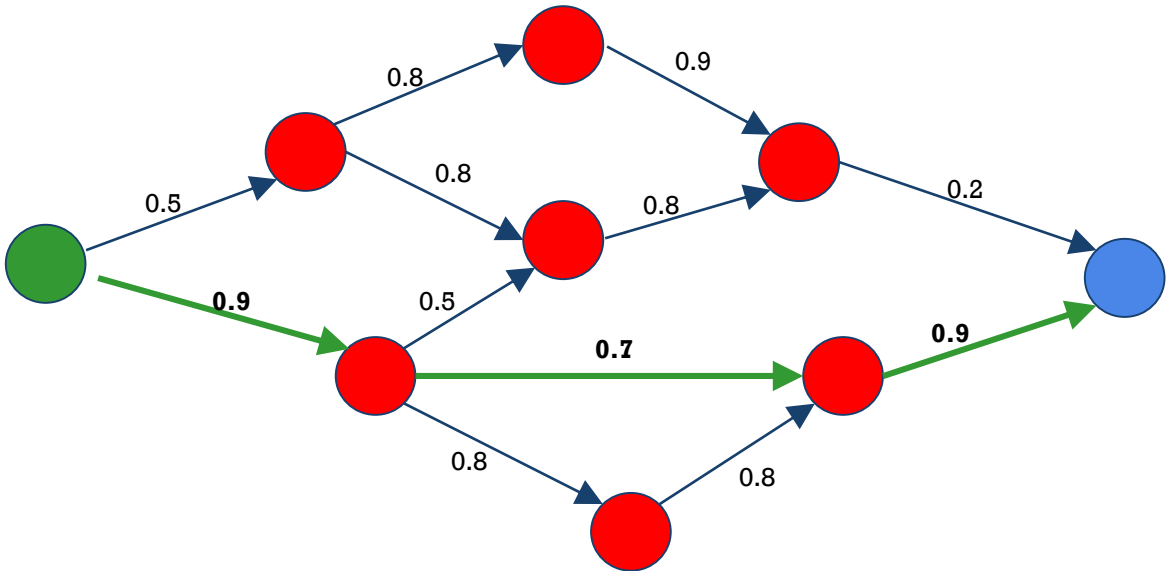
Grado en ingeniería informática
Artificial Intelligence 2021/2022

2.2. Informed search

Lara Quijano Sánchez



UAM
Universidad Autónoma
de Madrid

Problem solving using informed search

- Introduction
- Best-first search
- Iterative improvement algorithms

Readings

- CHAPTER 4 of Russell & Norvig
- CHAPTERS 9, 10, 11 of Nilsson

- Tools
  - http://qiao.github.io/PathFinding.js/visual/

Informed or heuristic methods

- Uninformed Methods
  - Very inefficient in most cases
  - In the face of combinatorial explosion, brute force is impractical
- Informed or heuristic methods
  - Uses domain knowledge to guide the search
    - The heuristic provides information about the proximity of each state to a target state
    - Using this information, the search can be oriented: Choosing as the next node to expand the one that is more "promising“
    - In case the heuristic is reliable it reduces the complexity of the combinatorial explosion of the exploration
      - It does not generate unpromising nodes and thus improves performance
      - We can find an acceptably good solution in a reasonable time
- Limitations
  - It does not prevent the combinatorial explosion: the complexity remains exponential.
  - If the heuristic is not reliable, the efficiency worsens.
  - In some cases they do not guarantee to find a solution (they may not be complete) and, if they do, they do not guarantee that it is optimal.

Heuristics

- From Greek *ευρισκω* = to find, to discover.
  - "EUREKA!" from Archimedes
  - Rules that generally (but not always) help directing the search towards the solution.
    - 1945, Georg Polya (American, born in Hungary): "How to Solve It" Methods for solving problems
      1. Understand the problem.
      2. Build a plan.
      3. Execute the plan
      4. Improve the plan.
    - 1958, Teddington Conference on the Mechanisation of Thought Processes (UK)
      - John McCarthy: "Common sense programs"
      - Oliver Selfridge: "Pandemonium"
      - Marvin Minsky: "Some methods of artificial intelligence and heuristic programming"
    - 1963, Newell
    - Mid 60's-80's: Stanford Heuristic Programming Project *(HPP),* led by E. Feigenbaum to develop rule-based expert systems (DENDRAL, MYCIN)

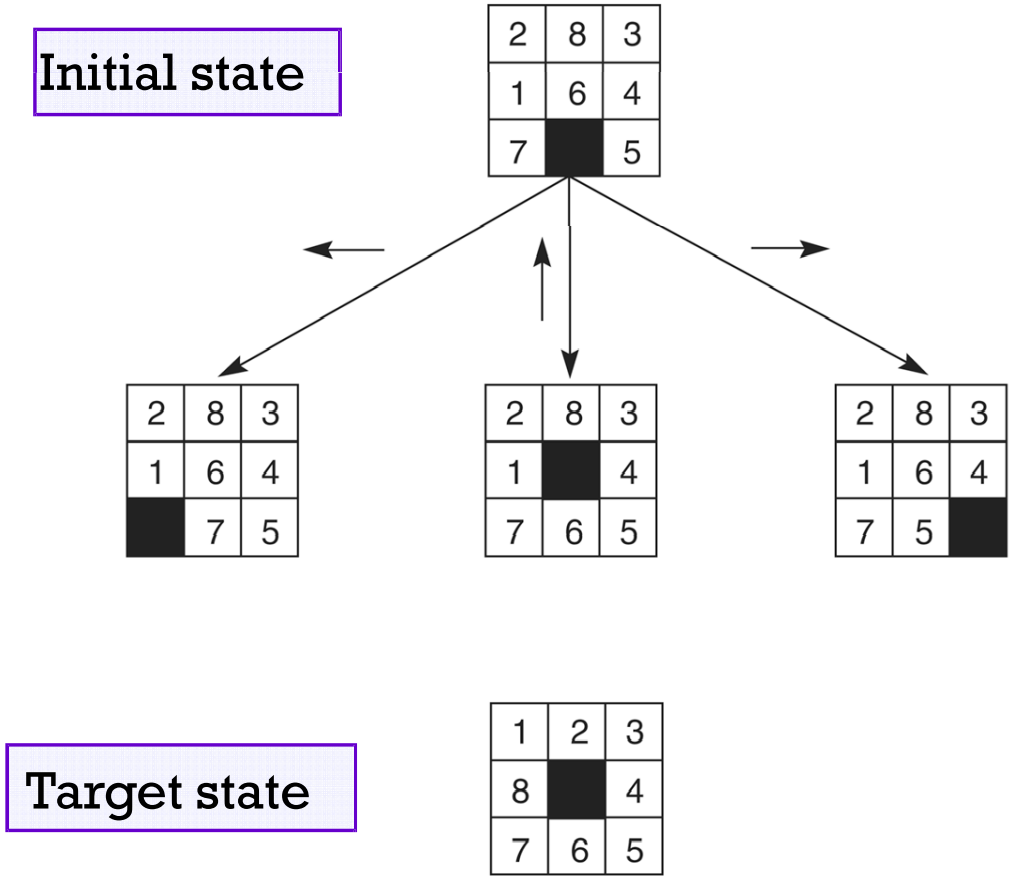
Heuristic search

- Heuristic search method:
  - Algorithm that uses a heuristic to guide the search in the state space
- Heuristics: technique that improves search efficiency
  - Uses domain-specific knowledge of the problem, beyond the definition of the problem itself.
  - Provides a guide to the search algorithm
    - Sort the generated nodes that are pending to be expanded (in open-list) so that those that are most promising are expanded first according to local information.
    - This local information is that which can be obtained directly from the current state without searching (without generating successors).
  - Heuristic search may not be complete or optimal
    - It is possible that no solution will be found, having it
    - It does not guarantee that, if a solution is found, this will be the least costly.
- Important to select a suitable heuristic function.
  - How to choose a good one?

Heuristic function: h (n)

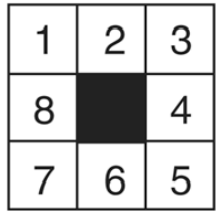
- h (n): The value of the heuristic function *h* evaluated at node *n* is a number that provides an estimation of how promising the state corresponding to that node is in reaching a target state.
- This information is used in the search algorithm to determine the expansion order of the *open-list* nodes.
- Two possible interpretations
  1. Estimation of the "quality" of a state
     - States with a higher value for the heuristic are preferred
  2. Cost estimation: lowest cost to reach the target state from a given state.
     - States with a lower heuristic value are preferred
  - Both points of view are complementary: A change of sign allows to pass from one perspective to the other
- Agreement: 2nd interpretation
  - Non-negative heuristic values (lower is better)
  - The states that meet the target test are assigned a value of 0 for the heuristic
  - **Attention**: there may be states that do not meet the target test and have a value of 0 for the heuristic.

Example: heuristics for the 8-puzzle



Example: heuristics for the 8-puzzle

- h_a = sum the distances of the pieces to their positions on the target board
  - Since there are no diagonal movements, the horizontal and vertical distances are added
  - Manhattan distance, (or taxi distance or city distance)
    - Example: 1 + 1 + 0 + 0 + 0 + 1 + 1 + 2 = 6



- h_b = n° of displaced pieces (with respect to the target board)
  - It is the simplest heuristic and seems quite intuitive
    - Example: 5
  - But it does not use the information related to the effort (number of movements) necessary to bring a tile to its place

Example: heuristics for the 8-puzzle

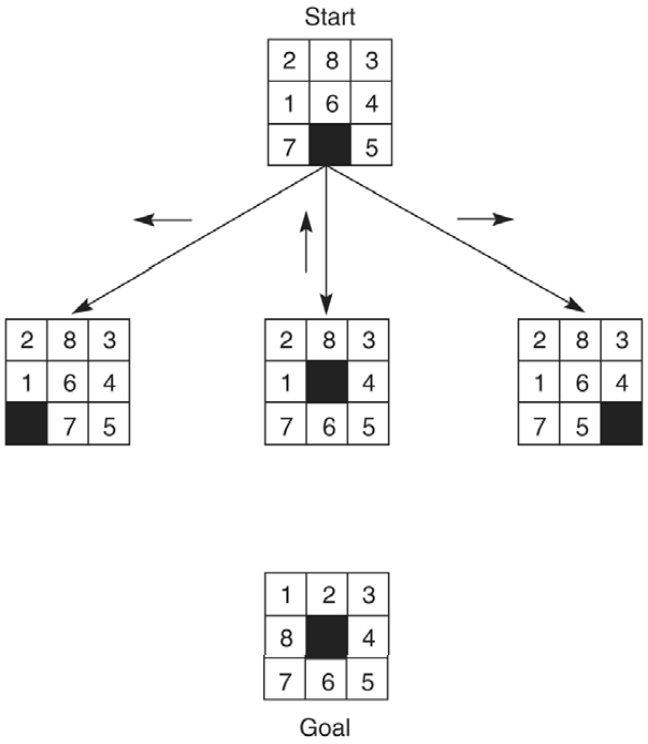
- Neither of these two heuristics gives importance to piece inversion
  - If 2 pieces are arranged contiguously, they have to exchange their positions to be ordered as in a target state, therefore, more than 2 moves are needed to reach said order.
- h_c = twice the number of pairs of pieces whose positions must be exchanged to reach the Target
  - This heuristic is also poor, since it focuses on a certain characteristic
    - In particular, many non-target boards have a value of 0
    - It can be used in combination with other distance measures to design a final heuristic function

- example: 2*1 = 2

1	2	3
8		4
7	5	6

Example: heuristics for the 8-puzzle

- $h_d(n) = h_a(n) + h_c(n)$
  - It is the finest heuristic, but requires more computational effort
  - Could still be improved …



	h_a	h_b	h_c	h_d
2 8 3 1 6 4 7 5	6	5	0	6
2 8 3 1 4 7 6 5	4	3	0	4
2 8 3 1 6 4 7 5	6	5	0	6

Informed search

- Uninformed (blind) search strategies are generally rather inefficient.
- The use of **specific knowledge** about the problem (beyond the definition of the problem) **to guide the search** can greatly improve its efficiency.
  - Informed version of general search algorithms: **best-first search**
    - Best-first greedy search
    - A* search
    - Heuristic search with bounded memory
      - IDA* (iterative-deepening A*)
      - RBFS (recursive best-first search)
      - MA* (A* with bounded memory)
      - SMA* (simplified MA*)
  - Heuristic functions
  - Local search and optimization.
  - Online search and exploration.

Problem solving using informed search

- Introduction
- Best-first search
- Iterative improvement algorithms

Best-first search

Choose the node to expand from *open-list* according to an **evaluation function**, f(*n*), that gives the cost of the lowest-cost path from the initial to a Target state that passes through node *n*.

- Implementation:
  - Use Graph-Search with a **priority queue** for the list of candidates to be expanded (*open-list*).

    Newly generated nodes are inserted into the queue in ascending order of their f values ⇒ nodes with the lowest f-values are expanded first.

- Performance:

  By definition it is optimal, complete and has the least possible complexity, but it is not a search.

Problem: f is generally unknown. We can only use estimates of the distance between a given node and the Target.

Best-first search

- First the apparently best:
  - The most promising node based on locally available information (no search; that is, no successors generated)
  - If we knew which is the best node to expand at each step, this would not be a search but a direct march to the Target
- Selection of the next node to expand according to an evaluation function $f(n)$ that takes into account
  - $g(n)$ = cost of the path from initial node to n
  - $h(n)$ = estimation of the cost necessary to arrive to the lowest cost solution from node n
- The node selected for the expansion will be the one among those in the open-list that minimizes the evaluation function
  - open-list is a priority queue: a list ordered from lowest to highest value for the evaluation function.
  - In this way, simply choose the first node in the list for expansion.
- We will assume that the operators' cost is non-negative (≥ 0).

Best-first search

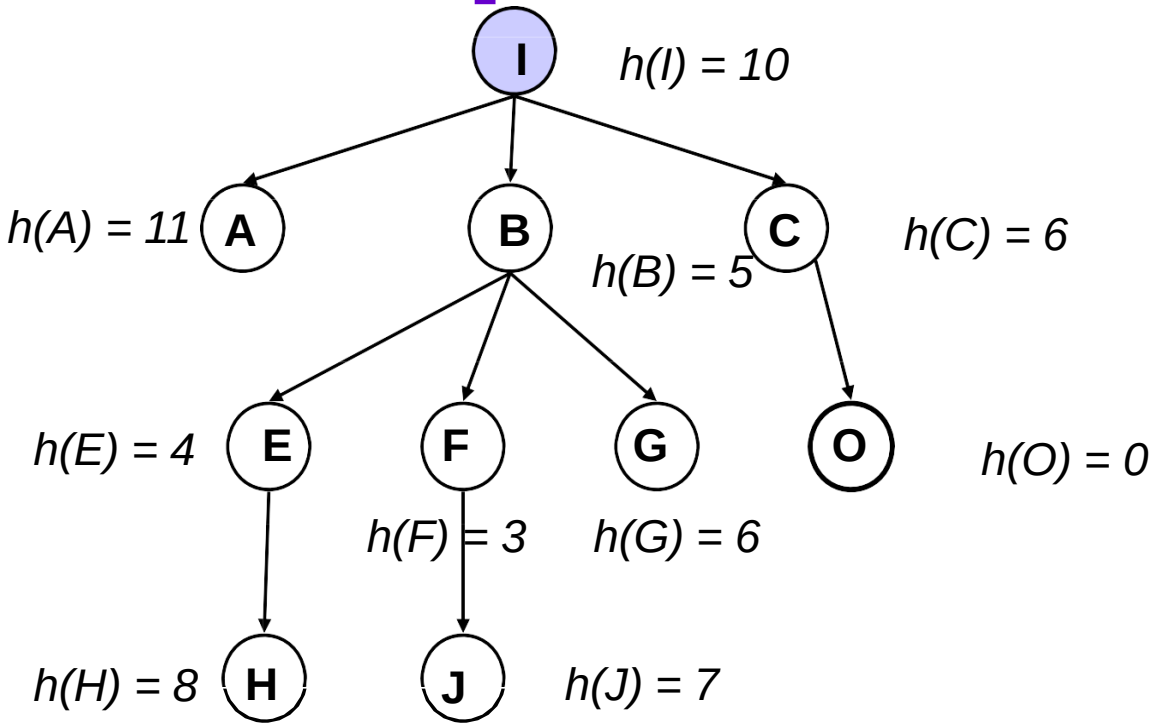
- The evaluation function *f* can be of two types:
  - $f(n) = h(n)$ [1. Greedy best-first search]
    - The cost of the path up to that moment is ignored and only considers what is "missing": the minimum estimated cost to reach a solution from node *n*.
  - $f(n) = g(n) + h(n)$ [2. A * Search]
    - Function *f* is made up of two components:
      - $g(n)$ = cost of the path from initial node to *n*
        - It is not an estimation, but an actual real cost calculated exactly
      - $h(n)$ = minimum estimated cost to reach the target node from *n*
    - $h(n)$ = estimates the total cost of the optimal path from the initial node to a target node passing node *n*.

