Course "Automated Planning: Theory and Practice"
Chapter 07: General Search Strategies

Teacher: Marco Roveri - marco.roveri@unitn.it
M.S. Course: Artificial Intelligence Systems (LM)
A.A.: 2023-2024
Where: DISI, University of Trento
URL: https://bit.ly/3zOkGk8

Last updated: Sunday 15th October, 2023

Terms of Use and Copyright

The material (text, figures) in these slides is authored by Jonas Kvarnström and Marco Roveri.

Important Distinction

Optimizing

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

Guaranteeing optimality is sometimes useful, always expensive!

Satisficing

- Satisficing (satisfy/suffice) in general:
  - *"Searching until an acceptability threshold is met"*
  - Motivation: High-quality non-optimal solutions are also useful
    - Can often be found in reasonable time
- Satisficing in planning (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 different points on the efficiency/quality spectrum!

Informed vs Uninformed Search

Uninformed Search

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

Informed Search

- Take additional information into account, such as heuristics!

Applicable to all search spaces we have seen so far

May work *better* in some of them...

Dijkstra's Algorithm

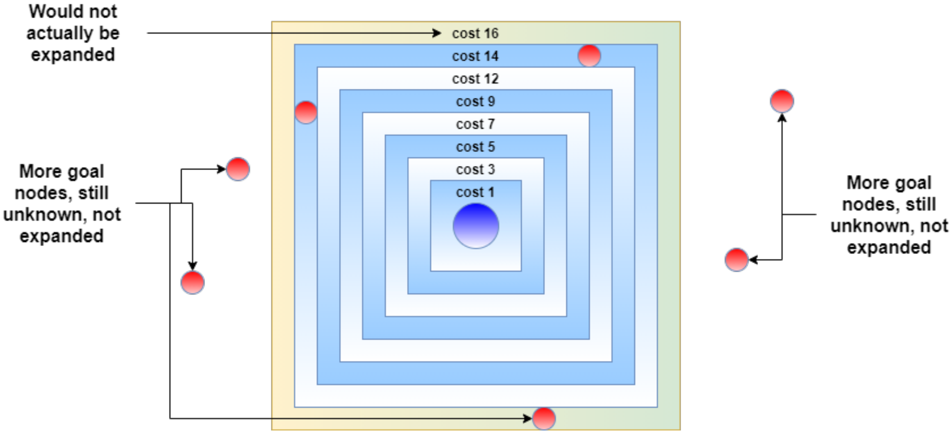
- Matches the forward search "template"
  - Use a "*simple*" strategy to select and remove a node n from open
  - Select a node n with minimal $g(n)$: Cost of reaching n from initial node
- Efficient graph search algorithm: $O(|E| + |V| \log(|V|))$
  - $|E|$ = number of edges (transitions), $|V|$ = number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

```
function SEARCH(problem)
    initial-node ← MAKE-INITIAL-NODE(problem)                    → [2]
    open ← {initial-node}
    while (open ≠ ∅) do
        node ← SEARCH-STRATEGY-REMOVE-FROM(open)                 → [6]
        if IS-SOLUTION(node) then                                → [4]
            return EXTRACT-PLAN-FROM(node)                       → [5]
        end if
        ...
    end while
    ...
end function
```

Typical Implementation
Priority Queue

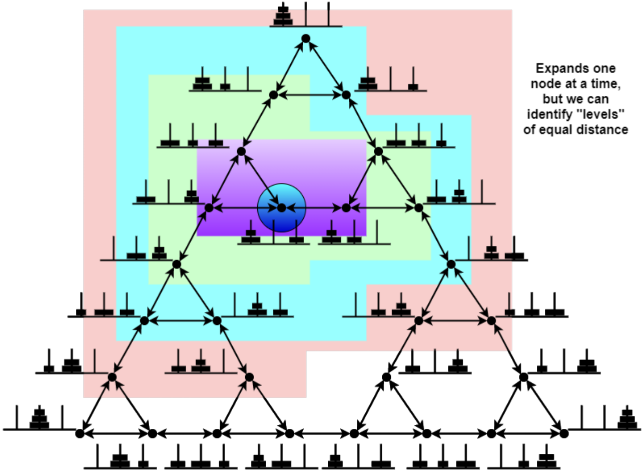
Dijkstra's Algorithm: Exploration order

- Explore nodes in increasing/decreasing order of cost!



