Linköping University

Automated Planning

General Search Strategies

Assumes you have some previous experience of search algorithms!

Jonas Kvarnström
Department of Computer and Information Science
Linköping University

jonas.kvarnstrom@liu.se – 2019

Important distinction:
Optimizing / Satisficing

Optimal Planning

- **<u>Optimal</u>** plan generation:
  - There is a **<u>quality measure</u>** for plans
    - (Minimal number of actions)
    - Minimal **sum of action costs**
    - …
  - We **<u>must</u>** find an optimal plan!
    - Suboptimal plans
      (0.5% more expensive):

Guaranteeing optimality is sometimes useful, always expensive

Satisficing Planning

- **<u>Satisficing</u>** (satisfy/suffice) in general:
  - *"Searching until an acceptability threshold is met"*
  - Motivation: High-quality non-optimal solutions are also useful
    - And can often be found in reasonable time

- Satisficing in **<u>planning</u>** (typically):
  - No well-defined threshold: **Any form of non-optimal planning**
  - *Try to find strategies and heuristics*
    that seem reasonably quick
    and give reasonable results in our tests

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

Important distinction:
Informed / Uninformed

Informed / Uninformed Search

- **Uninformed** search strategies:
  - No domain-specific knowledge
  - Can only take into account **search space structure** and **cost so far**
    - $g(n)$ = cost of reaching node n from starting point

- **Informed** search strategies:
  - Take additional information into account, such as heuristics

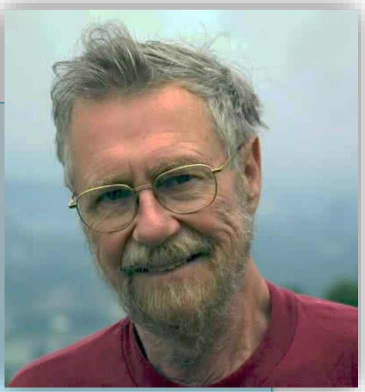
Applicable to all search spaces we have seen

May work *better* in some of them…

Dijkstra's Algorithm
(Optimal, Uninformed)

Dijkstra's Algorithm

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



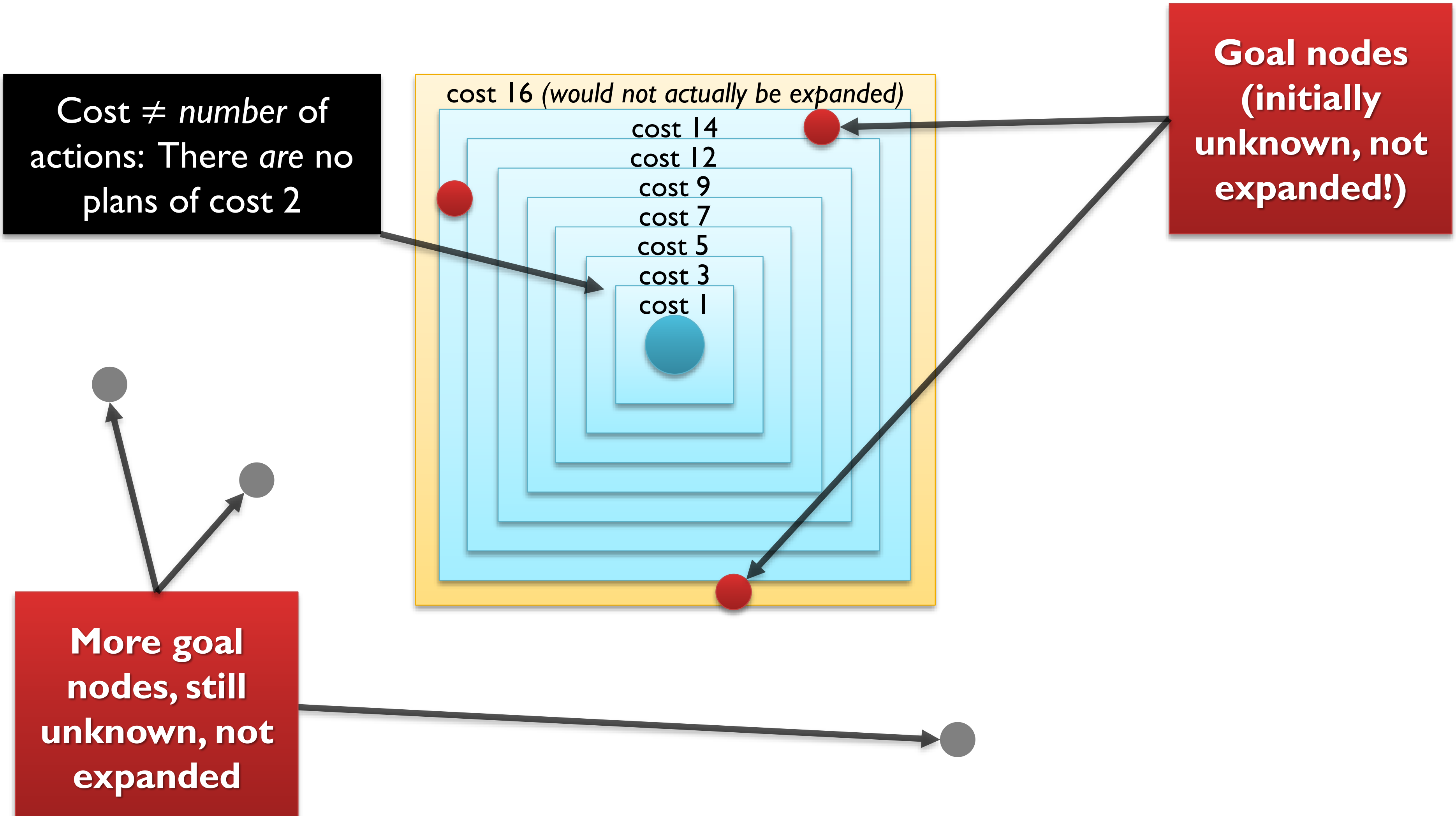
  - **Matches** the given forward search "template"
    - **use a strategy to select** and remove *node* from *open*
    - Selects a node n with minimal $g(n)$:
      Cost of reaching n from the initial node
  - **Efficient** graph search algorithm: $O(|E| + |V| \log|V|)$
    - $|E|$ = the number of edges (transitions), $|V|$ = the number of nodes (states)
  - **Optimal**: Returns minimum-cost plans

```
search(problem) {
    initial-node ← make-initial-node(problem) // [2]
    open ← { initial-node }
    while (open ≠ ∅) {
        node ← search-strategy-remove-from(open) // [6]
        if is-solution(node) then // [4]
            return extract-plan-from(node) // [5]
```

