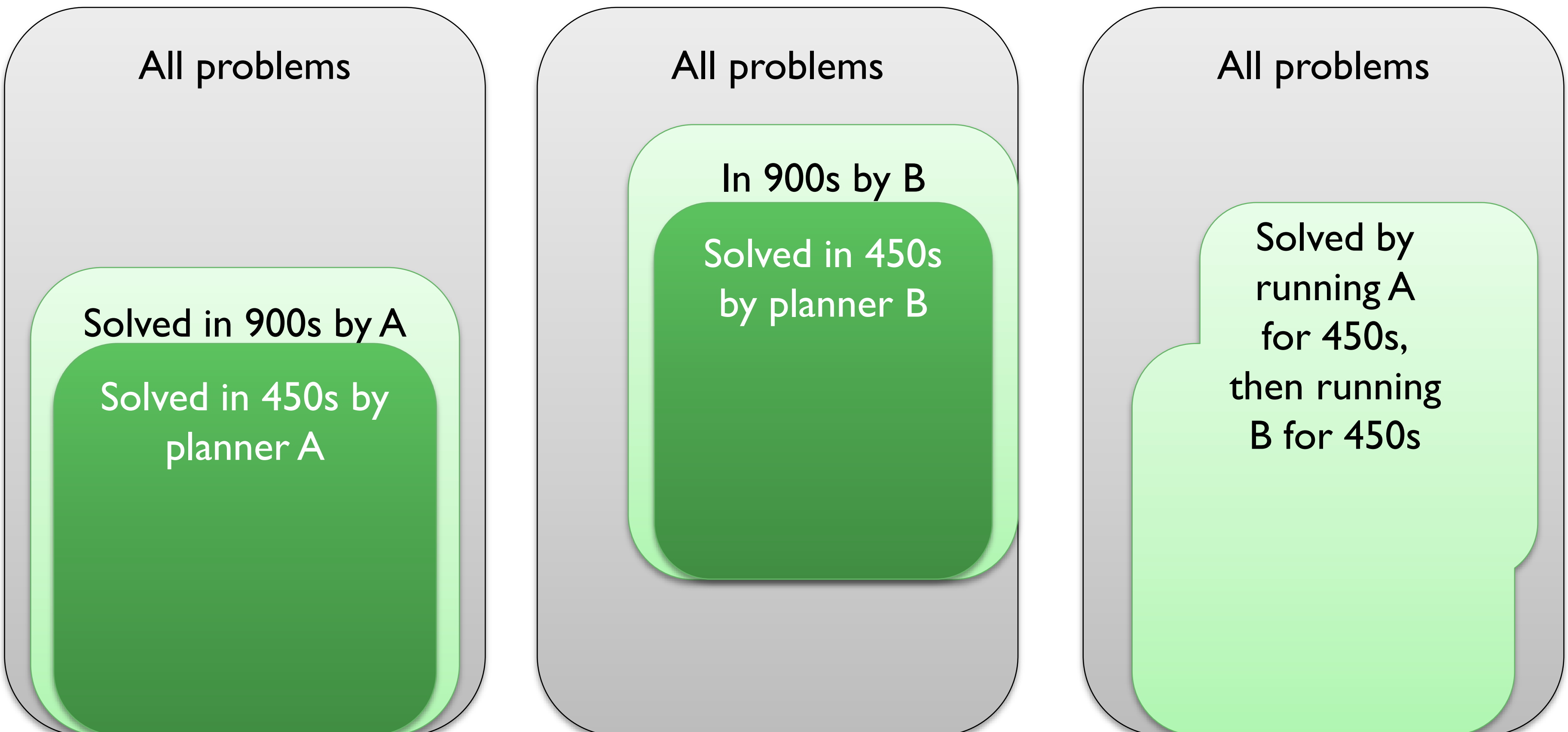
- Observation:
  - Different planners seem good in different domains!



Darker = better!