Dijkstra's Algorithm: Tower of Hanoi

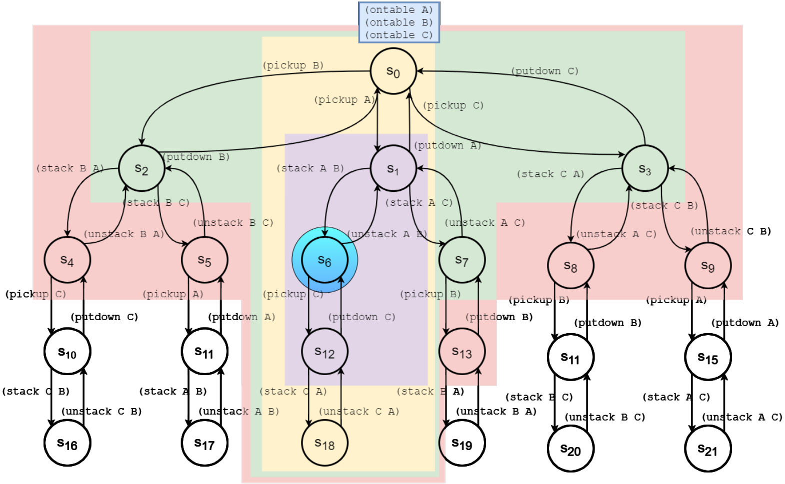
- Running Dijkstra, assuming all ToH actions are equally expensive



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

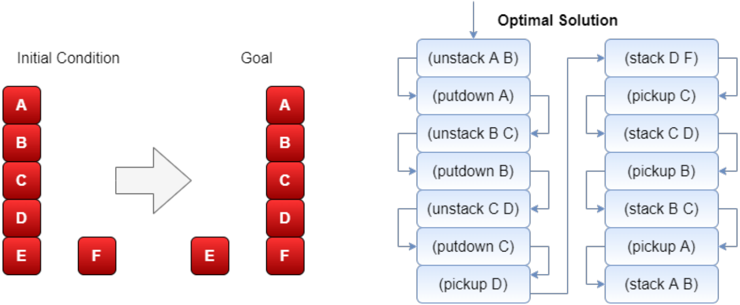
Dijkstra's Algorithm: Blocks World

- Running Dijkstra, assuming all BW actions are equally expensive



Dijkstra's Algorithm: Example

- A small instance



Dijkstra's Algorithm: Example

- A typical implementation: 8706 created states, 2692 visited/expanded
- BW 400
  - Standard formulation: $s^{n^2+3n+1} = 2^{161201} > 10^{48526}$ states
  - But we do not have to visit every one ... fewer reachable states!
- BW 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: = plans, 400*399 plans, ...
    - Stacking 2 blocks: > 400*399*399*398 plans, ...
    - Will visit more that $1.63 * 10^{1735}$

Dijkstra is efficient in terms of 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
  - $\Longrightarrow$ 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 $\mathit{cost} < \mathit{cost}(\mathit{optimal\ solution})$
- 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: Intuition

```
function SEARCH(problem)
    initial-node ← MAKE-INITIAL-NODE(problem)
    open ← {initial-node}
    while (open ≠ ∅) do
        node ← SEARCH-STRATEGY-REMOVE-FROM(open)
        if IS-SOLUTION(node) then
            return EXTRACT-PLAN-FROM(node)
        end if
        for each newnode ∈ SUCCESSORS(node) do
            open ← open ∪ {newnode}
        end for
    end while
    return Failure
end function
```

- Keep track of a set of open nodes
- Use an 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 Dijkstra's algorithm etc,.. which consider cost so far, and having no idea where to go next!
  - As opposed to hill climbing and others that "throw away nodes instead of keeping all nodes in open!