Typical implementation: **Priority queue**

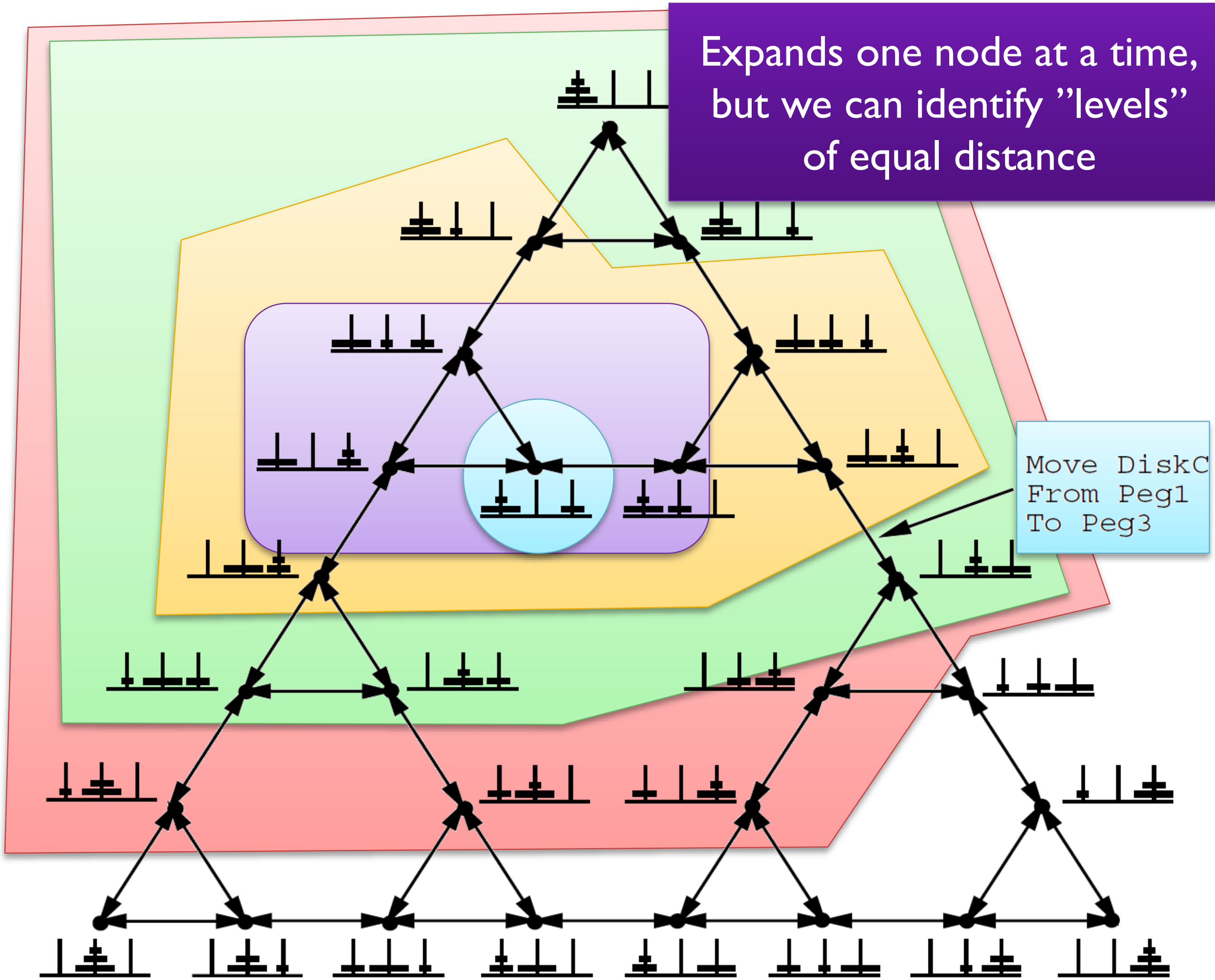
Dijkstra's Algorithm

- Explores nodes in order of cost



Dijkstra: ToH

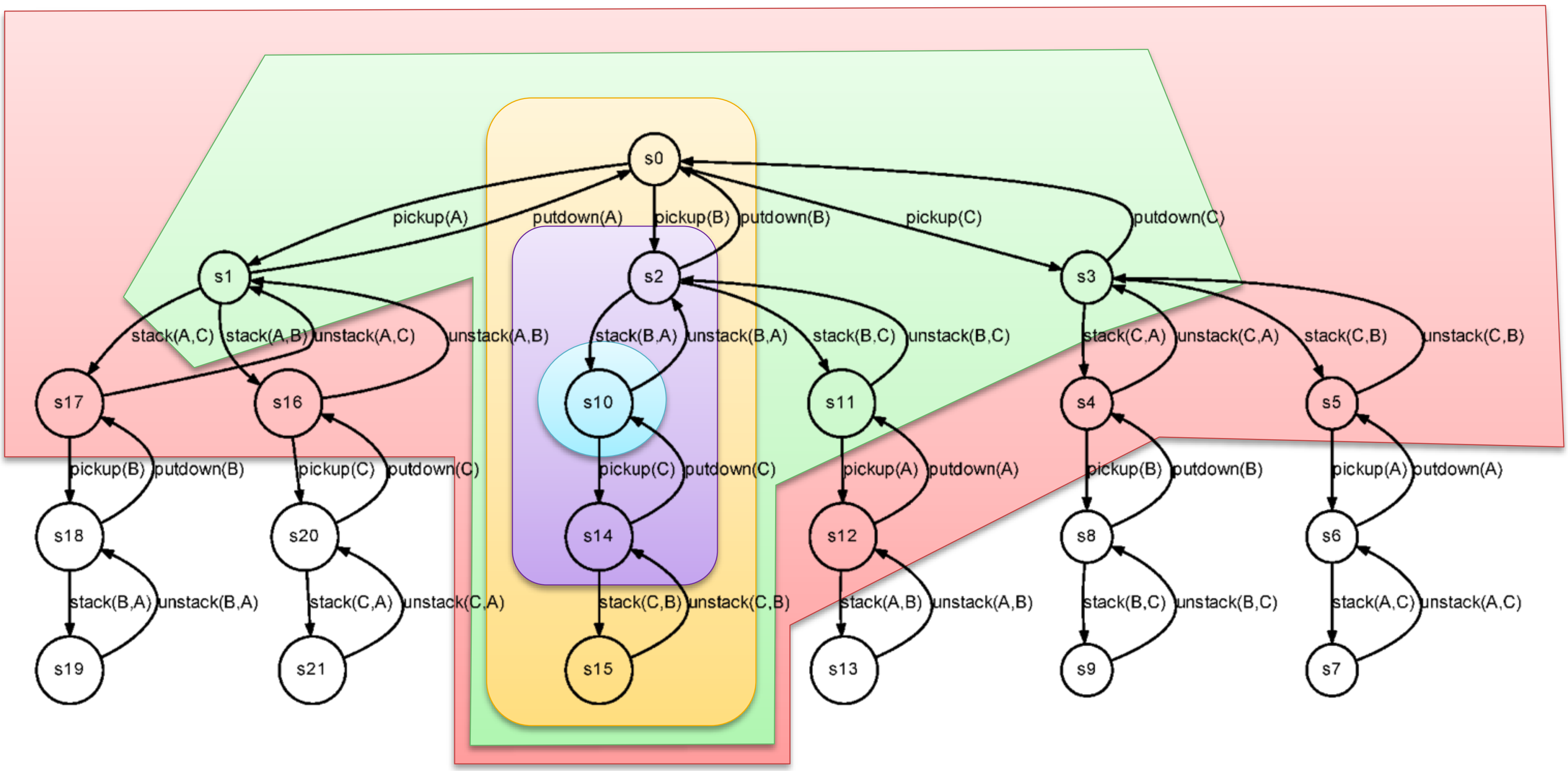
- Running Dijkstra, assuming all ToH actions are equally expensive:



Expands one node at a time, but we can identify "levels" of equal distance

Dijkstra: Blocks World

- Running Dijkstra, assuming all BW actions are equally expensive:

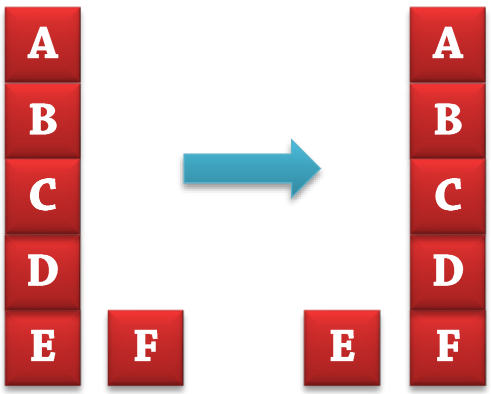


No problems?

Dijkstra's Algorithm
and the Difficulty of Planning

Dijkstra's Algorithm: Example

- A small instance:



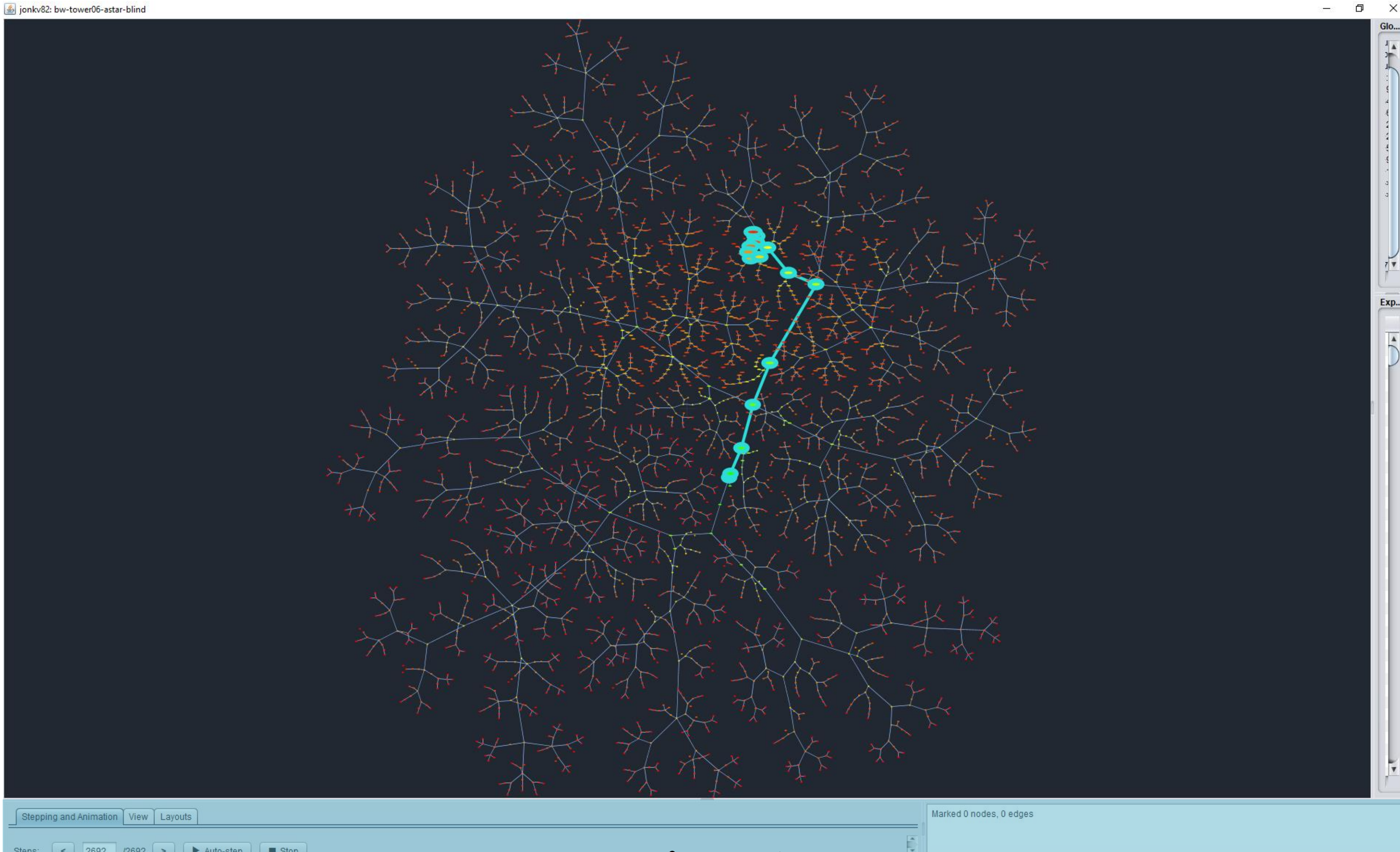
Goal

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

Optimal solution

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

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



Actions: 14

States: 8706 created, 2692 visited/expanded

400 blocks

- Blocks world, 400 blocks
  - Standard formulation: $2^{n^2+3n+1} = 2^{161201} > 10^{48526}$ states
  - **But we don't have to visit every one… Fewer reachable states!**

400 blocks

- Blocks world, 400 blocks initially on the table, goal is a 400-block tower
  - Given state space search with uniform action costs (same cost for all actions), 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, …
  - Will **visit** more than
    1630569839078931058645796793733472877564594841634782672258624197623042639942079976642582139557665811636541371 18
    163119220488226383169161648320459490283410635798745232698971132939284479800304096674354974038722588873480963719
    240642724363629154726632939764177236010315694148636819334217252836414001487277618002966608761037018087769490614
    847887418744402606226134803936935233568418055950371185351837140548515949431309313875210827888943337113613660928
    318086299617953892953722006734158933276576470475640607391701026030959040303548174221274052329579637773658722452
    549738459404452586503693169346 091275485326579590911344408444175566421 1796
    274320256992992317773749830373 488265744484456318793090777966157299028 9194
    810585217819146476629300233604 137235056874866524902199184976064698803 1691
    394386551194171193333144031542 130264943230562021556885065768422967838 5177
    725358933986112127352452988033 308720174243236072916252738750807322557 8630
    777685901637435541458440833878709344174983977437430327557534417629122448835191721077333875230695681480990867109
    051332104820413607822206465635272711073906611800376194410428900071013695438359094641682253856394743335678545824
    320932106973317498515711006719985304982604755110167254854766188619128917053933547098435020659778689499606904157
    077005797632287669764145095581565056589811721520434612770594950613701730879307727141093526534328671360002096924
    483494302424649061451726645947585860104976845534507479605408903828320206131072217782156434204572434616042404375
    21105232403822580540571315732915984635193126556273109603937188229504400 states

$1.63 * 10^{1735}$

Dijkstra is 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)
- If we don't guarantee optimality: **Depth first search**?
  - *Could* be faster, by pure luck…
    but normally finds **very** inefficient plans

The state space is fine,
but we need some *guidance*!

Best First Search
(a general idea)

Best First Search: A Very General Idea


```
search(problem) {
    initial-node ← make-initial-node(problem) // [2]
    open ← { initial-node }
    while (open ≠ ∅) {
        node ← search-strategy-remove-from(open) // [6]
        if is-solution(node) then  // [4]
            return extract-plan-from(node) // [5]

        foreach newnode ∈ successors(node) { // [3]
            add newnode to open
        }
    }
    // Expanded the entire search space without finding
    return failure;
}
```

Keep track of a set of open nodes

Use a heuristic function $h(node)$ to select the open node that seems "best"

(As opposed to depth first, breadth first, … which only consider tree structure!)

(As opposed to hill climbing and others that "throw away" nodes instead of keeping all nodes in open)

(As opposed to Dijkstra's algorithm etc, considering cost so far but having no idea where to go next)

Greedy Best First Search
(Non-Optimal, Informed, Greedy)

Greedy Best First


```
search(problem) {
    initial-node ← make-initial-node(problem) // [2]
    open ← { initial-node }
    while (open ≠ ∅) {
        node ← search-strategy-remove-from(open) // [6]
        if is-solution(node) then  // [4]
            return extract-plan-from(node) // [5]

        foreach newnode ∈ successors(node) { // [3]
            add newnode to open
        }
    }
    // Expanded the entire search space without finding
    return failure;
}
```

Choose an open node that minimizes $h(n)$

Ignore the cost of reaching the node, g(n)

Try to minimize the (apparent) amount of <u>search</u> left to do

A* – Another Best First Search Algorithm (Optimal, Informed, Non-Greedy)

A* (1)

- Optimal Plan Generation: Often uses **A***
  - A* focuses **entirely** on **optimality**
    - Expand from the initial node, systematically checking possibilities
    - No point in trying to find a "reasonable" plan *before* the optimal one!
  - Requires **admissible** heuristics to guarantee optimality: $\forall n.\, h(n) \leq h^*(n)$
    - Reason: Heuristic used for *pruning* (skipping some search nodes + all descendants)

$h^*(n)$ = cost of *optimal* plan from n

Essential: How does admissibility help?

Suppose we found a solution,
exact cost = 12

Another node, n:
g(n) = cost of reaching node = 10
h(n) = heuristic value = 5

h(n) admissible,
never overestimates,
so any solution found from here
would cost at least 10+5=15

No need to investigate
successors of this node!

A* (2)

- **A* strategy:**
  - Pick nodes from **open** in order of increasing $f(n) = g(n)$ [actual cost] + $h(n)$ [heuristic]
  - Works like a priority queue

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

If a heuristic never <u>under</u>estimates costs:

Suppose we found a solution,
exact cost = 12

Another node, n:
g(n) = cost of reaching node = 10
h(n) = heuristic value = 5

h(n) never <u>under</u>estimates,
so any solution found from here
would cost at <u>most</u> 10+5=15

Doesn't help!