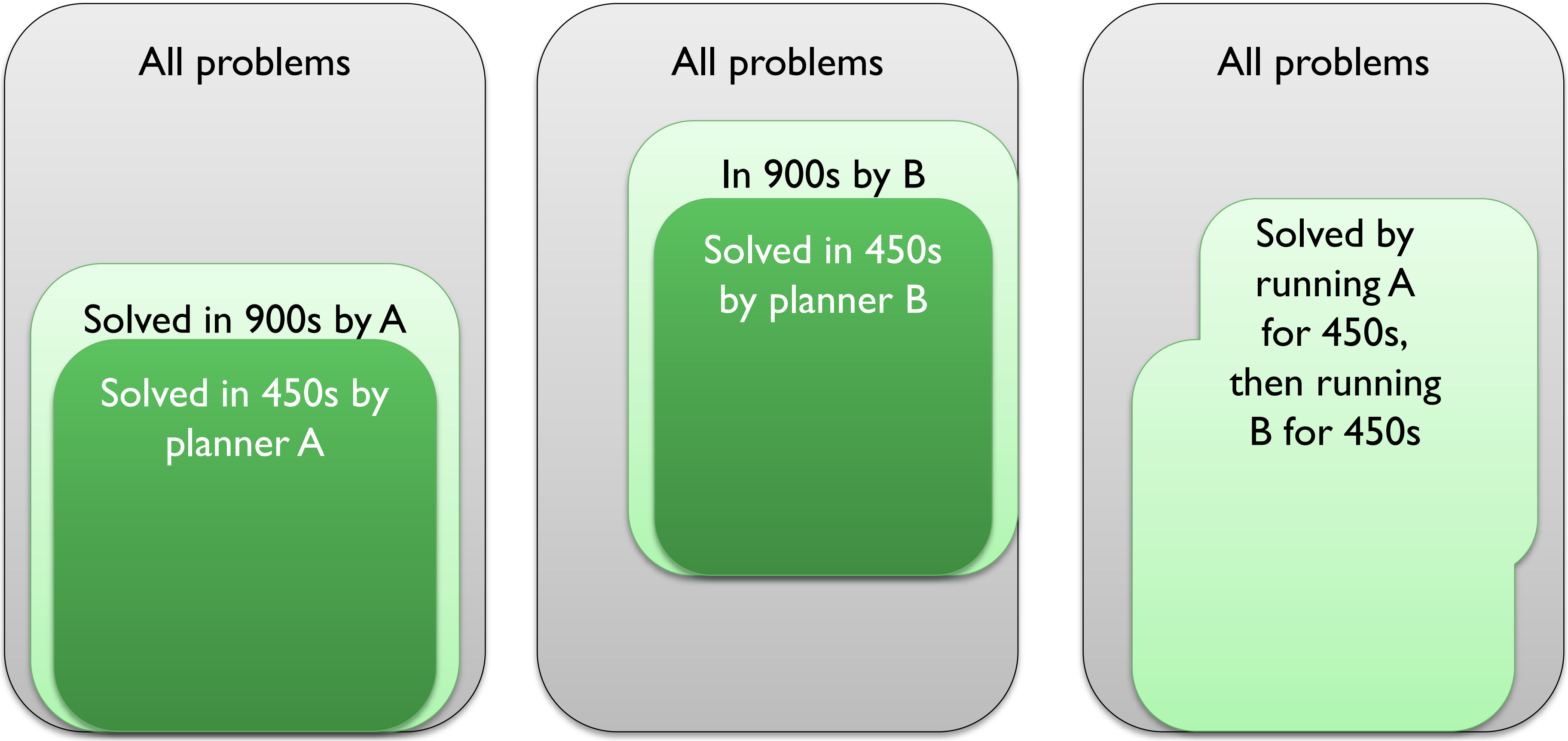
Portfolio Planning (2)

- Further analysis would show:
  - Even if two planners solve equally many problems in one domain, they may solve **<u>different</u>** problems
  - Also, planners often return plans **<u>quickly</u>** or **<u>not at all</u>**



Portfolio Planning (3)

- The competition has a fixed time limit
  - Can benefit from splitting this across **multiple algorithms**!
  - ➔ **Portfolio** planning



Portfolio Planning (4)

- Fast Downward Stone Soup: **<u>Learning</u>**
  - Which configurations to use
  - How much time to assign to each one
  - Given test examples from older domains