Greedy Best First Search: Intuition

```
function SEARCH(problem)
    initial-node ← MAKE-INITIAL-NODE(problem)
    open ← {initial-node}
    while (open ≠ ∅) do
        node ← SEARCH-STRATEGY-REMOVE-FROM(open)
        if IS-SOLUTION(node) then
            return EXTRACT-PLAN-FROM(node)
        end if
        for each newnode ∈ SUCCESSORS(node) do
            open ← open ∪ {newnode}
        end for
    end while
    return Failure
end function
```

- Choose an open node minimizing $h(n)$

- Ignore the cost $g(n)$ of reaching the node
- Try to minimize the (apparent) amount of search left to do

A*

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

- How admissibility helps?
  - Let 12 be the cost of optimal solution
    - Another node n with $g(n) = 10$ and $h(n) = 5$
    - $h(n)$ admissible, never overestimates, so any solution from here would cost at least 10+5=15
    - No need to investigate successors of this node!
  - If $h(n)$ does not underestimate, it does not help!
    - Could find solutions of cost 10 as descendants of node $n \Longrightarrow$ must keep searching!

A* Strategy

- Pick nodes from open in order of increasing $f(n) = g(n) + h(n)$
  - $g(n)$ 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 *underestimates* so solution found by expanding these nodes will cost $\geq\ g + h$ (and we have one of cost $\leq\ g + h$)	

- If an heuristic never <u>under</u>estimates costs:
  - Let 12 be the cost of a solution
    - Another node, n: $g(n) = 10$, and $h(n) = 5$
    - $h(n)$ never <u>under</u>estimates, so any solution found from here on would cost at <u>most</u> 15
    - Does not help! Could find solutions of cost 10 as descendant of node n, must keep searching!

A*: Dijkstra's vs A* – essential difference

Dijkstra

- Selects from open a node n with minimal $g(n)$
  - Cost of reaching n from initial node

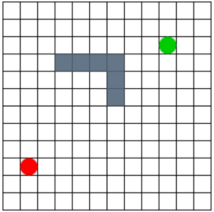
Uninformed - blind -

A*

- Selects from open a node with minimal $g(n) + h(n)$
  - + underestimate cost of reaching a goal from n

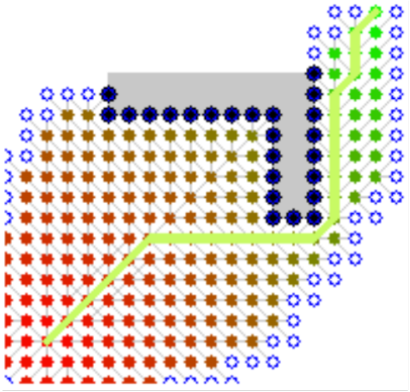
Informed

- Example:
  - Hand-coded heuristic function
  - Can move diagonally $\implies h(n) = \max(abs(n.x - g.x), abs(n.y - g.y))$ – Chebyshev distance
  - Related to Manhattan Distance = $abs(n.x - g.x) + abs(n.y - g.y)$



A*

- Given an admissible heuristic h, A* is optimal in two ways
  - Guarantee an optimal plan is extracted
  - Expands the minimum number of nodes required to *guarantee optimality* with the given heuristic!
- Still may expand many "unproductive" nodes in the maze example
  - The heuristic is not perfectly informative
  - Does not take obstacles into account
- If we knew actual remaining cost $h^*(n)$:
  - Expand optimal path to the goal!



Variations of A*

- Weighted A*
  - Use $f(n) = g(n) + w \cdot h(n)$
    - Weight $w > 1$ place greater emphasis on being close to the goal! I.e., you *believe* to be close to the goal!
    - $\Longrightarrow$ At most w times more expensive!
- Repeated Weighted A*
  - Consider an ordered set of weights, and try to repeatedly solve problem using one weight from the set!
  - **for** $w \in \{5.0, 3.0, 2.0, 1.0\}$ **do**
    solve problem with Weighted A* using w
  - Rationale
    - 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 a single weight? $\Longrightarrow$ Can't predict how much time any given weight will require!
- More variants are discussed in the path planning robotic course!

With Open List

```
function SEARCH(problem)
    initial-node ← MAKE-INITIAL-NODE(problem)
    open ← {initial-node}
    while (open ≠ ∅) do
        node ← SEARCH-STRATEGY-REMOVE-FROM(open)
        if IS-SOLUTION(node) then
            return EXTRACT-PLAN-FROM(node)
        end if
        ...
    end while
    ...
end function
```

- With an Open List, we have no "current position" during the search!
  - We choose from all open nodes, not from the nearest one!