1. Greedy search

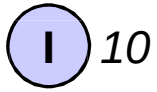
- $f(n) = h(n)$
  - Represents the estimated minimum cost to get from n to a target node (by the least cost path)
  - $h(n) = 0$ for the target nodes
- It is a greedy algorithm
- Properties
  - It is neither optimal nor complete
    - It does not take into account the actual real cost from the initial node to the current node, only an estimation of the cost of the optimal path from the current node to the target.
  - Worst case: if the heuristic is poor, it may have a higher computational cost than uninformed (unrealistic) search
    - Time: $O(b^m)$, where **m** is the maximum depth of the search space
    - Space: $O(b^m)$ (keeps all generated nodes if GraphSearch)
  - If heuristics are good, computational costs could be reduced
    - This depends on the problem and the quality of the heuristic
    - If the heuristics are good, it expands few nodes: low computational cost.

Example: greedy search

State Space



Search Tree

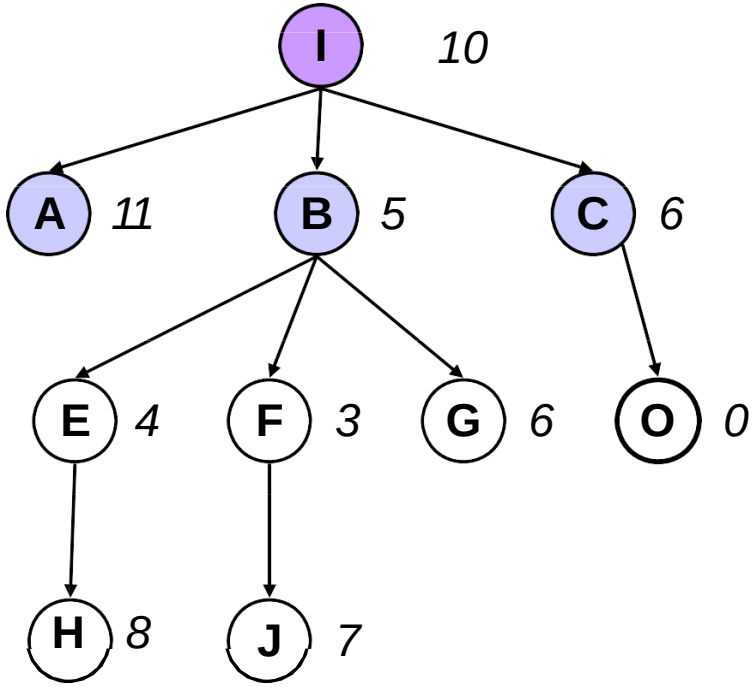


opened-list (priority queue): (I_{10})

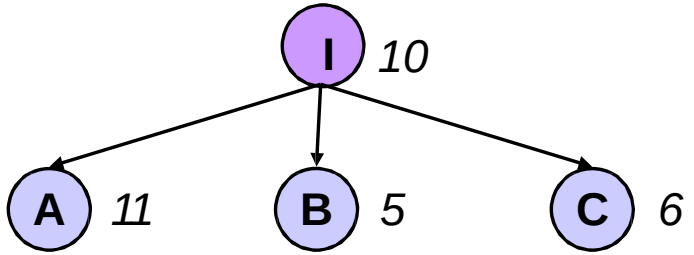
closed-list : ( )

Example: greedy search

State Space



Search Tree

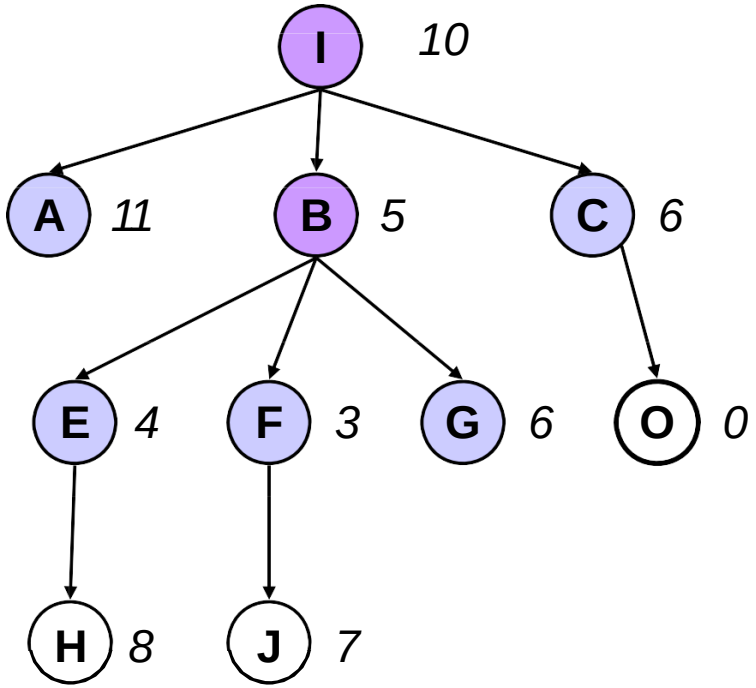


opened-list (priority queue): (B_5 C_6 A_{11})

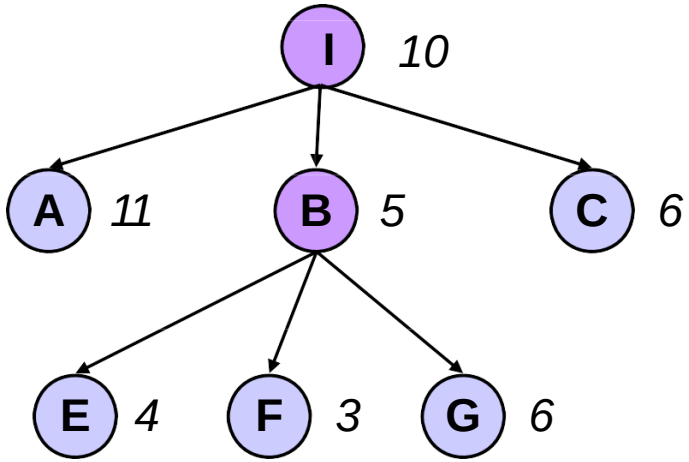
closed-list : (I)

Example: greedy search

State Space



Search Tree

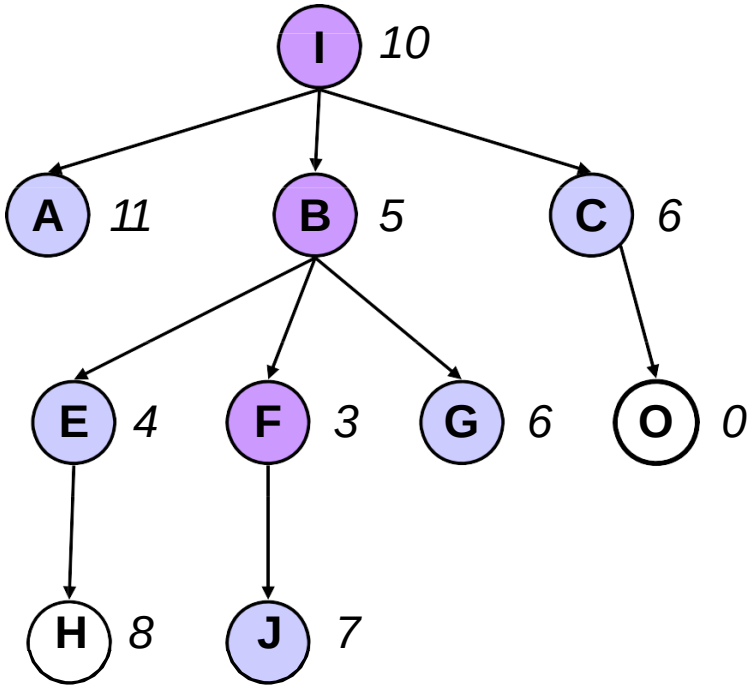


opened-list (priority queue): (F_3 E_4 C_6 G_6 A_{11})

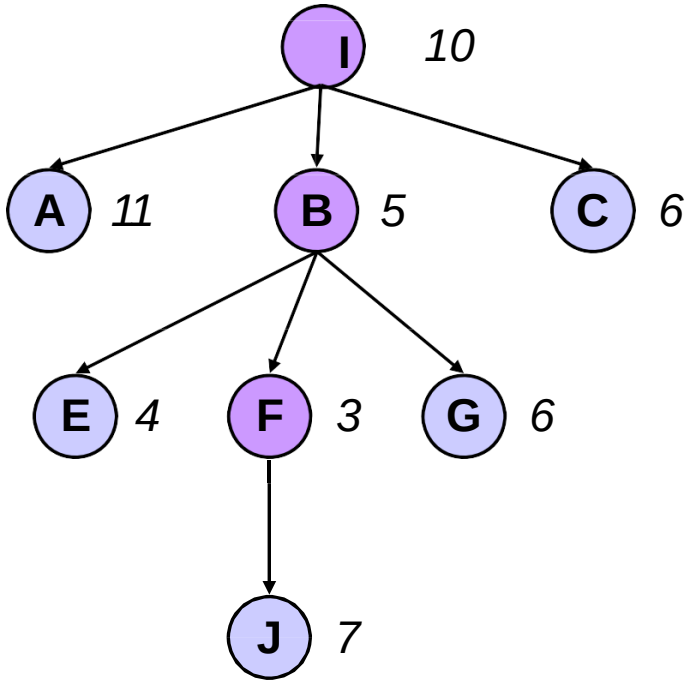
closed-list : (B I)

Example: greedy search

State Space



Search Tree

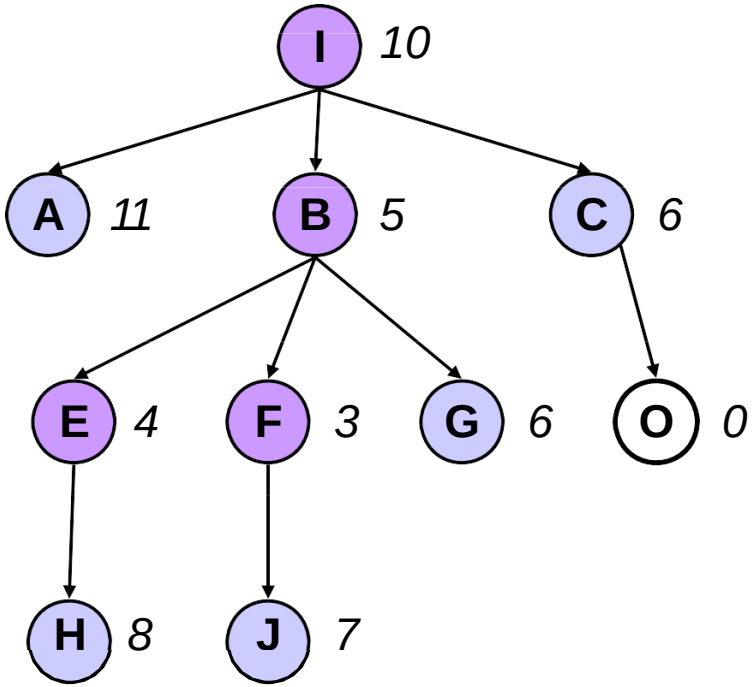


opened-list (priority queue): (E_4 C_6 G_6 J_7 A_{11})

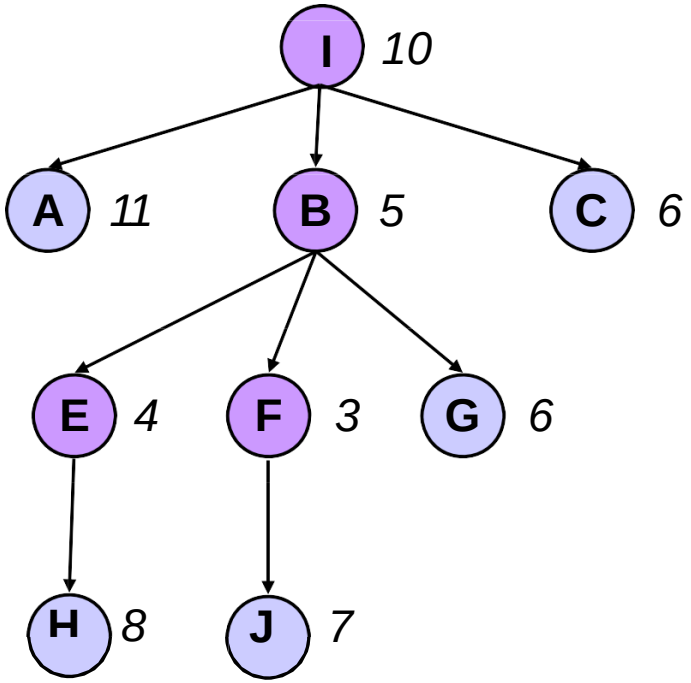
closed-list : (F B I)

Example: greedy search

State Space



Search Tree

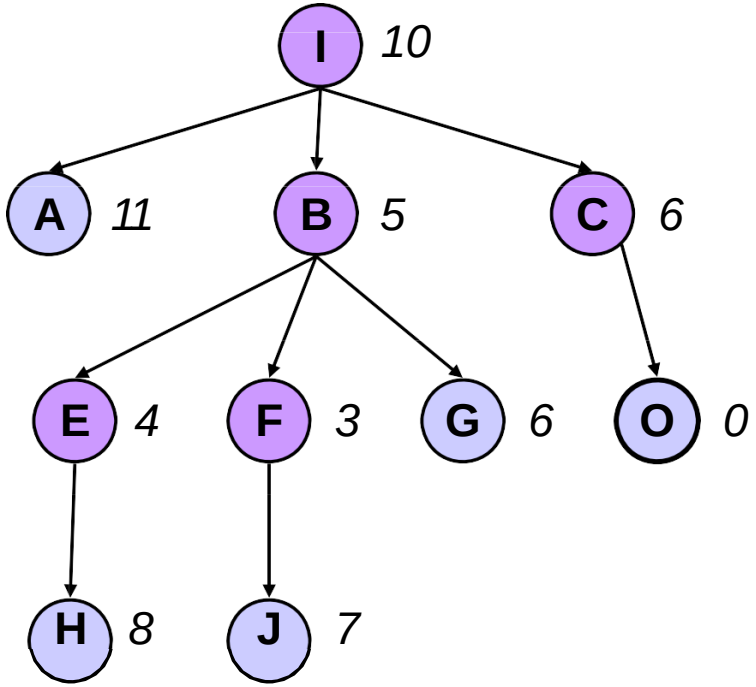


opened-list (priority queue): (C_6 G_6 J_7 H_8 A_{11})

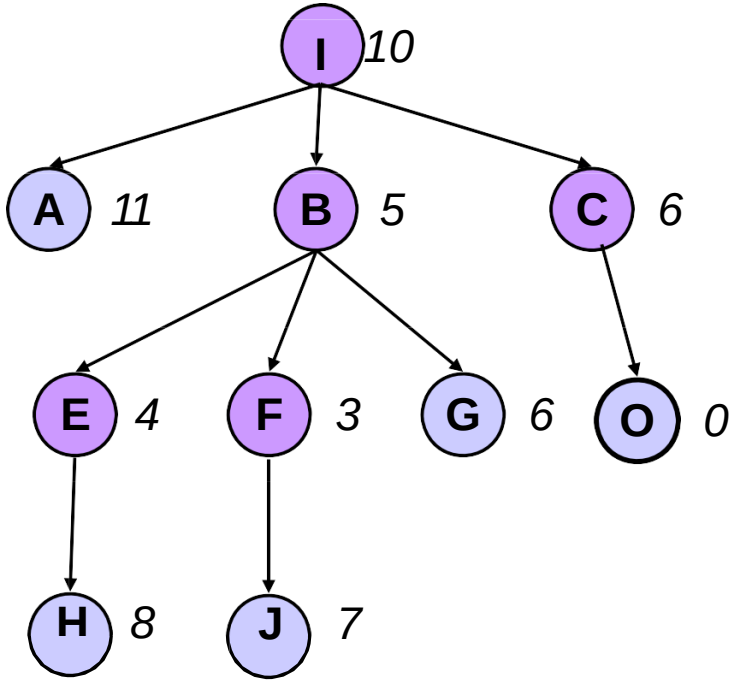
closed-list : (E F B I)

Example: greedy search

State Space



Search Tree

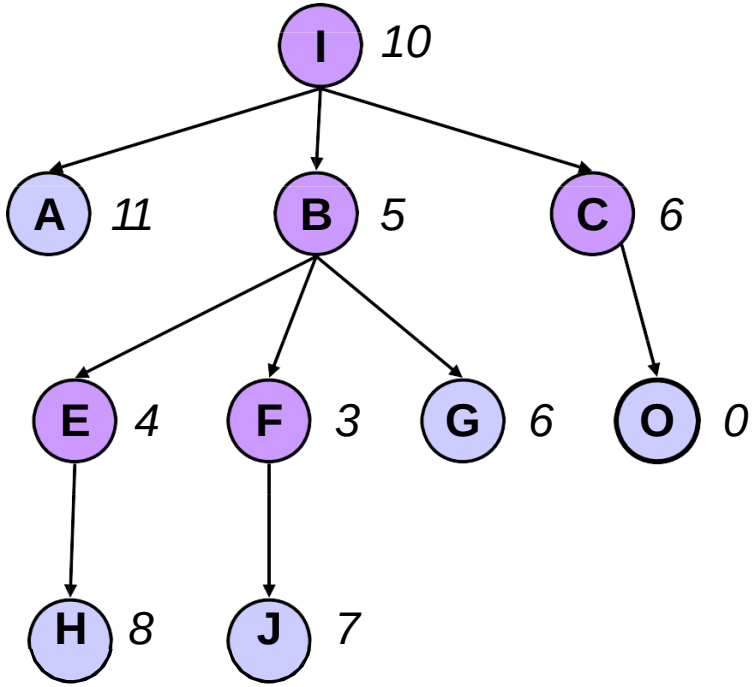


opened-list (priority queue): (O_0 G_6 J_7 H_8 A_{11})