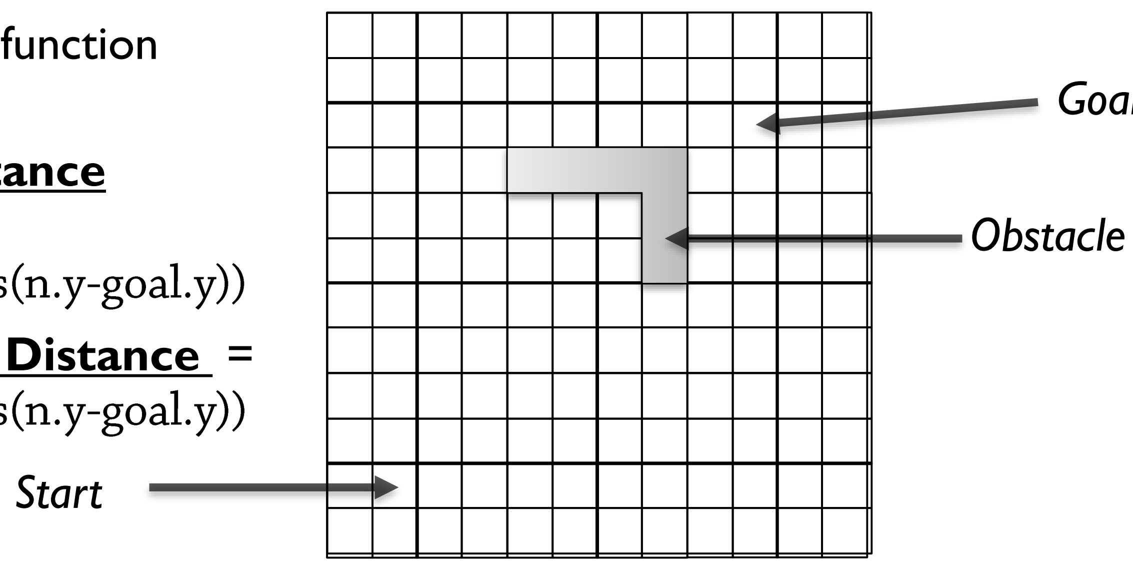
Could find solutions of cost 10
as descendants of node n,
must keep searching

A* (3)

- Dijstra vs. A*: The essential difference

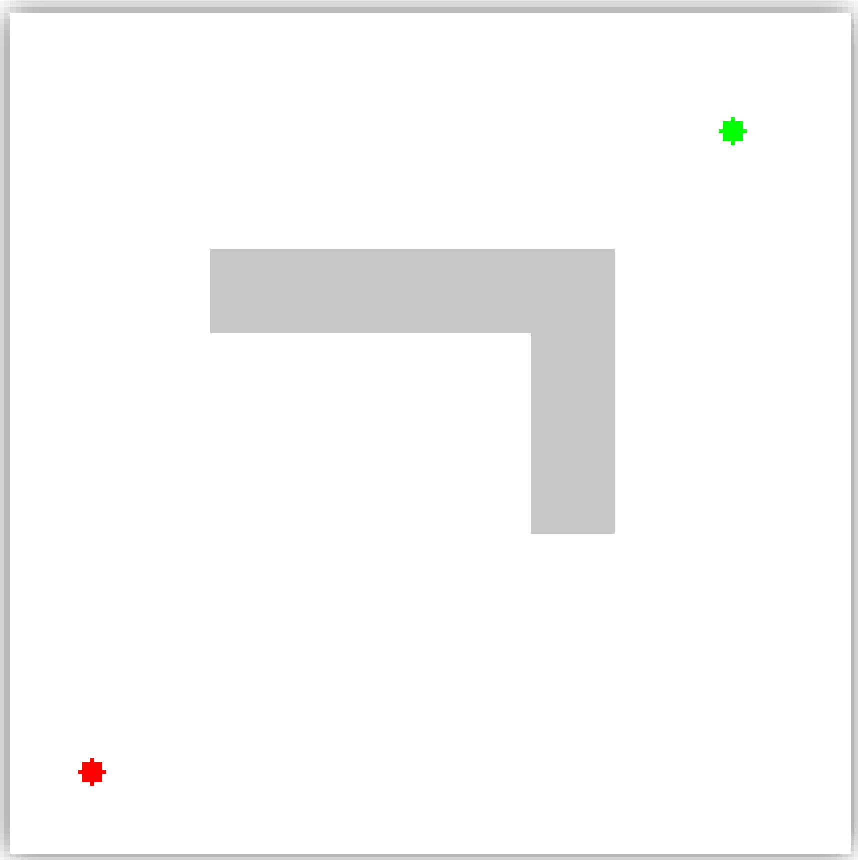
Dijkstra	A*
Selects from *open* a node *n* with minimal g(*n*) – Cost of reaching *n* from initial node	Selects from *open* a node *n* with minimal g(*n*) **+ h(*n*)** – + **underestimated cost of reaching a goal from *n***
Uninformed (blind)	**Informed**

- 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* (4)

- A* Search:



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

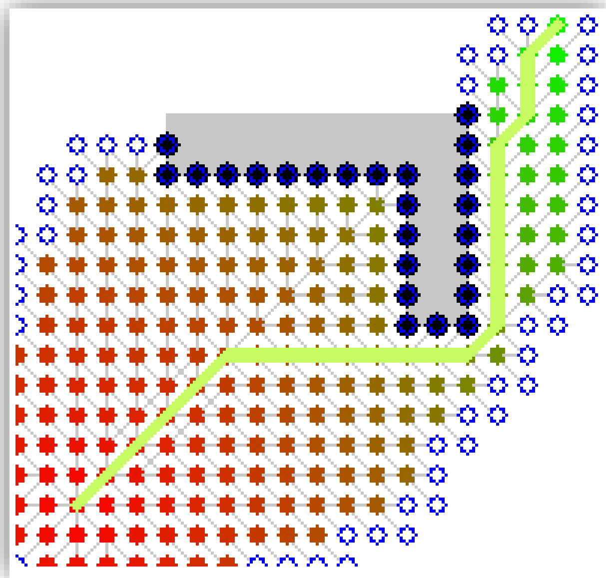
<u>In general</u>:
Many nodes where
all successors will
increase g+h
(cost + heuristic)

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

A* (5)

- 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

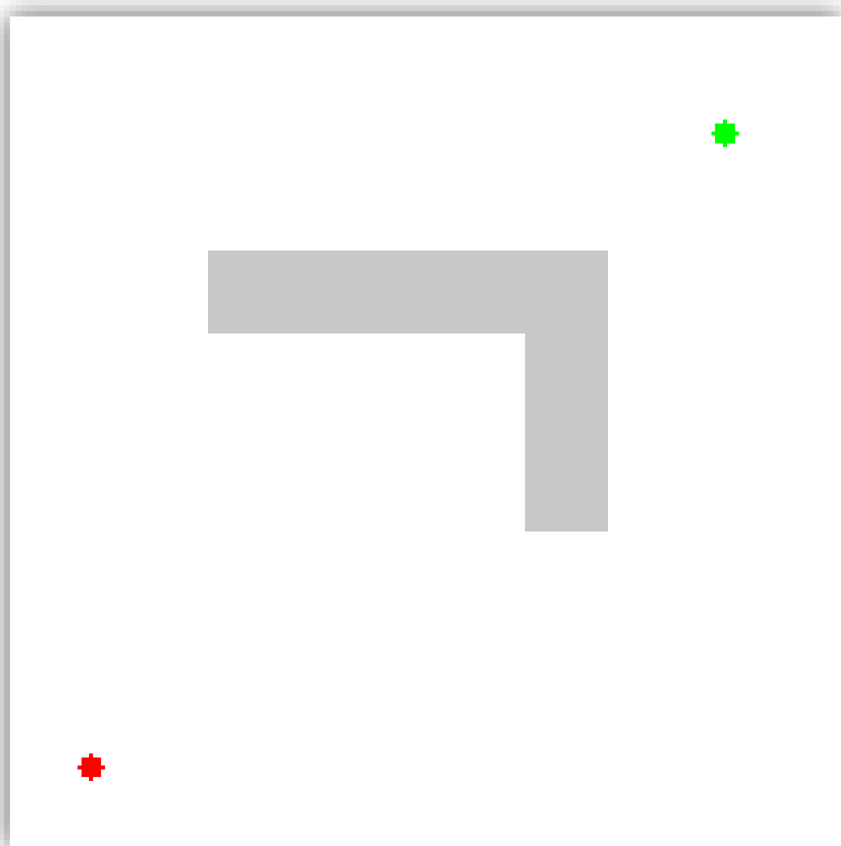
- Still expands many "unproductive" nodes in the example
  - Because the heuristic is **<u>not perfectly informative</u>**
    - Even though it is hand-coded
    - Does not take **<u>obstacles</u>** into account
  - If we knew actual remaining costs h*(n):
    - Expand optimal path to the goal