Without Open List

```
function DEPTH-FIRST-SEARCH(problem)
    initial-node ← MAKE-INITIAL-NODE(problem)
    return DEPTH-FIRST-SEARCH-REC(initial-node)
end function

function DEPTH-FIRST-SEARCH-REC(node)
    if IS-SOLUTION(node) then
        return EXTRACT-PLAN-FROM(node)
    end if
    for each newnode ∈ SUCCESSORS(node) do
        solution ← DEPTH-FIRST-SEARCH-REC(newnode)
        if solution ≠ null then
            return solution
        end if
    end for
    return null
end function
```

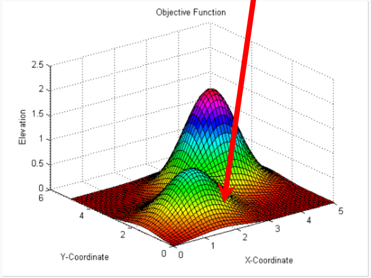
- Depth First Search can use open list or recursive search!
    - We can only look at the successors of *current* node
    - No possibility to postponing a node until later
    - Introduces backtracking: going back from *where you are*
        - Not such concept exists with open list!

Steepest Ascent Hill Climbing

- Greedy local search algorithm for optimization problems
  - (i) Start in some current location

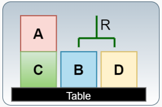
function SteepestAscentHillClimbing(problem)
  n ← initial-node
  ...

2D Example



http://www.willmcginnis.com/2012/05/12/272/

State Space Example

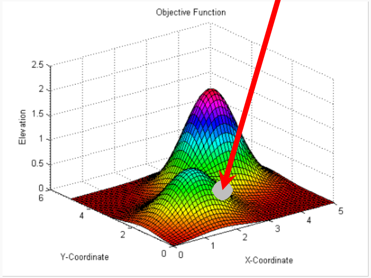


Steepest Ascent Hill Climbing (cont.)

- (ii) Find the local neighborhood, with nodes that can be reached in one step

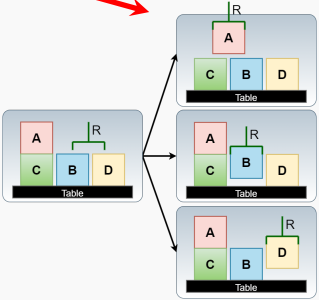
function SteepestAscentHillClimbing(problem)
  n ← initial-node
  while True **do**
    if *n* is a solution **then return** *n*
    expand children of *n*

2D Example



http://www.willmcginnis.com/2012/05/12/272/

State Space Example



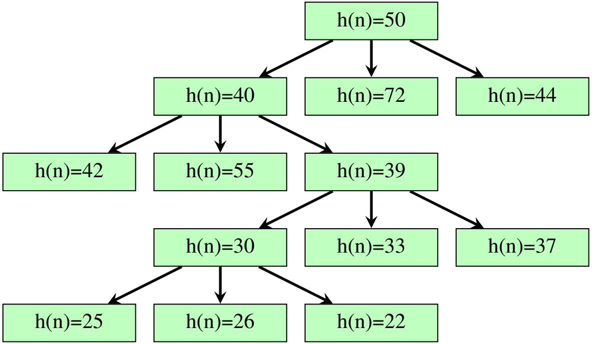
Steepest Ascent Hill Climbing (cont.)

- (iii) Try to improve using local optimal choice:
  - Choose the successor/neighbor that is *best in this step*
  - $\Longrightarrow$ Don't care about the *future*

```
function SteepestAscentHillClimbing(problem)
    n ← initial-node
    while True do
        if n is a solution then return n
        expand children of n
        calculate h for children
        if some child decreases h(n) then
            n ← a child minimizing h(n)
        else  ??
            ...
```

- Search nodes have no absolute quality
  - They are *solutions* or useless *non-solutions*
- But we can *estimate* the quality using heuristics (leading towards the goal)!

Steepest Ascent Hill Climbing: Example



Steepest Ascent Hill Climbing (cont.)

