Linköping University

Automated Planning

3. Planning as Search, Forward State Space Search

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

Planning as Search

Planning as Search

- Planning algorithms are often based on **search**

Search space
Classical planning: Finite number of search nodes

Initial search node 0
(contains some information, depending on the search space…)

We usually don't have all search nodes explicitly represented:
We start with a single **initial search node**

Child node 1

Child node 2

A **successor function** / **branching rule** returns all successors of any search node
➔ can build the graph *incrementally*

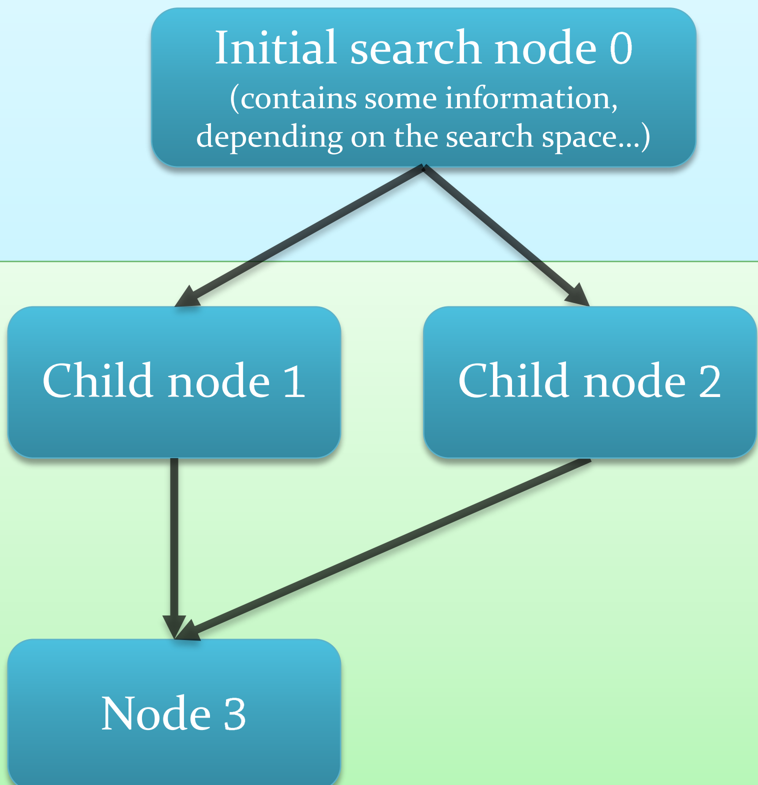
Expand a node = generate its successors

Now we have *multiple* unexpanded nodes!
A **search strategy** chooses which one to expand next

Planning as Search (2)

Search space

Classical planning: Finite number of search nodes



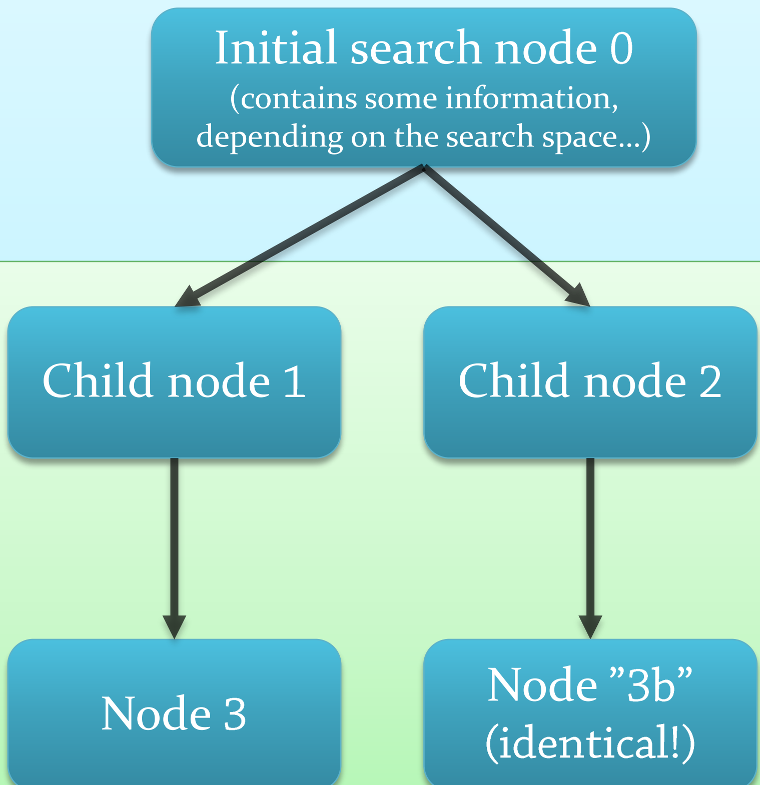
Two nodes might have the *same* successor!
Option 1: Keep track of all visited nodes, *detect* when the same successor is generated again

- ➔ Requires a lot of memory
- ➔ Only investigate a given node once, second time: backtrack
- ➔ The search space is a general *graph*

Planning as Search (3)

Search space

Classical planning: Finite number of search nodes



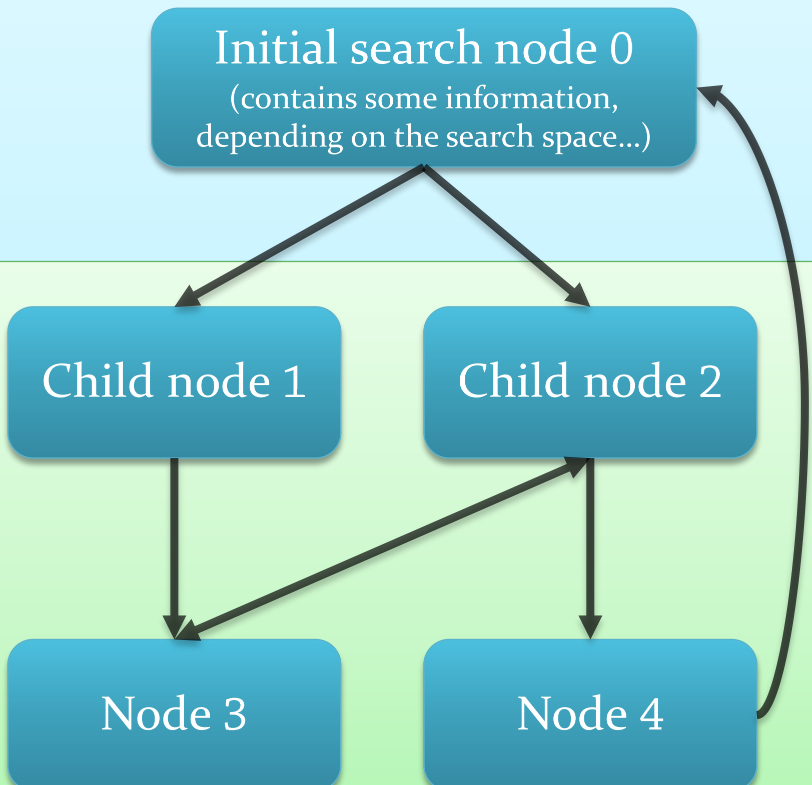
<u>Option 2</u>:
<u>Don't</u> keep track of visited nodes

- ➔ Saves memory
- ➔ Investigate some nodes multiple times
- ➔ The search space is a *tree*

Planning as Search (4)

Search space

Classical planning: Finite number of search nodes



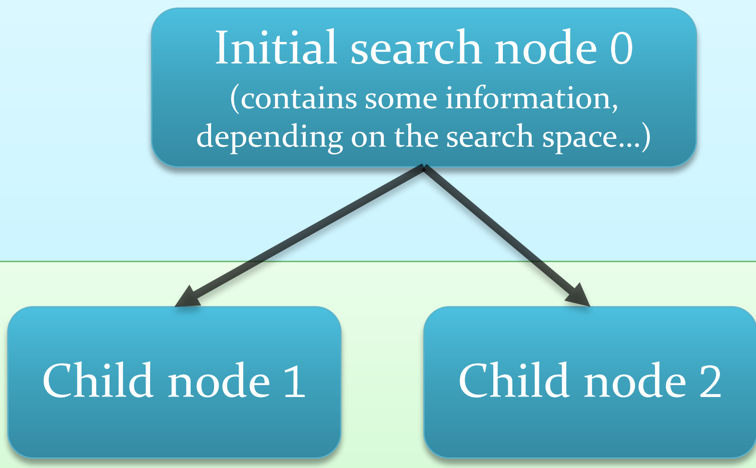
An **ancestor** may also be a **successor**
➔ **loops** in the search graph

Depending on the search algorithm,
it may or may not be necessary
to detect and handle this

Planning as Search (5)

Search space

Classical planning: Finite number of search nodes



<u>Additional requirements</u>:

- ➔ A *goal criterion*, detecting whether a node satisfies the goal
- ➔ A *"plan extractor"*, telling us which plan a goal node corresponds to

Planning as Search (6)

- General **Search-Based Planning Algorithm**:
  - **search**() {

```
search() {
    open ← { initial-node }
    while (open ≠ emptyset) {
        use a search strategy to select and remove node from open
        if goal-satisfied-by(node) then return path

        foreach mod ∈ possible-modifications-to(node) {
            node' ← apply(mod, node) // dynamically generate a successor
            add node' to open
        }
    }
    return failure;
}
```

Expand the node

Planning as Search (7)

- To keep track of visited nodes:
  - **search**() {

```
open  ← { initial-node }
added ← { initial-node }
while (open ≠ emptyset) {
    use a search strategy to select and remove node from open
    if goal-satisfied-by(node) then return path

    foreach mod ∈ possible-modifications-to(node) {
        node' ← apply(mod, node) // dynamically generate a successor
        if not (node' ∈ added)
            add node' to open
            add node' to added
    }
}
return failure;
}
```

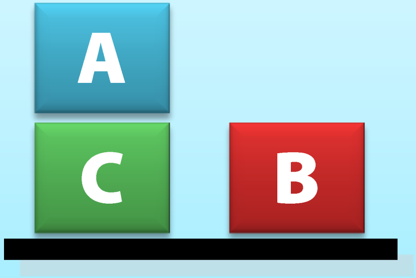
Forward State Space Search

Blocks World (1)

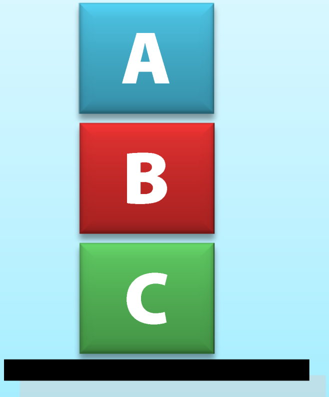
- Our next example domain: The **Blocks World**
  - A **simple** example domain
    allowing us to focus on algorithms and concepts, not domain details

You

Initial State

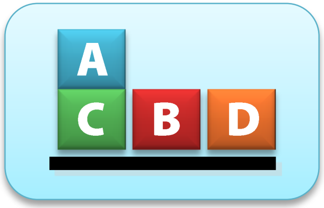


Your greatest desire



Blocks World (2)

- We will generate classical **sequential plans**
  - A common blocks world version, with **4 operators**
    - (**pickup** ?x) – takes ?x from the table
    - (**putdown** ?x) – puts ?x on the table
    - (**unstack** ?x ?y) – takes ?x from on top of ?y
    - (**stack** ?x ?y) – puts ?x on top of ?y
  - Predicates used:
    - (**on** ?x ?y) – block ?x is on block ?y
    - (**ontable** ?x) – ?x is on the table
    - (**clear** ?x) – we can place a block on top of ?x
    - (**holding** ?x) – the robot is holding block ?x
    - (**handempty**) – the robot is not holding any block



(not (exists (?y)
(on ?y ?x)))

(not (exists (?x)
(holding ?x)))

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

Blocks World (3): Operator Reference


```
(:action pickup
  :parameters (?x)
  :precondition (and (clear ?x) (on-table ?x)
                     (handempty))
  :effect
  (and (not (on-table ?x))
       (not (clear ?x))
       (not (handempty))
       (holding ?x)))
```

```
(:action putdown
  :parameters (?x)
  :precondition (holding ?x)

  :effect
  (and (on-table ?x)
       (clear ?x)
       (handempty)
       (not (holding ?x))))
```

```
(:action unstack
  :parameters (?top ?below)
  :precondition (and (on ?top ?below)
        (clear ?top) (handempty))
  :effect
  (and (holding ?top)
       (clear ?below)
       (not (clear ?top))
       (not (handempty))
       (not (on ?top ?below))))
```

```
(:action stack
  :parameters (?top ?below)
  :precondition (and (holding ?top)
                     (clear ?below))
  :effect
  (and (not (holding ?top))
       (not (clear ?below))
       (clear ?top)
       (handempty)
       (on ?top ?below)))
```

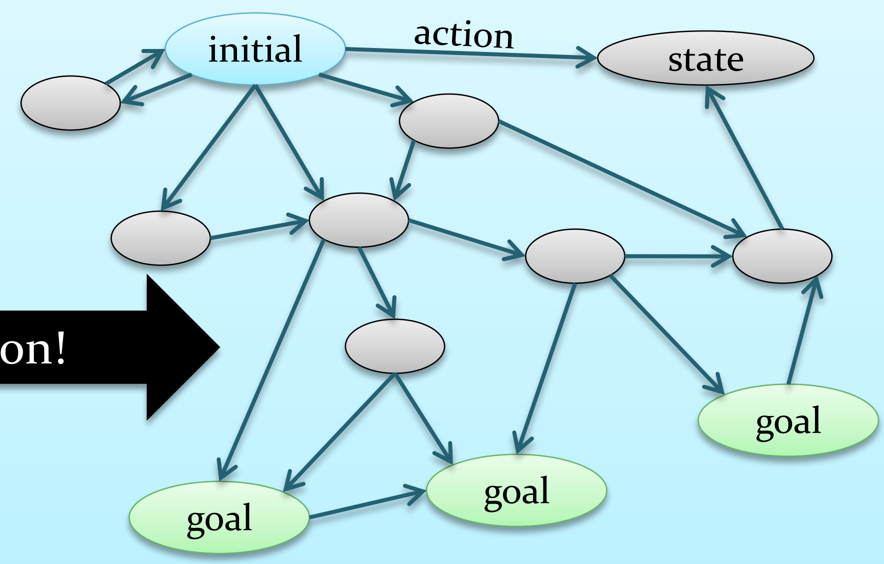
Representation and Model

Classical representation, *structured*

```
(define (domain bw)
  (:requirements :strips)
  (:predicates
    (on ?x ?y)       ;; x is on top of y
    (ontable ?x)     ;; x is on the table
    (clear ?x)       ;; nothing on x, not holding it
    (handempty);; not holding any block
    (holding ?x)) ;; holding block x
  (:action pickup
     :parameters (?x)
     :precondition (and (clear ?x)
                        (ontable ?x) (handempty))
     :effect …)
   …
)
(define (problem bw42)
    (:domain bw)
   …)
```

Simple translation!

Formal model: State transition system

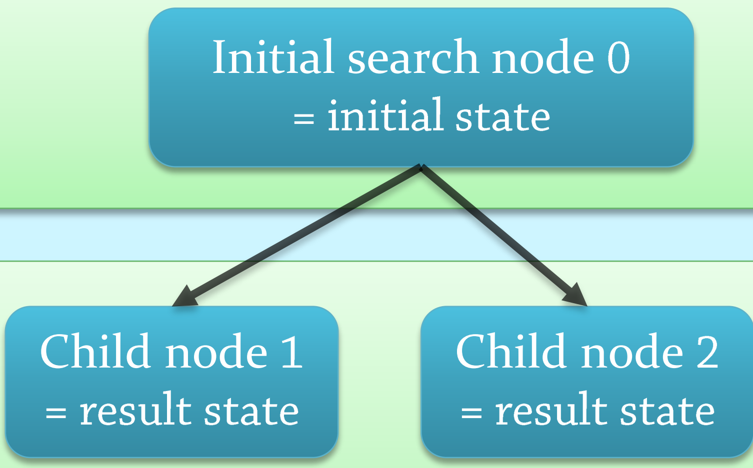


The model itself
is a possible search space!

Forward State Space

Forward State Space

Forward planning, forward-chaining, progression: Begin in the **initial state**



Corresponds directly to the initial state

Edges correspond to actions

The *successor function* / *branching rule*:

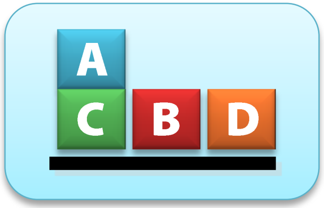
Given a state *s*,
generate **all states** that result from
applying an action that is applicable in s

Goal criterion: *The state of the node satisfies the goal formula*

Plan extraction: *Generate the sequence of all actions on the path to the goal node*

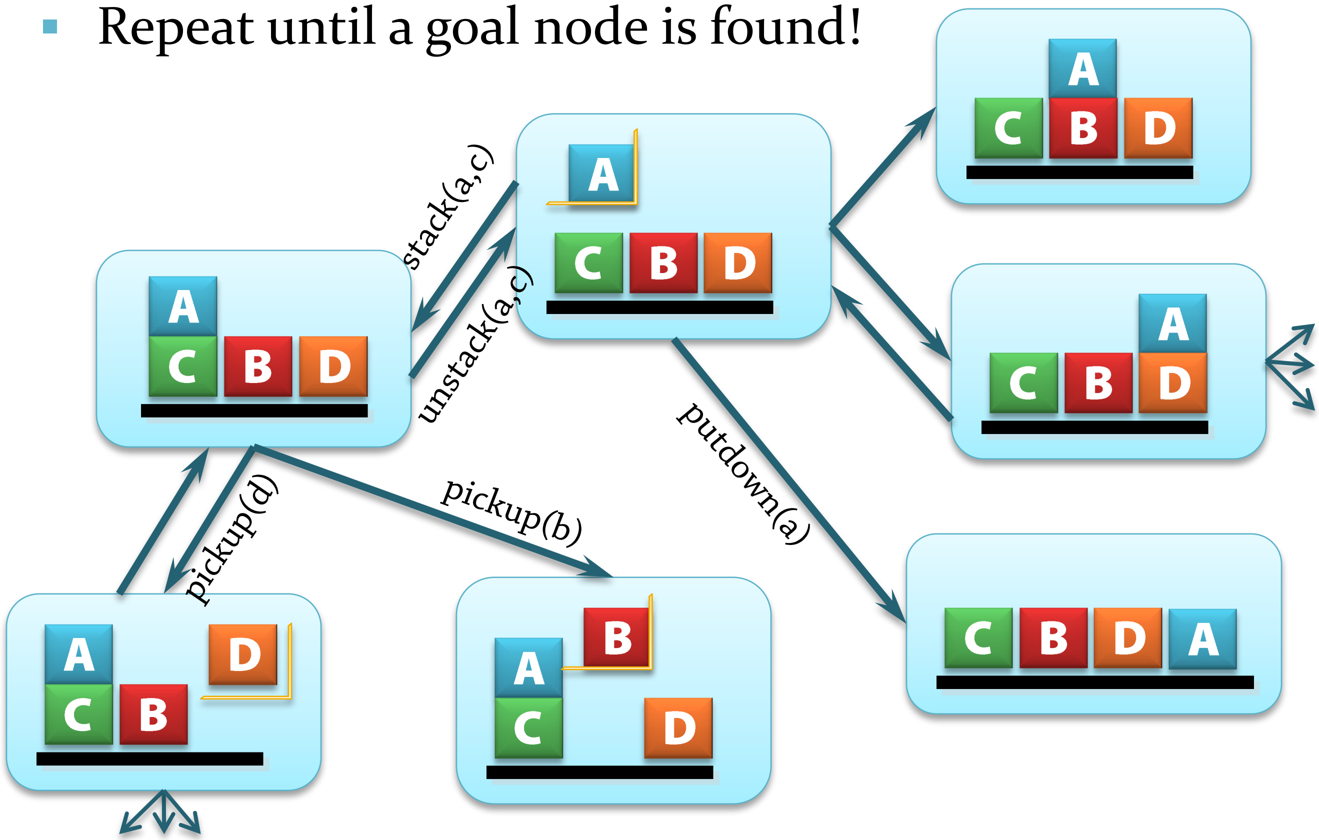
Forward State Space Search (1)

- Blocks world example:
  - **Generate** the initial state = initial node
    from the initial state **description** in the problem



Forward State Space Search (2)

- **<u>Incremental expansion</u>**: **<u>Choose</u>** a node
  - First time, the initial state – other times, depends on the **<u>search strategy</u>** used
- Expand all possible successors
  - *"What actions are applicable in the current state, and where will they take me?"*
  - Generates new states by applying effects
- Repeat until a goal node is found!