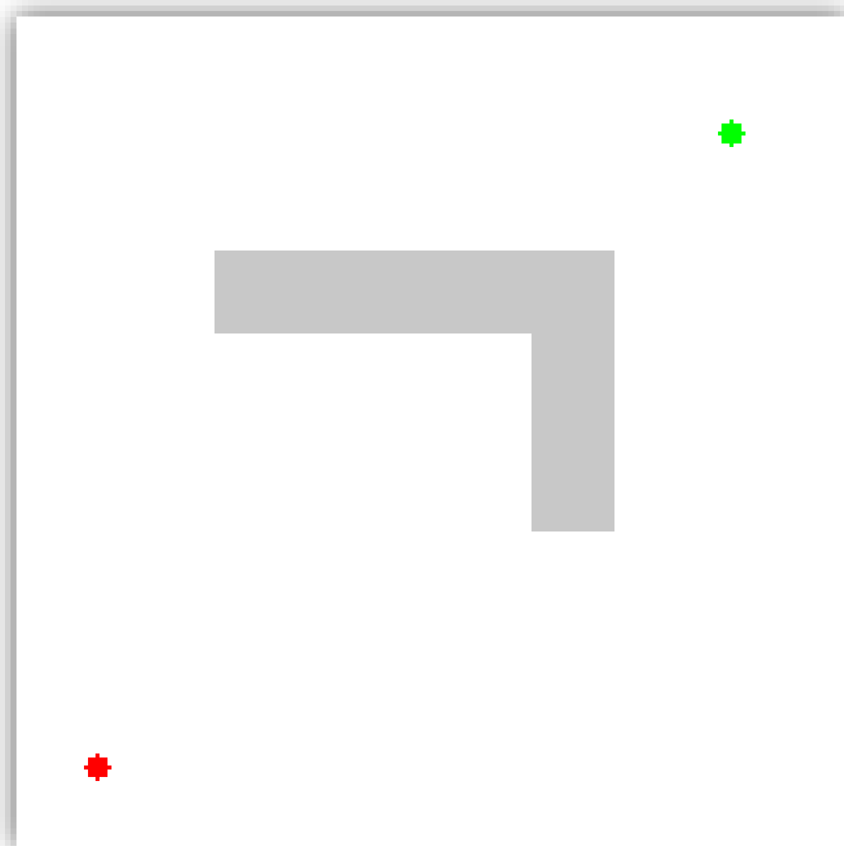


*Variations of A**

Weighted A*

- Weighted A*: Use $f(n) = g(n) + \boxed{w \cdot} h(n)$
  - Weight $w > 1$ places greater emphasis on being (*believing you are*) close to the goal
  - Result: At most w times more expensive





Repeated Weighted A*

- Repeated weighted A* -- example:
  - **<u>for</u>** *w* in (5.0, 3.0, 2.0, 1.5, 1.2, 1.0):
    solve problem using given *w*
  - Why?
    - Each pass is much faster than the next
    - Try to *approach* optimality,
      while still being able to *return a plan quickly* if necessary
  - Why not just specify a single weight?
    - Can't predict how much time any given weight will require

More variations will be discussed in the path planning lecture

Observations about the Open List

Open Lists


```
search(problem) {
    initial-node ← make-initial-node(problem) // [2]
    open ← { initial-node }
    while (open ≠ ∅) {
        node ← search-strategy-remove-from(open) // [6]
        …
    }
}
```

With an **OPEN list**, we have no "current position" during search

We choose from **all** open nodes, not from the nearest one

Without Open Lists


```
depth-first-search(problem) {
    initial-node ← make-initial-node(problem) // [2]
    return depth-first-search(initial-node)
}

depth-first-search(node) {
    if is-solution(node) then  // [4]
        return extract-plan-from(node) // [5]

    foreach newnode ∈ successors(node) { // [3]
        solution ← depth-first-search(newnode)
        if solution ≠ null {
            return solution
        }
    }
    return null
}
```

Depth First Search can use open lists or **recursive** search

We can **only** look at the successors of the *current node*

No possibility of postponing a node until later

Introduces **backtracking**: Going back from *where you are* (*no such concept* with open lists!)

Hill Climbing
in HSP, Heuristic Search Planner

(Non-Optimal, Informed)

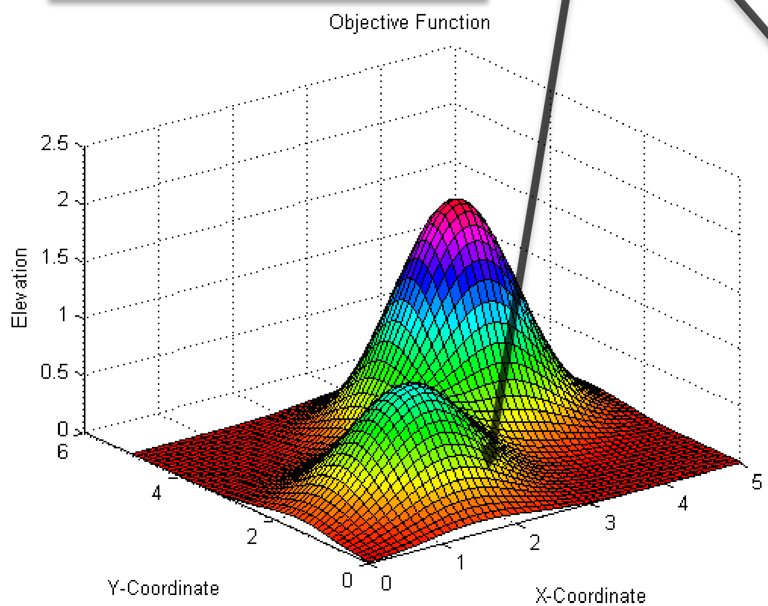
Hill Climbing (1)

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

Steepest Ascent Hill-climbing
$n \leftarrow$ initial node

$n = (x, y)$

State space example: n={on(A,C),…}

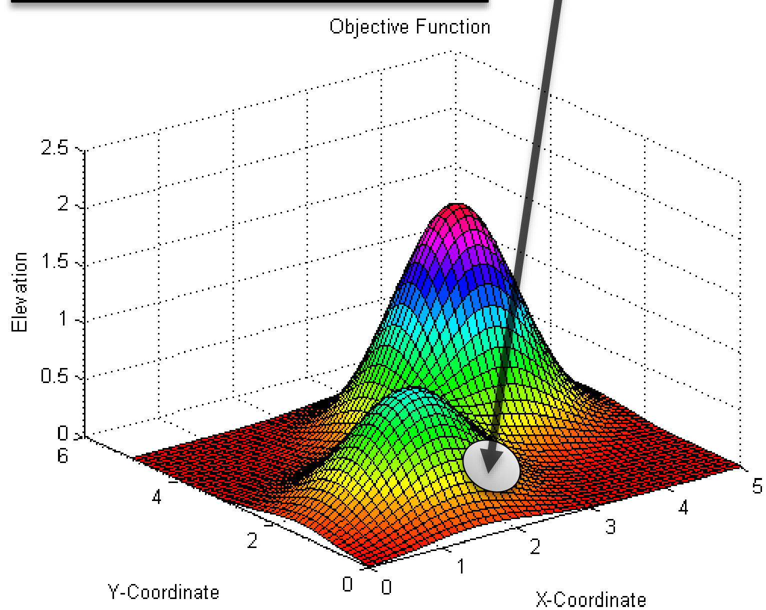


Hill Climbing (2)

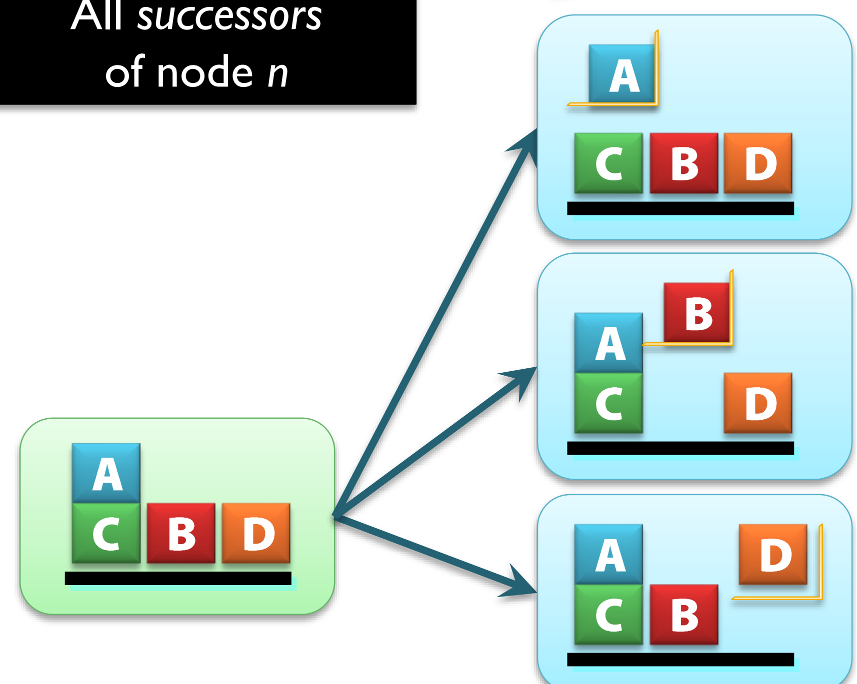
- (2) Find the **<u>local neighborhood</u>**, with nodes that you can reach in one "step"