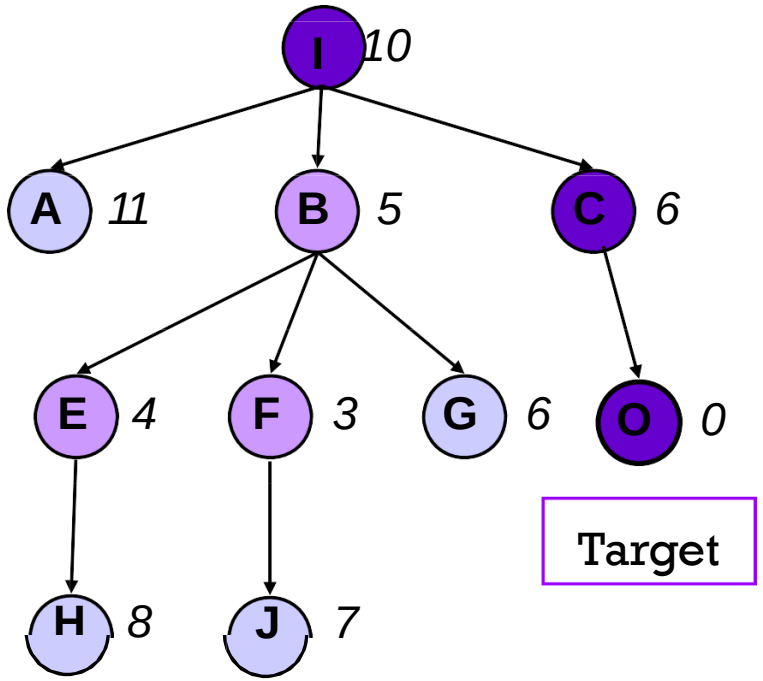
closed-list : (C E F B I)

Example: greedy search

State Space



Search Tree

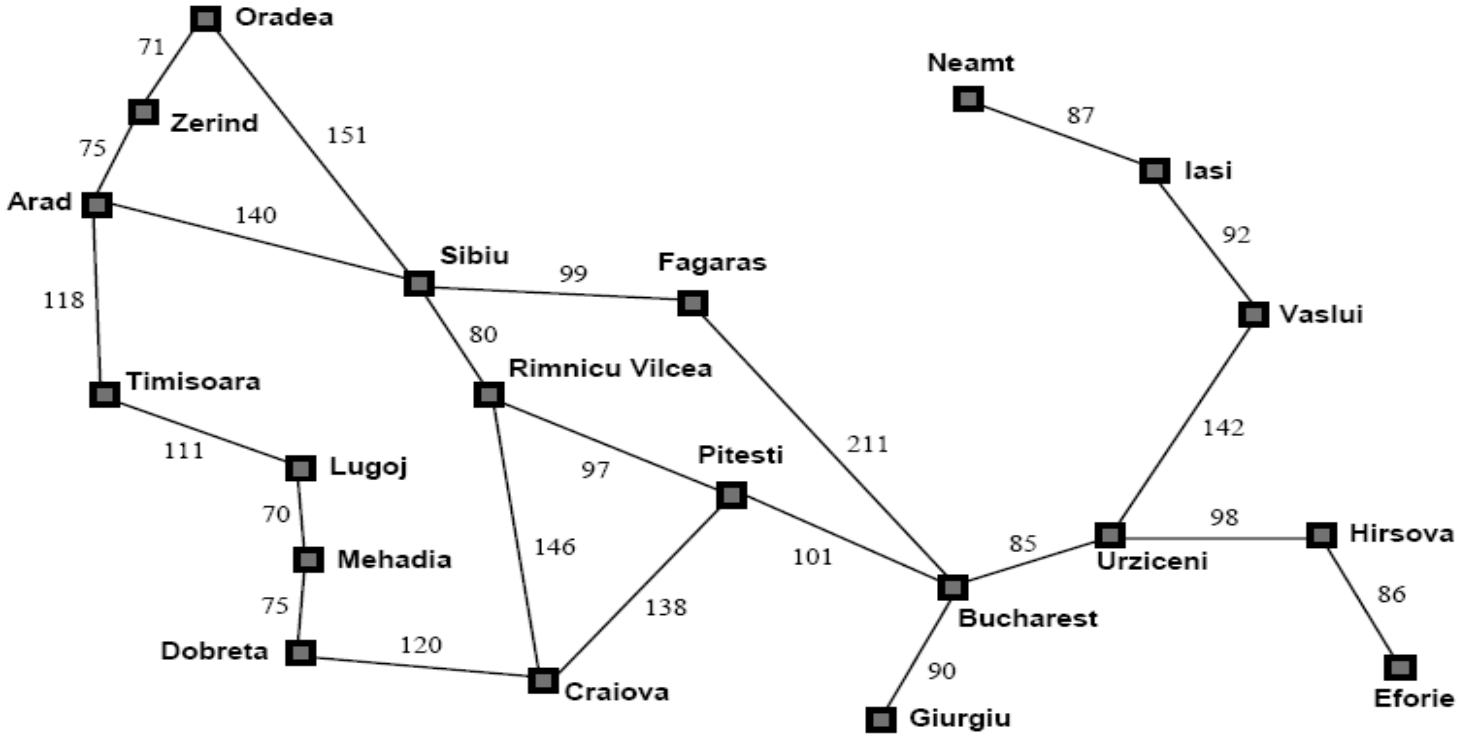


opened-list (priority queue): (G_6 J_7 H_8 A_{11})

closed-list : (C E F B I)

The roadmap problem

- Find the best itinerary between two cities in Romania

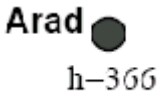


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

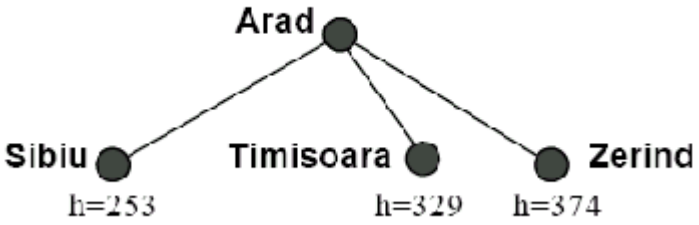
- Heuristic function (admissible, monotonic)
  - h(n) = straight line distance from city n to **Bucarest.**

Greedy search for the roadmap problem

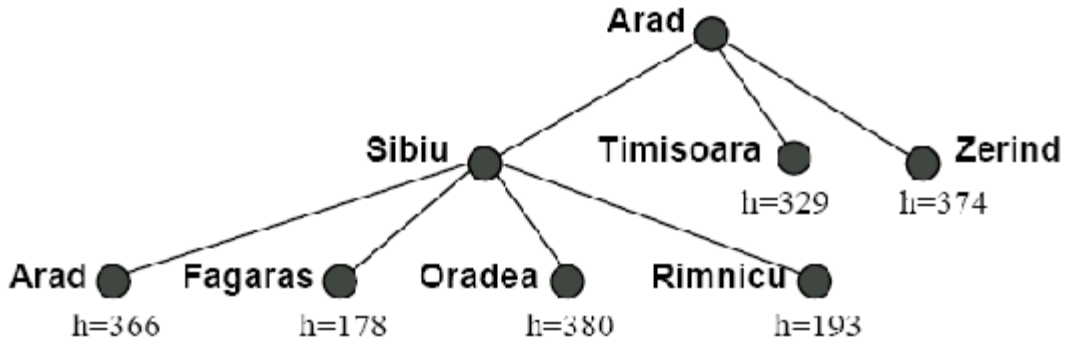
Initial state:
Departure city



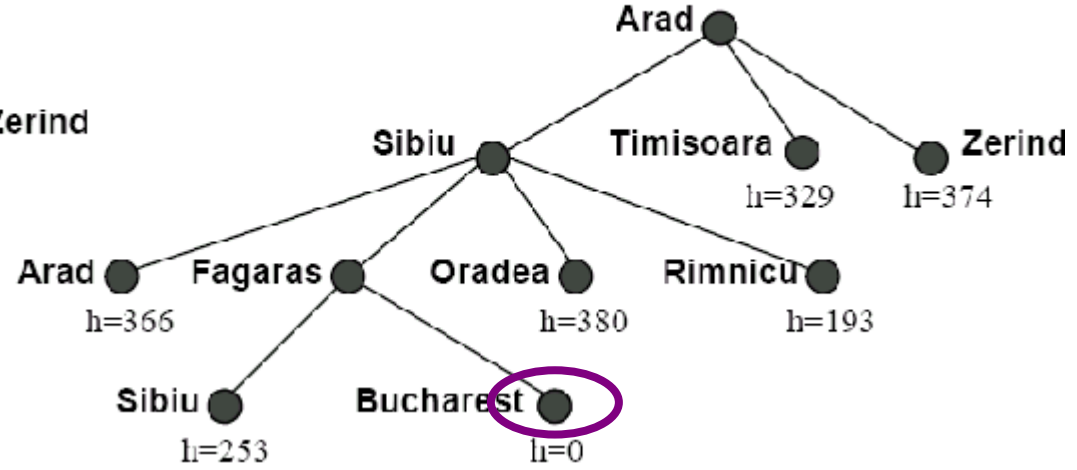
1. Expand Arad



2. Expand Sibiu (leaf node with lowest h, 253)



3. Expand Fagaras (leaf node with lowest h, 178)

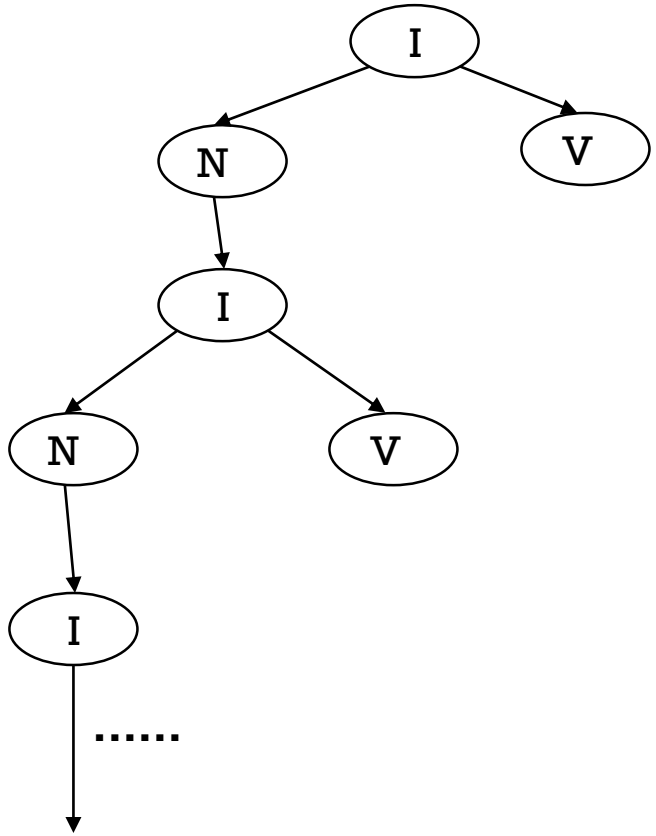


4. Target achieved: Bucharest

Greedy search: repeated states

If we do not keep track of possible loops, greedy **best-first** search can fail to find a solution

- Example: find the itinerary between "**Iasi**" and "**Fagaras**":



I: Iasi
N: Neamt
V: Vaslui

2. A* search

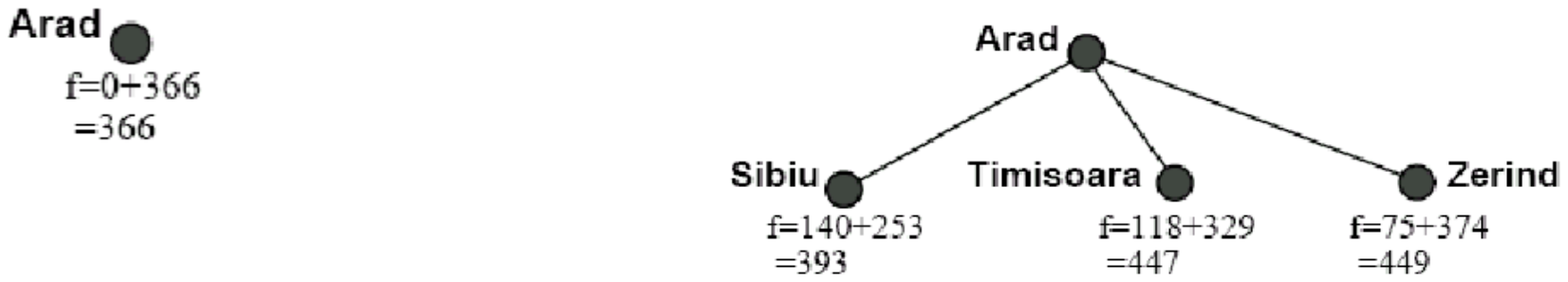
- $f(n) = g(n) + h(n)$
  - f (n) = estimates the total minimum cost (from initial node to target) of any solution that passes through node *n*
    - $g(n)$ = real cost of the path to *n*
    - $h(n)$ = estimation of the minimum cost from *n* to a target node
  - If $h = 0$ => uniform cost search ("uniformed": 1st lowest cost)
  - If $g = 0$ => greedy search
- Combines breadth-first search with depth-first search
  - The *g* component of *f* gives it a realistic touch
  - Preventing it from being guided exclusively by an overly optimistic h
    - *h* tends to depth-first
    - *g* tends to breadth-first : forces backward when dominating *h*
- It changes its path each time there are other more promising

Conditions on h

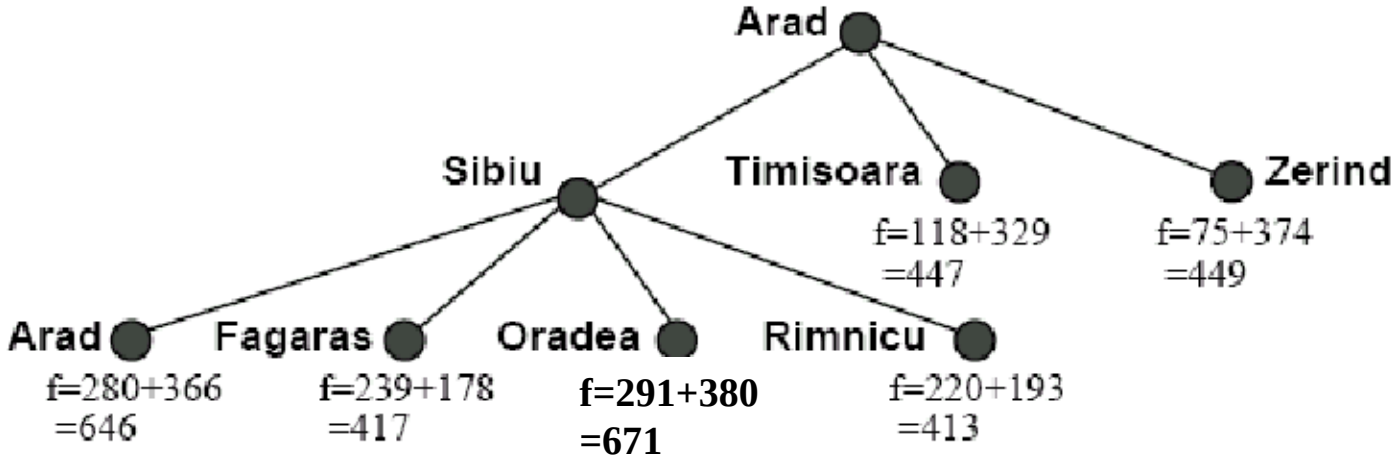
- *h* is an admissible heuristic if $h(n) \leq g^*(n)$ for all *n*
  - $g^*(n)$ is the real cost to go from node *n* to a target node by the least cost path (optimal cost).
  - *h* does not overestimate for any node the optimal cost to achieve the objective.
  - It is, therefore, an optimistic heuristic.
- The A * search without elimination of repeated states and with an admissible heuristic is optimal (that is, it guarantees to find the lowest cost solution)