- Notice that the BW lacks dead ends.
- In fact, it is even *symmetric*.
- This is not true for all domains!

Forward State Space Search (3)

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

    open ← { <s_0, ε> }
    while (open ≠ empty set) {
    use a strategy to select and remove <s,path> from open
    if goal g satisfied in state s then **return** path

    foreach a ∈ { ground instances of *operators* applicable in state s } {
    s' ← apply(a, s) // dynamically generate a new state
    path' ← append(path, a)
    add <state', path'> to open
    }
    }
    return failure;
    }

What **strategies** are available and useful?

Expand the node

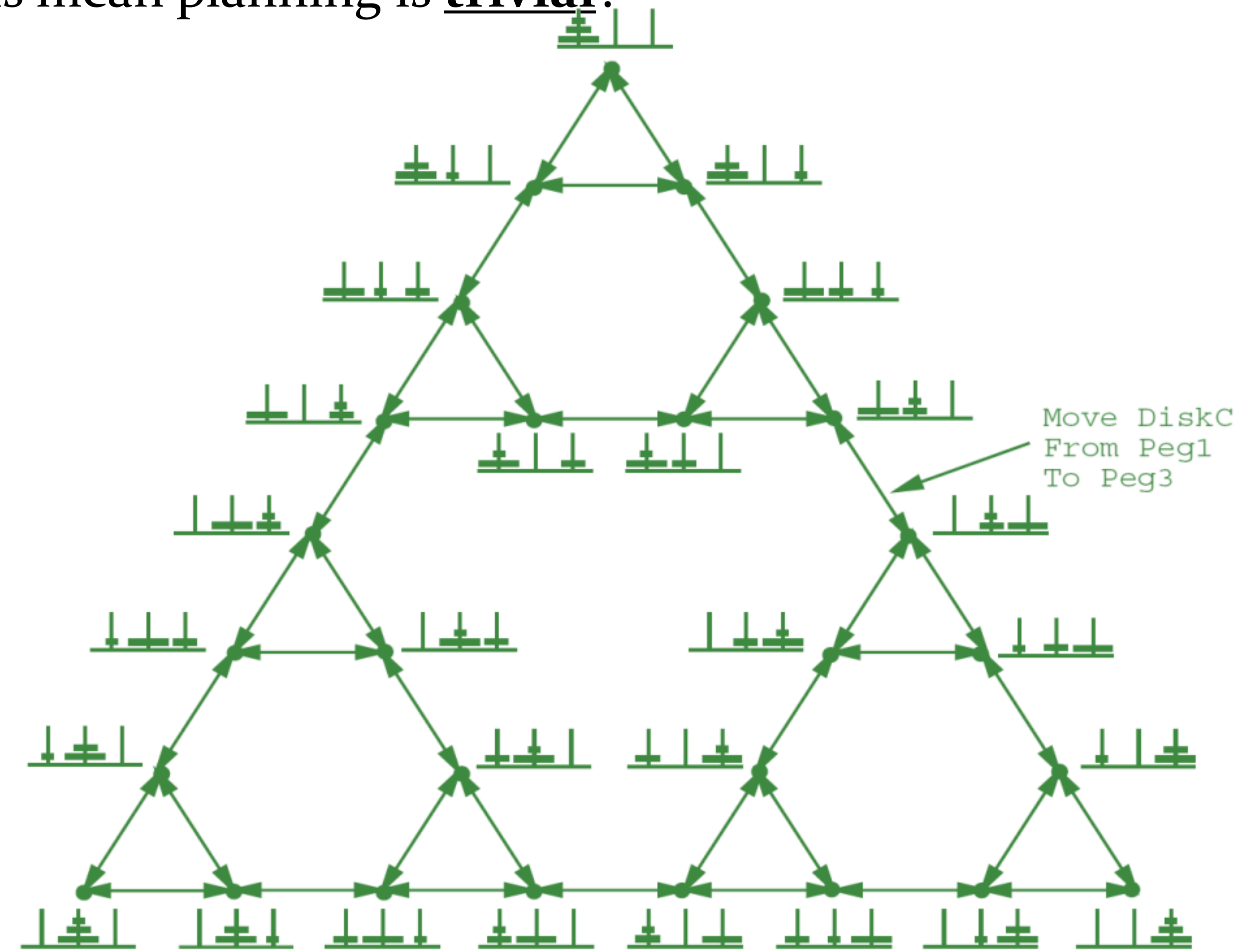
To simplify extracting a plan, a state space search node could include the plan to reach that state!

Still generally called state space search…

Is always **sound**
Completeness depends on the strategy

Forward State Space Search: Trivial?

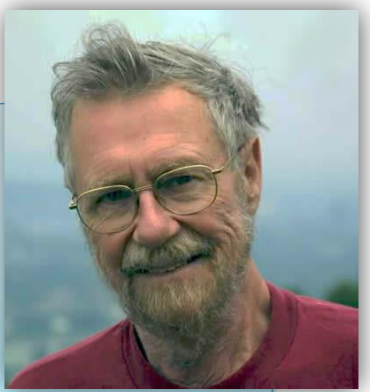
- We see that for **classical** planning problems, we can **search** directly **in the formal model** – the STS
  - Does this mean planning is **trivial**?



Forward State Space Search: Search Strategies and the Difficulty of Planning

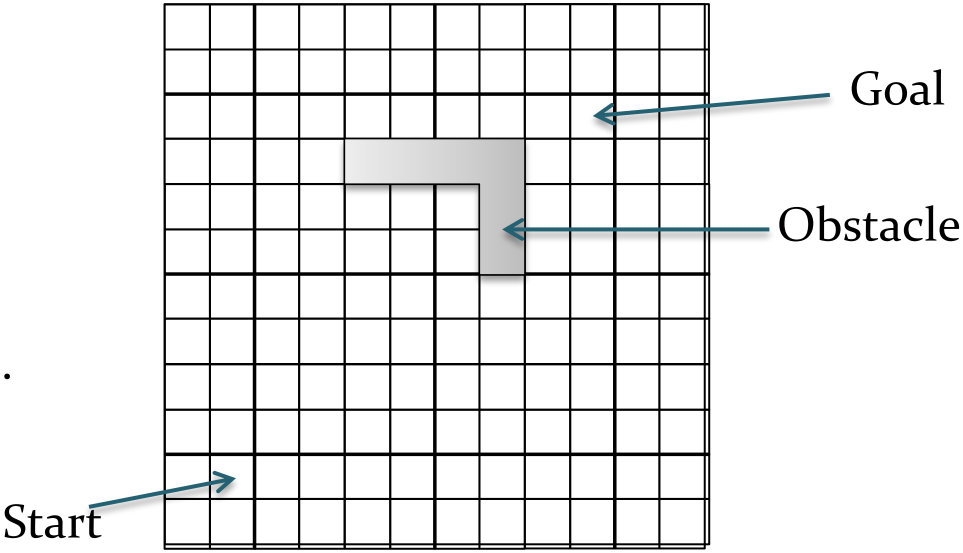
Forward State Space Search: Dijkstra

- First search strategy: **Dijkstra's algorithm**



  - **Matches** the given forward search "template"
    - Selects from *open* a node *n* with minimal g(*n*): Cost of reaching *n* from the starting point
  - **Efficient** graph search algorithm: O(|E| + |V| log |V|)
    - |E| = the number of edges, |V| = the number of nodes
  - **Optimal**: Returns minimum-cost plans

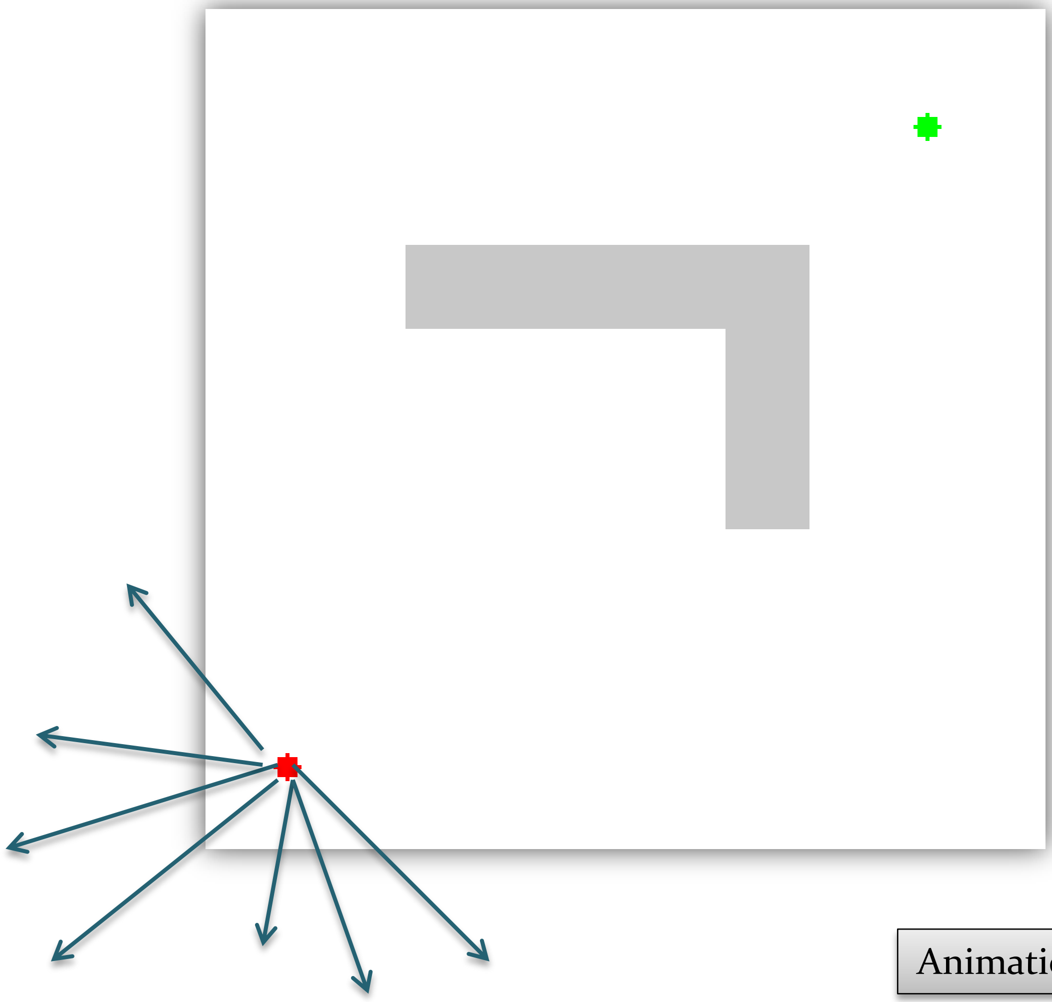
- Simple problem, for illustration:
  - **Navigation** in a grid
  - Each state specifies only the **coordinates of the robot**: Two state variables
  - **Actions**: Move left, move right, … (cost = 1)
  - Single goal node



Dijkstra's Algorithm (2)

- Dijkstra's Algorithm:

Search in all possible directions!

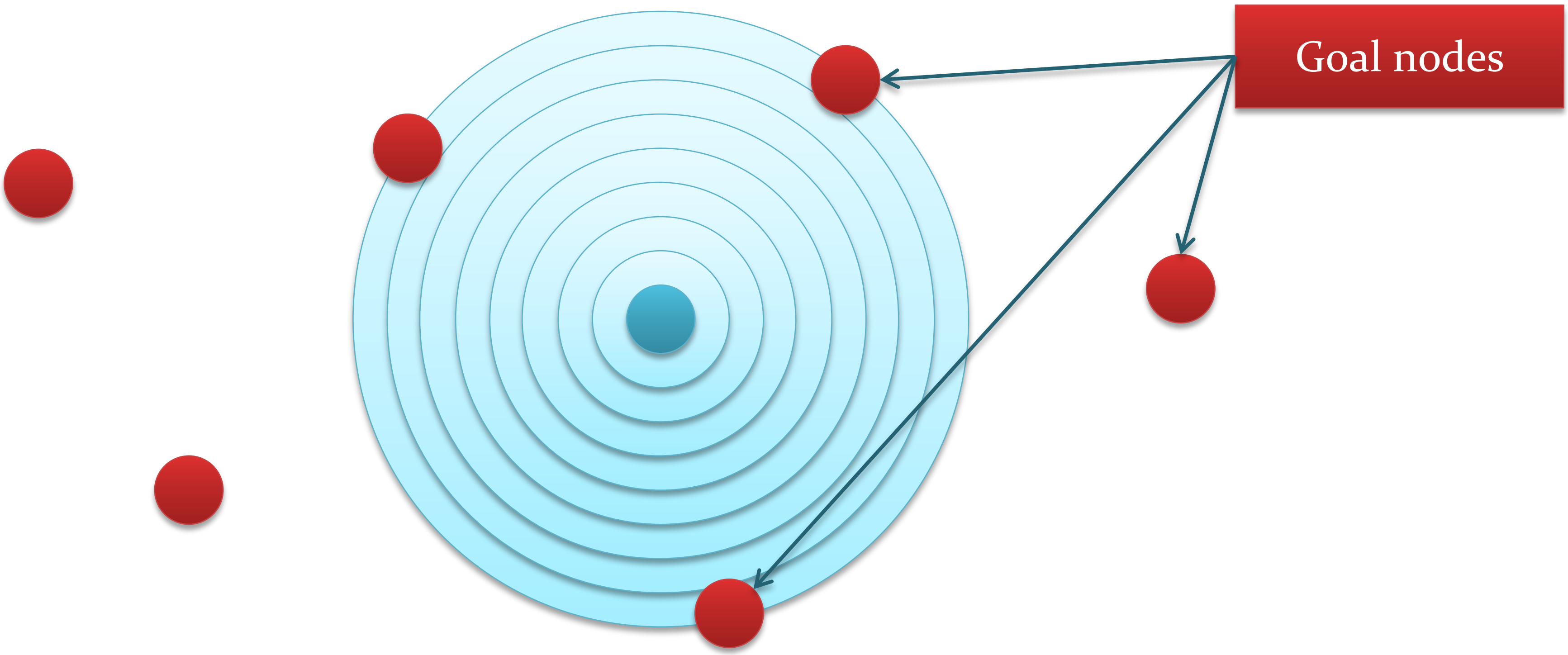


Animation from Wikimedia Commons

Dijkstra's Algorithm (3)

- Explores **all** states that can be reached more cheaply than the cheapest goal node

Usually we have many more "dimensions", many more nodes within a given distance (this was just a trivial 2-dimensional 8-connected example)!



Dijkstra's Algorithm (4)

- Blocks world, 400 blocks initially on the table, goal is a 400-block tower
  - Given uniform action costs,
    Dijkstra will **always** consider **all** plans that stack **less than 400 blocks**!
    - Stacking 1 block: = 400*399 plans, …
    - Stacking 2 blocks: > 400*399 * 399*398 plans, …
  - More than
    163056983907893105864579679373347287756459484163478267225862419762304263994207997664258213955766581163654137118163119220488226383169161648320459490283410635798745232698971132939284479800304096674354974038722588873480963719240642724363629154726632939764177236010315694148636819334217252836414001487277618002966608761037018087769490614847887418744402606226134803936935233568418055950371185351837140548515949431309313875210827888943337113613660928318086299617953892953722006734158933276576470475640607391701026030959040303548174221274052329579637773658722452549738459404452586503693 …13918091275485326579590911344408444175566421179627432025699299231777374 …07108548826574448445631879309077796615729902891948105852178191464766293 …42465441372350568748665249021991849760646988031691394386551194171193333 …30203244130264943230562021556885065768422967838517772535893398611212735 …91029206930872017424323607291625273875080732255786307776859016374355414584408338787093441749839774374303275575344176291224488351917210773338752306956814809908671090513321048204136078222064656352727110739066118003761944104289000710136954383590946416822538563947433356785458243209321069733174985157110067199853049826047551101672548547661886191289170539335470984350206597786894996069041570770057976322876697641450955815650565898117215204346127705949506137017308793077271410935265343286713600020969244834943024246490614517266459475858601049768455345074796054089038283202061310722177821564342045724346160424043752110523240382258054057131573291598463519312655627310960393718822950440

$1.63 * 10^{1735}$

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

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

Fast Computers, Many Cores

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

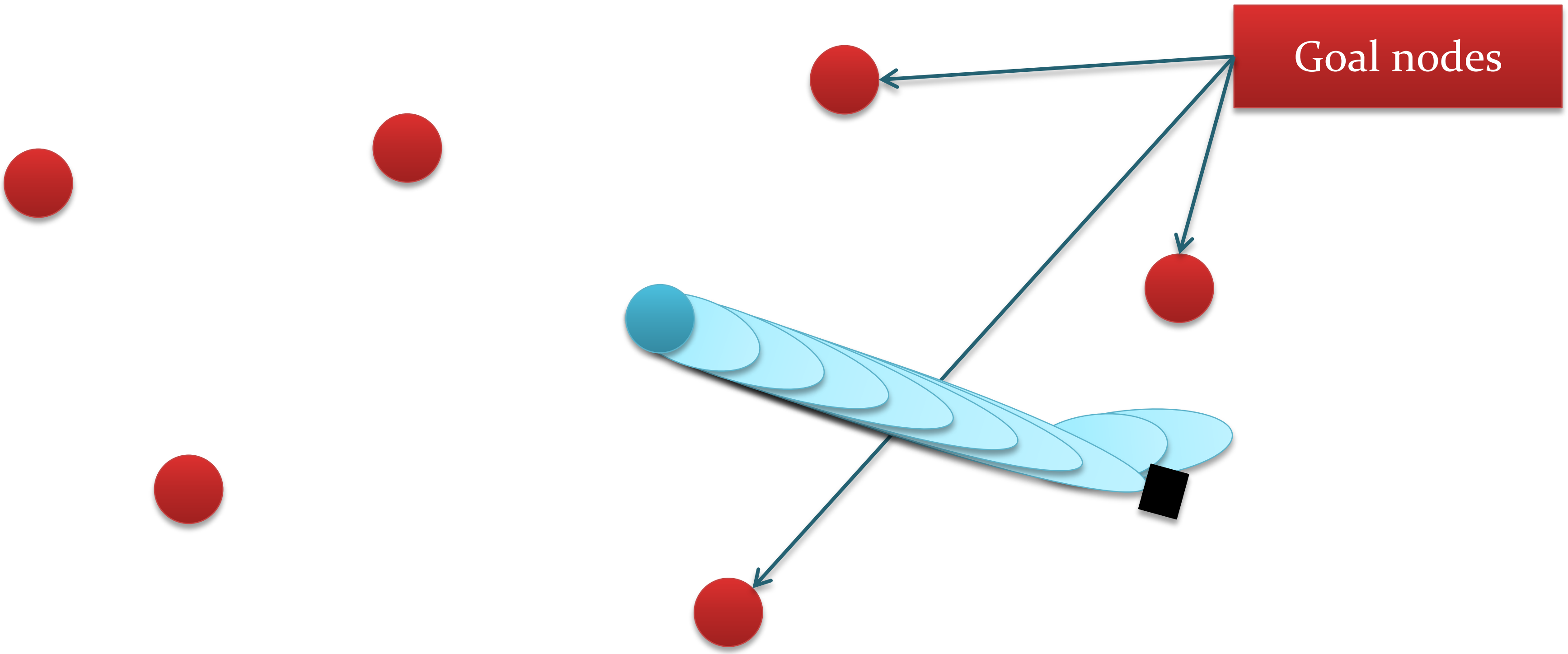
- But we have **multiple cores**!
  - The universe has at most 10^87 particles, including electrons, …
  - Let's suppose every one is a CPU core
  - ➔ only 10^1628 seconds > 10^1620 years
  - The universe is around 10^10 years old

Impractical Algorithms

- Dijkstra's algorithm is **completely impractical** here
  - Visits all nodes with cost < cost(optimal solution)
- **Breadth first** would not work
  - Visits all nodes with length < length(optimal solution)
- **Iterative deepening** would not work
  - Saves space, still takes too much time
- **Depth first** search would **normally** not work
  - Could work in *some* domains and *some* problems, by pure luck…
  - Usually either doesn't find the goal,
    or finds **very** inefficient plans
  - [movies/4_no-rules]

Depth First Search Example

- Depth first search:
  - Always prefers **adding a new action** to the current action sequence
  - Always adds the **first action** it can find



Problems and Problem Statements

We discussed problem sizes before!