<u>Steepest Ascent Hill-climbing</u>
n ← initial node
while True:
  if n is a solution **then return** n
  <u>expand</u> children of n

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

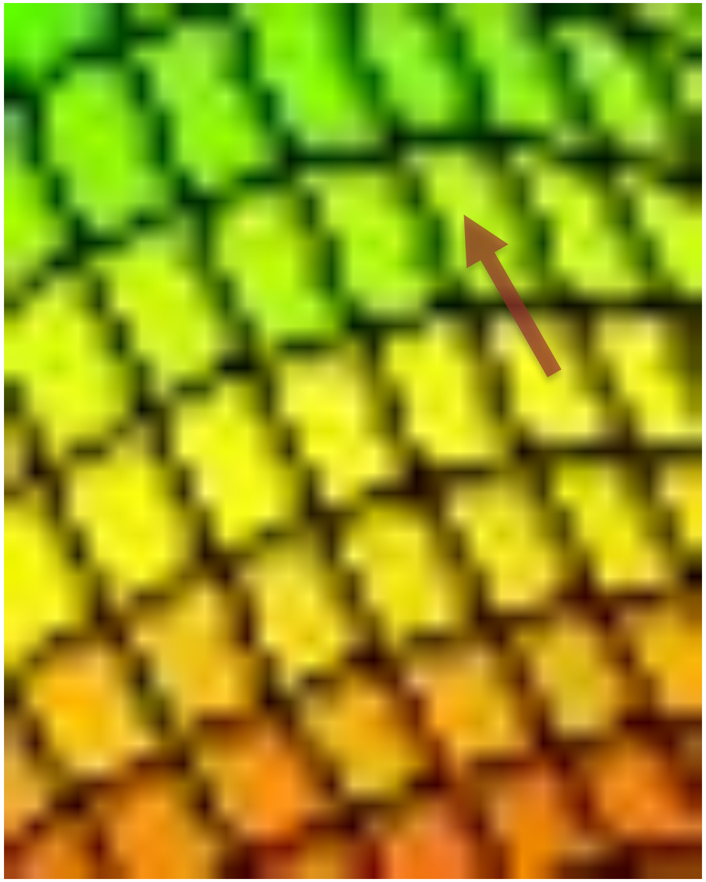


All *successors* of node *n*



Hill Climbing (3)

- (3) Try to **<u>improve</u>** using a **<u>locally optimal</u>** choice:
  Choose the successor/neighbor that is *best in this step*
  (don't care about the *future*)



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

n ← initial node
while True:
  if *n* is a solution **then return** *n*
  <u>expand</u> children of *n*

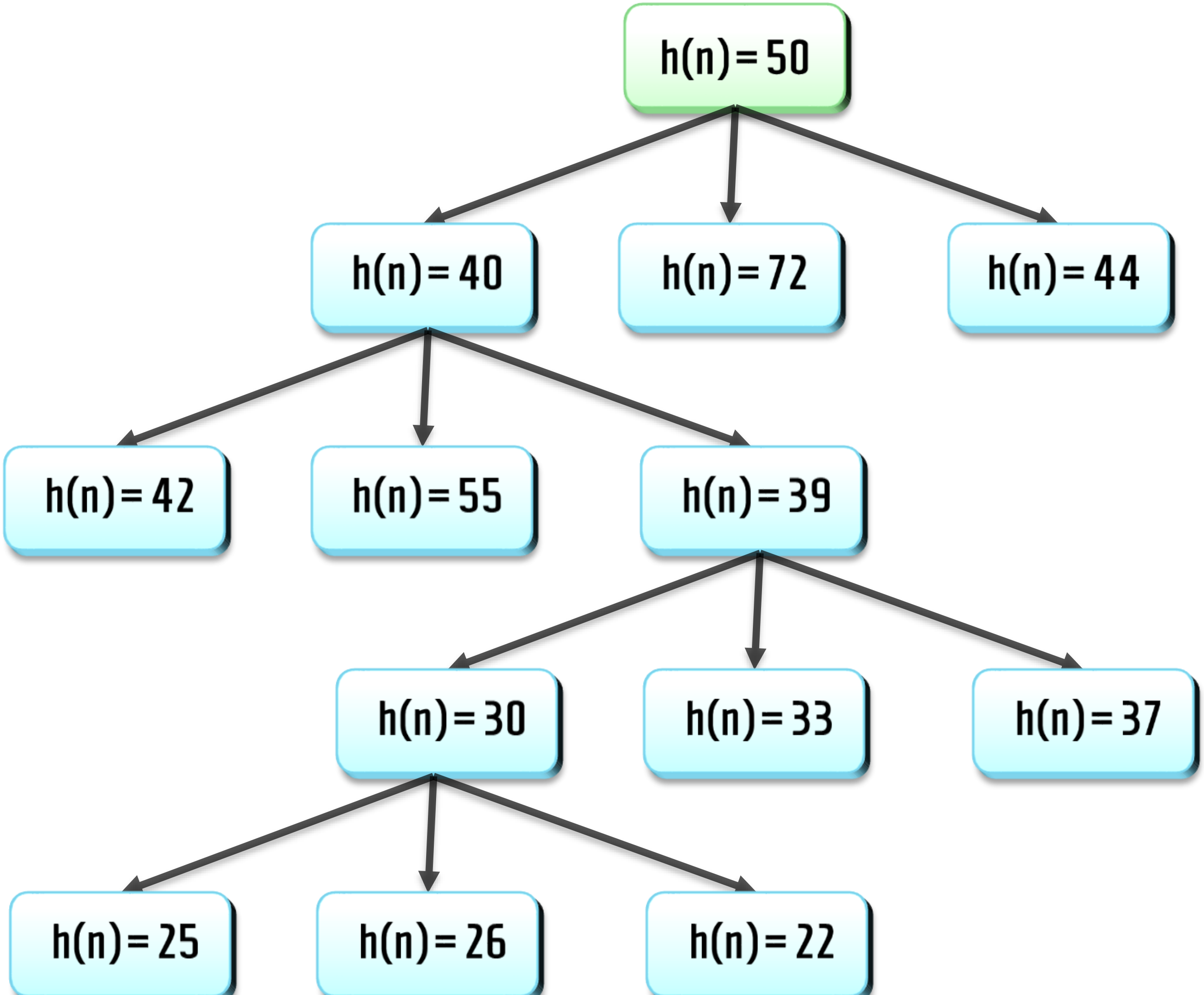
  <u>calculate</u> *h* for children
  if (some <u>child</u> decreases h(*n*)):
    n ← a child minimizing h(*n*)
  else …?
end loop

Search nodes have no "absolute" quality:
They are *solutions*
or useless *non-solutions*

But we can *estimate* quality
using heuristics (leading towards goals)

Hill Climbing (4)

- Example of hill climbing search:



Hill Climbing (5)

Greedy Best First search:

n ← initial node
open ← ∅
loop
  if *n* is a solution **then return** *n*
  expand children of *n*
  calculate *h* for children

  add children to *open*
  n ← a node in *open* minimizing h(*n*)
end loop

Steepest Ascent Hill-climbing
n ← initial node

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

  if (some child decreases h(*n*)):
    n ← a child minimizing h(*n*)
  else stop // local optimum
end loop