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

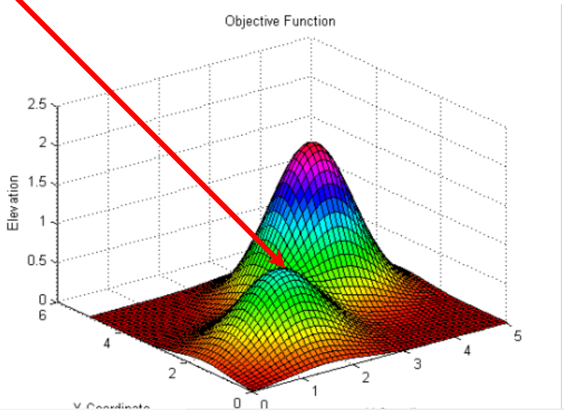
function GreedyBestFirstSearch(problem)
  n ← initial-node
  open ← ∅
  while True **do**
    if n is a solution **then return** n
    expand children of n
    calculate h for children
    add children to open
    $n \leftarrow$ a node in open minimizing $h(n)$

function SteepestAscentHillClimbing(problem)
  n ← initial-node
  while True **do**
    if n is a solution **then return** n
    expand children of n
    calculate h for children
    if some child decreases $h(n)$ **then**
      $n \leftarrow$ a child minimizing $h(n)$
    else **stop** → *Local optimum*
      ...

Chose best among childrens

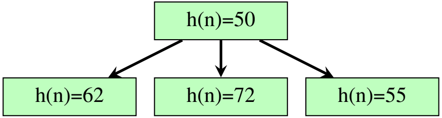
Local Optima

- (iv) When there is nothing strictly better nearby: Stop!
  - Standard Hill Climbing 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 (cont.)

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

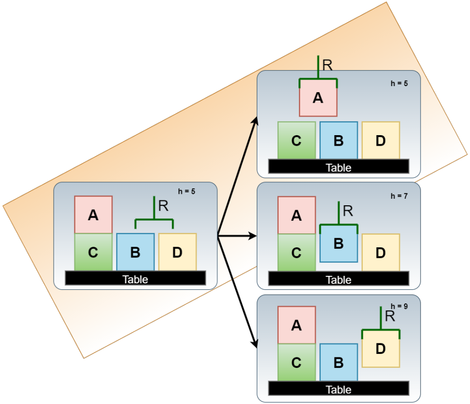


Local Optima (cont.)

- 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...

```
function SteepestAscentHillClimbing(problem)
    n ← initial-node
    while True do
        if n is a solution then return n
        expand children of n
        calculate h for children
        if some child decreases h(n) then
            n ← a child minimizing h(n)
        else
            n ← some random state
```

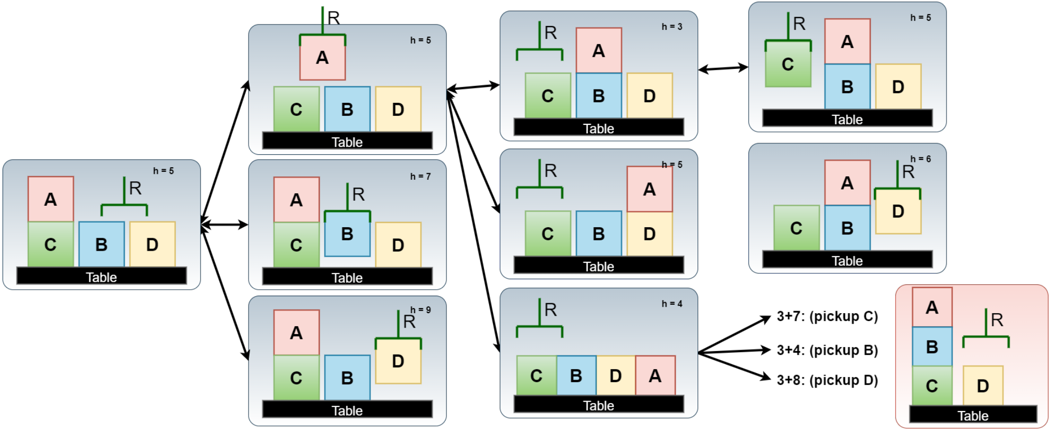
Hill Climbing with h_{add}: Plateaus



- No successor improves the heuristic value: some are equal!
  - We have a plateau

- Jump to a random node *immediately*?
  - No! The heuristic is not so accurate – may be some child *is closer* to the goal even though $h(n)$ is not lower!
  - $\Longrightarrow$ keep exploring: allow some consecutive moves across plateaus!

Hill Climbing with h_{add}: Local Optima



- If we continue, all successors have higher heuristic values!
  - We have a local optimum… *Impasse* = optimum or plateaus $\Longrightarrow$ Some *impasses* allowed!