Search space = transition system (>10^1700 plans)

PDDL description = problem statement (400 blocks)

Real World + current problem

Abstraction

Approximation

Language L defined by predicates, objects

Planning Problem
$\mathcal{P} = (\Sigma, s_0, S_g)$

Equivalence

Problem Statement
$P=(O,s_0,g)$

Trillions of states in $\Sigma = (S,A,\gamma)$ would be a rather small planning **problem**

Trillions of state transitions in γ would also correspond to a small planning problem

Thousands of constants and predicates in L would be a rather large **classical** planning problem **statement**

Hundreds of operators would correspond to a very large classical planning problem statement

Hopeless?

- Is there still hope for planning?
  - Of course there is!
  - Our trivial planning method uses **blind** search – tries **everything**!
  - **We** wouldn't choose such silly actions – so why should the computer?

- **Planning is part of Artificial Intelligence!**
  - We should develop methods to **judge** what actions are **promising** given our goals

Search Guidance

Two Types of Guidance

Two distinct **types** of guidance

Binary decision: Is this search node **definitely bad** or **possibly good**?	On a scale: **How promising** is this search node?
Definitely bad ➔ remove the node, prune the tree ➔ never have to consider the node again! Possibly good ➔ keep the node	A **heuristic function**, used to prioritize the *search order* Low value ➔ try earlier High value ➔ keep, possibly try later
Potentially very effective	Resilient: Prioritize in the wrong order ➔ can come back later
A single mistake, removing a *good* node ➔ might not find a solution at all! Therefore, difficult to find good *domain-independent* pruning rules	Less efficient: Have to keep all nodes in case you need to go back later
	For now, we will focus on heuristics!

Two Aspects of Guidance

Two **aspects** of guiding search

Defining a **search strategy** that takes guidance into account

Examples:

A* uses a heuristic (function)
Hill-climbing uses a heuristic… differently!

Generating the actual **guidance** as input to the search strategy

Example:

Finding a suitable heuristic function 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 algorithms

Two Uses for Guidance

Two distinct **objectives** for guidance

Find a solution **quickly**	Find a **good** solution
Prioritize nodes that appear to be **close to a goal node** in the search space	Prioritize nodes that appear to lead to **good solutions**, even if finding those solutions will be difficult

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

Node: Plan length 50, estimated goal distance 10

Node: Plan length 5, estimated goal distance 30

Heuristics for Forward State Space Search: True Costs and Heuristic Estimates

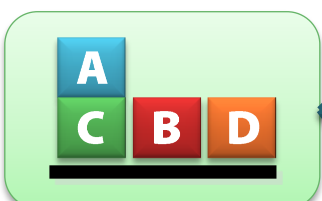
True Goal Distances

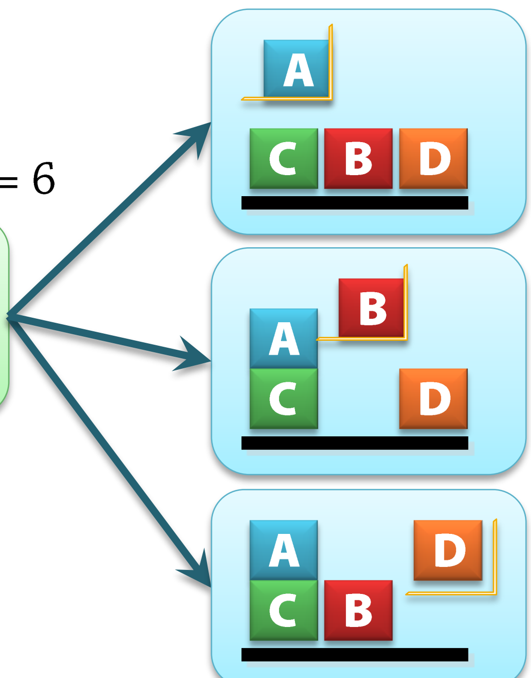
For now: A solution is **better** if it has **lower cost when executed**

Let h*(*n*) be the **actual cost** of reaching a goal from *n*

Cost = sum of **action costs** for cheapest solution starting in *n*
In the example, each action has a cost of 1
We don't *explicitly* consider computational costs of *finding* solutions!

h*(initnode) = 6





Cheapest solution starting here:
putdown(A); pickup(B); stack(B,C);
pickup(A); stack(A,B) ➔ h*(thisnode) = 5

putdown(B); unstack(A,C);
putdown(A); pickup(B); stack(B,C);
pickup(A); stack(A,B) ➔ 7

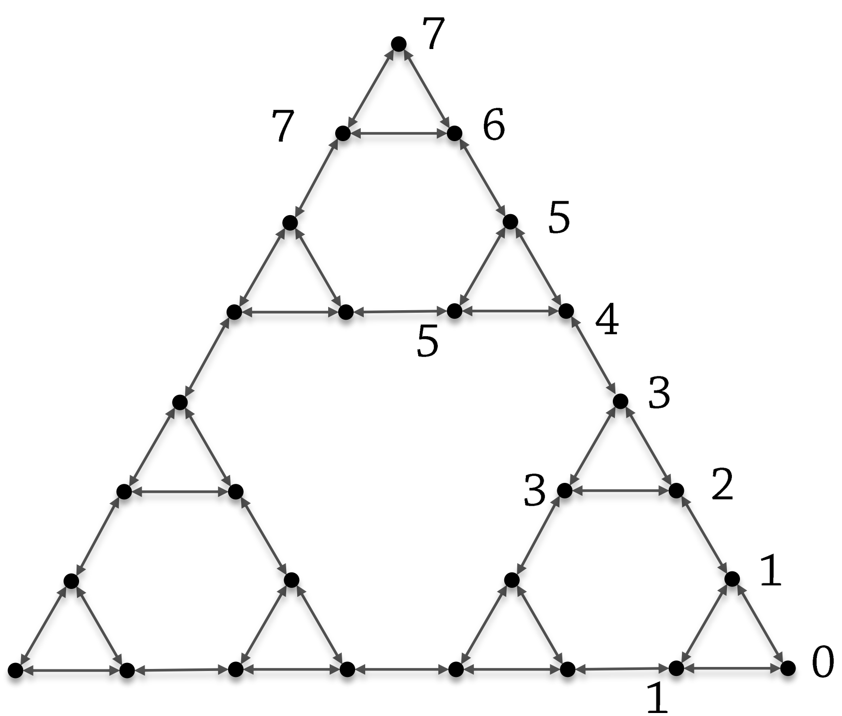
putdown(D); unstack(A,C);
putdown(A); pickup(B); stack(B,C);
pickup(A); stack(A,B) ➔ 7

Planning given True Goal Distances

If we *knew* the true goal distances h*(n):

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!



Heuristics Estimate True Goal Distances

- So regardless of method, computing h* is **as hard as optimal planning**!
  - Planning is PSPACE-complete in general…
    (in terms of input size = representation size)

Heuristics should quickly provide good estimates of h*

- A **heuristic function** h(*n*):
  - An approximation of h*(*n*)
  - Often used together with g(*n*), the known cost of *reaching* node *n*
- **Admissible** if $\forall n.\ h(n) \leq h^*(n)$
  - Never overestimates – important for *some* search algorithm

General Heuristic Forward Search

General Heuristic Forward Search Algorithm

```
heuristic-forward-search(ops, s0, g) {
    open ← { <s0, ε> }
    while (open ≠ emptyset) {
        use a heuristic search strategy to select and remove <s,path> from open
        if path is cyclic then skip it
        if goal-satisfied(g, s) then return path

        foreach a ∈ groundapp(ops, s) {
            s′ ← apply(a, s)
            path′ ← append(path, a)
            add <state′, path′> to open
        }
    }
    return failure;
}
```

- The **strategy** selects nodes from the *open* set depending on:
  - h(*n*)
  - Possibly other factors such as g(n)
- What is a *good* heuristic depends on:
  - The algorithm (examples later)
  - The purpose (good solutions / finding solutions quickly)

A*, simulated annealing, hill-climbing, …

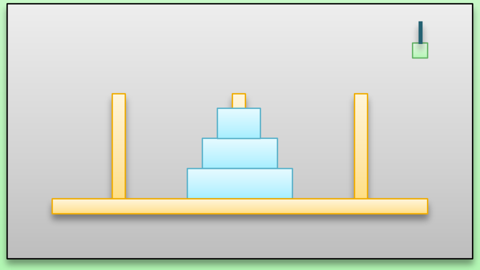
A Simple
Domain-Independent Heuristic

Heuristics given Structured States

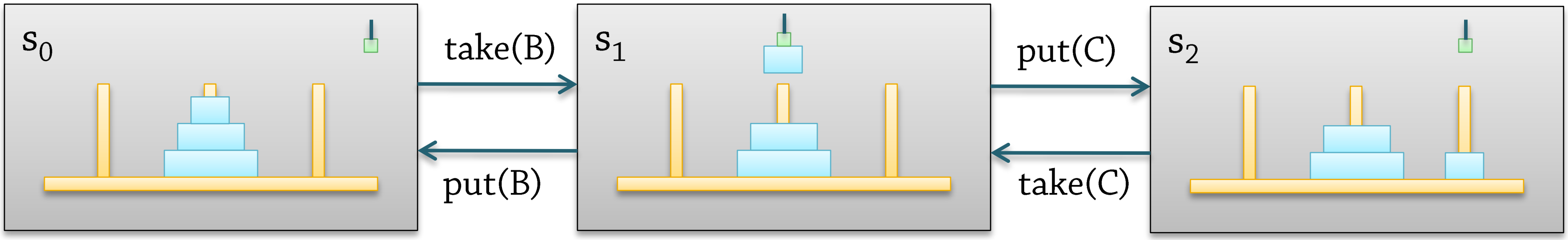
- In planning, we often want **domain-independent** heuristics
  - Should work for **any** planning domain – how?
- Take advantage of **high-level representation**!

- **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 Sg={ 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?
  - …



Heuristics given Structured States (2)



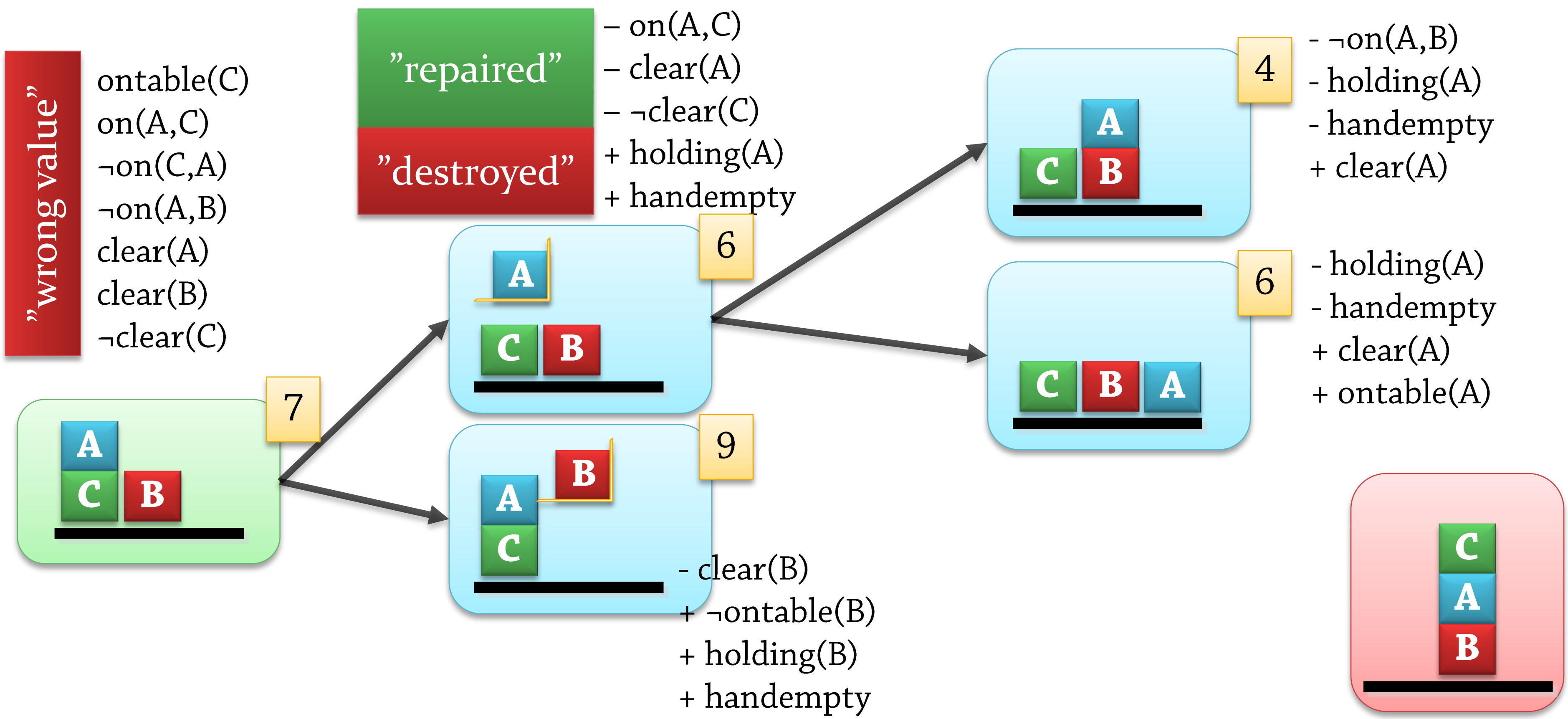
- All facts can be "tested" independently of each other
  - **What is the difference between states 0 and 1?**
    Only that in state 1, disk 1 is being **carried**
    instead of being **on top of disk 2 on peg B**
    (so the states are *very similar*)
- We can see "how close" a state is to the goal
  - "Almost all disks are in the right place, only C needs to be moved"
- We see **actions** as having structure: Parameters, conditions, effects
  - Can see that in state s_0, we cannot execute take(2,b),
    because the precondition top(2) is not true
    (there is something on top of disk 2)

This can be used as a basis for our heuristics!

Counting Remaining Goals

- A very simple **domain-independent** heuristic:
  - **Count** the number of facts that are "wrong"
    - Competely independent of the domain

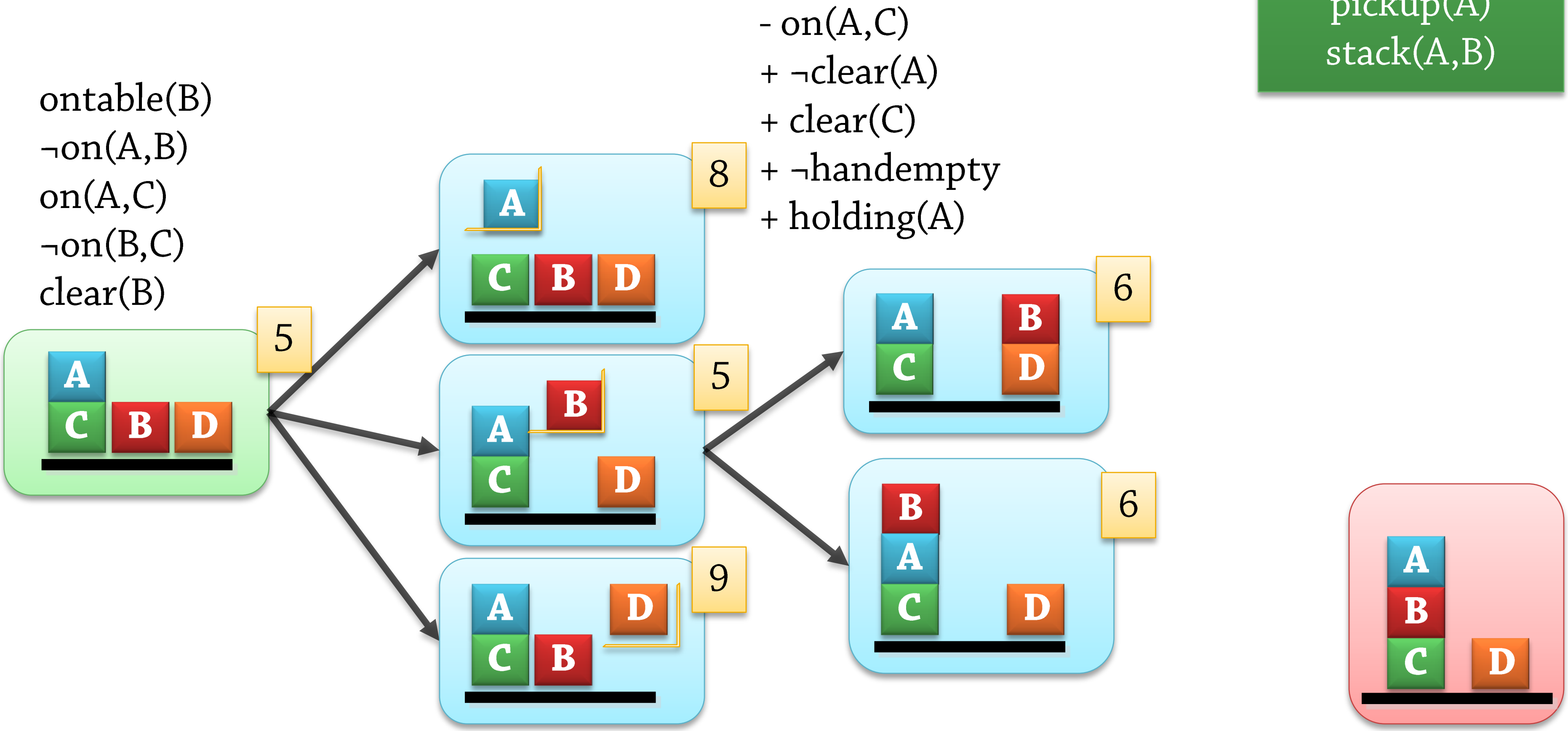
Optimal:
unstack(A,C)
stack(A,B)
pickup(C)
stack(C,A)



Counting Remaining Goals (2)

- A **perfect** solution? No!
  - We must often go **away** from the goal before we can approach it again

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



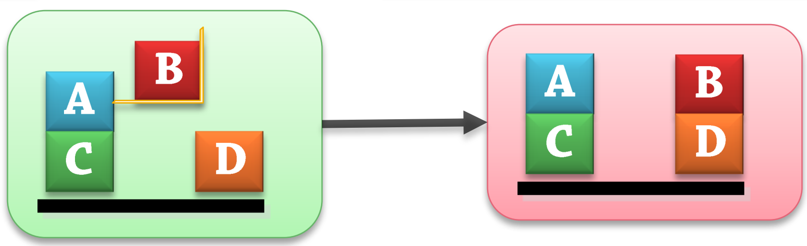
Counting Remaining Goals (3)

- Not admissible!
  - Matters to some heuristic search algorithms (not all)

”wrong value”

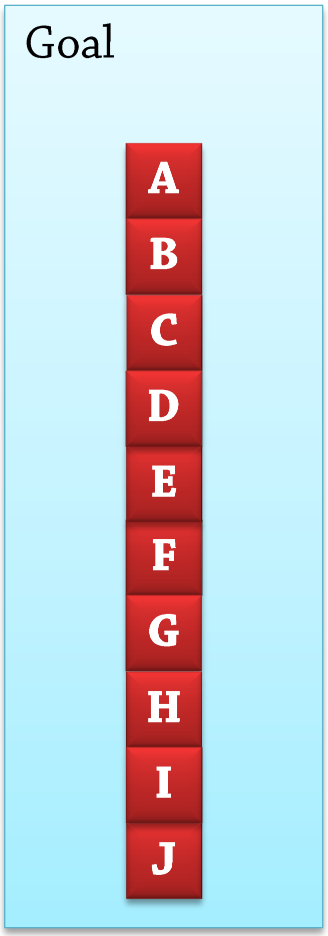
¬on(B,D)
¬handempty
¬clear(B)
clear(D)

4 facts are ”wrong”, can be fixed with a single action



Counting Remaining Goals (4)

- In the scenario below:
  - Facts to add: on(I,J)
  - Facts to remove: ontable(I), clear(J)
  - Heuristic value of 3 – but is it close to the goal?