A* search: Roadmap I

1. Expand Arad

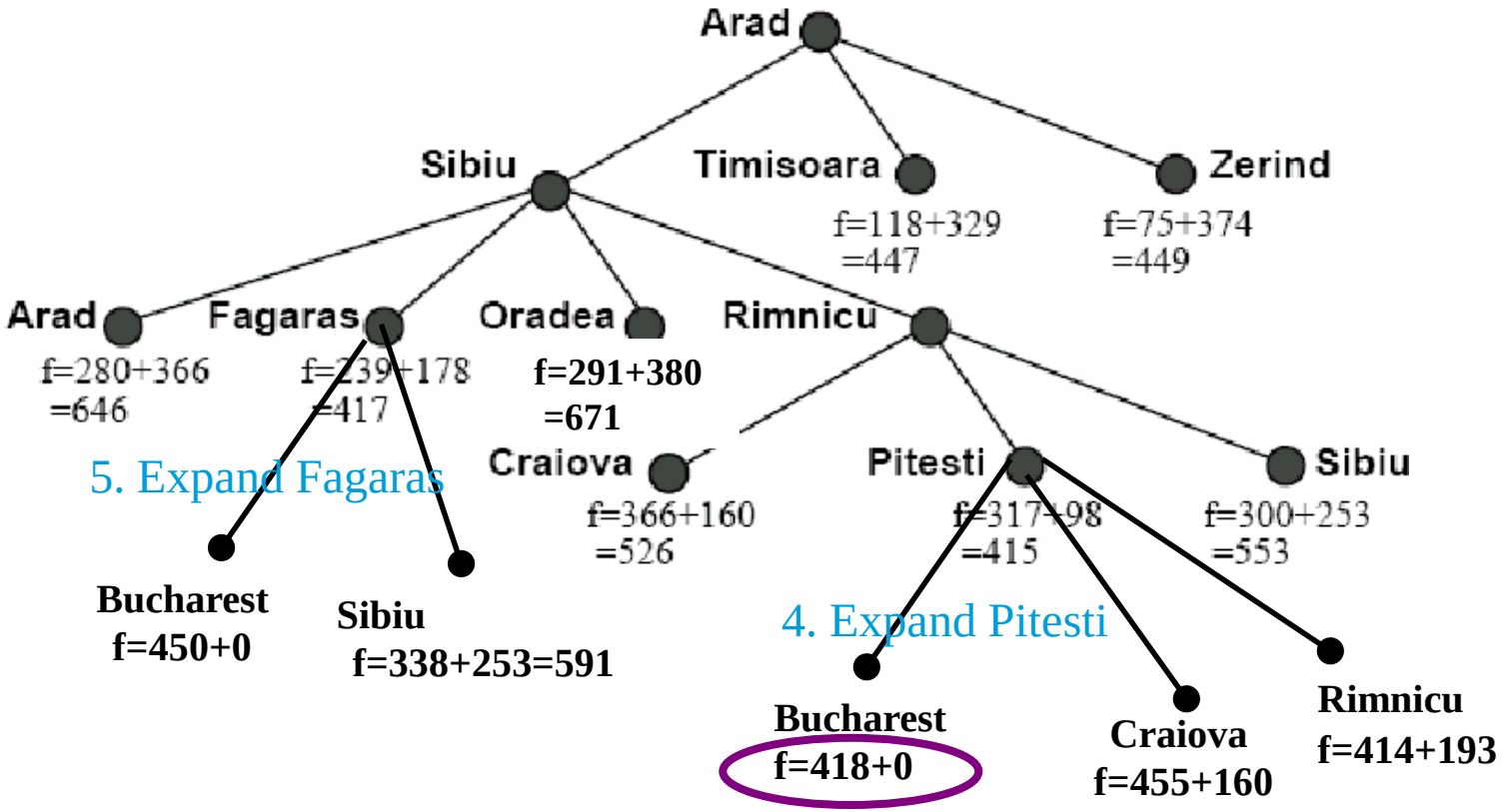


2. Expand Sibiu



A* search: Roadmap II

3. Expand Rimnicu (lower f, 413)



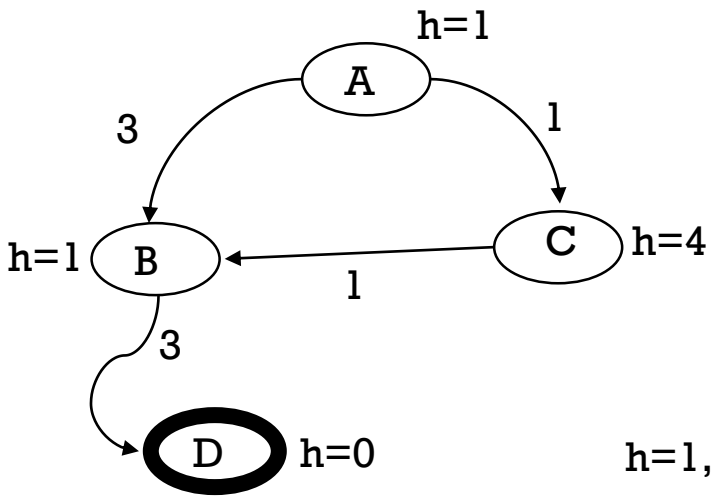
6. Target found: Bucharest

A* Implementation

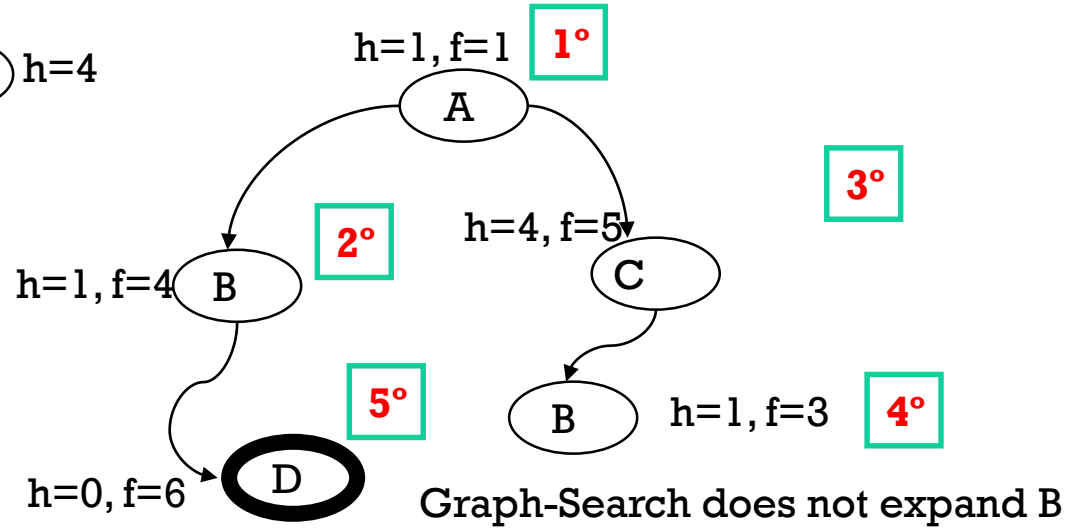
- A * can be implemented as
  - *Tree-search* (no removal of repeated states)
  - *Graph-search* (with elimination of repeated states)
- In the *Graph-search* implementation we need:
  - Information stored for node n:
    - Description of the status corresponding to that node.
    - Reference to the parent node: to rebuild the path when finding the solution
    - Values of **f (n) = g (n) + h (n)**
  - Two structures to store the nodes:
    - opened-list: (top of the search tree): priority queue in which the nodes generated but not yet expanded are ordered from lowest to highest value of the evaluation function (f)
    - Closed-list: Nodes already expanded.
    - They are not removed when generated, but when trying to expand them.
  - PROBLEM with the elimination of repeated states:
    - Even though with an admissible heuristic, the optimal solution may be obtained by exploring the node that is chosen to be eliminated because it has already been expanded.

A* + admissible heuristic

- If **Graph-Search** (i.e. with elimination of repeated states) is used, even if **h is admissible, A* may not be optimal:** Suboptimal solutions can be generated if the optimal path to a repeated state is not the first one generated.



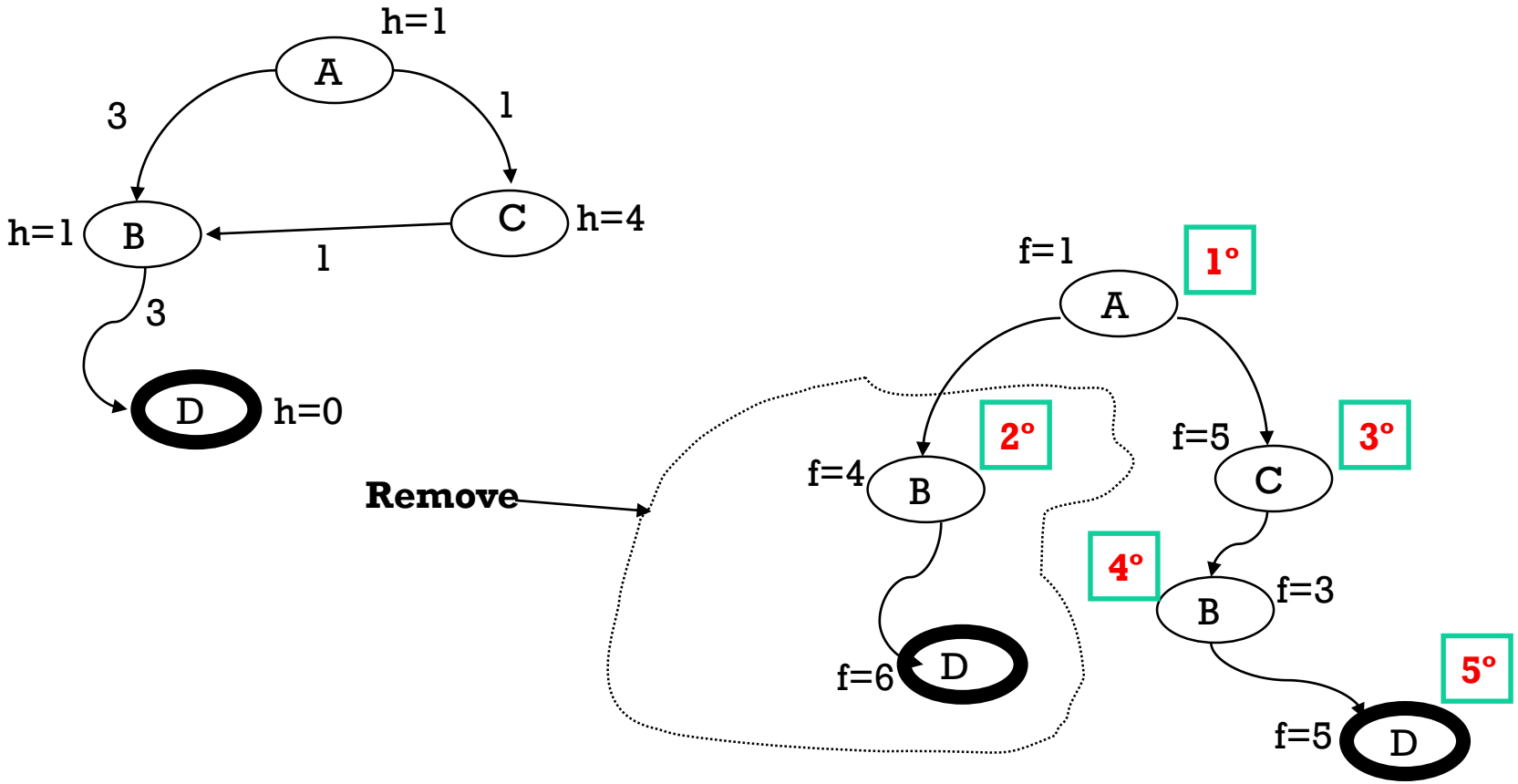
The solution found is suboptimal: cost = 6



Graph-Search does not expand B (B is in closed-list)

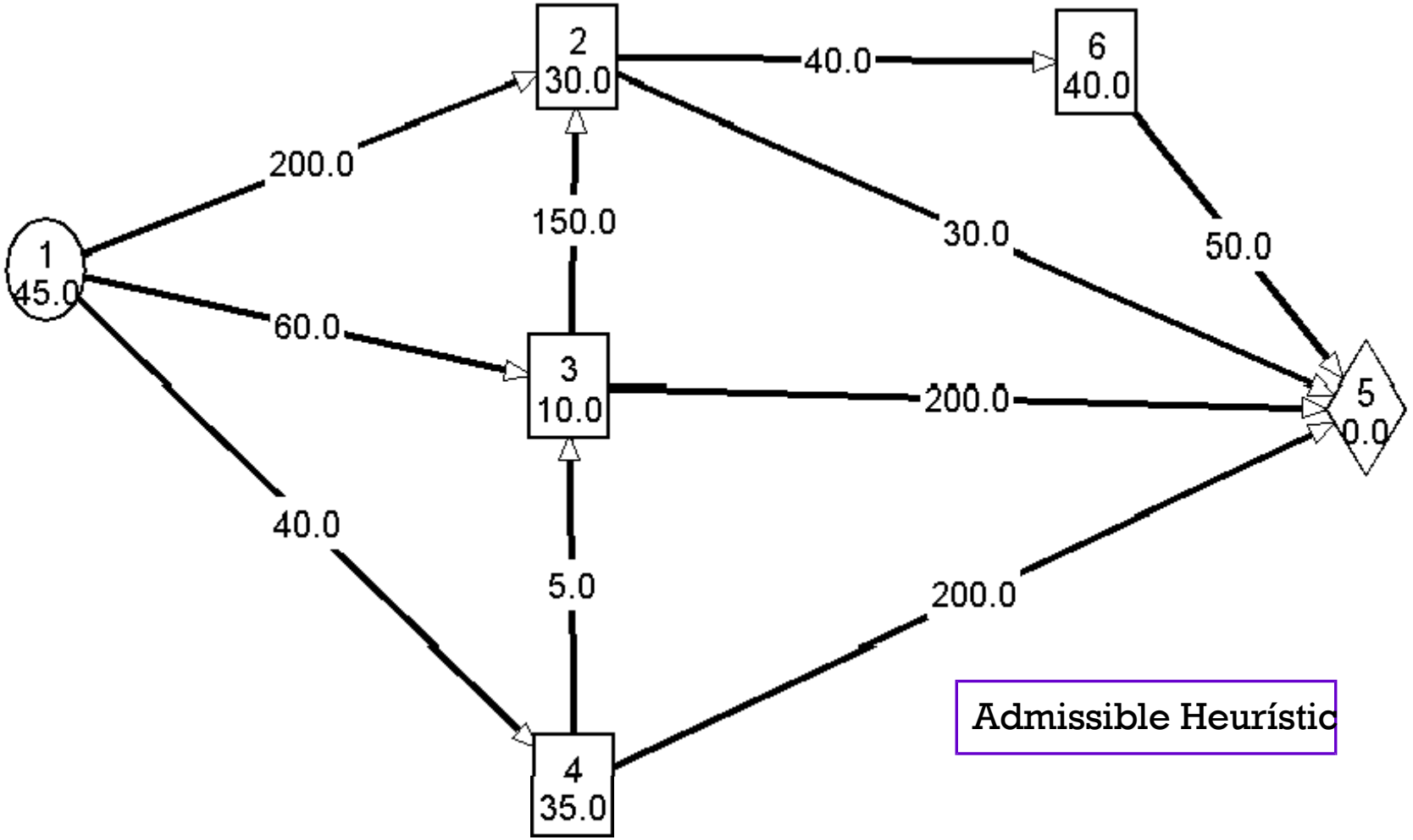
A* + admissible heuristic

- Solution: Discard the **path** with highest cost.
  - Increases the complexity of the algorithm: it needs to eliminate from open-list the node with highest cost and its descendants.