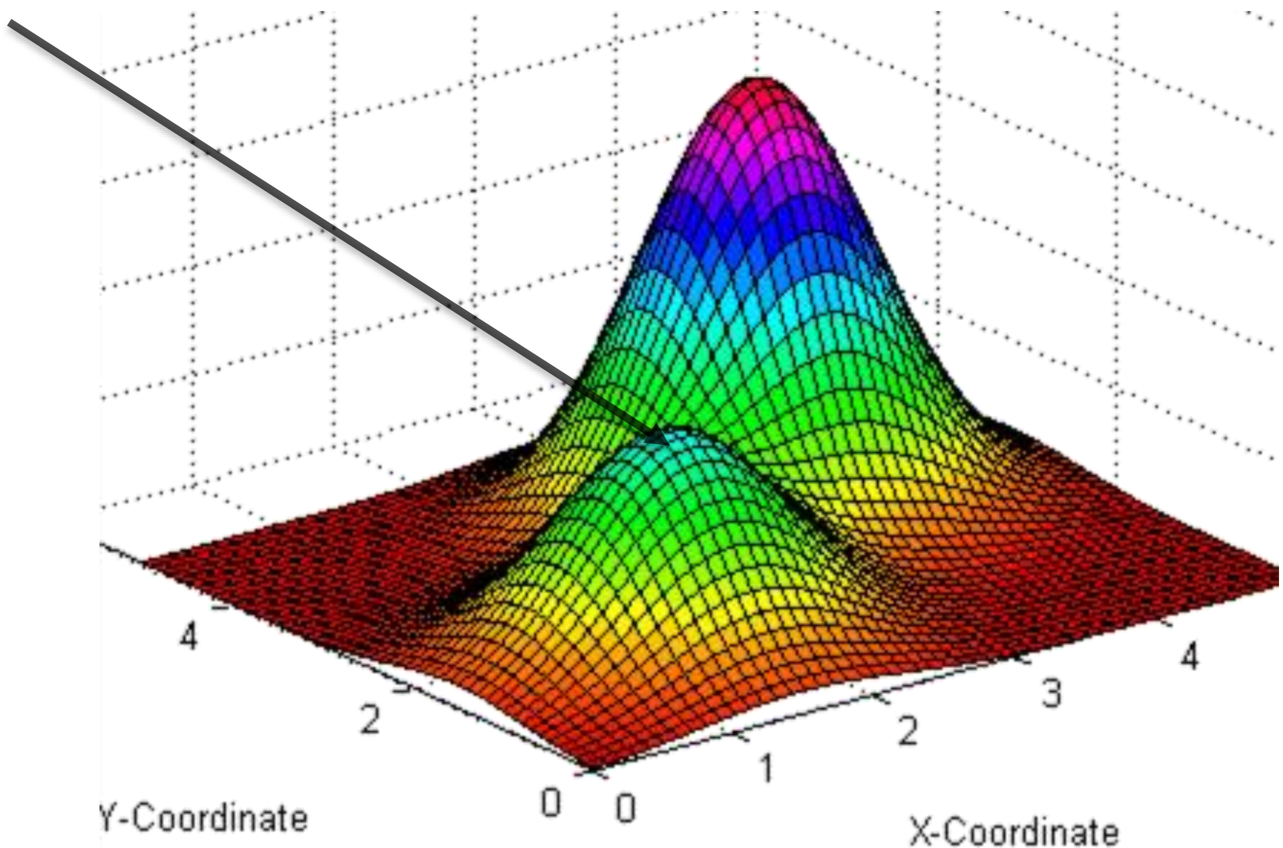
Be stubborn: Only consider children of this node, don't keep track of open nodes to return to

Choose best among children

Local Optima and Plateaus

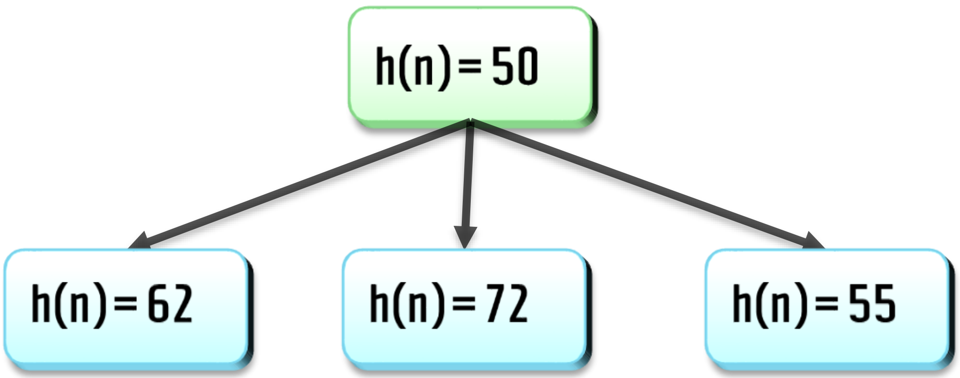
Local Optima (1)

- (4) When there is **nothing strictly better** nearby: Stop!
  - Standard HC is used for *optimization*
    - Any point is a *solution*, we search for a *good* 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*



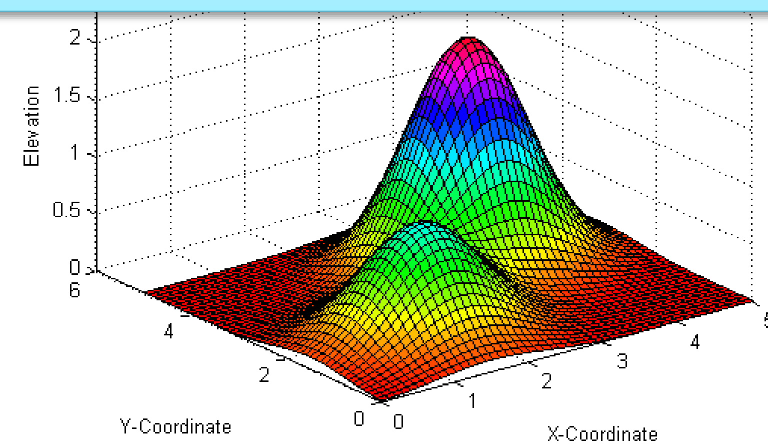
Local Optima (3)

- Standard solution to local optima:
  - Randomly choose another node
  - Continue searching from there
  - Hope you find a global optimum eventually
- In **planning**:
  - Must choose a node that you have actually created during expansion…

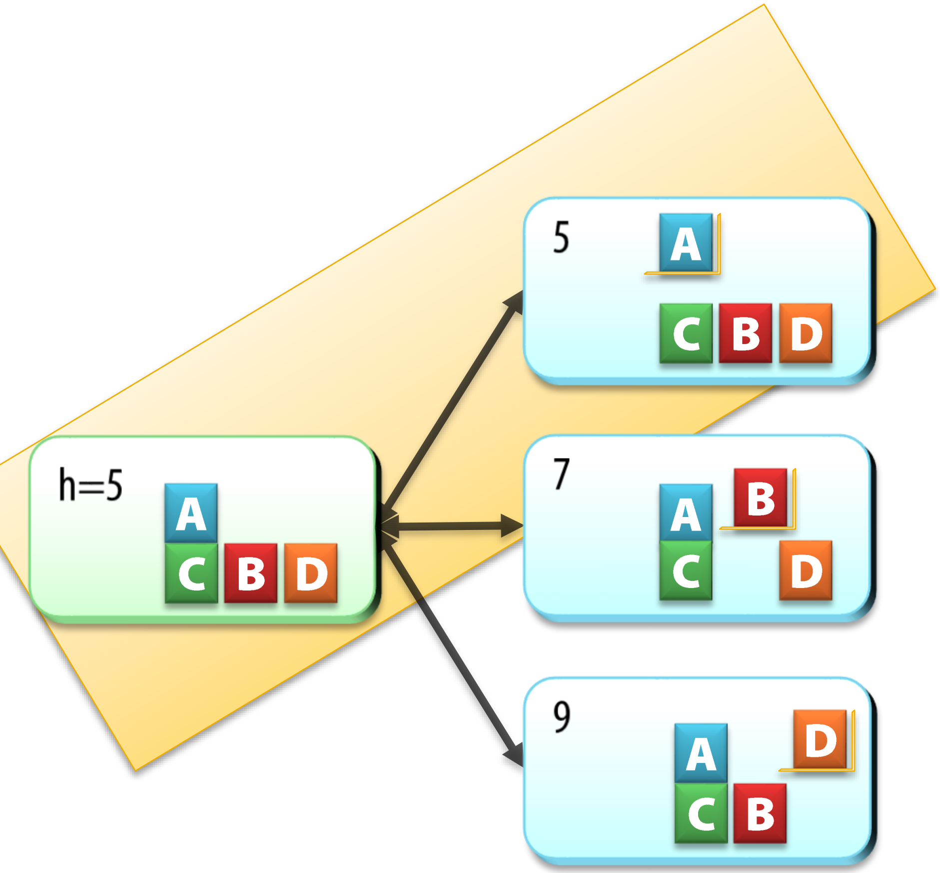
Steepest Ascent
Hill-climbing with Restarts

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

    if (some child decreases h(*n*)):
        n ← a child minimizing h(*n*)
    else *n* ← **some rnd. state**
end loop



Hill Climbing with h_{add}: Plateaus



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

We have a **<u>plateau</u>**…

Jump to a random node *immediately*?