Counting Remaining Goals (5): Analysis

- What we see from **this** analysis is…
  - 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

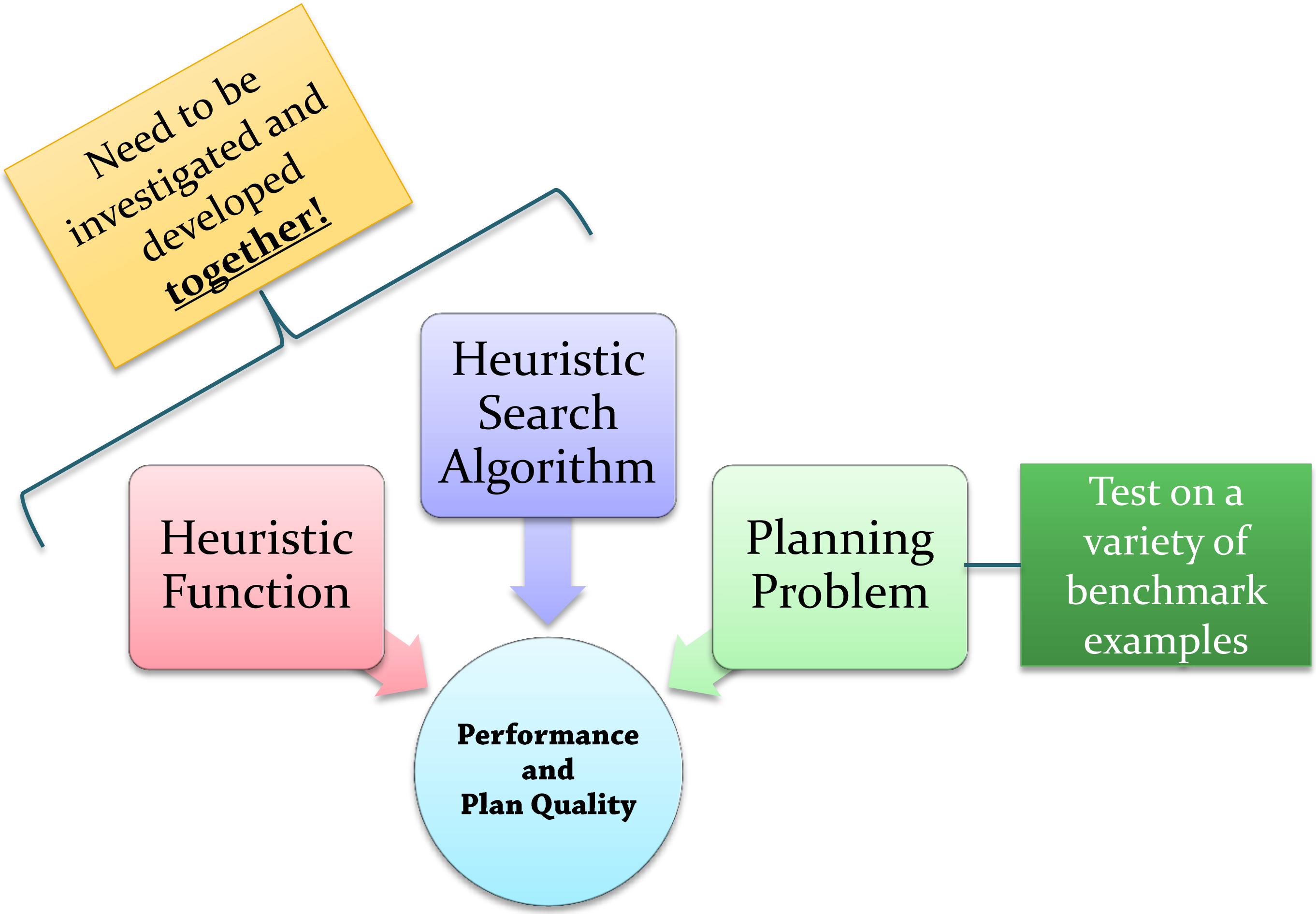
- Planning Competition 2011: Elevators domain, problem 1
  - A* with goal count heuristics
    - States: 108922864 generated, gave up
  - LAMA 2011 planner, good heuristics, other strategy
    - Solution: 79 steps, 369 cost
    - States: 13236 generated, 425 evaluated/expanded
- Elevators, problem 5
  - 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/expand

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…

Some Desired Properties (1)

- What properties do **good heuristic functions** have?
  - **Informative**: Provide guidance to the search strategy
  - In what sense? Depends on the strategy (examples later)!



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**…
    - 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

Heuristic Search: Difficult

<u>Good</u> domain-independent heuristics were difficult to find…

- Bonet, Loerincs & Geffner, 1997:
  - Planning problems are **<u>search problems</u>**:
    - There is an *initial state*,
      there are *operators* mapping states to successor states,
      and there are *goal states* to be reached.
  - Yet planning is **<u>almost never formulated in this way</u>**
    in either textbooks or research.
  - The reasons appear to be two:
    - the specific nature of planning problems, that calls for decomposition,
    - and the **<u>absence of good heuristic functions</u>**.

Alternative Approaches

- At the time, research diverged into **alternative approaches**

Use another search space to find plans more efficiently

Backward state search
Partial-order plans
Planning graphs
Planning as satisfiability
...

Include more information in the problem specification

(Domain-specific heuristics)
Hierarchical Task Networks
Control Formulas

But that was 15 years ago!

Heuristics have come a long way since then...

Heuristics and Search Strategies for Optimal Forward State Space Planning

A Well Known Heuristic Search Algorithm: A*

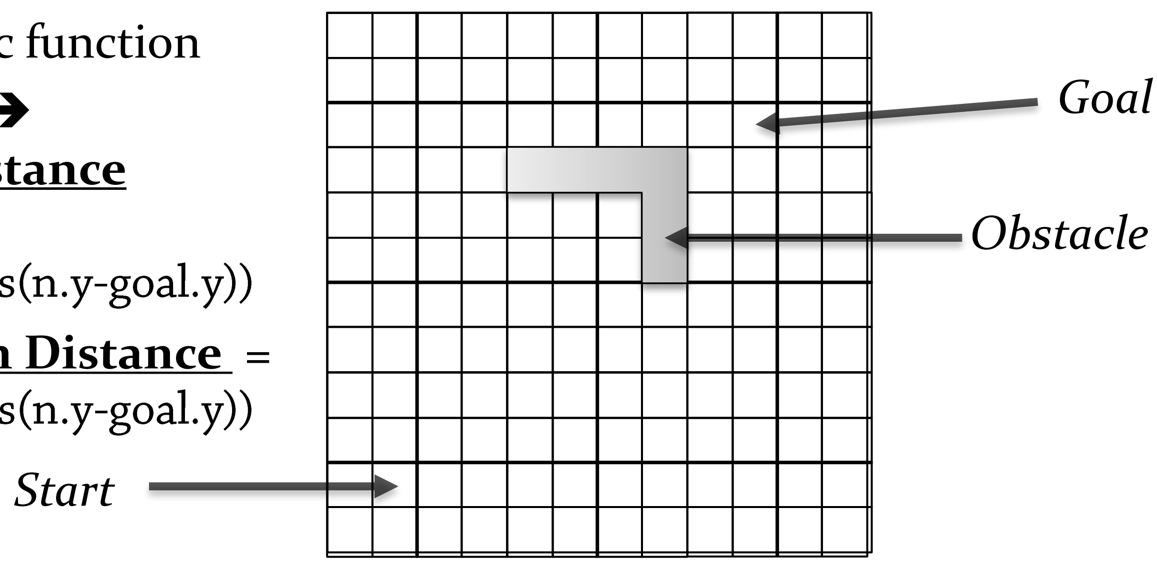
Used in many **optimal** planners

A* (1)

- Dijstra vs. A*: The essential difference

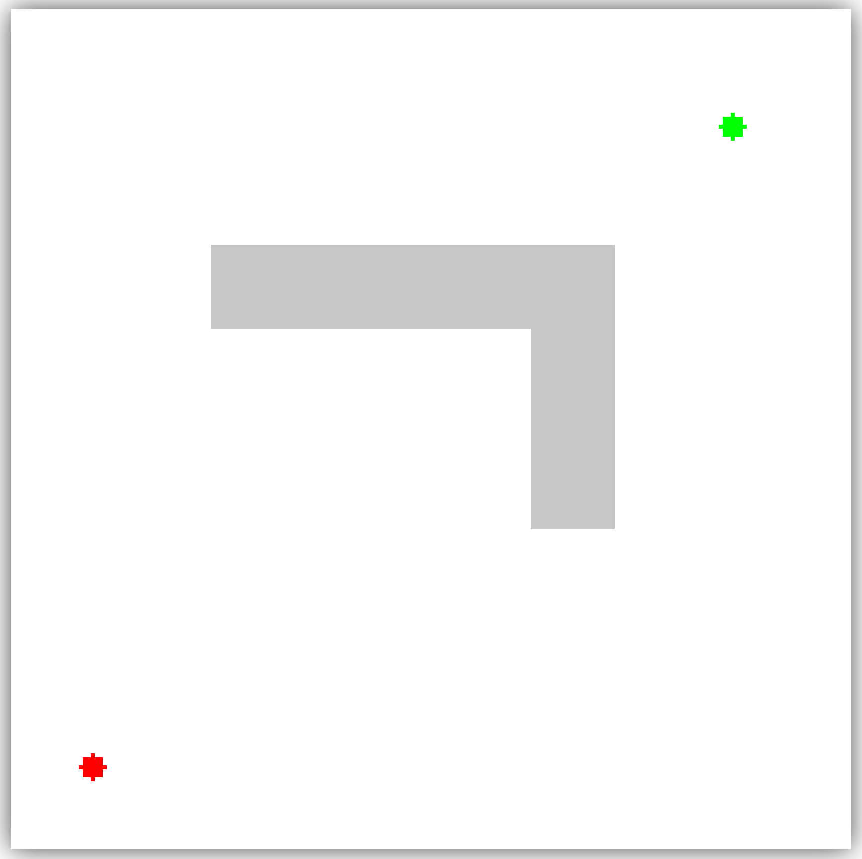
Dijkstra	A*
Selects from *open* a node *n* with minimal f(*n*) = g(*n*) - Cost of reaching *n* from initial node	Selects from *open* a node *n* with minimal f(*n*) = g(*n*) **+ h(*n*)** - + **estimated cost of reaching a goal from *n***
Uninformed (blind)	Informed

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



A* (2)

- A* Search:



<u>Here:</u>
A single physical obstacle

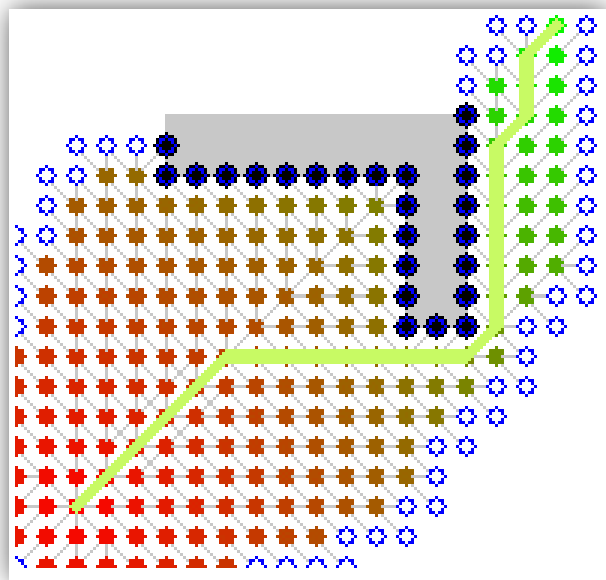
<u>In general</u>:
Many states where all available actions will increase g+h (cost + heuristic)

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

A* (3)

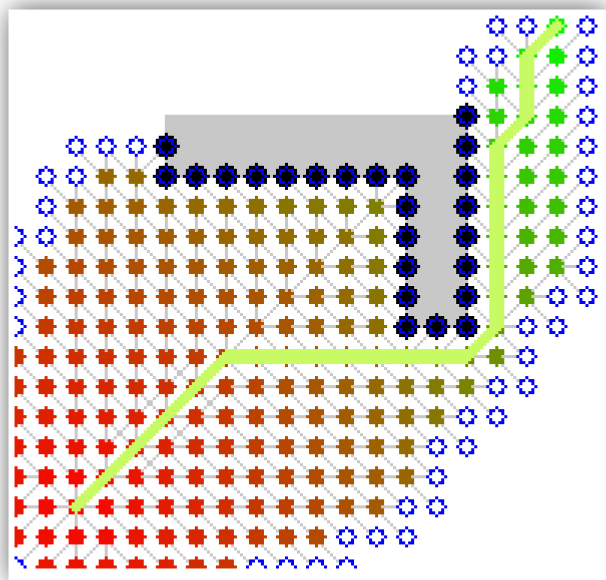
- Given an admissible heuristic *h*, A* is **optimal in two ways**
  - Guarantees an **optimal** plan
  - Expands the **minimum number of nodes**
    required to guarantee optimality when this heuristic is used

- Still expands many ”unproductive” nodes in the example
  - Because the heuristic is **not perfectly informative**
    - Even though it is hand-coded
    - Does not take **obstacles** into account



A* (4)

- What is an **informative** heuristic for A*?
  - As always, h(n) = h*(n) would be perfect – but maybe not attainable…
  - But 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
  - Sounds obvious
    - But not true for all search strategies!

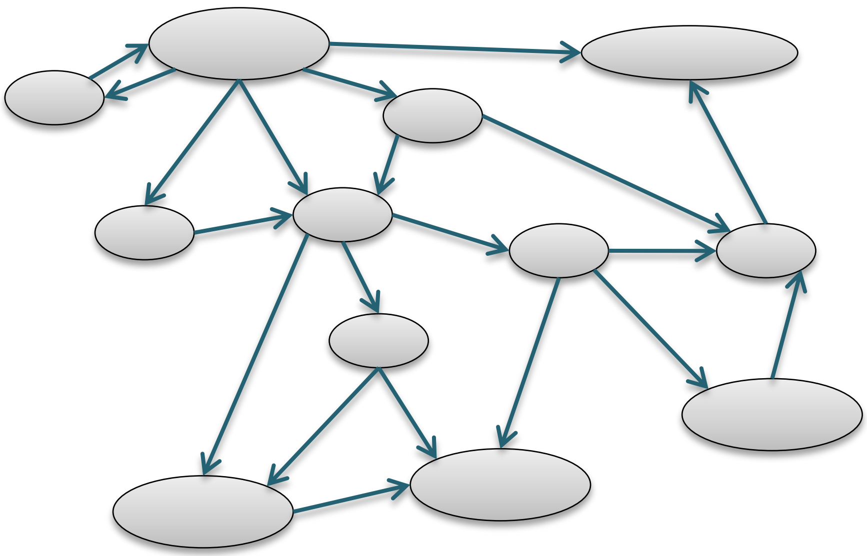


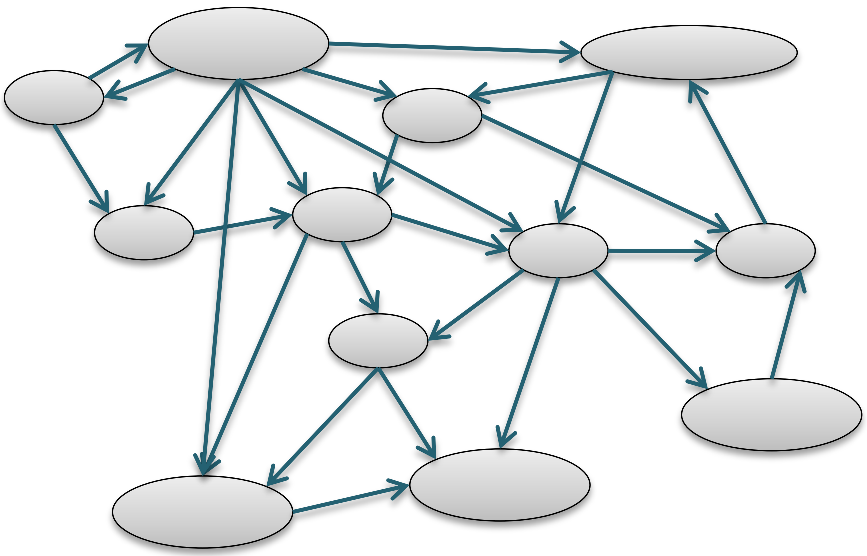
Creating Admissible Heuristics: The Relaxation Principle

Relaxation 1: Intro

Suppose we have a planning problem P…

…and we add **more edges** (transitions), resulting in P'





The problem is simpler, **the constraints are relaxed**:
All old solution plans remain valid, new solutions become possible!

An **optimal** solution for P'
can **never** be more expensive than the corresponding optimal solution for P

Relaxation 2: Generalization

Suppose we have a planning problem P…

…and we add **more solutions**, resulting in P’

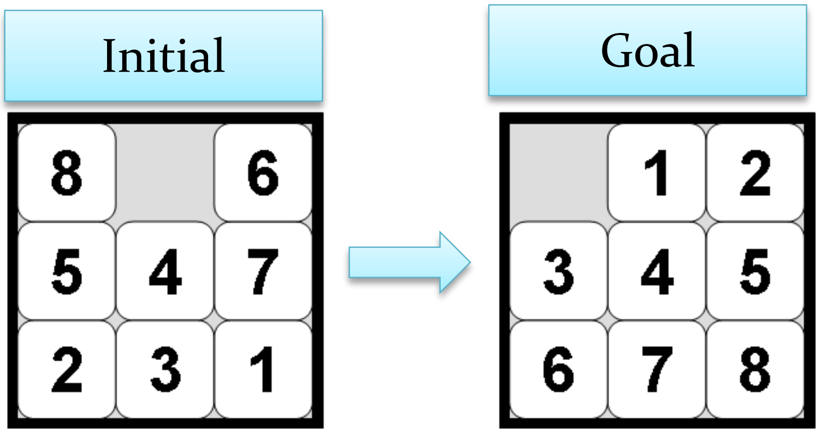
No matter how this is done:
Changing existing transitions,
using different states altogether, …

As long as all old solution plans remain solutions for P’:

The **optimal** solution for P’
can **never** be more expensive than the optimal solution for P

Relaxation 3: Example

- Classical example: The **8-puzzle** (15-puzzle, …)



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

  - Relaxation: **Suppose that tiles can be moved across each other**
    - Now we have 21 possible first moves!
  - 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"

The **optimal** solution for modified 8-puzzle
can **never** be more expensive than the optimal solution for original 8-puzzle

Relaxation 4: Admissible Heuristic

- We want:
  - A heuristic h for P that is **admissible**: $\forall n.\ h(n) \leq h^*(n)$

Original 8-puzzle

- We know:
  - An optimal solution for P'
    can **never** be more expensive than the corresponding optimal solution for P
  - $\neg\exists n.\ h^{*\prime}(n) > h^*(n)$
  - $\forall n.\ h^{*\prime}(n) \leq h^*(n)$: **h*'(n) is an admissible heuristic for P**

Relaxed 8-puzzle

How does this help?

$h^{*\prime}(n)$ may be much easier to calculate than $h^*(n)$

Relaxation 5: Example

- Let's analyze the **relaxed 8-puzzle**...
  - 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!

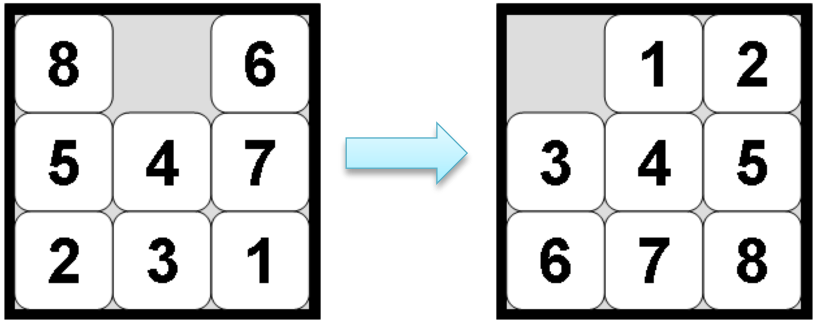
- ➔ **optimal cost** for relaxed problem
  = sum of Manhattan distances

- ➔ **admissible heuristic**
  for *original* problem
  = sum of Manhattan distances

- Can be **coded procedurally** in a solver – efficient!
  - (Though we'd prefer to extract heuristics automatically – later!)