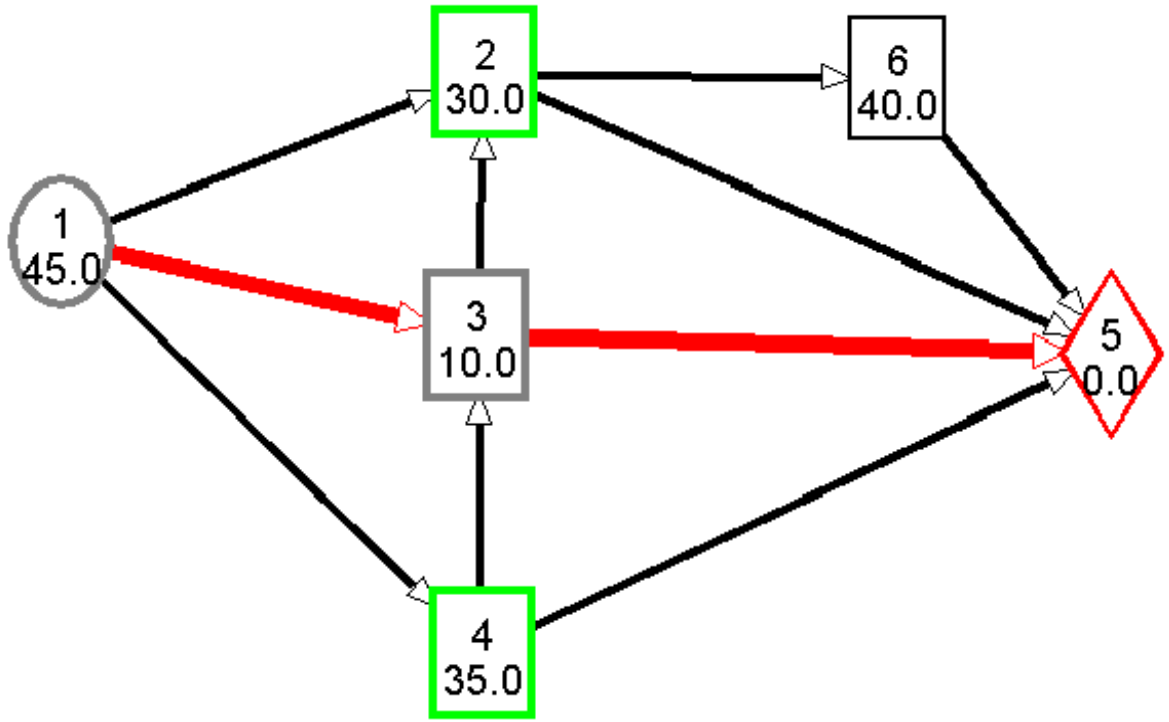
The solution found is optimal: cost = 5

Example: learning about admissible heuristics



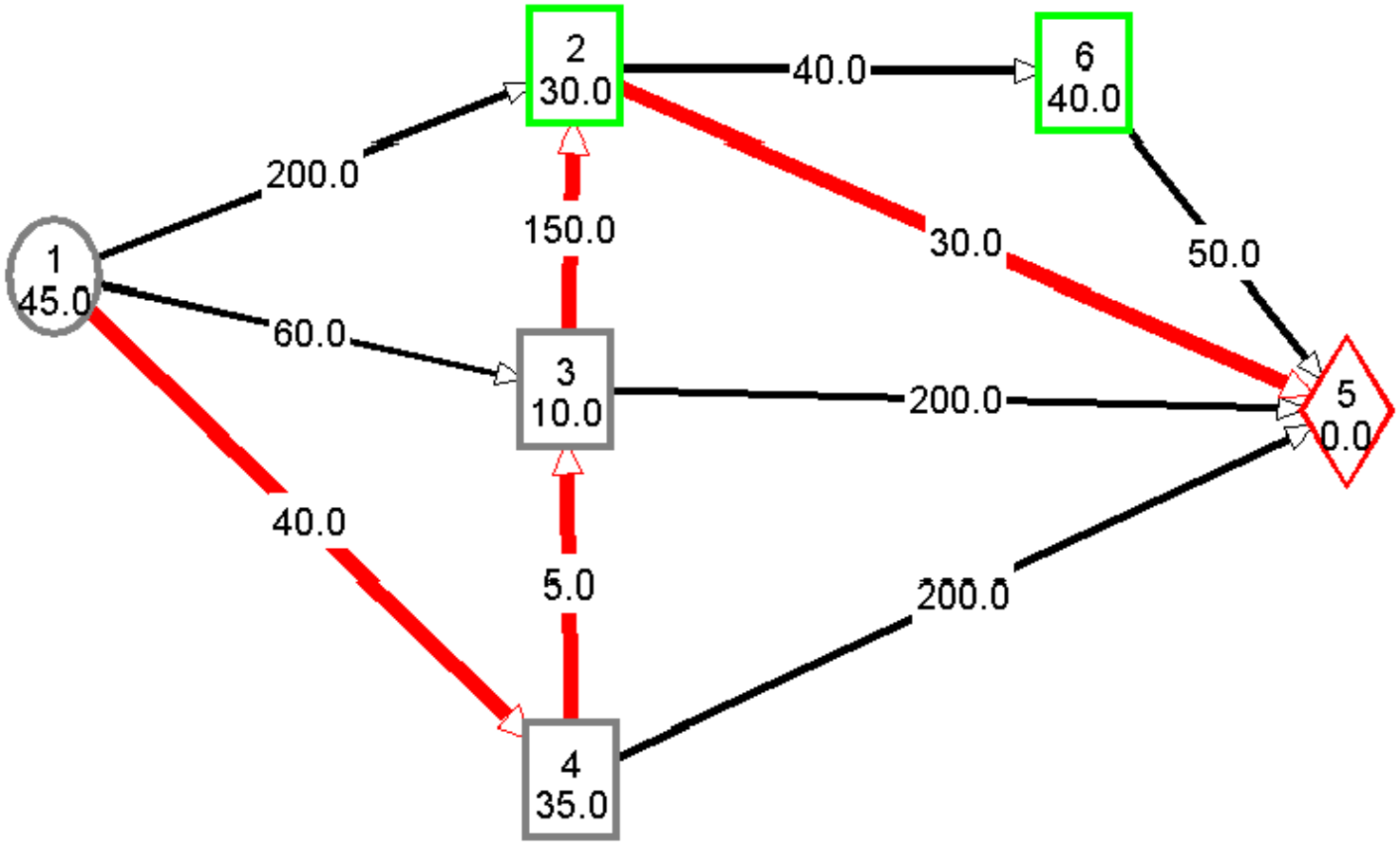
Always optimistic
→ heuristic less or equal than the real cost from there to solution

Solution with greedy search



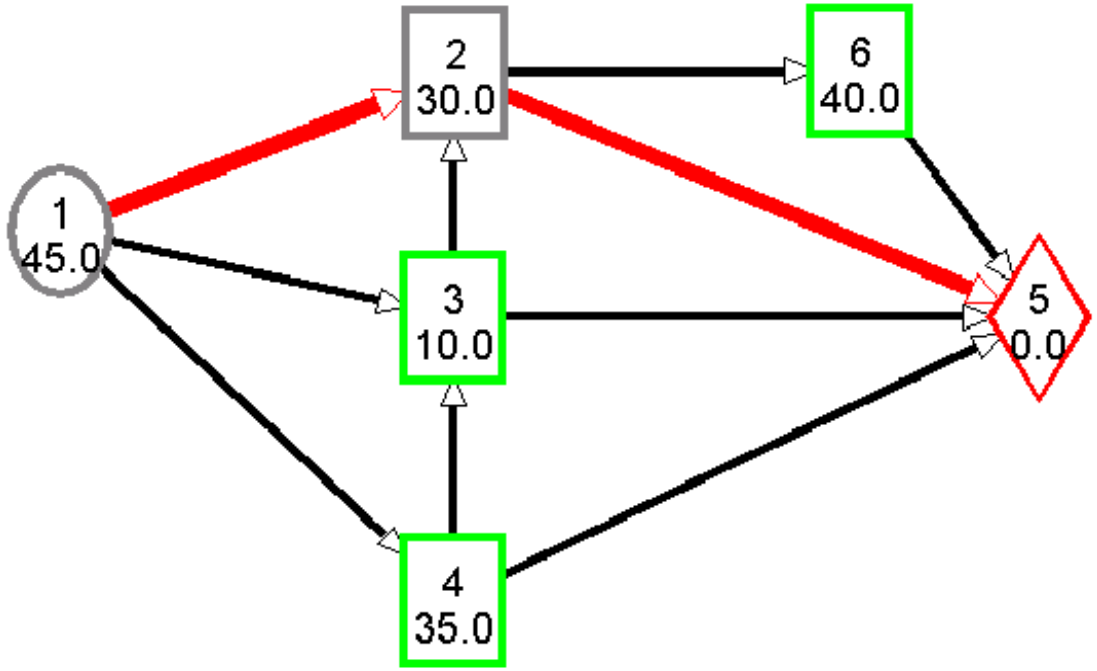
Greedy search: 1-3-5 Cost (not taken into account): 60+200 = 260

Solution with A* + tree-search (without elimination of repeated states)



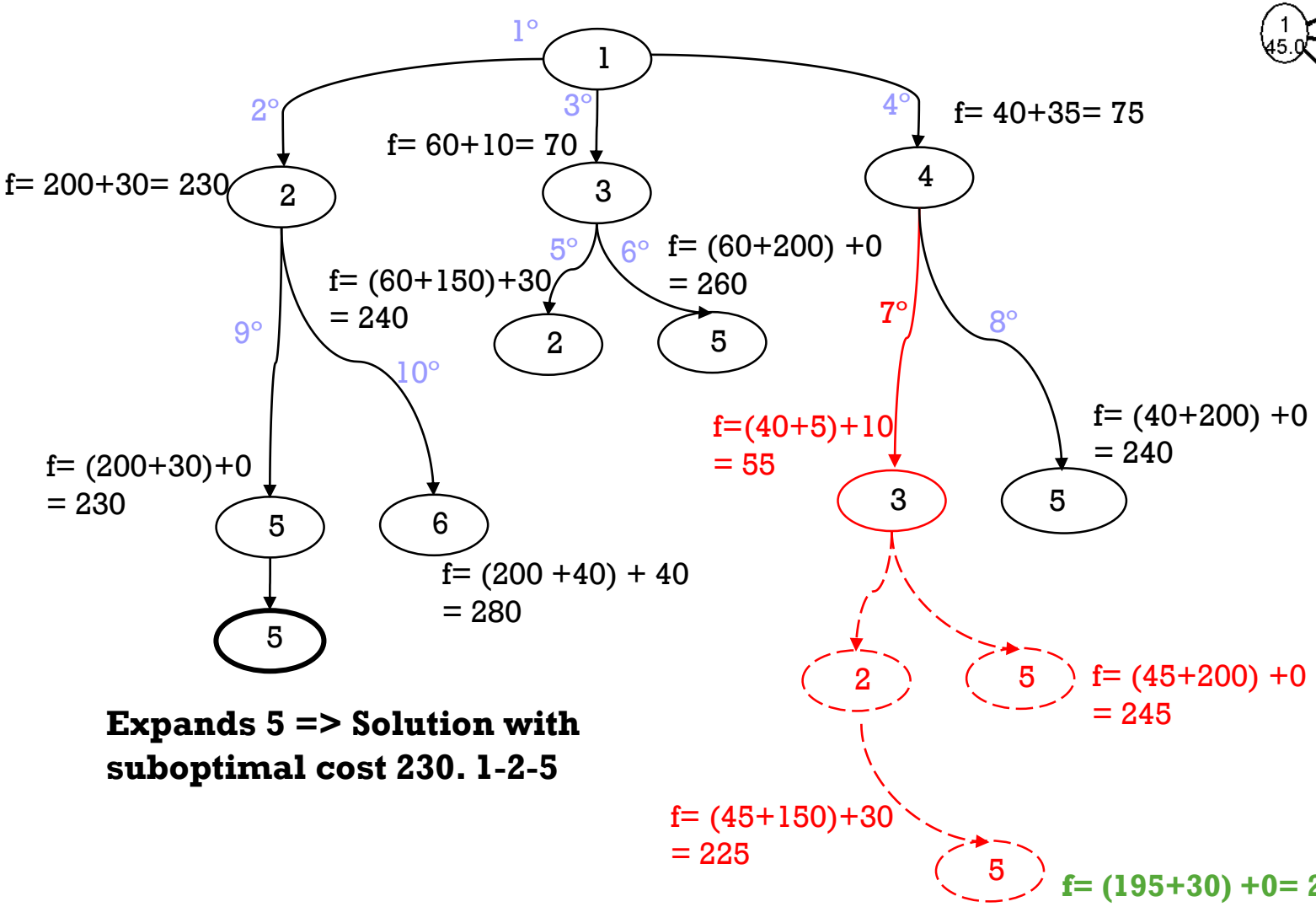
Solution with A* + tree-search: 1-4-3-2-5
Cost: 40+5+150+30 = 225 (optimal)

Solution with A* + graph-search (with elimination of repeated states)

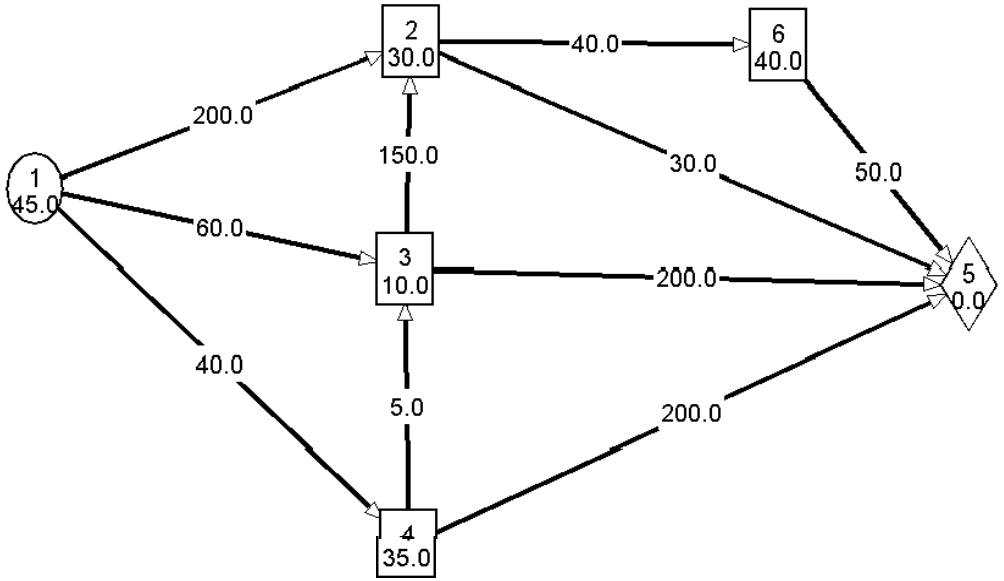


Solution with A* + graph-search: 1-2-5
Cost: 200+30 = 230 (suboptimal)

In detail

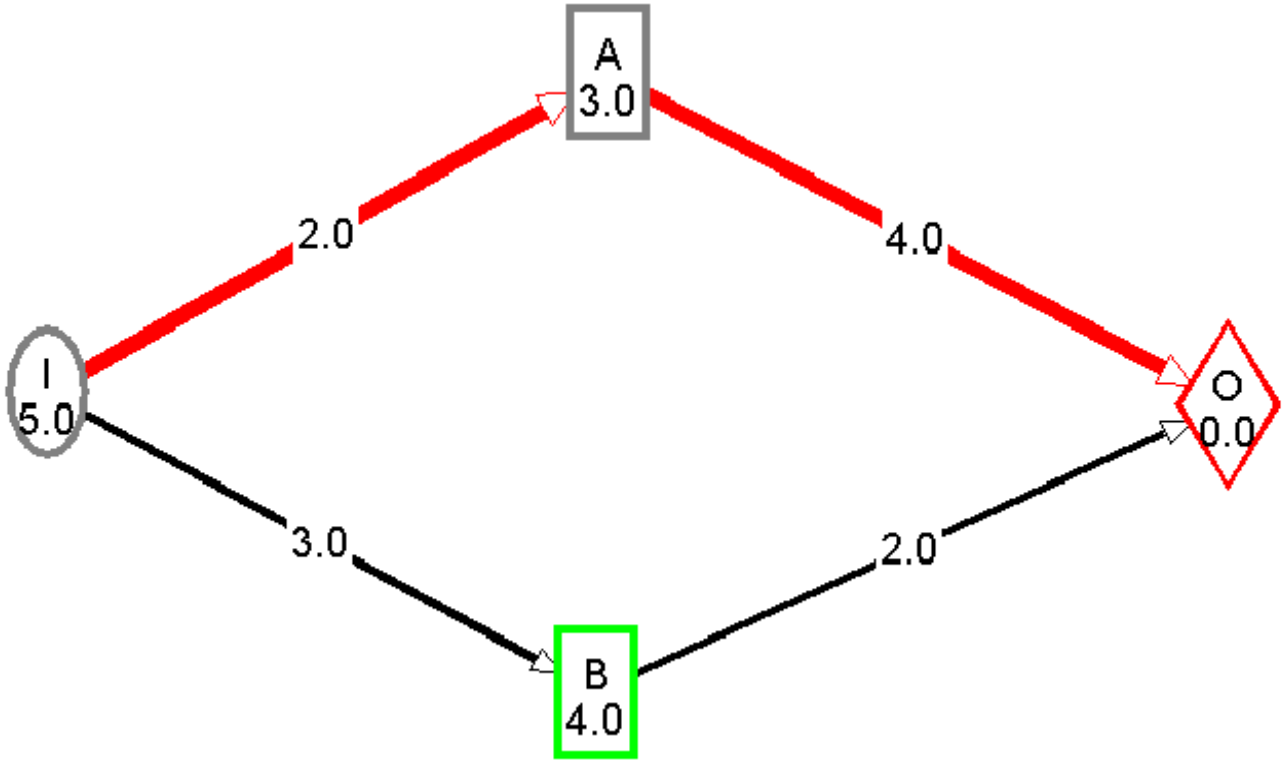


Expands 5 => Solution with suboptimal cost 230. 1-2-5



Graph-search does not expand 3 again (it's closed-list) so red path never happens

A * + elimination of repeated states + inadmissible heuristic does not guarantee finding the lowest cost solution



Inadmissible heuristic: does not guarantee finding the lowest cost solution

Properties of A*

- Theorem:

  Using **tree-search** (without elimination of repeated states) and if **h is admissible** $h(n) \leq h^*(n), \forall n$ $\Rightarrow$ **A* is complete and optimal**

Demonstration:

- C* : cost of the optimal solution
- Consider a suboptimal target node G_2 (i.e. $g(G_2) > C^*$, $h(G_2) = 0$) that is included in the fringe of the search tree:

  $f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^* \Rightarrow f(G_2) > C^*$ **(1)**

- Consider the node n on the fringe of the tree that is in the optimal solution path.
  - Since n is in the optimal solution path, $g(n) = g^*(n)$
  - Since h is admissible: $h(n) \leq h^*(n)$

  $f(n) = g(n) + h(n) \leq g^*(n) + h^*(n) = C^* \Rightarrow f(n) \leq C^*$ **(2)**

(1)+(2) $\Rightarrow$ $f(n) \leq C^* < f(G_2)$ and n is explored before G_2

Monotonic heuristic (or "consistent")

- A **heuristic function** h(n) is **monotonic** if it satisfies the following **triangular inequality:**

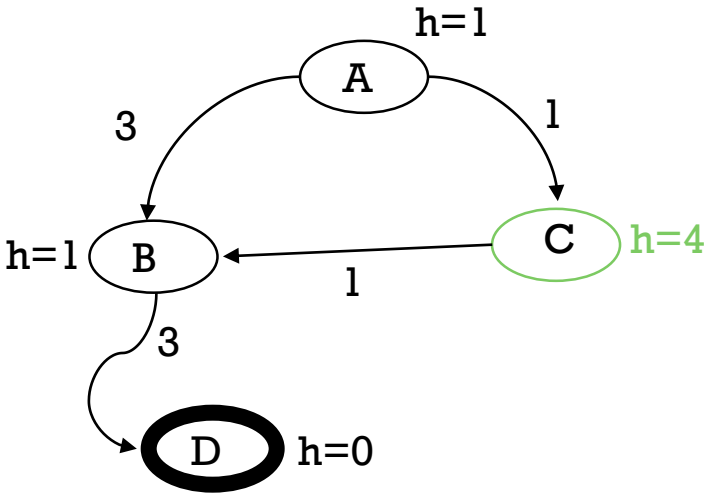
$$h(n) \leq \text{cost}(\text{n} \rightarrow \text{n}') + h(n'),$$
$$\forall n, n'[n' \text{ successor of } \text{n}]$$

  - Example: In the roadmap problem the straight line distance is a monotonic heuristic function.