Impasses and Restarts

- What if there are many impasses?
  - May be we are in the wrong part of the search space after all....
  - $\Longrightarrow$ Select another *promising* expanded node where search continues...

HSP 1: Heuristic Search Planner

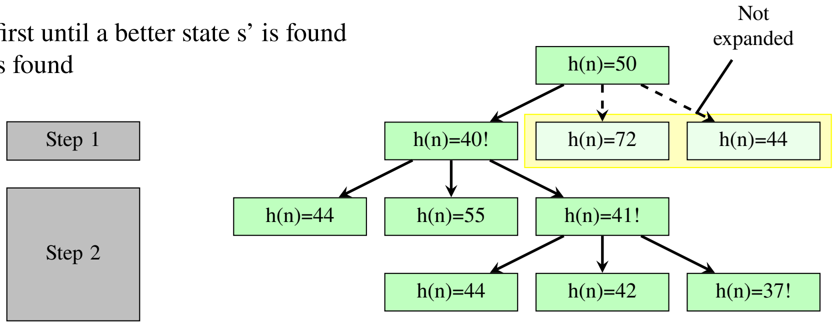
- HSP 1.x: h_{add} heuristic + hill climbing + modifications

```
function SteepestAscentHillClimbing(problem)
    impasses ← 0
    unexpanded ← ∅
    current ← initial-node
    while (not yet reached the goal) do
        children ← expand(current)                          → Apply all applicable actions
        if (children = ∅) then                               → Dead end ⟹ restart!
            current ← pop(unexpanded)
        else
            bestChild ← best(children)                       → Child with the lowest heuristic value
            add other childrens to unexpanded in order of h(n)   → Keep for restarts!
            if (h(bestChild) ≥ h(current)) then              → Essentially HC, but not all steps have to move "up"
                impasses++
                if (impasses = threshold) then               → Too many downhill/plateau moves ⟹ escape!
                    current ← pop(unexpanded)                → Restart from another node!
                    impasses ← 0
                else
                    current ← bestChild
        else
            current ← bestChild
```

Simple structure, but highly competitive at its introduction!

Enforced Hill Climbing

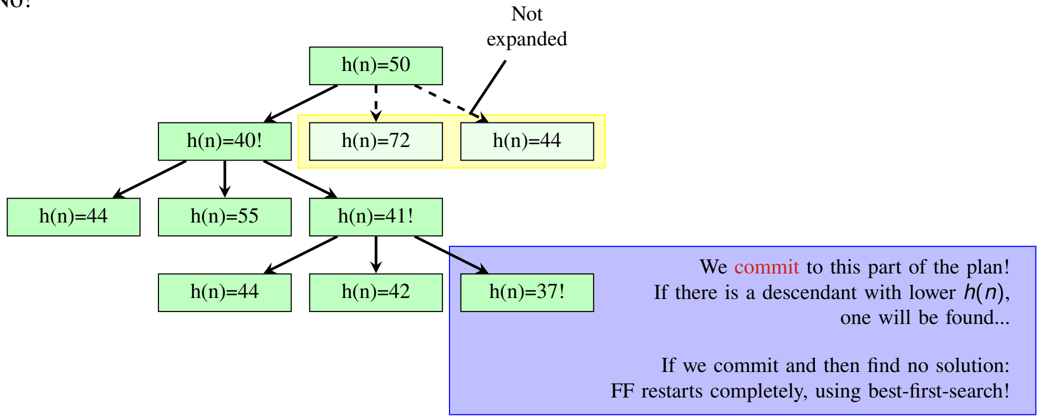
- FastForward (FF) [1] uses enforced hill climbing – approximately
  - s ← init-state
  - **repeat**
    expand breadth-first until a better state s' is found
    until a goal state is found



Wait longer to decide which branch to take! $\Longrightarrow$ Do not restart – keep going!

PROPERTIES OF ENFORCED HILL-CLIMBING

- Is Enforced Hill-Climbing complete?
  - No!



References I

[1] FF. The Fast Forward Planner. `https://fai.cs.uni-saarland.de/hoffmann/ff.html`, 2001. 34

[2] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013. ISBN 9781608459698. doi: 10.2200/S00513ED1V01Y201306AIM022. URL `https://doi.org/10.2200/S00513ED1V01Y201306AIM022`.

[3] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

[4] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016. ISBN 978-1-107-03727-4. URL `http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/automated-planning-and-acting?format=HB`.