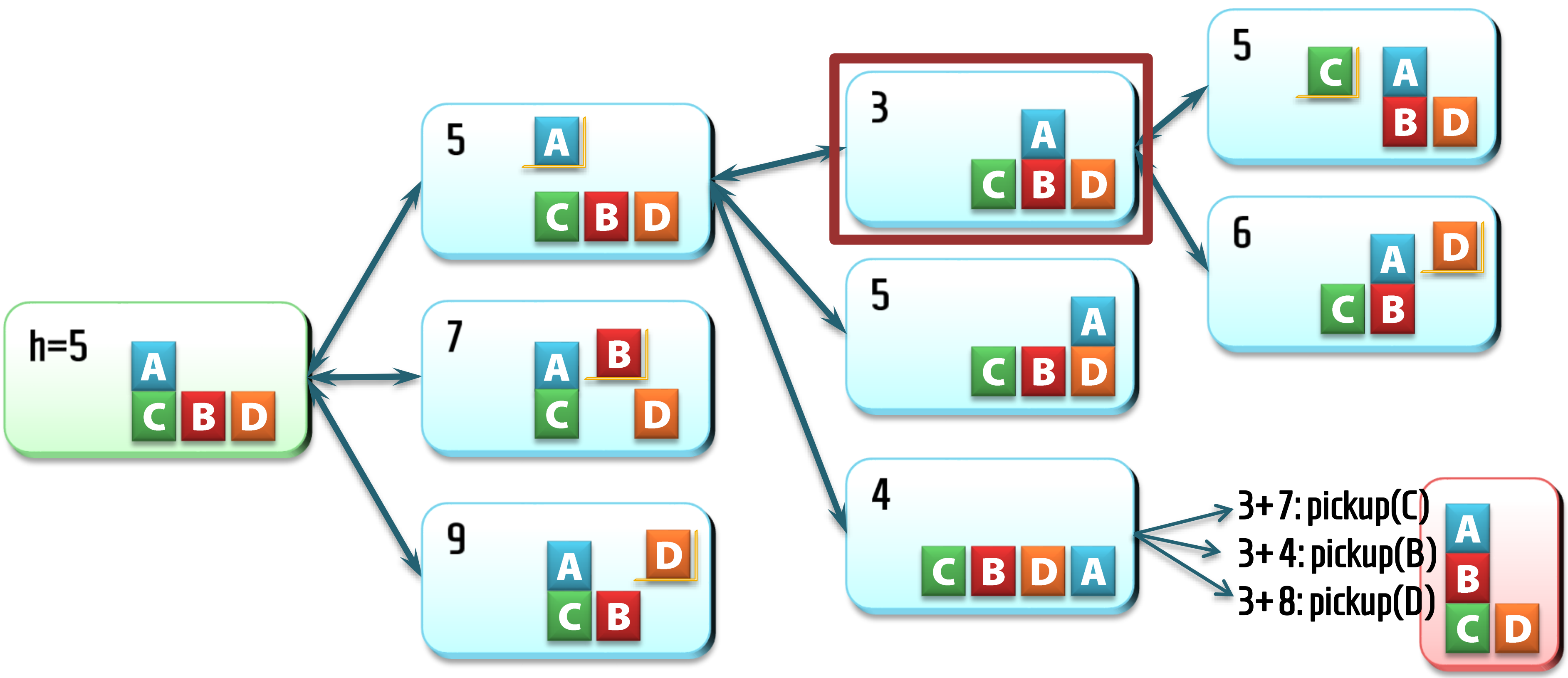
No: the heuristic is not so accurate – maybe some child *is* closer to the goal even though h(n) isn't lower!

➔ Keep exploring: Allow *some* consecutive **<u>moves across plateaus</u>**

Hill Climbing with h_{add}: Local Optima

If we continue, all successors have higher heuristic values!

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



Impasses and Restarts

- What if there are **many** impasses?
  - Maybe we *are* in the wrong part of the search space after all…
  - → Select another *promising* expanded node where search continues

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;
                  } else current = bestChild;
              } else current = bestChild;
          }
      }
      ```

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
      ```

Enforced Hill-Climbing

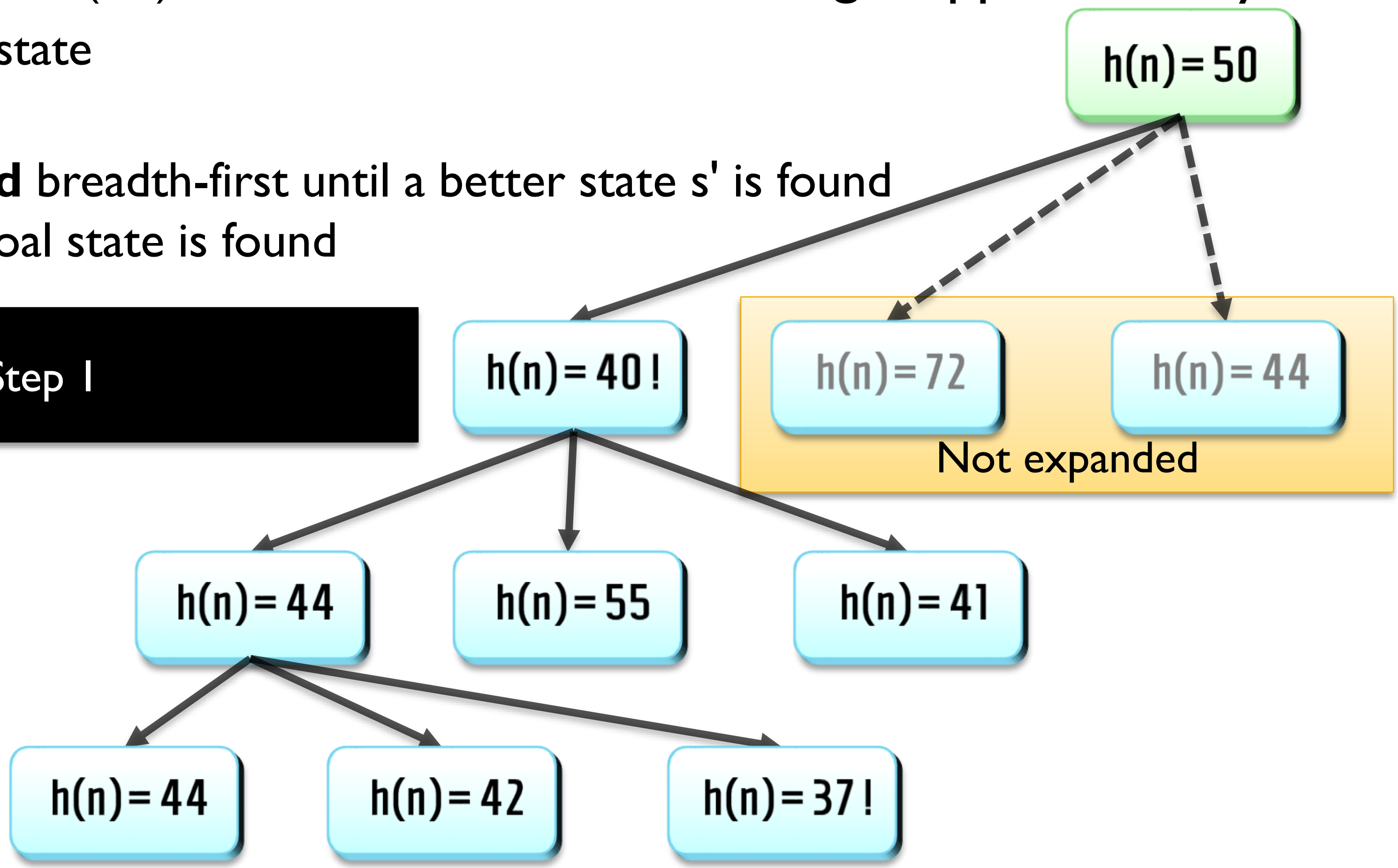
(Non-Optimal, Informed)

Enforced Hill Climbing

- FastForward (FF) uses **enforced** hill climbing – approximately:
  - s ← init-state
  - **repeat**
    expand breadth-first until a better state s' is found
    until a goal state is found

Step 1

Step 2



Wait longer to decide which branch to take
Don't restart – keep going

Properties of EHC

- Is Enforced Hill-Climbing **complete**?
  - No!