- If **h is monotonic** $\Rightarrow$ **h is admissible**

  [Exercise: Demonstrate this]

- There are admissible heuristic functions which are not monotonic



A* + monotonic heuristic

- THEOREM:

  If **h is monotonic** $\Rightarrow$ the values of **f(n)** along the path searched by A* are **non-decreasing**

  Proof:

  Suppose that n' is a successor of n

  $f(n') = g(n') + h(n') = g(n) + cost(n \rightarrow n') + h(n')$

  $\geq g(n) + h(n) = f(n) \quad \Rightarrow \quad \mathbf{f(n') \geq f(n)}$

- THEOREM:

  If **h is monotonic** $\Rightarrow$ **A*** using **Graph-Search (with elimination of repeated states) is complete and optimal.**

  Proof:

  Since f(n) is non-decreasing the first expanded target node must be the one corresponding to the optimal solution.

A* + monotonic heuristic

- THEOREM: If h is monotonic and A* has expanded a given node n, then g(n)=g*(n)

  Being g*(n) the cost of the optimal path between the initial node and n

  Proof: Let us consider the same problem with same initial state but having node n as the target state. Let us define the new heuristic for this new problem:

  $$h'(m) = h(m) - h(n), \qquad \forall m \ f(m) \leq f(n).$$

  Since the difference between h and h' is a constant:

  - The searches (A* with h) and (A* with h') starting from the same initial state, explore the same sequence of nodes before expanding n.
  - h' is a monotonic heuristic for the new problem.

  Since A* with a monotonic heuristic is complete and optimal, the search (A* with h') finds the optimal path between the initial state and the node n. This path will also be the optimal one for the search (A* with h) $\Rightarrow$ g(n)=g*(n).

- THEOREM: **A*** is optimally efficient.

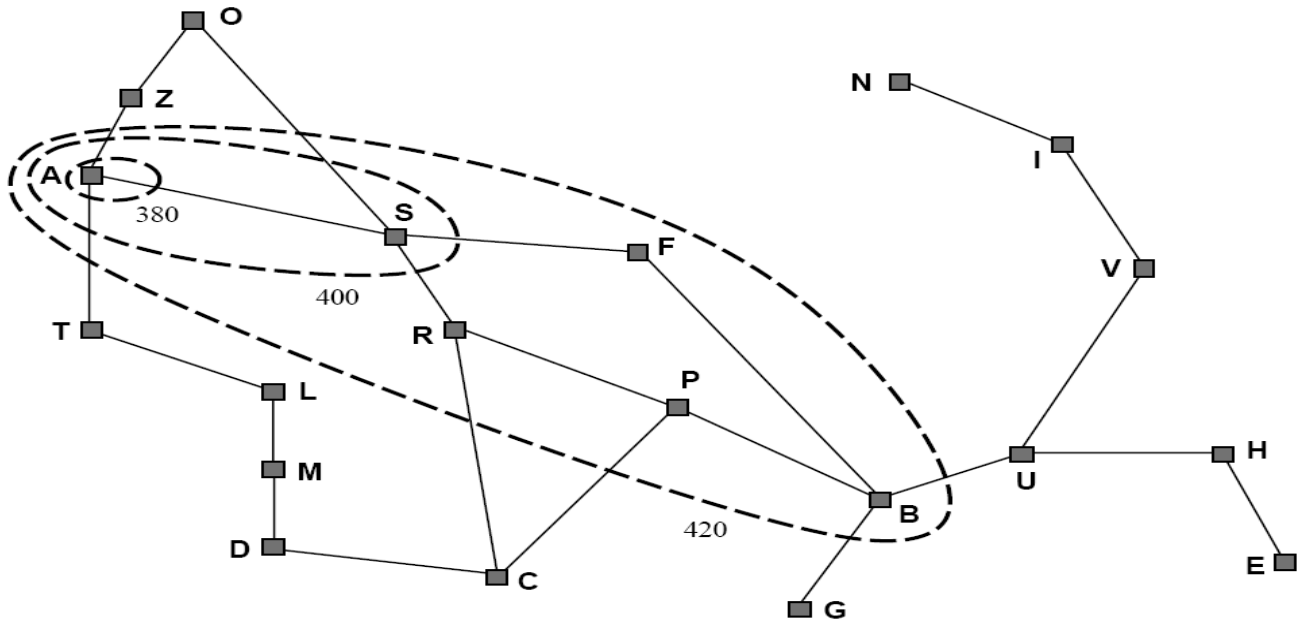
  For a given heuristic, no other algorithm will expand fewer nodes than those expanded by A * (except for possible ties)

  Proof: if there are nodes that do not expand between the origin and the optimal level curve, it is not guaranteed that the algorithm finds the optimal solution.

  [Dechter, Pearl, 1985]

A* + monotonic heuristic

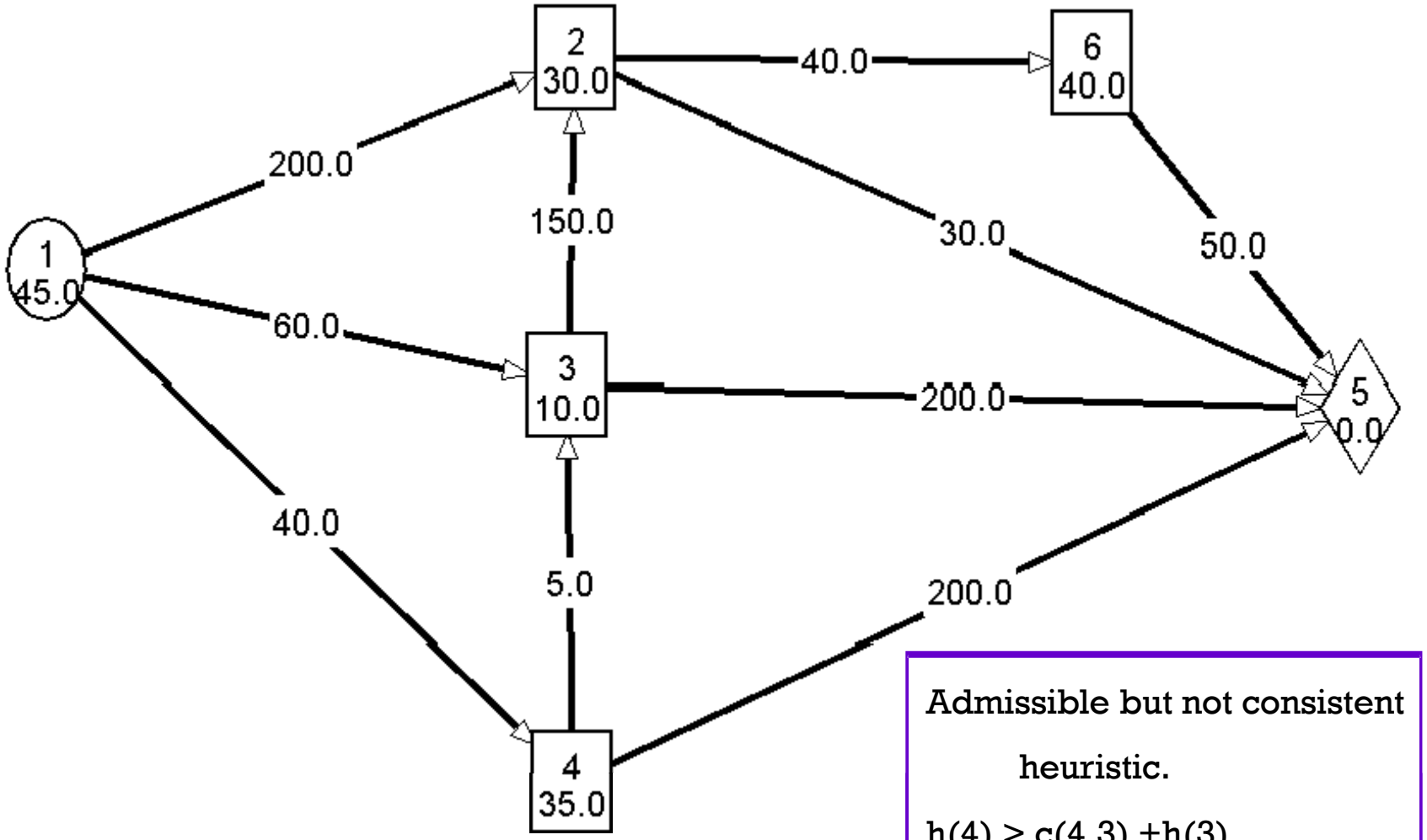
- If h is a monotonic heuristic, the exploration is performed in contour lines with increasing values of f(n).
  - In a uniform cost search [h (n) = 0], the contours are "concentric" around the starting state.
  - In better heuristics these curves form bands that extend into the target state.



Summary. Optimality of A *. GraphSearch

- Consistency of h (or monotony of f)
  - h is consistent if, for each node *n* and each successor *n'* of *n*, the estimated cost of reaching the Target from *n* is not greater than the real cost of reaching *n'* plus the estimated cost of reaching the target from n '
    - h (n) $\leq$ c (n, n ') + h (n') (*triangular inequality*)
    - All consistent heuristics are also admissible (*but not the other way around*)
    - If *h* is consistent then the values of *f* along any path do not decrease (*monotonic non-decreasing f*)
  - If f is monotonic non-decreasing, the sequence of nodes expanded by A * will be in non-decreasing order of *f (n): f (n) <= f (n ')*
    - The first target node selected for expansion should be an optimal solution, since all subsequent ones will be at least as expensive as it
  - If *h* is consistent, each time you expand a node
    - It will have found an optimal path to that node from the initial
    - It increases efficiency by not needing to revisit nodes: 1st expansion, the best

In the example before..



Admissible but not consistent heuristic.

$h(4) > c(4,3) + h(3)$

Optimality of A * according to its implementations

- A * + tree-search (without elimination of repeated states)
  - It is optimal if the heuristic is admissible.
- A * + graph-search (with elimination of repeated states)
  - It is optimal if the heuristic is consistent.

To guarantee the optimality of A *

- If h is not consistent, but it is admissible, we can make it consistent
  - *h* can be dynamically modified during search so that satisfies the consistency condition $h(n) \leq c(n, n') + h(n')$
    - At each step, we check the values of *h* for the successors of node n that has just been expanded
    - If for any of these values of h it is true that $h(n') < h(n) - c(n, n')$ then we make $h(n') = h(n) - c(n, n')$
      - In the example: new value of h (3) = h (4) - c (4, 3) = 35 - 5 = 30
- Behaviour of A * with consistent h
  - If f * is the cost of the optimal solution, then
    - A * expands all nodes with f (n) <f *
    - A * could expand some nodes directly on the "target contour line "(where f (n) = f *) before selecting a target node
    - A * will not expand any node with $f(n) > f^*$ (there is the pruning)

Completeness and efficiency of A * with consistent heuristics

- Completeness
  - If there is a solution, it will reach a target node, unless there is an infinite sequence of nodes n in which f (n) ≤ f * is fulfilled
  - This can happen if
    - There are nodes with infinite branching factor, or
    - If there are finite cost paths with an infinite number of nodes
  - A * is complete
    - if the branching factor b is finite
    - and there exists a constant ε> 0 such that
      - the cost of any operator is always ≥ ε
- It is optimally efficient
  - No other optimal algorithm guarantees to expand fewer nodes than A *
    - Except perhaps tie-breaks between nodes with the same value of f
    - This is because any algorithm that does not expand all nodes with f (n) <f * runs the risk of missing the optimal solution

A* complexity

- With the established constraints, the A * search is complete, optimal and optimally efficient,
  - but A * is not the answer to all search needs
- In the worst case it is still exponential: $O(b^{\tilde{d}})$, $\tilde{d} = \frac{C^*}{\varepsilon}$

  C* = optimal cost; ε = minimum cost per action

- Exponential growth does not occur if
  - The error in the heuristic does not grow faster than the logarithm of the actual cost $|\, h(n) - h^*(n) \,| \leq O(\log h^*(n))$
- In practice, for almost all heuristics, the error is at least
  - proportional to the cost of the path $|\, h(n) - h^*(n) \,| \approx O(h^*(n))$ *and not to its logarithm*
- Exponential growth exceeds the capacity of any computer ➔ exponential in time and space
  - We need to keep all generated nodes in memory
  - Not suitable for big problems

Variants to solve exponential growth

- In practice, it is not convenient to insist on optimality
  - Variants of A * are used: they quickly find suboptimal solutions
  - A * is used with slightly inadmissible heuristics to obtain slightly suboptimal solutions
    - By limiting the excess of h over h we can limit the excess in cost of the solution reached with respect to the cost of the optimal solution
  - In any case, using good heuristics provides huge savings compared to using an uninformed search
- Some variants of A *:
  - RTA (Real Time A): limits the time
    - Real-time tasks: require a decision every so often
  - IDA * (Iterative Deepening A *): limits the cost
    - Limit with f : expands only states with cost less than that limit
  - SMA * (Simplified Memory-bounded A *): limits the space
    - If when generating a successor there is a lack of memory
      - less promising opened nodes are broken free

IDA* search

- Iterative-deepening A* search:

  Perform a depth-first search with a depth given by f, and increase this value.

  - n_0= initial node.
  - Having

$$C_0 = f(n_0);$$
$$C_k = \min_n \{ f(n) \mid f(n) > C_{k-1} \}; \quad k = 1,2,\ldots$$

  - Iteration k: expand all nodes that satisfy {n | f(n) ≤ C_k}

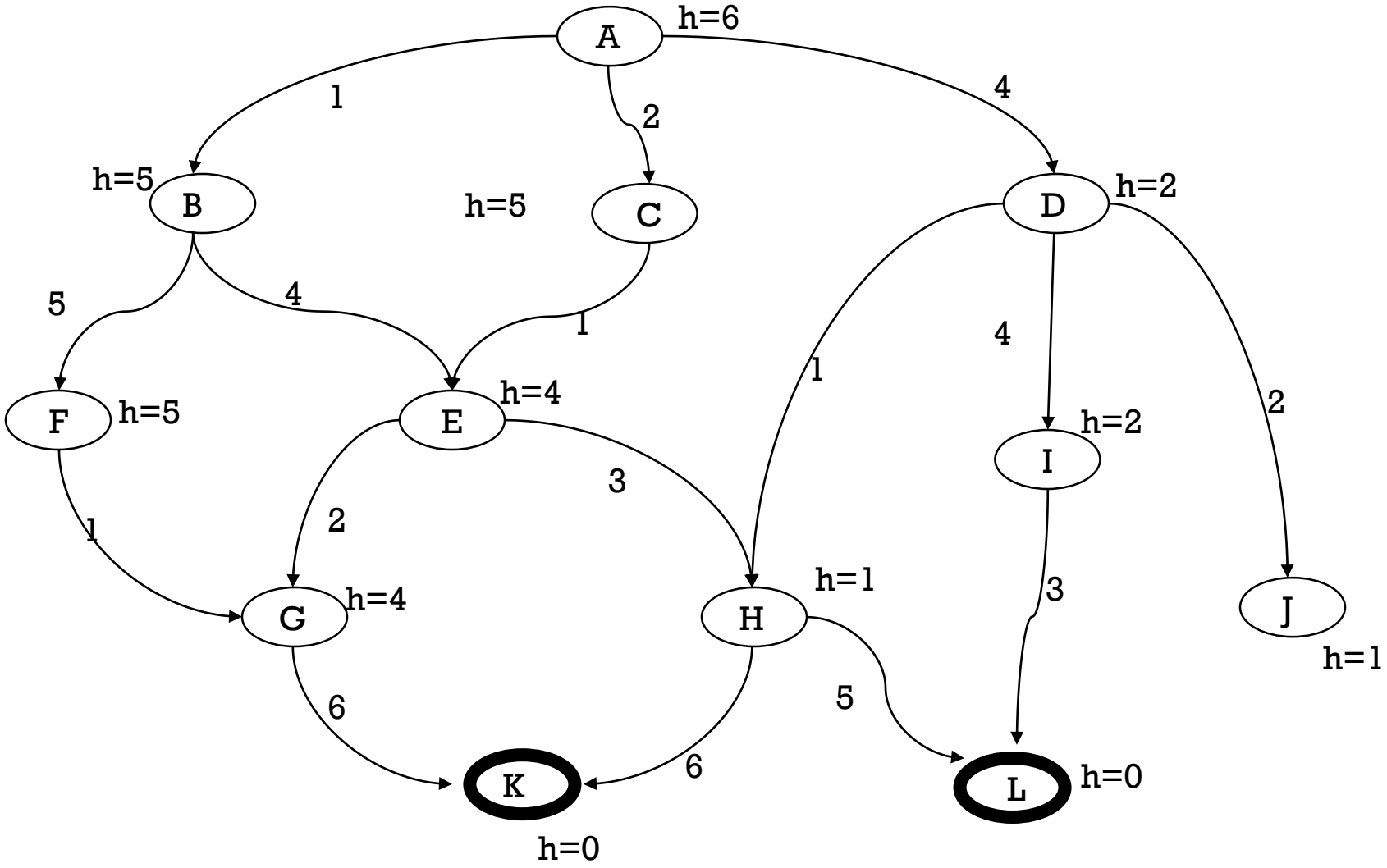
- Properties:
  - If h is monotonic ⇒ IDA* is complete and optimal.
  - Spatial complexity: O(b·d); d = C*/ε.