Rapid calculation
is the *reason* for relaxation

Shorter solutions
are an *unfortunate side effect*:
Leads to less informative heuristics



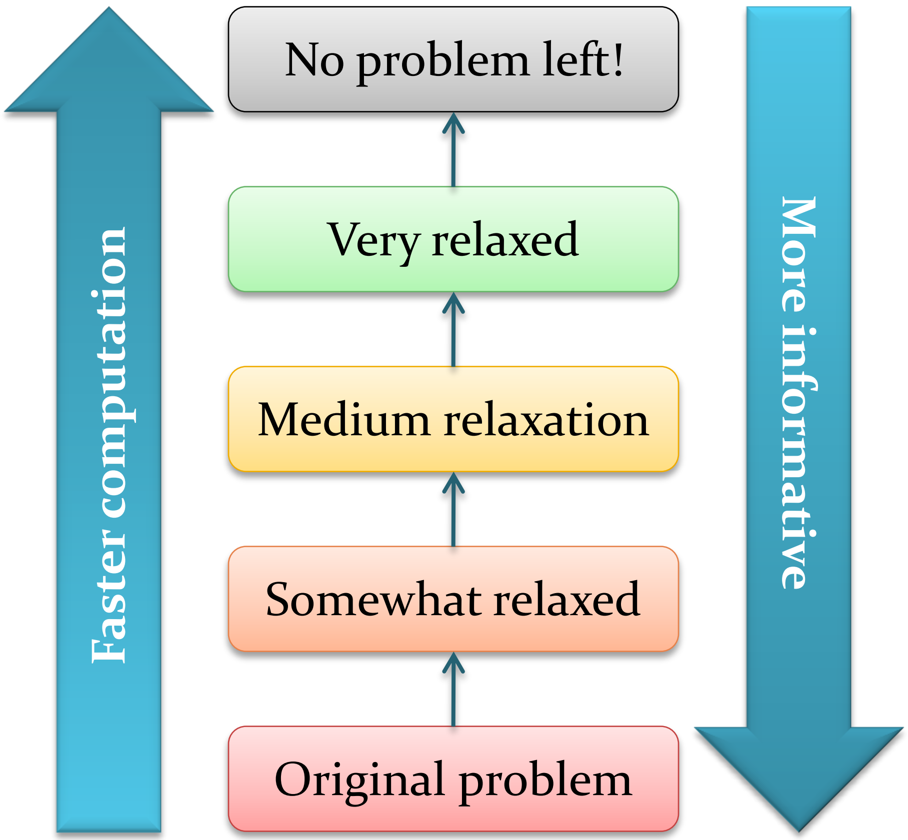
Relaxation 6: Principle

- **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)
    - **Calculating the cost** of 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

Relaxation is not the only method
used to derive new heuristics!

Relaxation 7: Balance

- Should be easy to calculate – but must find a balance!
  - Relax too much ➔ not informative
    - Example: Any piece can teleport into the desired position
      ➔ h(n) = *number* of pieces left to move



Relaxation 8: Important Issues!

- Important:

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.

Solving the relaxed problem
can result in a more expensive solution
➔ inadmissible!

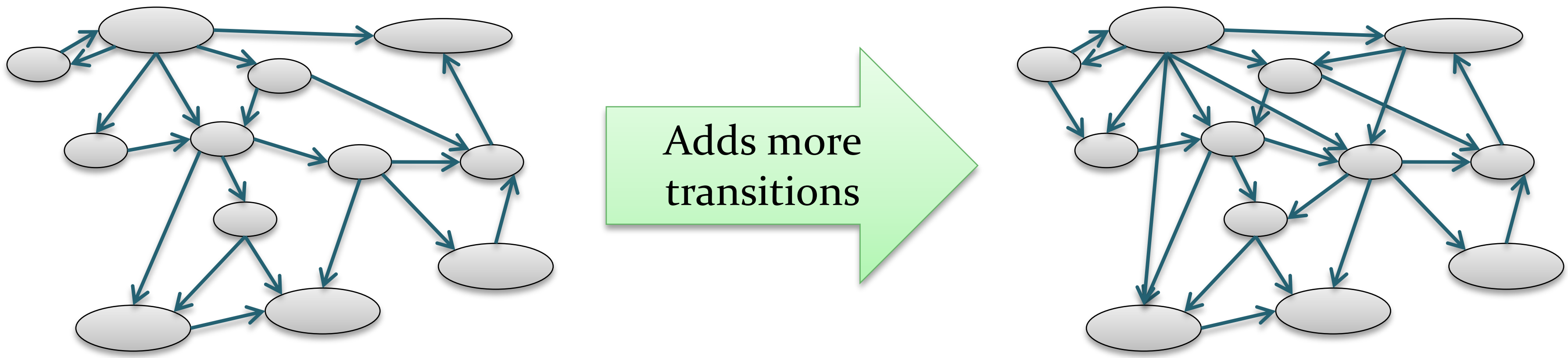
You have to solve it optimally to get the admissibility guarantee.

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

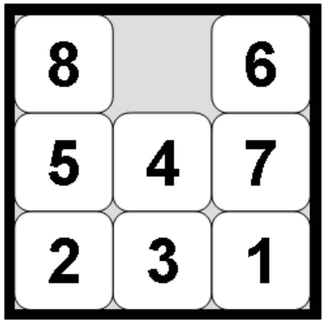
General Domain-Independent Techniques: Precondition Relaxation, Delete Relaxation

Precondition Relaxation

- What about **domain-independent** heuristics?
  - Planners don't reason:
    "Suppose that tiles can be moved across each other"...
  - One general technique: **Precondition relaxation**
    - Remove some preconditions
    - Solve the resulting problem in a standard optimal planner
    - Return the cost of the optimal solution



Example: 8-puzzle



- (**define** (domain strips-sliding-tile)

```
(define (domain strips-sliding-tile)
  (:requirements :strips)
  (:predicates
    (tile ?x) (position ?x)
    (at ?t ?x ?y) (blank ?x ?y)
    (inc ?p ?pp) (dec ?p ?pp))
  (: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)))
  …)
```

Remove this ➔ **exactly** the same relaxation that we hand-coded!

Problem 1: How can a planner *automatically determine* which preconditions to remove/relax?

Problem 2: Need to actually *solve* the resulting planning problem (unlikely that the planner can automatically find an efficient closed-form solution!)

Delete Relaxation (1)

- Second general technique: **delete relaxation**
  - Assume a pure "old-fashioned" STRIPS problem with:
    - **Positive** preconditions
    - **Positive** goals

Then a state where additional facts are true can be better, but never worse!

$s \supset s' \Rightarrow h^*(s) <= h^*(s')$

  - Why?
    - If *adding* a fact to a state makes an action *inapplicable*, this has to be due to a negative precondition
    - If *adding* a fact to a state makes a goal *inachievable*, this has to be due to a negative goal

Delete Relaxation (2)

- Assume we have both **negative and positive** effects
  - The relaxation: **remove all negative effects** (all "delete 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))

- Modifies the state transition system, *moves* existing transitions!

Delete Relaxation (3): 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)

STS for the delete-relaxed 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

pickup(B) →

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

No physical correspondence!

<u>All</u> the same actions applicable – and more!

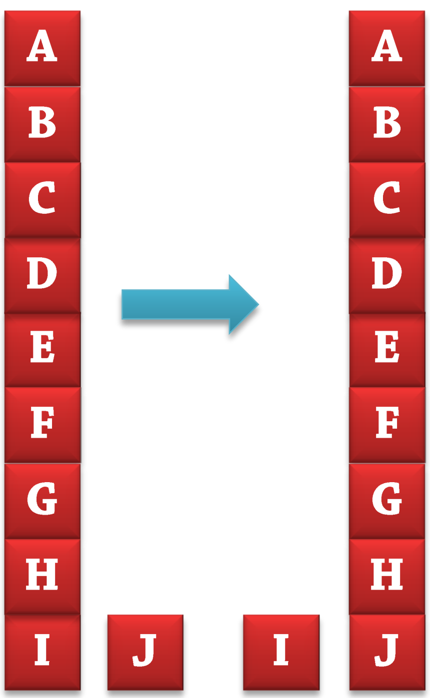
Delete Relaxation (4): Heuristic

- **Analysis**:
  - "All the same actions applicable – and more"
  - In fact, given any **action sequence**:
    - If it is applicable P, it is applicable in P'
    - If it results in a goal state in P, it results in a goal state in P'
    - ➔ This *is* a relaxation!
- **Easy to apply mechanically**
  - Remove **all** negative effects
- If **only** this relaxation is applied:
  - Gives us 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)?
  - **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:

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

Relaxed:

unstack(A,B)
unstack(B,C)
unstack(C,D)
unstack(D,E)
unstack(E,F)
unstack(F,G)
unstack(G,H)
unstack(H,I)
stack(H,J)
DONE!

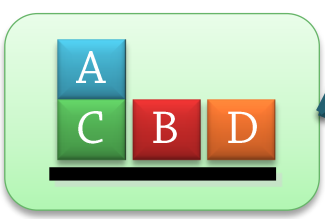
Accuracy of h+ in Selected Domains (2)

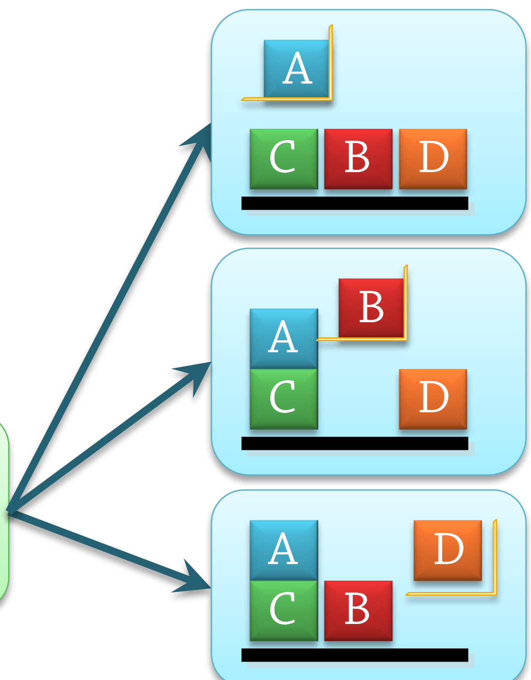
- **<u>How close</u>** is h+(*n*) to the true goal distance h*(*n*)?
  - **<u>Asymptotic accuracy</u>** as problem size approaches infinity:
    - Blocks world: 1/4 ➔ h+(*n*) ≥ 1/4 h*(n)
    - Gripper domain: 2/3 (single robot moving balls)
    - Logistics domain: 3/4 (move packages using trucks, airplanes)
    - 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

Example of Accuracy

- Delete relaxation example
  - **Accuracy** will depend on the domain and problem instance!
  - **Performance** also depends on the search strategy
    - How sensitive it is to specific types of inaccuracy

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

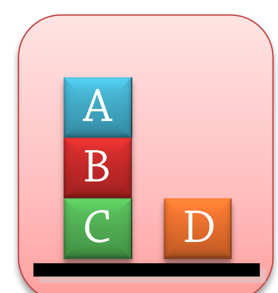




pickup(B); stack(B,C); stack(A,B)
➔ h+ = 3 [h* = 5]
Good action!

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]



Calculating h+

- **Why** is h+(*n*) **easier to calculate** than the true goal distance?
  - Only positive effects remain
    - ➔ The set of **true facts** increases monotonically
  - Only positive preconditions exist
    - ➔ The set of **applicable actions** increases monotonically
    - ➔ **If** a solution contains actions a1+a2, **then** the order of addition is irrelevant
  - Still **difficult** to calculate in general!
    - Remains a **planning problem**
    - NP-equivalent (reduced from PSPACE-equivalent),
      since you must find **optimal** solutions to the relaxed problem
      in order to guarantee admissibility
    - Even a constant-factor approximation is NP-complete to compute!
  - Therefore, not **directly** useful
  - But forms the **basis** of many other heuristics such as h1(n), h2(n)

Important Issues

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

Pattern:

Precondition relaxation	ignores/removes/relaxes some preconditions
Delete relaxation	ignores/removes/relaxes all ”delete effects”

Optimal Classical Planning Using Admissible h_m Heuristics

The h_m Heuristics

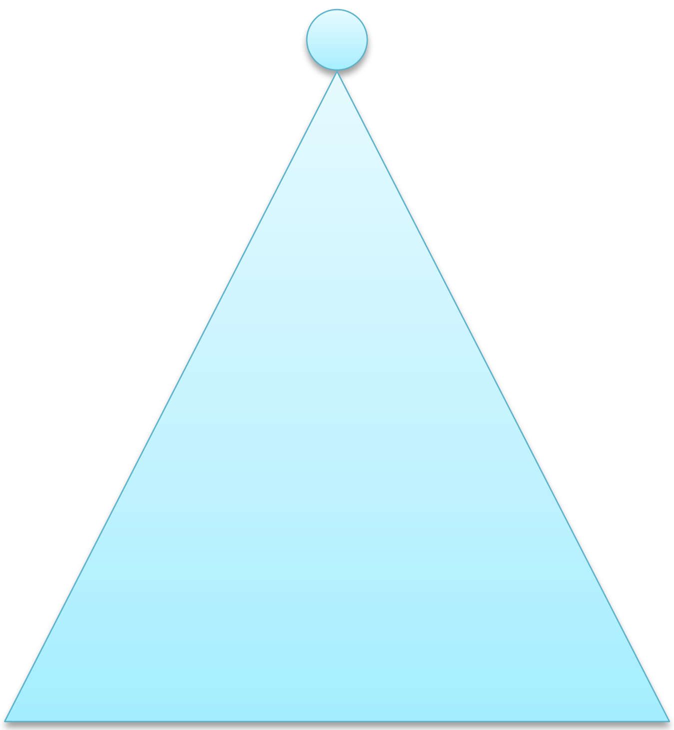
- For optimal planning,
  we need a "faster" admissible heuristic than h+ !
  - Idea in **HSPr***:
    Compute the cost of achieving **subsets of the goal**
    - $h_1(s)=\Delta_1(s,g)$: The most expensive atom
    - $h_2(s)=\Delta_2(s,g)$: The most expensive pair of atoms
    - $h_3(s)=\Delta_3(s,g)$: The most expensive triple of atoms
    - …
    - ➔ A **family** of **admissible** heuristics $h_m = h_1, h_2, \ldots$
      for **optimal** classical planning

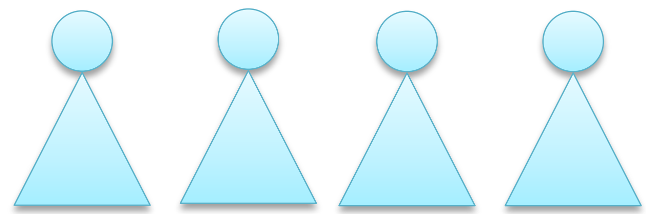
The h_m Heuristics: Essential Difference

- Basic idea: Try to achieve **individual goals**; sum their costs

h+(n) (optimal delete relaxation):
Remove delete effects,
find a single long plan

h_m(n): Solve each **goal subset** of size m
Take the **maximum** of their costs



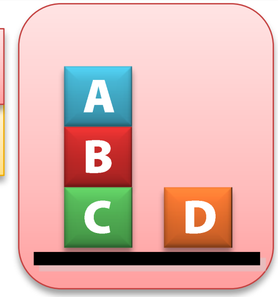


Much easier,
given that search trees tend to be wide

A plan that achieves **all goals**
must be a valid solution for any **subset**
➔ This is a relaxation

The h_1 Heuristic: Example

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

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)

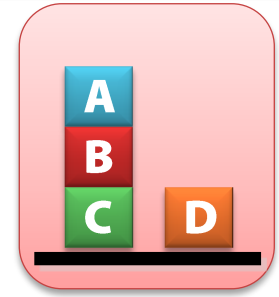
handempty	*clear(B)*
cost 0	cost 0

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_1 Heuristic: Important Property 1



on(B,C)
cost 2

Each goal considered separately!

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)*

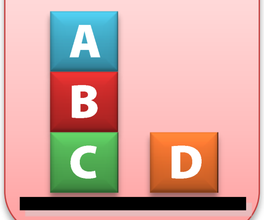
We don't search for a **valid plan** achieving on(B,C)!

Then we would need putdown(A)…

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

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

The h_1 Heuristic: Important Property 2



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

stack(A,B)

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

stack(B,C)

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

unstack(A,C)

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

Cheaper!

The same action can "occur" twice!

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

unstack(A,C)

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

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 ∈ g may actually delete p2 ∈ 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

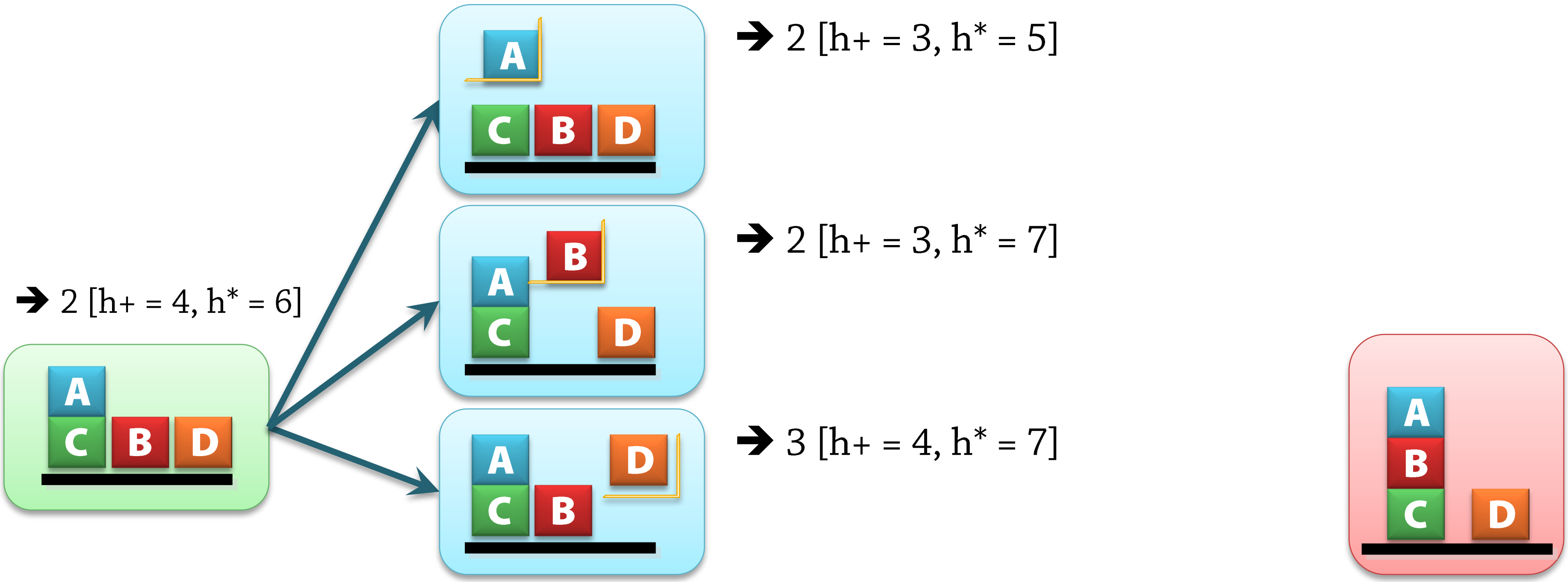
- For a **single proposition** p to be achieved:
  - $\Delta_1(s, p)$ = the cost of **achieving p from s**
    - $\Delta_1(s, p) = 0$ if $p \in s$ // *Already achieved p*
    - $\Delta_1(s, p) = \infty$ if $\forall a \in A.\ p \notin effects^+(a)$ // *Unachievable*
    - Otherwise:
      $\Delta_1(s,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

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*:
  - Δ_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:
  - **Easier** to calculate than the optimal delete relaxation heuristic h+
  - **Admissible** (never overestimates the cost)
  - Somewhat **useful** for this simple BW problem instance
  - **Not sufficiently informative** in general

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
- $\Delta_2(s, g) = \textbf{max} \{ \Delta_2(s,p,q) \mid p,q \in g \}$ 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 $\forall a \in A.\ q \notin effects^+(a)$
- $\Delta_2(s, p, q) = \textbf{min} \{$
  $\min \{ cost(a) + \Delta_2(s, precond(a)) \mid a \in A \text{ and } p,q \in effects^+(a) \}$,
  $\min \{ cost(a) + \Delta_2(s, precond(a) \cup \{q\}) \mid a \in A,\ p \in effects^+(a),\ q \notin effects^-(a) \}$,
  $\min \{ cost(a) + \Delta_2(s, precond(a) \cup \{p\}) \mid a \in A,\ q \in effects^+(a),\ p \notin effects^-(a) \}$
  $\}$

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

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 Delete Relaxation

- In the book:
  - $\Delta_2(s, p, q) =$ **<u>min</u>** {
    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 **<u>not</u>** how the heuristic is normally presented!
  - Corresponds to applying (full) delete relaxation
  - Fixed 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)

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

Cost of cheapest action taking you from s to s'

Accuracy of h_m in Selected Domains

- **<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 ➔ $h_m(n) \geq 0\, h^*(n)$
    - For any constant m!

Accuracy of h_m in Selected Domains (2)

- Consider a constructed **family of problem instances**:
  - 10*n* 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)

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

Still **useful** – this is a **worst-case** analysis as **sizes approach infinity**!
+ 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

- **<u>Experimental</u>** 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!