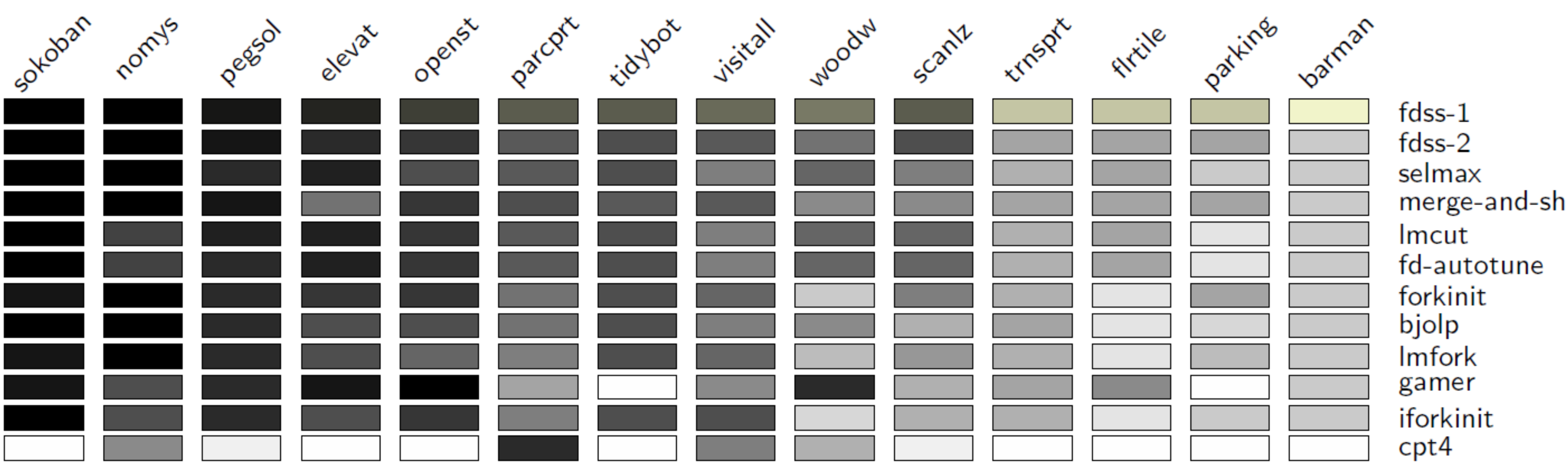
Algorithm	Score	Time	Marginal
BJOLP	605	455	46
RHW landmarks	597	0	—
LM-cut	593	569	26
h^1 landmarks	588	0	—
M&S-bisim 1	447	175	8
$h^{\max}$	427	0	—
M&S-bisim 2	426	432	20
blind	393	0	—
M&S-LFPA 10000	316	0	—
M&S-LFPA 50000	299	0	—
M&S-LFPA 100000	286	0	—
Portfolio	654	1631	
"Holy Grail"	673		

Configurations learned for sequential optimal planning

Portfolio Planning (5)

- Results from IPC-2011:

Sequential Optimization track: Results



Portfolio Planning (6)

- Results from IPC-2014:
  - Sequential Satisficing Track
    - **#1: IBaCoP -- portfolio planner, 12 planners, 150 seconds per planner**
      - ARVAND (Nakhost, Valenzano, and Xie 2011)
      - FD-AUTOTUNE 1 & 2 (Fawcett et al. 2011)
      - FD STONE SOUP (FDSS) 1 & 2 (Helmert et al. 2011)
      - LAMA 2008 & 2011 (Richter, Westphal, and Helmert 2011)
      - PROBE (Lipovetzky and Geffner 2011)
      - MADAGASCAR (Rintanen 2011)
      - RANDWARD (Olsen and Bryce 2011)
      - YAHSP2-MT (Vidal 2011)
      - LPG-TD (Gerevini et al. 2004)
    - **#2: IBaCoP2 -- portfolio planner**
      - Before the competition: Extracted interesting properties of planning problems; used ML to learn which planners were most likely to solve them
      - At the competition: Used the learned model to classify new problems; applied the 5 planners that seemed most useful (360 seconds each)