    C* = optimal cost; ε = minimum cost per action
  - Temporal complexity: In the worst case a single new node is expanded in each iteration. If the last expanded node satisfies the Target test, then the number of performed iterations is 1+2+…+N ~O(N^2)
    - Variation of IDA*:

$$C_k = f(n_0) + k\Delta C; \quad k \geq 0$$

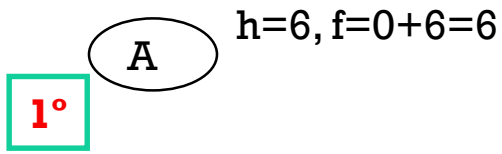
    - Number of iterations ~ O(C*/ΔC)

Example (A*, IDA*)

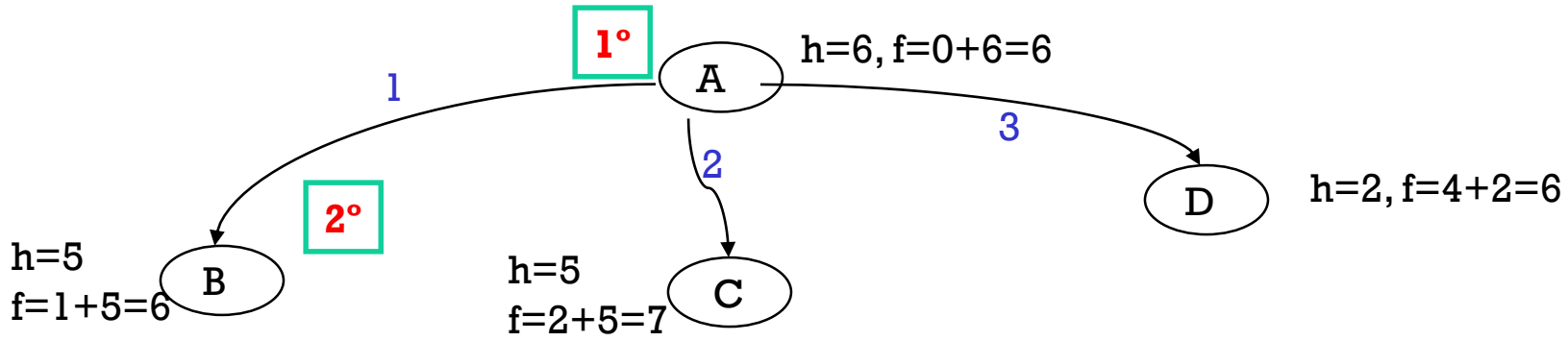


Final states: K, L

A* + tree search

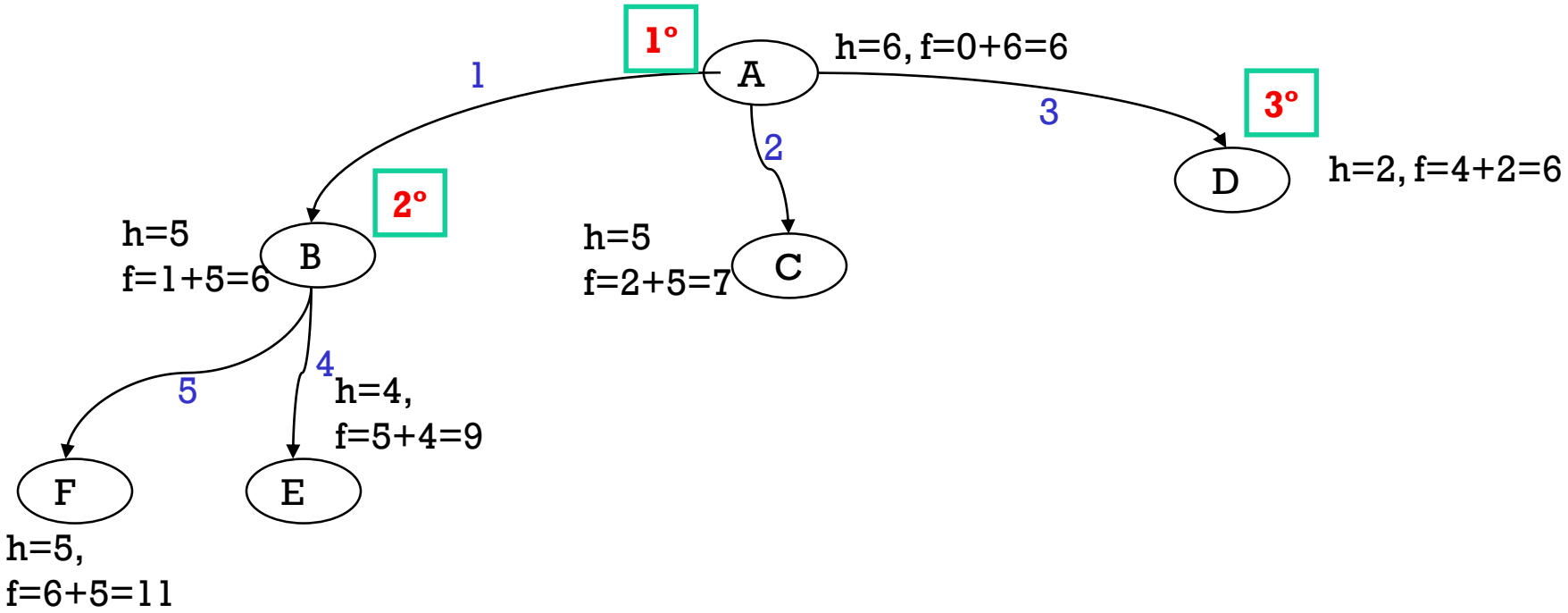


A* + tree search



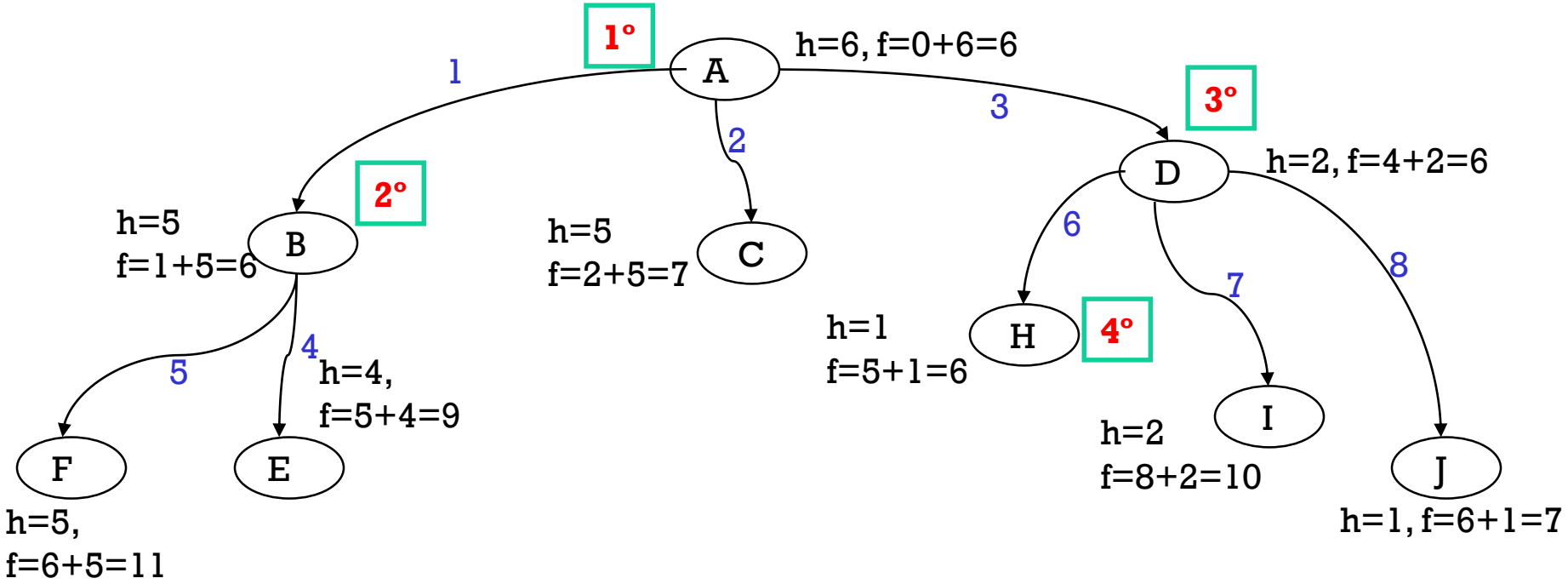
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



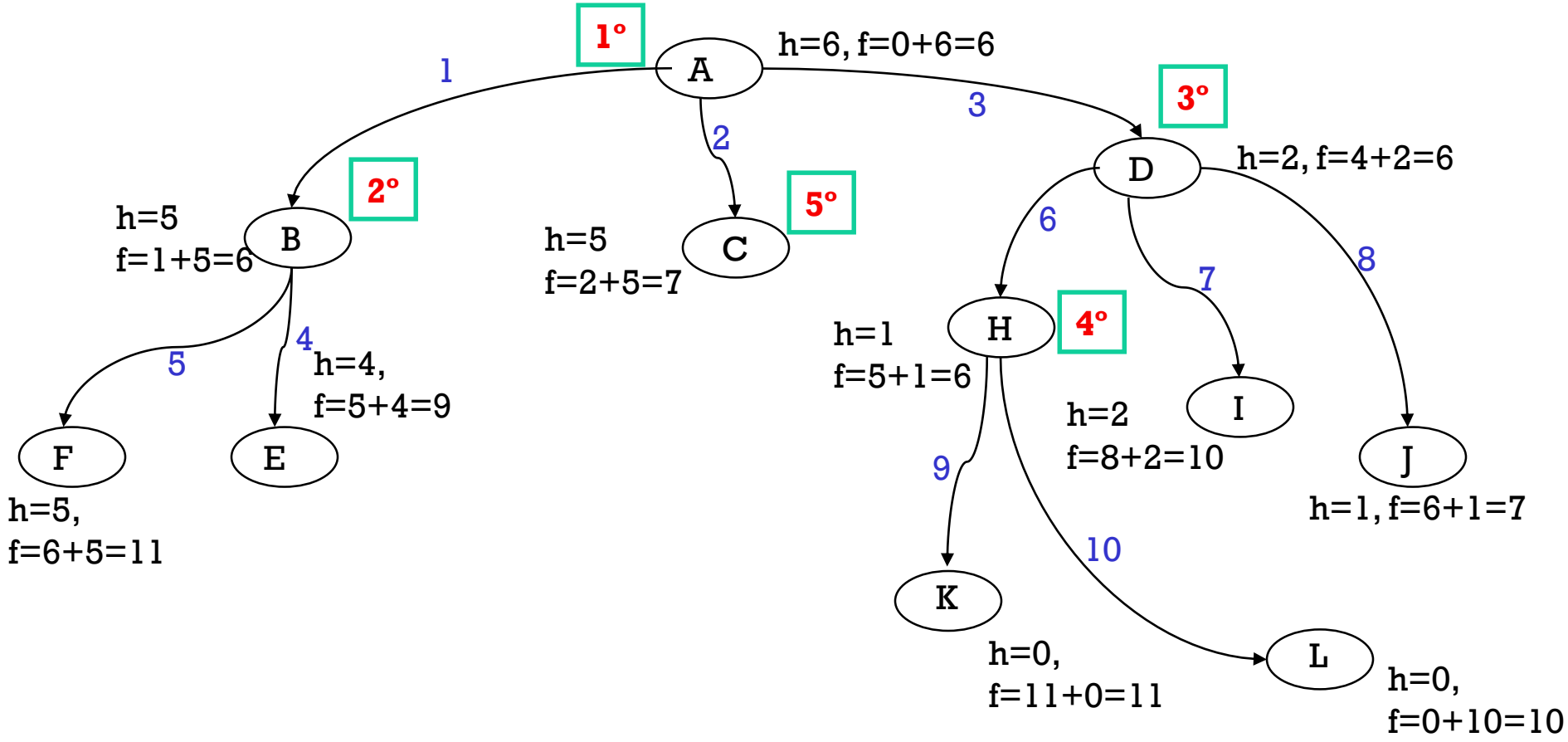
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



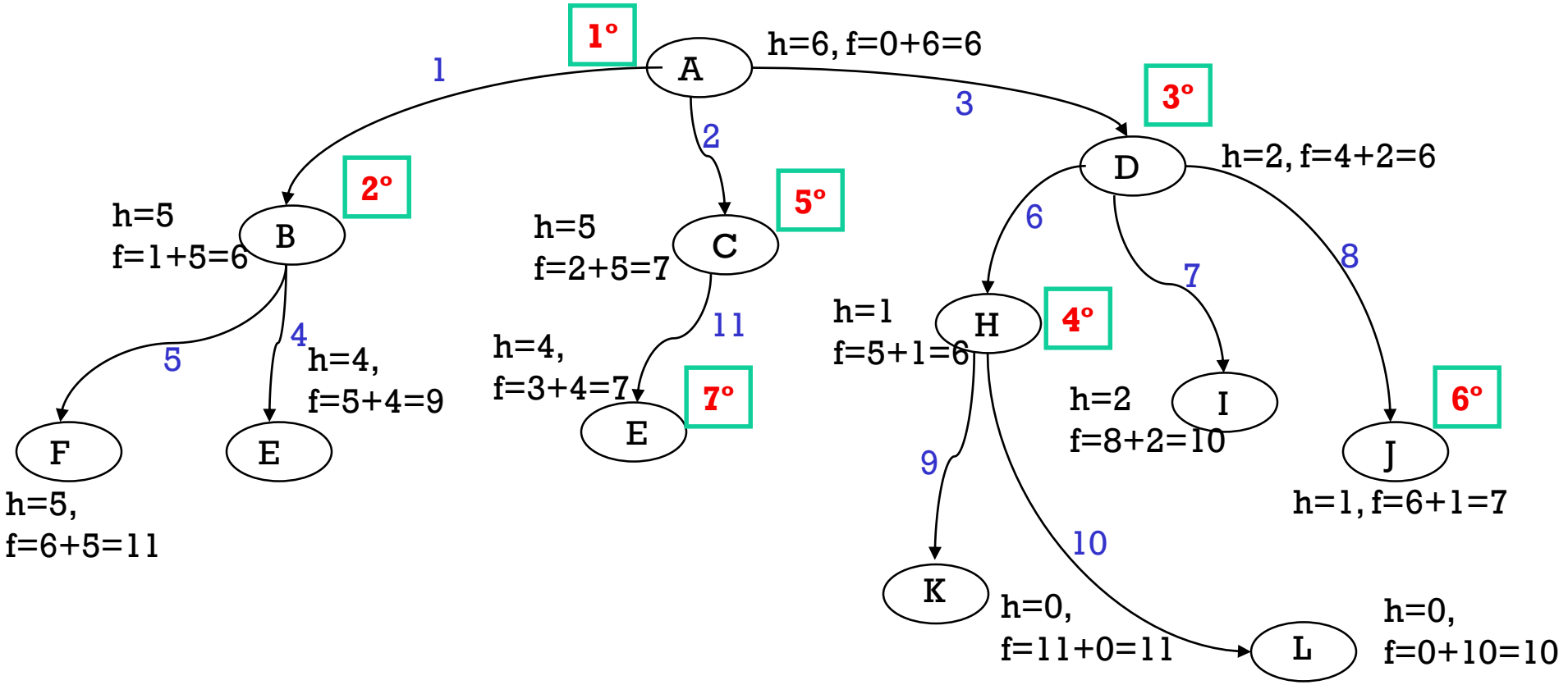
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