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 and HSP (Heuristic Search Planner)

Also called h_0

Background

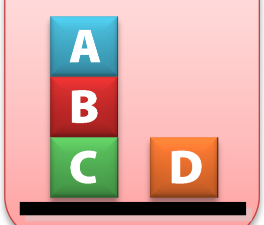
- h_m heuristics are **admissible**, but not very **informative**
  - Only measure the **most expensive** goal subsets

- For satisficing planning, we do not need admissibility
  - Let's consider a modification:
    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:

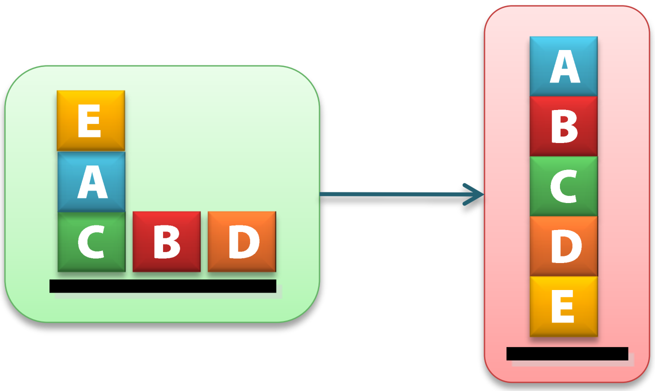
      $\Delta_0(s, p) = \min \{ cost(a) + \Delta_1(s, precond(a)) \mid a \in A \text{ and } p \in effects^+(a) \}$

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

- $h_{add}(s) = \Delta_0(s, g)$
  - 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!

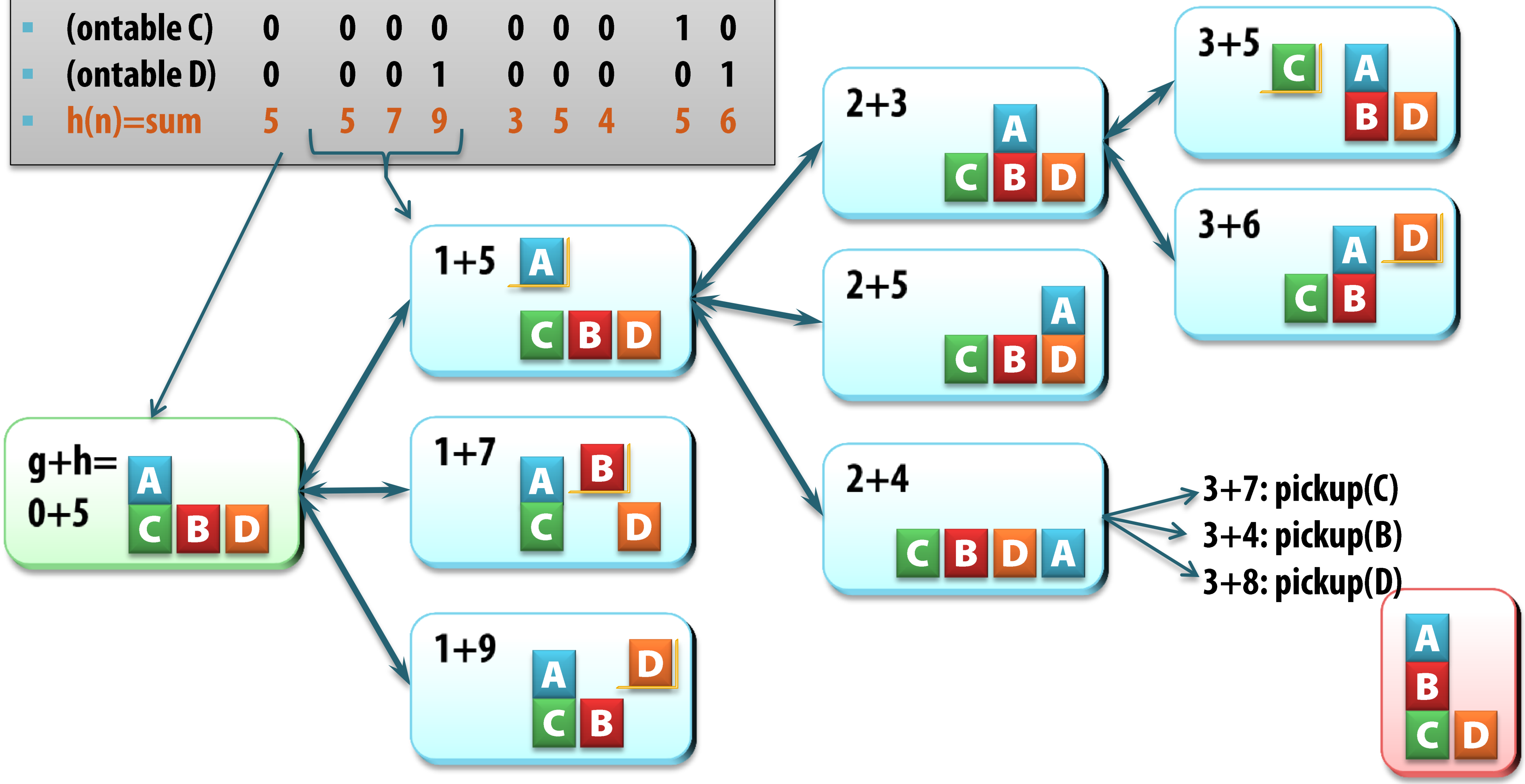
The h_{add} Heuristic: Admissibility

- 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: causes p
    - A2: causes q
    - A3: causes p and q
    - To achieve p: Use A1 – No specific action used twice
    - To achieve q: Use A2 – Still misses interactions

The h_{add} Heuristic: Using A*

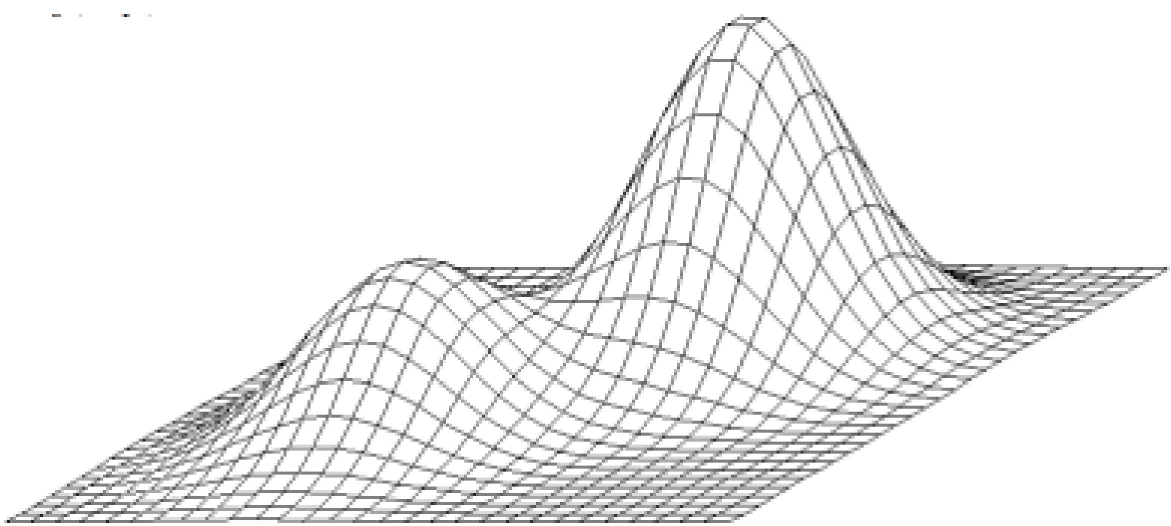
(on A B)	2	1	3	3	0	2	2	0	0
(on B C)	3	3	4	4	3	2	2	4	4
(clear A)	0	1	0	0	0	0	0	0	0
(clear D)	0	0	0	1	0	1	0	0	1
(ontable C)	0	0	0	0	0	0	0	1	0
(ontable D)	0	0	0	1	0	0	0	0	1
h(n)=sum	5	5	7	9	3	5	4	5	6

Seems to work well, but no optimality guarantee: h_{add} is **informative** but not **admissible**. Are there alternatives?



Hill Climbing (1)

- What about **Hill Climbing**?
  - **Greedy** algorithm:
    - Searches the **local neighborhood** around the current solution
    - Makes a **locally optimal** choice at each step
    - ➔ **Climbs the hill** towards the top,
      without exploring as many nodes as A*



Hill Climbing (2)

A* search:

n ← initial state
open ← ∅
loop
  if *n* is a solution then return *n*
  expand children of *n*
  calculate *h* for children
  add children to *open*
  n ← node in *open*
    minimizing f(n) = g(n) + h(n)
end loop

Plain Hill-climbing

n ← initial state

loop
  if *n* is a solution then return *n*
  expand children of *n*
  calculate *h* for children

  if (some child decreases h(*n*)):
    n ← child with minimal h(*n*)
  else stop // local minimum
end loop

Be stubborn: Only search among children of this node (like depth first), never mind other open nodes

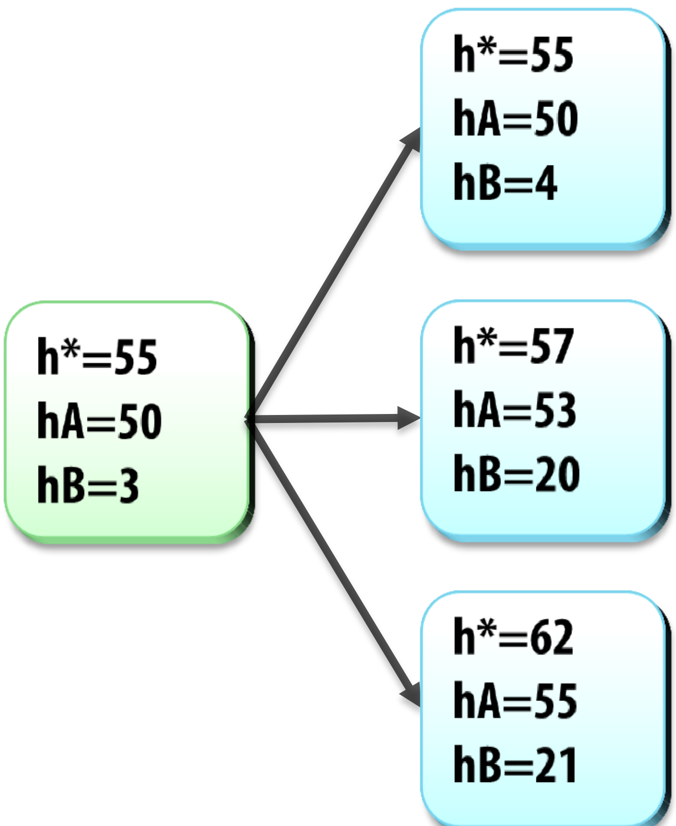
Ignore g(n): prioritize **finding a plan quickly** over **finding a good plan**

- Which objective function for planning?
  - –h(s): We want to minimize heuristic value

Heuristics for HC Planning

- What is a **good heuristic** for HC in **planning**?

Which is best, hA or hB?



Equally good!

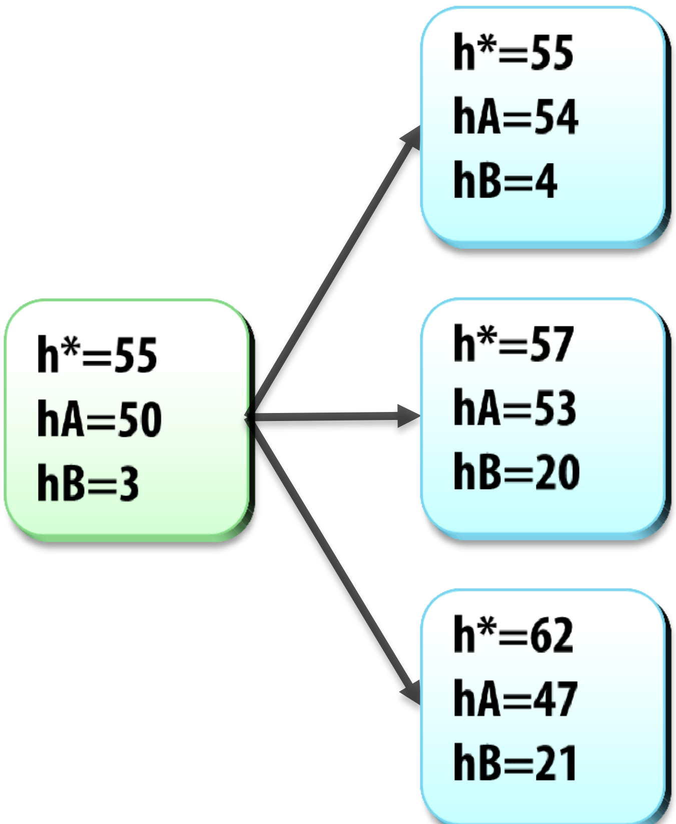
HC only cares about
the *relative* quality
of the children of one node…

For A*, hA is *much* better:
Much closer to real costs

Heuristics for HC Planning (2)

- What is a **good heuristic** for HC in **planning**?

Which is best, hA or hB?



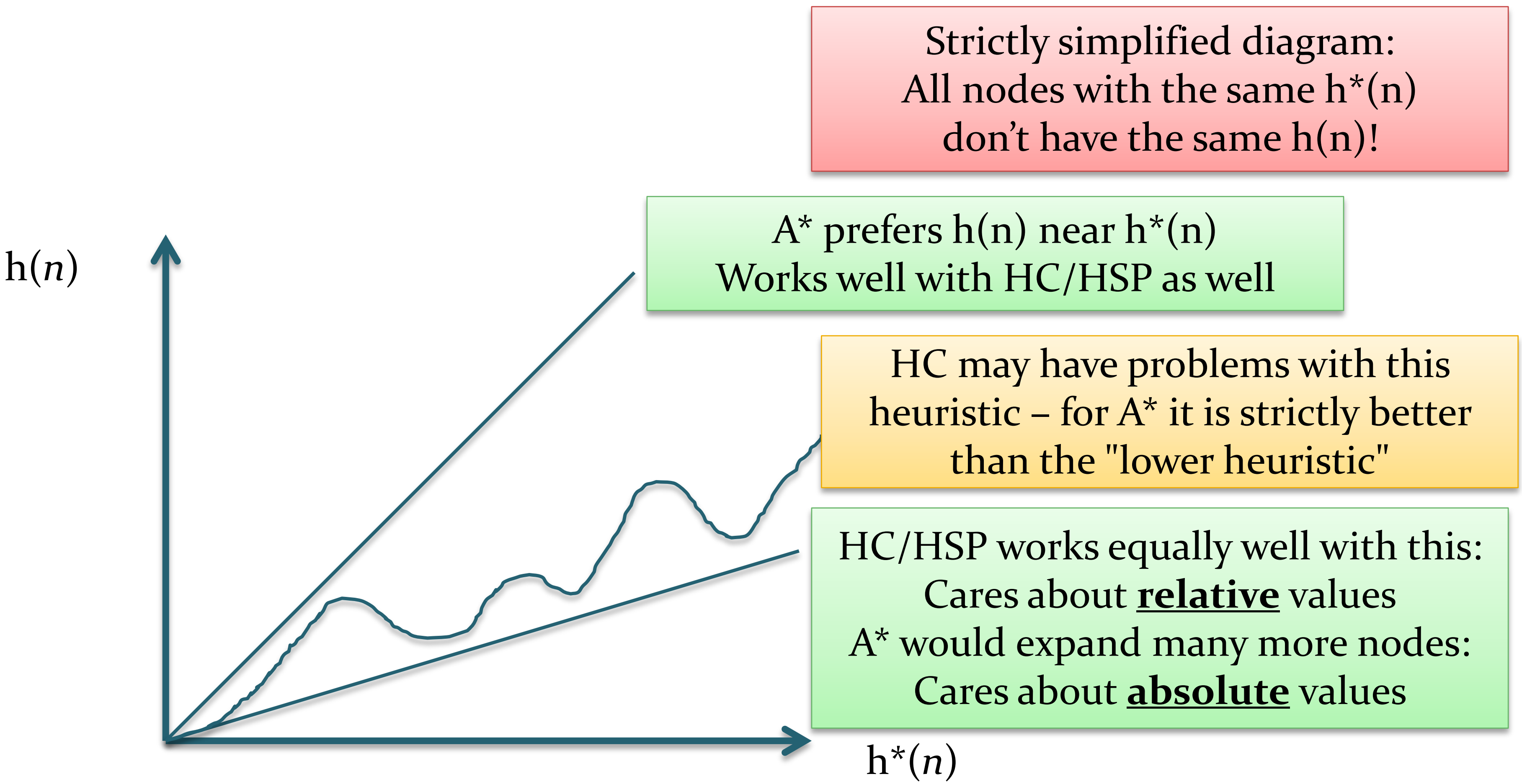
hB is better!

hA prioritizes children in the *opposite* order…

For A*, hA is *much* better: Much closer to real costs

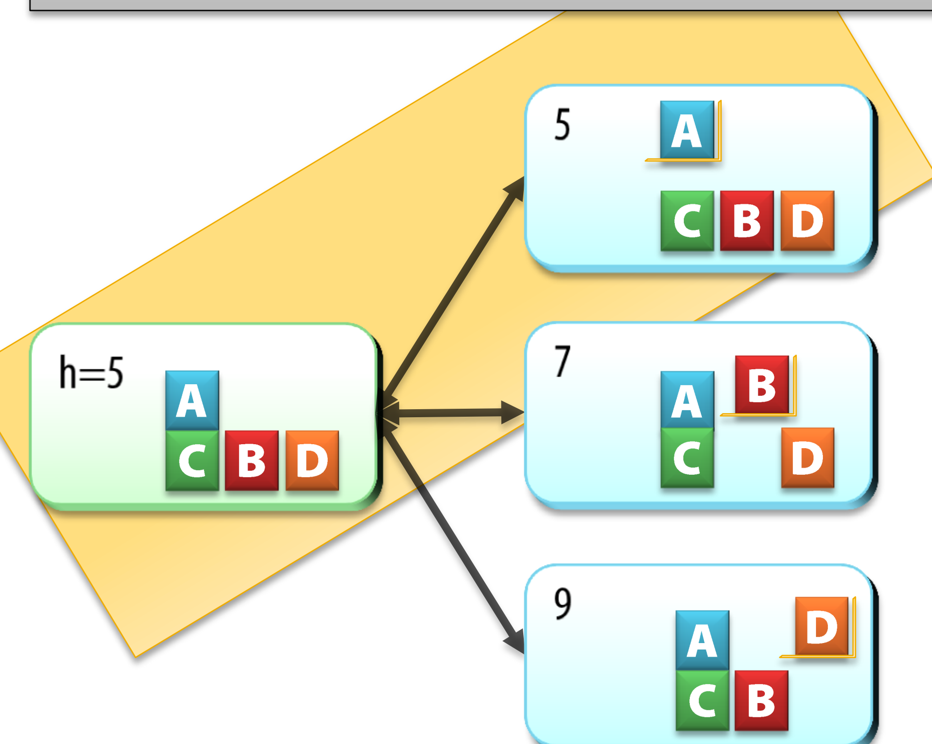
Heuristics for HC Planning (3)

- What is a **good heuristic** for HC in **planning**?



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 improves the heuristic value; some are equal!

We have a **plateau**...

Standard hill climbing:
"Can't improve ➔
Jump to a random state"

But the heuristic is not so accurate – maybe some child *is* closer to the goal even though h(n) isn't lower!

➔ Let's allow a small number of consecutive **moves across plateaus**

Plateaus

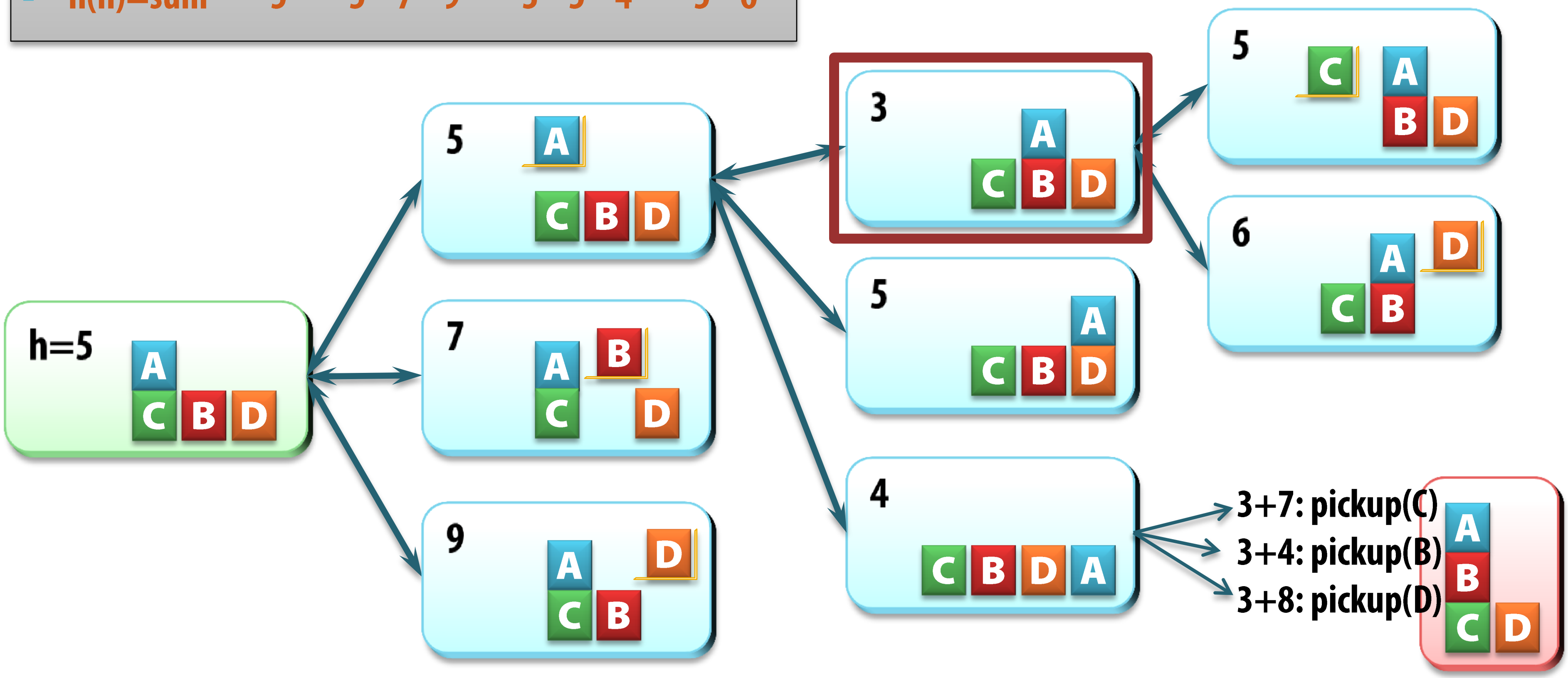
- A plateau…

Hill Climbing with h_{add}: Local Optima

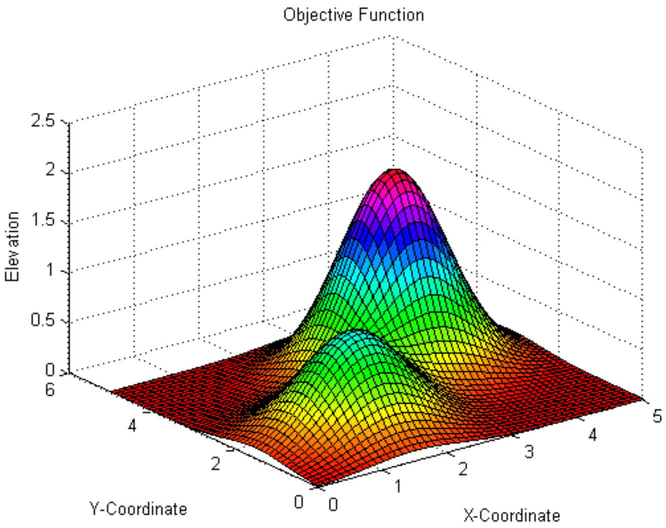
(on A B)	2	1	3	3	0	2	2	0	0
(on B C)	3	3	4	4	3	2	2	4	4
(clear A)	0	1	0	0	0	0	0	0	0
(clear D)	0	0	0	1	0	1	0	0	1
(ontable C)	0	0	0	0	0	0	0	1	0
(ontable D)	0	0	0	1	0	0	0	0	1
h(n)=sum	5	5	7	9	3	5	4	5	6

If we continue, all successors have higher heuristic values!

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



Local Optima

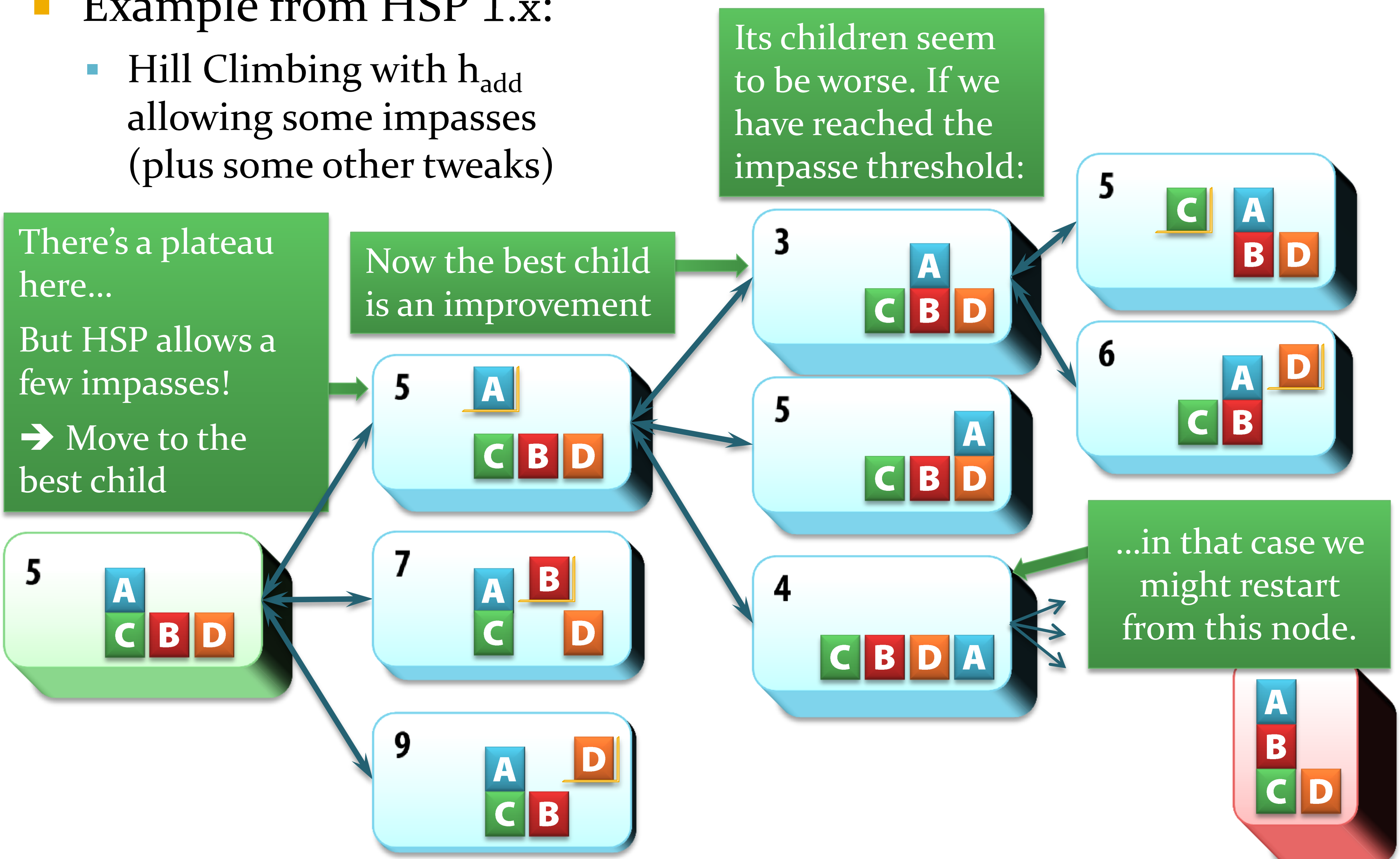


Impasses and Restarts

- What if there are **<u>many</u>** 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)



HSP 1: Heuristic Search Planner

- HSP 1.x: h_{add} heuristic + hill climbing + modifications
  - Works **approximately** like this (some intricacies omitted):
    - greedy = true; impasses = 0; unexpanded = { initialNode }; node = pop(unexpanded);

```
while (not yet reached the goal) {
    children ← expand(node);                          // Apply all applicable actions
    add children to unexpanded in order of h(n);      // Keep track of visited nodes for "random" restarts!
    if (|children| = 0) {                             // Dead end
        node = pop(unexpanded);                       // Restart from the next node (fail if none available)
    } else if (greedy) {
        best Child ← first(children);                 // Child with the lowest heuristic value, hill-climbing-style
        remove bestChild from unexpanded;
        if (h(bestChild) >= h(node)) {
            impasses++;
            if (impasses == threshold) greedy = false;
        }
    } else {
        node = pop(unexpanded);                       // Restart from another node (fail if none available)
        greedy = true;                                // Go back to hill-climbing search
        impasses = 0;
    }
}
```

Dead end ➔ restart

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

Too many downhill/plateau moves ➔ escape

Pure HC with limited domain-indep. heuristics ➔ jump around too much!
Allow limited downhill/plateau moves ➔ be a bit more persistent, but eventually try another path

HSP (2): Heuristic Search Planner

- Late 1990s: "State-space planning **too simple** to be efficient!"
  - Most planners used very elaborate and complex search methods

- HSP:
  - Simple search space: Forward-chaining
  - Simple search method: Hill-climbing with limited impasses + restarts
  - Simple heuristic: Sum of distances to propositions (still spends 85% of its time calculating h_{add}!)
  - ➔ Very clever combination

- **Planning competition** 1998:
  - HSP solved more problems than most other planners
  - Often required a bit more time, but still competitive
  - (Later versions were considerably faster)

An Overview of
Pattern Database Heuristics

Introduction

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

Subproblem for the h2 heuristic:

Pick two **goal literals**
Ignore the others
Solve the problem optimally

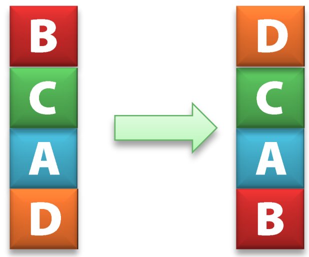
Subproblem for Pattern Database Heuristics

Pick some **state atoms**
Ignore the others
Solve the problem optimally

Database:
Solve for all values of the state atoms
Store in a database
Look up values quickly during search

Pattern Database Heuristics (1)

- **Pattern Database Heuristics**:
  - Example problem:



  - If you use the classical (predicate) representation:
    - Reduce state space size: Partition atoms into **mutually exclusive groups**
    - In all states **reachable** from s0 using available actions, exactly one atom in each group is true!

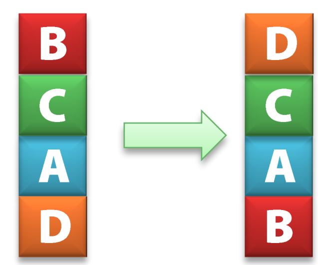
– $G_1 = \{$(on c a), (on d a), (on b a), (clear a), (holding a)$\}$,
– $G_2 = \{$(on a c), (on d c), (on b c), (clear c), (holding c)$\}$,
– $G_3 = \{$(on a d), (on c d), (on b d), (clear d), (holding d)$\}$,
– $G_4 = \{$(on a b), (on c b), (on d b), (clear b), (holding b)$\}$,
– $G_5 = \{$(ontable a), true$\}$,
– $G_6 = \{$(ontable c), true$\}$,
– $G_7 = \{$(ontable d), true$\}$,
– $G_8 = \{$(ontable b), true$\}$, and
– $G_9 = \{$(handempty), true$\}$,

{p} represents that p always holds,
{p,true} represents that
p may or may not hold

Pattern Database Heuristics (2)

- Every group can be seen as a single **state variable**
  - Variable G1 has 5 possible values:
    - v1, v2, v3, v4, v5
  - **Equivalent** way of viewing the problem!
    - (on c a) ⇔ G1 = v1
      (on d a) ⇔ G1 = v2
    - (on b a) ⇔ G1 = v3
    - Many modern planners work with this representation internally, even if they don't use PDBs

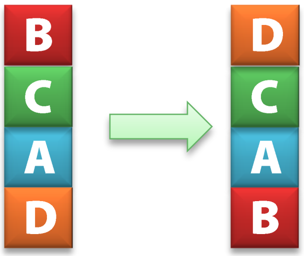
– G_1 = {(on c a), (on d a), (on b a), (clear a), (holding a)},
– G_2 = {(on a c), (on d c), (on b c), (clear c), (holding c)},
– G_3 = {(on a d), (on c d), (on b d), (clear d), (holding d)},
– G_4 = {(on a b), (on c b), (on d b), (clear b), (holding b)},
– G_5 = {(ontable a), true},
– G_6 = {(ontable c), true},
– G_7 = {(ontable d), true},
– G_8 = {(ontable b), true}, and
– G_9 = {(handempty), true},



Pattern Database Heuristics (2)

- Every group can be seen as a single **state variable**
  - Variable G1 has 5 possible values:
    - on-c-a, on-d-a, on-b-a, clear-a, and holding-a
  - **Equivalent** way of viewing the problem!
    - (on c a) ⇔ G1 = on-c-a
    - Many modern planners work with this representation internally, even if they don't use PDBs

- $G_1 = \{(\texttt{on c a}), (\texttt{on d a}), (\texttt{on b a}), (\texttt{clear a}), (\texttt{holding a})\}$,
- $G_2 = \{(\texttt{on a c}), (\texttt{on d c}), (\texttt{on b c}), (\texttt{clear c}), (\texttt{holding c})\}$,
- $G_3 = \{(\texttt{on a d}), (\texttt{on c d}), (\texttt{on b d}), (\texttt{clear d}), (\texttt{holding d})\}$,
- $G_4 = \{(\texttt{on a b}), (\texttt{on c b}), (\texttt{on d b}), (\texttt{clear b}), (\texttt{holding b})\}$,
- $G_5 = \{(\texttt{ontable a}), \texttt{true}\}$,
- $G_6 = \{(\texttt{ontable c}), \texttt{true}\}$,
- $G_7 = \{(\texttt{ontable d}), \texttt{true}\}$,
- $G_8 = \{(\texttt{ontable b}), \texttt{true}\}$, and
- $G_9 = \{(\texttt{handempty}), \texttt{true}\}$,



Pattern Database Heuristics (3)

- Why change the representation like this?
  - Original: 25 atoms, 2^25 = 33554432 states
  - Now: 5^4 * 2^5 = 20000 states
    - Remove a lot of "useless" unreachable states

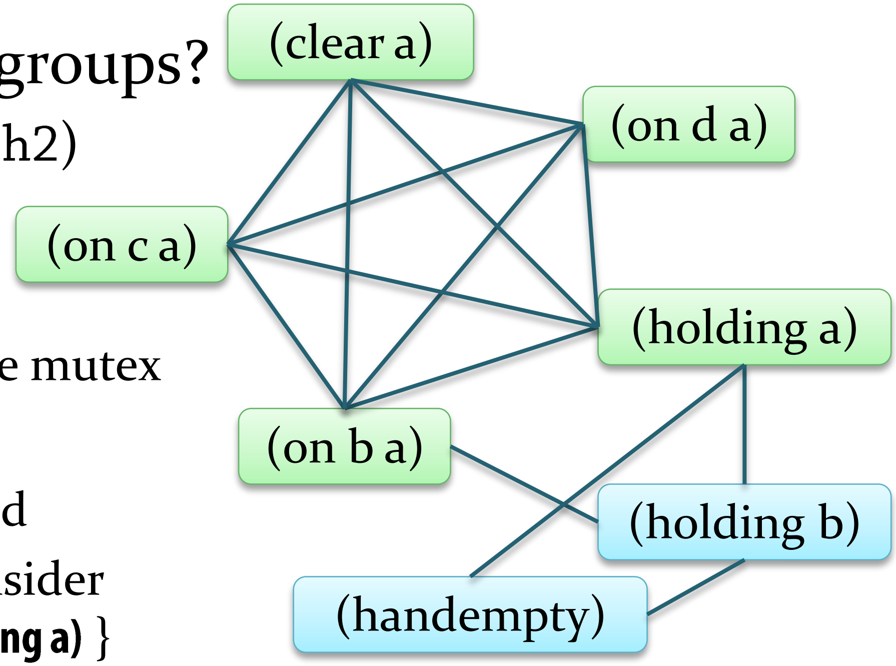
Not important for **search**: We would never have reached an unreachable state…

Helps when creating **pattern databases**

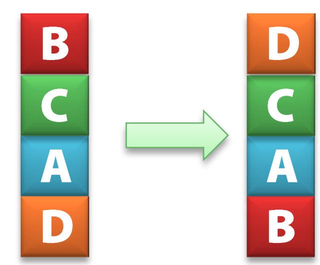
- $G_1 = \{$`(on c a)`, `(on d a)`, `(on b a)`, `(clear a)`, `(holding a)`$\}$,
- $G_2 = \{$`(on a c)`, `(on d c)`, `(on b c)`, `(clear c)`, `(holding c)`$\}$,
- $G_3 = \{$`(on a d)`, `(on c d)`, `(on b d)`, `(clear d)`, `(holding d)`$\}$,
- $G_4 = \{$`(on a b)`, `(on c b)`, `(on d b)`, `(clear b)`, `(holding b)`$\}$,
- $G_5 = \{$`(ontable a)`, `true`$\}$,
- $G_6 = \{$`(ontable c)`, `true`$\}$,
- $G_7 = \{$`(ontable d)`, `true`$\}$,
- $G_8 = \{$`(ontable b)`, `true`$\}$, and
- $G_9 = \{$`(handempty)`, `true`$\}$,

Pattern Database Heuristics (4)

- How to find mutually exclusive groups?
  - Find **pairwise** mutexes (e.g., using h2)
  - Create a graph:
    - One node per atom
    - Edge (p←→q) iff p and q are pairwise mutex
  - Find *maximal cliques*
    - Groups where *all* nodes are connected
    - Does not give a unique solution: Consider { **(on a b), (on a c), (on a d), (ontable a), (holding a)** }

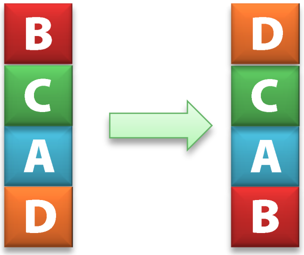


- $G_1 = \{$(on c a), (on d a), (on b a), (clear a), (holding a)$\}$,
- $G_2 = \{$(on a c), (on d c), (on b c), (clear c), (holding c)$\}$,
- $G_3 = \{$(on a d), (on c d), (on b d), (clear d), (holding d)$\}$,
- $G_4 = \{$(on a b), (on c b), (on d b), (clear b), (holding b)$\}$,
- $G_5 = \{$(ontable a), true$\}$,
- $G_6 = \{$(ontable c), true$\}$,
- $G_7 = \{$(ontable d), true$\}$,
- $G_8 = \{$(ontable b), true$\}$, and
- $G_9 = \{$(handempty), true$\}$,



Pattern Database Heuristics (5)

- A **<u>planning space abstraction</u>** "ignores" some groups
  - A mapping ϕ from atoms to atoms + {true}, where for each group *G*:
    - Either $\forall f \in G$: $\phi(f) = f$ – all atoms in the group are preserved
    - Or $\forall f \in G$: $\phi(f) = \text{true}$ – all atoms in the group are ignored
    - Results in an *exponentially smaller state space*
  - Suppose ϕ preserves all *even groups*
    - Real goal = { (on d c), (on c a), (on a b) }
    - Relaxed goal = { (on d c), true, (on a b) }
    - pickup(a):
      - No longer requires (ontable a): In group 5
      - No longer causes (holding a): In group 1
  - The resulting mini-problem is called a **<u>pattern</u>**
    - Matches many states that we might reach in the complete problem!



- $G_1 = \{$(on c a), (on d a), (on b a), (clear a), (holding a)$\}$,
- $G_2 = \{$(on a c), (on d c), (on b c), (clear c), (holding c)$\}$,
- $G_3 = \{$(on a d), (on c d), (on b d), (clear d), (holding d)$\}$,
- $G_4 = \{$(on a b), (on c b), (on d b), (clear b), (holding b)$\}$,
- $G_5 = \{$(ontable a), true$\}$,
- $G_6 = \{$(ontable c), true$\}$,
- $G_7 = \{$(ontable d), true$\}$,
- $G_8 = \{$(ontable b), true$\}$, and
- $G_9 = \{$(handempty), true$\}$,

Pattern Database Heuristics (6)

- Using these abstractions for **heuristics** – general idea:
  - Automatically generate a set of planning space abstractions
    - Set of selections of groups/variables
    - Difficult issue – different approaches exist
  - Each abstraction results in a **much smaller** abstract state space
    - Complete state space: 5^4 * 2^5 = 20000 states
    - Abstraction containing *all even groups*: 5*5*2*2 states = 100 states

– $G_1 = \{$(on c a), (on d a), (on b a), (clear a), (holding a)$\}$,
– $G_2 = \{$(on a c), (on d c), (on b c), (clear c), (holding c)$\}$,
– $G_3 = \{$(on a d), (on c d), (on b d), (clear d), (holding d)$\}$,
– $G_4 = \{$(on a b), (on c b), (on d b), (clear b), (holding b)$\}$,
– $G_5 = \{$(ontable a), true$\}$,
– $G_6 = \{$(ontable c), true$\}$,
– $G_7 = \{$(ontable d), true$\}$,
– $G_8 = \{$(ontable b), true$\}$, and
– $G_9 = \{$(handempty), true$\}$,

Pattern Database Heuristics (7)

- For each abstraction, compute a **pattern database**
  - Exhaustive search: Cheapest way of achieving **any** state in the pattern
    - Assigns a cost to each *abstract state*
  - To be computable in polynomial time:
    - Each individual pattern must have at most *logarithmic size*

- To **calculate a heuristic**:
  - From the current state, generate the **corresponding abstract state**
  - Look up its **precalculated cost**
    - Using perfect hash function: Near constant time lookups

  - Each such cost is an admissible heuristic
    - Therefore the **maximum** over many different abstractions is also an admissible heuristic

Pattern Database Heuristics (8)

- **<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
    - So h(n) <= cost of reaching the most expensive subgoal of size O(log n)
  - Problem size grows much faster than h(n)
    - ➔ For a *single* pattern, asymptotic accuracy is 0

Example

- Example:
  - pickup(A) affects holding(A), ontable(A), clear(A), handempty
  - If we use pickup(A) in abstraction 1:
    - It must affect some fact that is part of abstraction 1
  - "Suppose every action affects atoms in at most *one* of them"
    - So pickup(A) can't affect any atom used in abstraction 2
    - So it isn't used in any optimal plan in abstraction 2

Pattern Database Heuristics (9)

- ➔ Given several abstractions:
  - Suppose every action affects atoms in at most *one* of them
    - Then optimal solutions from distinct abstractions can't share actions
    - Therefore, the abstractions are *additive:* The **sum** of the corresponding heuristics is admissible

- If 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

Pattern Database Heuristics (10)

- **How close** to h*(n) can an admissible 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

  - **Assuming** that the planner finds the best combination of abstractions!

An Overview of Landmark Heuristics

Landmark Heuristics (1)

<u>Landmark</u>:

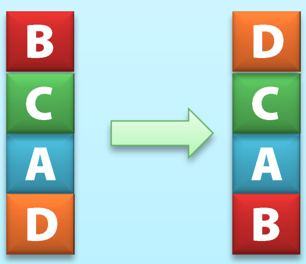
”a geographic feature used by explorers and others to find their way back or through an area”

Landmark Heuristics (2)

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

Landmark:

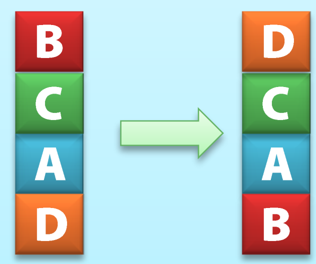
A **formula** that must be achieved
in *every* solution



clear(A)
holding(C)
...

Action Landmark:

An **action** that must be used
in *every* solution



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

...so their preconds and effects are *landmarks*!

Landmark Heuristics (3)

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

Current planning problem, P

Modified planning problem, P'

Removed all actions adding atom A

If this (P') is unsolvable...

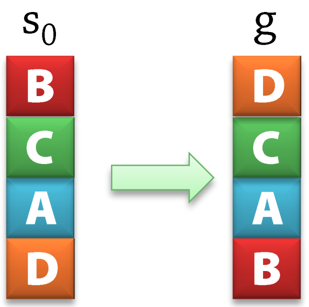
Delete relaxation of P' is unsolvable,
or $h_m(s_o) = \infty$,
or ...
➔ P' is unsolvable

...then **every** solution to P must use one of the actions adding A

➔ **Atom A is a landmark**

Landmark Heuristics (4)

- Discover landmarks using (1) **means-ends analysis**



The goals are (obviously) landmarks:
clear(D), on(D,C), on(C,A), on(A,B), ontable(A)

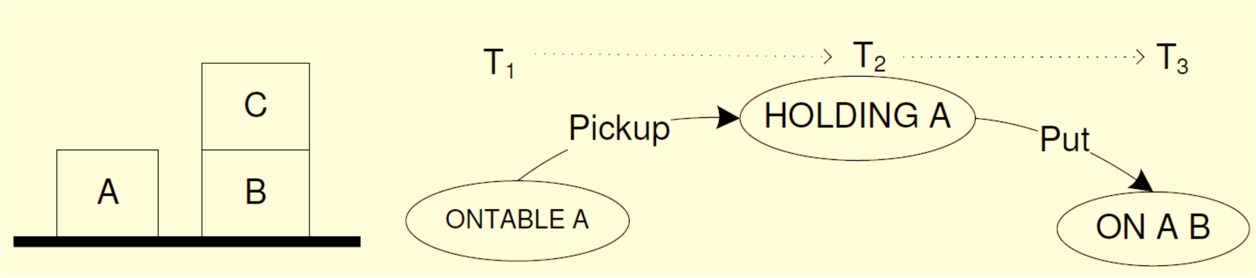
on(D,C) is a landmark,
on(D,C) is not true in the current state (s0)
➔ we must *cause* on(D,C) with an action

All actions causing on(D,C) require holding(D)
➔ **holding(D) is a landmark**!

holding(D) is not true in the current state,
all actions causing holding(D) require handempty
➔ **handempty is a landmark**

Landmark Heuristics (5)

- Discover landmarks using (2) **domain transition graphs**
  - Use **state variables**, or generate mutually exclusive sets of atoms
    - { ontable(A), holding(A), on(A,B) }
  - Add **transitions** caused by actions



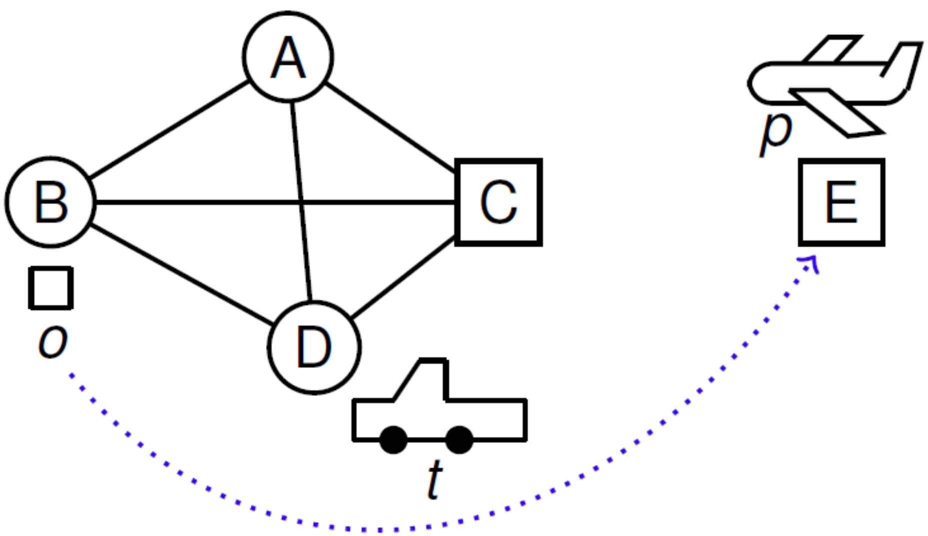
  - ➔ If A is on the table **now** and must be on B **in the goal**, then at some point we must be holding A (all paths pass through this node!)

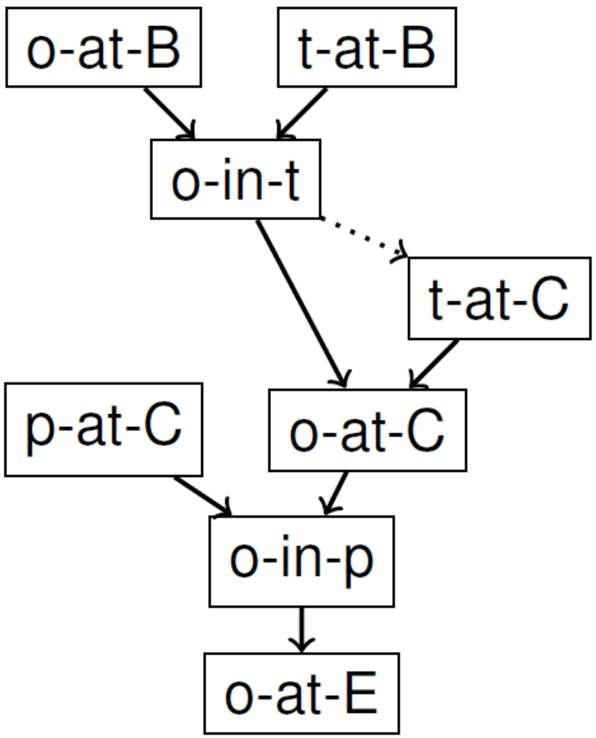
- ...and other methods.
- Can sometimes find or approximate **necessary orderings**
  - We must achieve holding(A), *then* holding(B)

Using Landmarks as Subgoals

Landmarks as Subgoals (1)

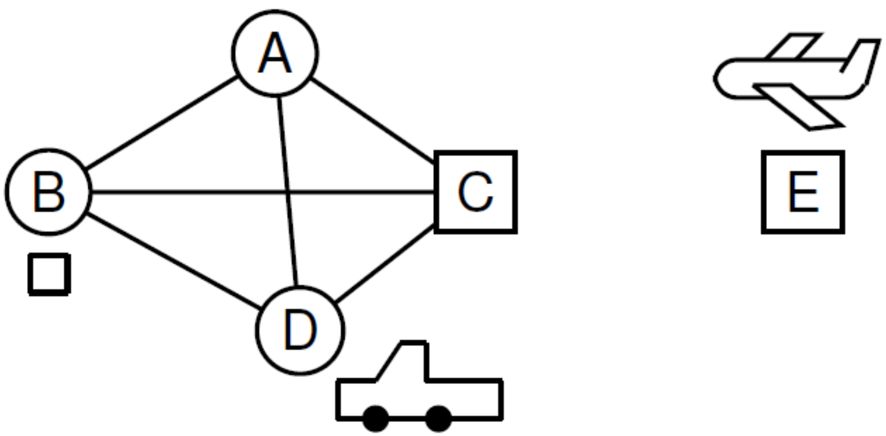
- Use of landmarks:
  - As **<u>subgoals</u>**: Try to achieve each landmark in succession, using inferred landmark orderings
    - Example from *Karpas & Richter: Landmarks – Definitions, Discovery Methods and Uses*

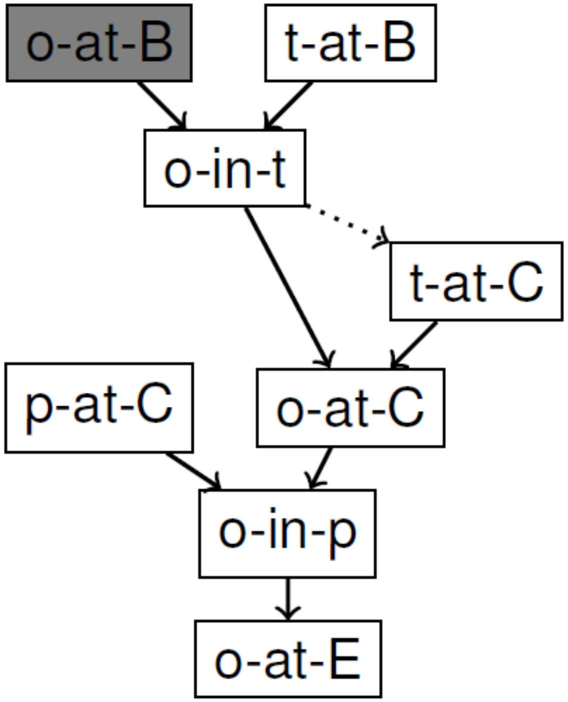




Landmarks as Subgoals (2)

Already true when we start

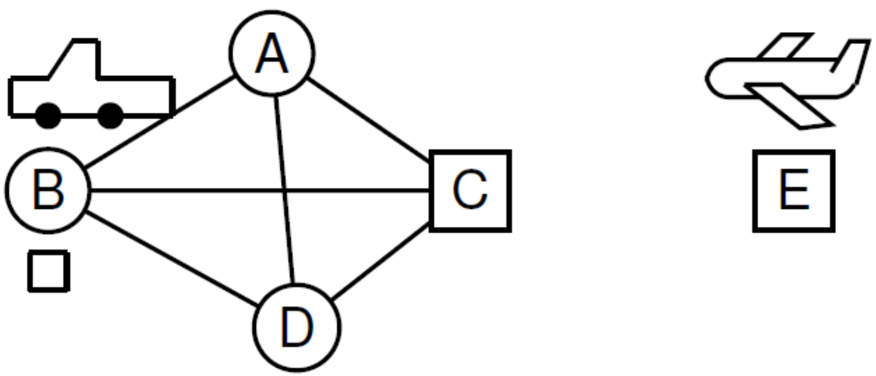


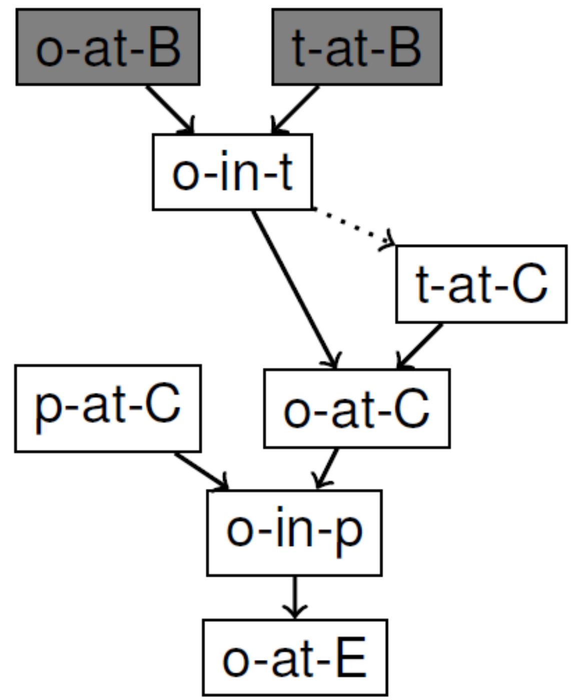


Current goal: t-at-B *or* p-at-C (disjunctive!)

Landmarks as Subgoals (3)

Suppose we begin with drive(t, B)

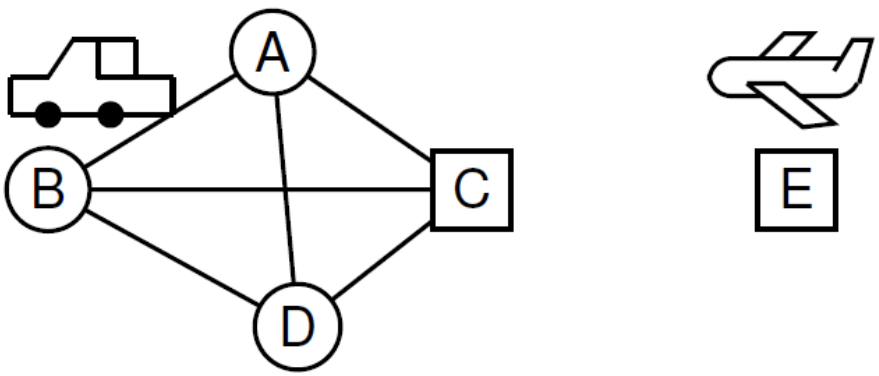


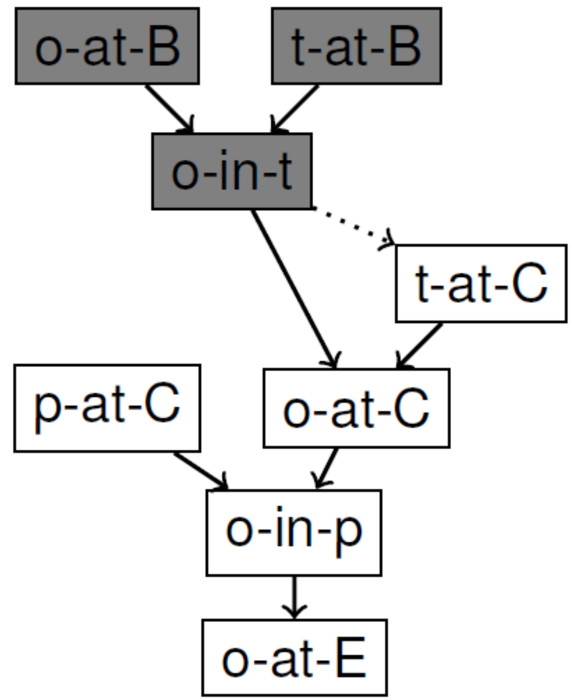


Current goal: o-in-T *or* p-at-C

Landmarks as Subgoals (4)

Suppose we continue with load-truck(o,t,B)





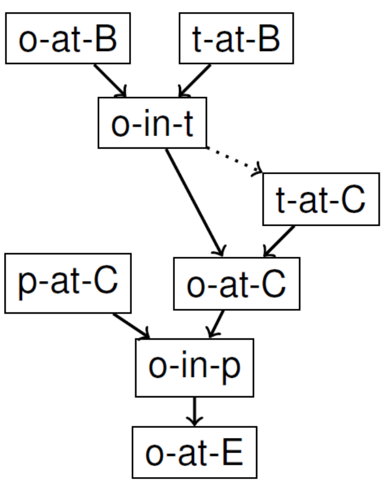
Landmarks as Subgoals (5)

- Sometimes very helpful
  - But there are choices to be made
  - Simply achieving each landmark in some permitted order can lead to long plans or even incompleteness…

Landmark Counts and Costs

Landmark Counts and Costs (1)

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



(L \ Accepted(s,π)): All discovered landmarks, minus those that are *accepted* as achieved (has become true *after* predecessors are achieved!)

ReqAgain(s,π): 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 planner**:
  - Won the *sequential satisficing* track
    of the International Planning Competition in 2008, 2011
  - **Heuristics combining**:
    - FF heuristics (discussed later)
    - The **number** of landmarks still to be achieved in a state
  - 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

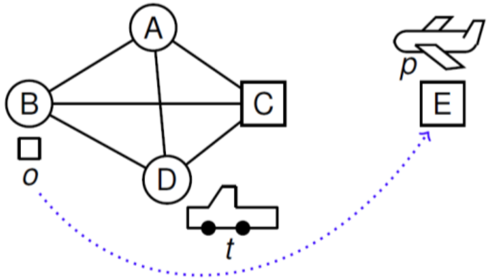
  - **Search strategy**:
    - First, **greedy best-first** (create a solution as quickly as possible)
    - Then, **repeated weighted A*** search with decreasing weights
      (iteratively improve the plan – anytime planning)

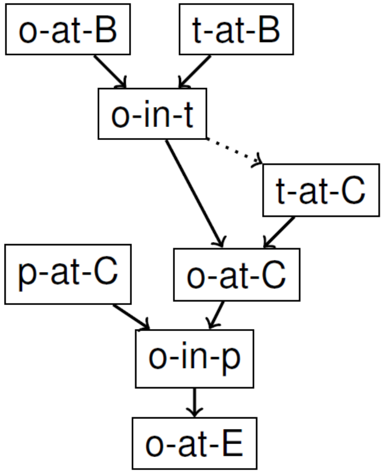
Landmark Counts and Costs (3)

- Use of landmarks:
  - As a basis for **admissible heuristic estimates**

  - Idea: The cost of each action is *shared* 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

- Use of landmarks:
  - As a basis for a **modified planning problem**
    - For example, add new predicates "achieved-landmark-*n*"
    - Each action achieving a landmark makes the corresponding predicate true
    - The goal requires all such predicates to be true
    - ➔ Other heuristics can be applied to the modified problem