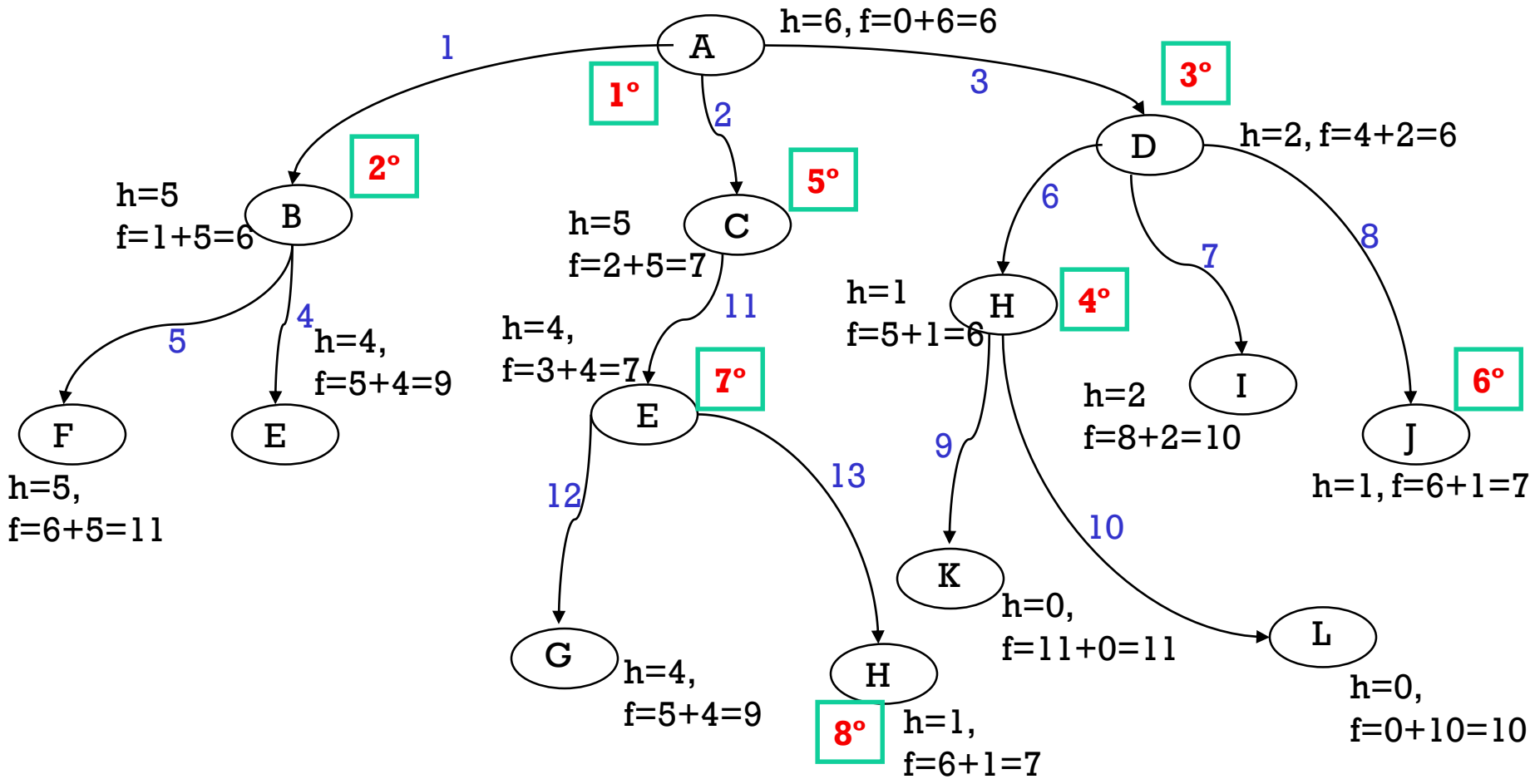
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



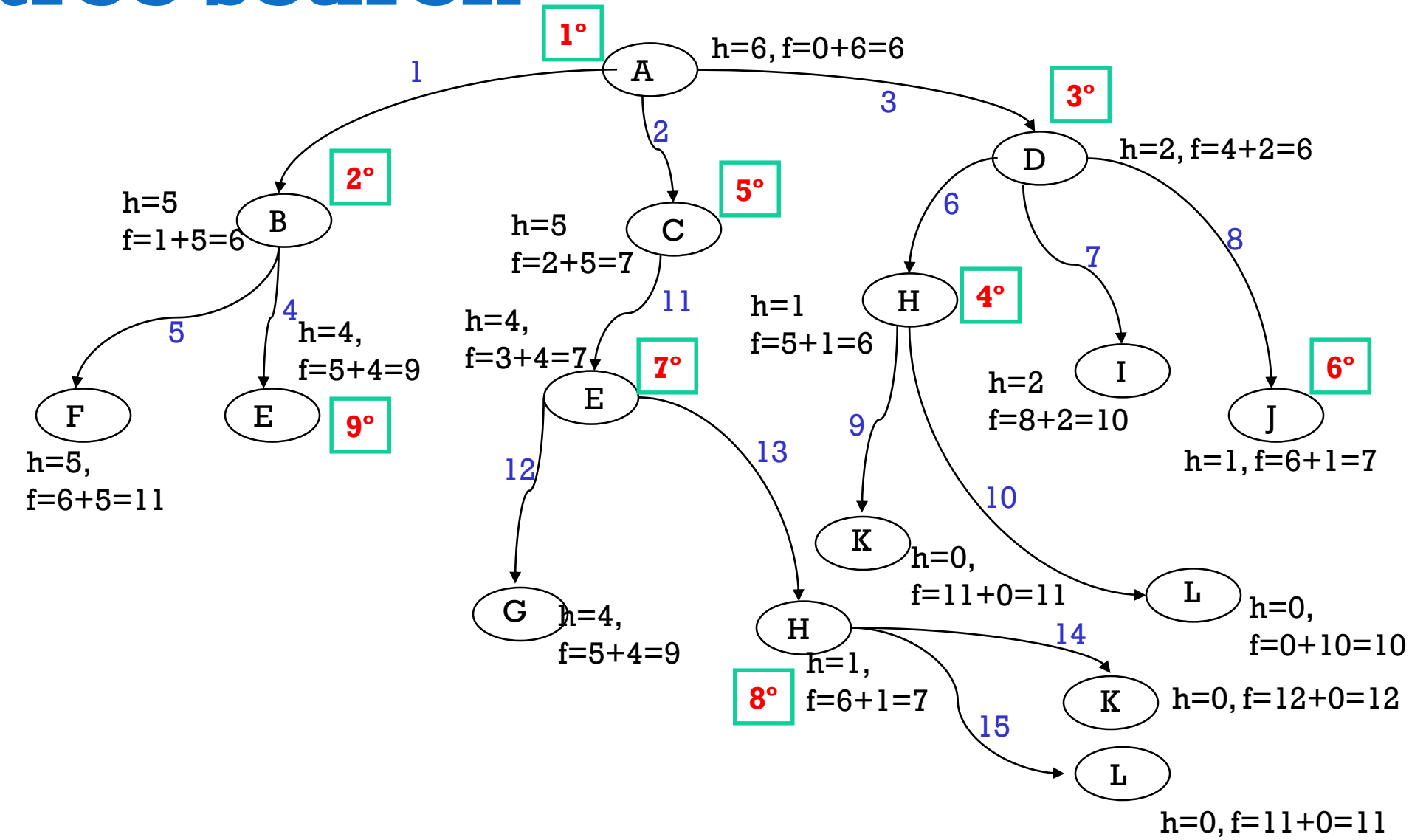
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



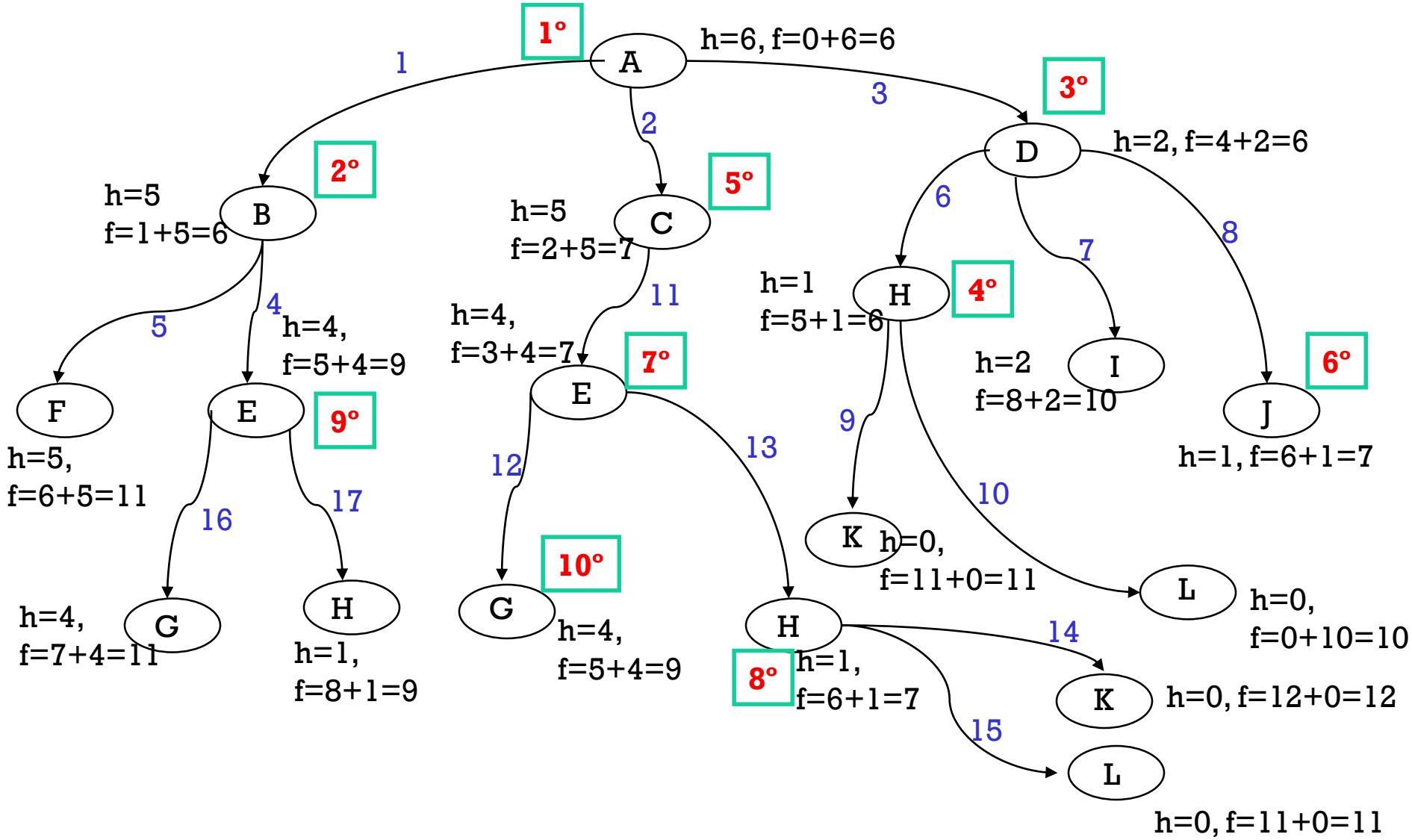
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



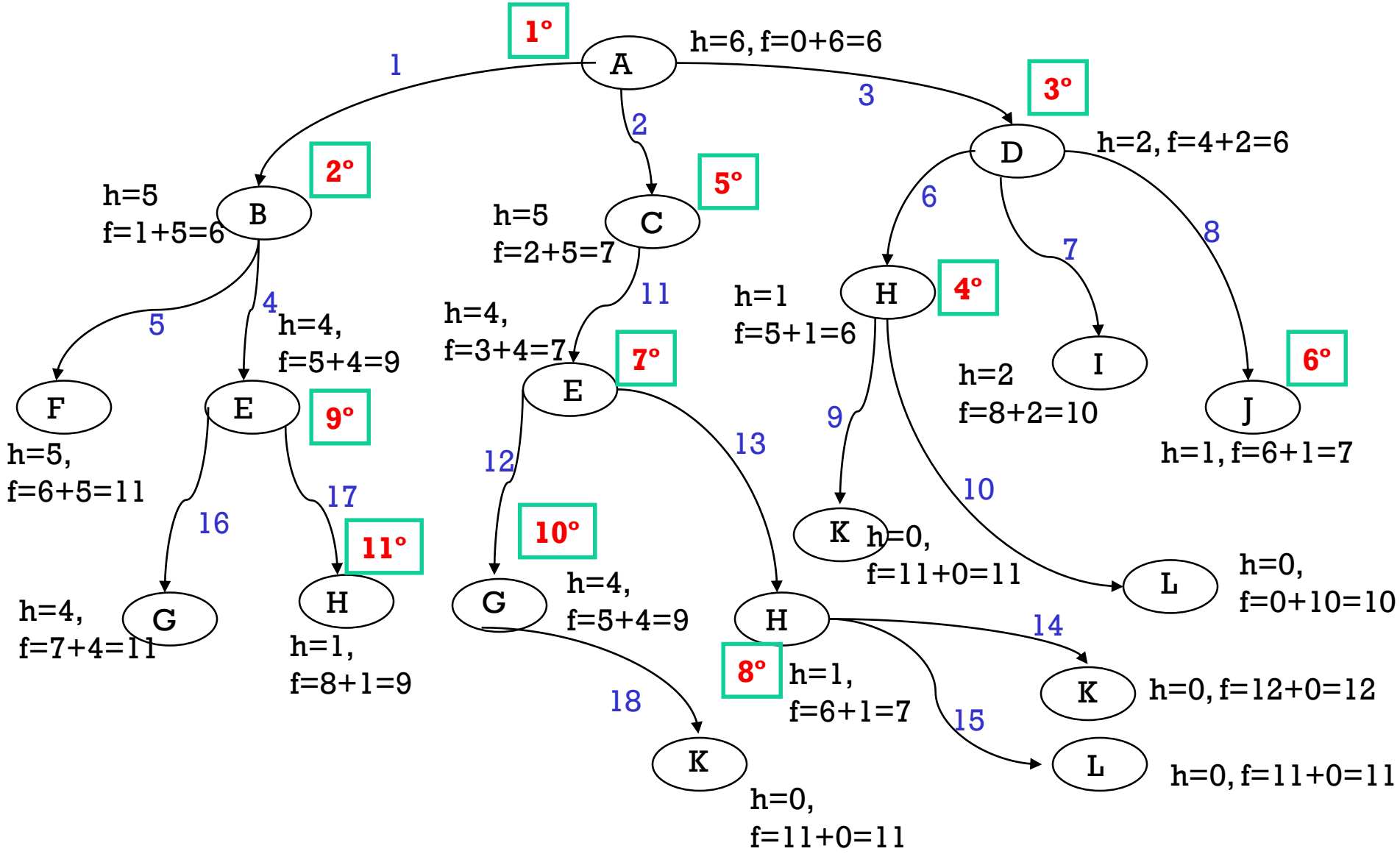
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



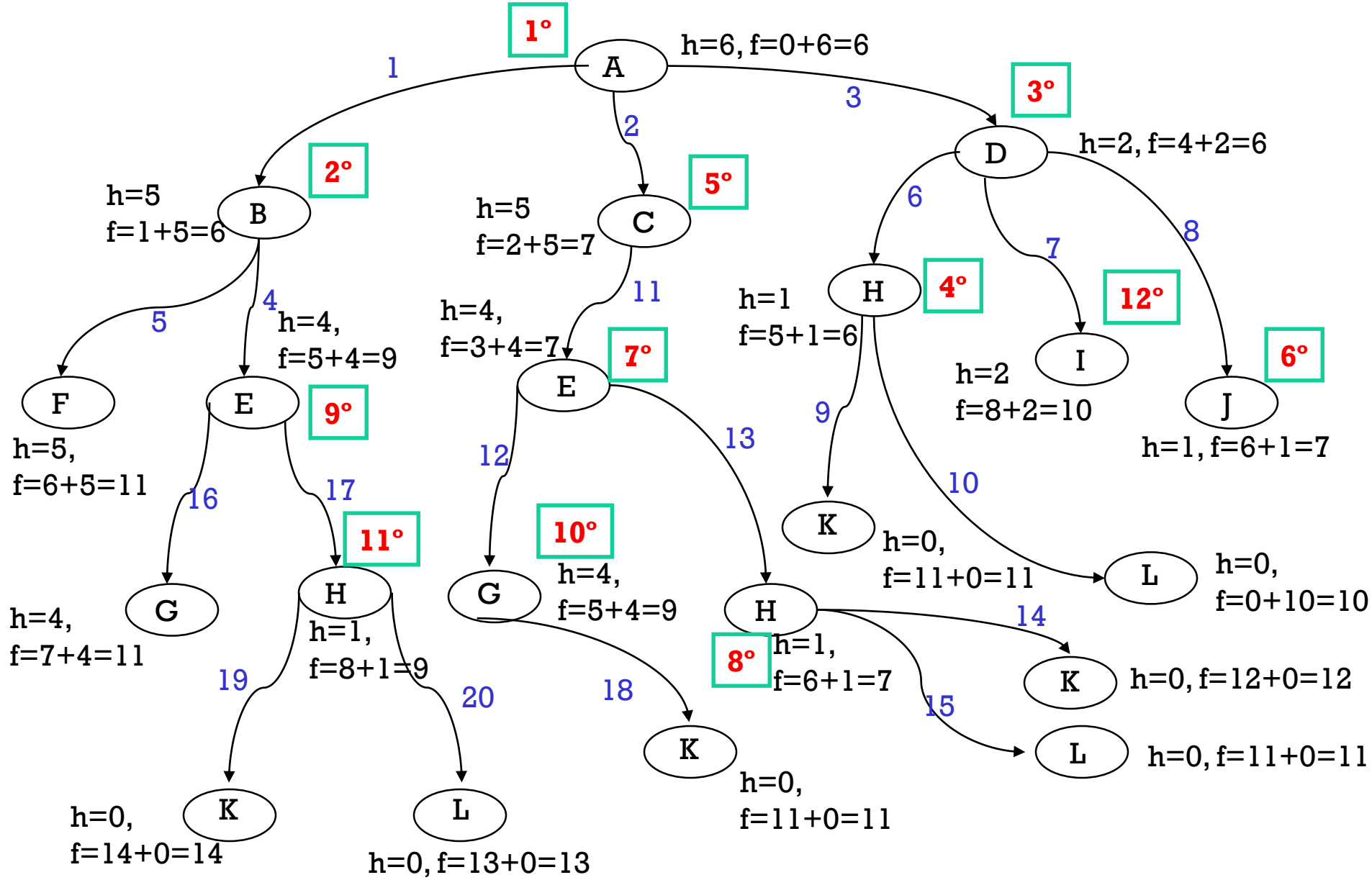
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



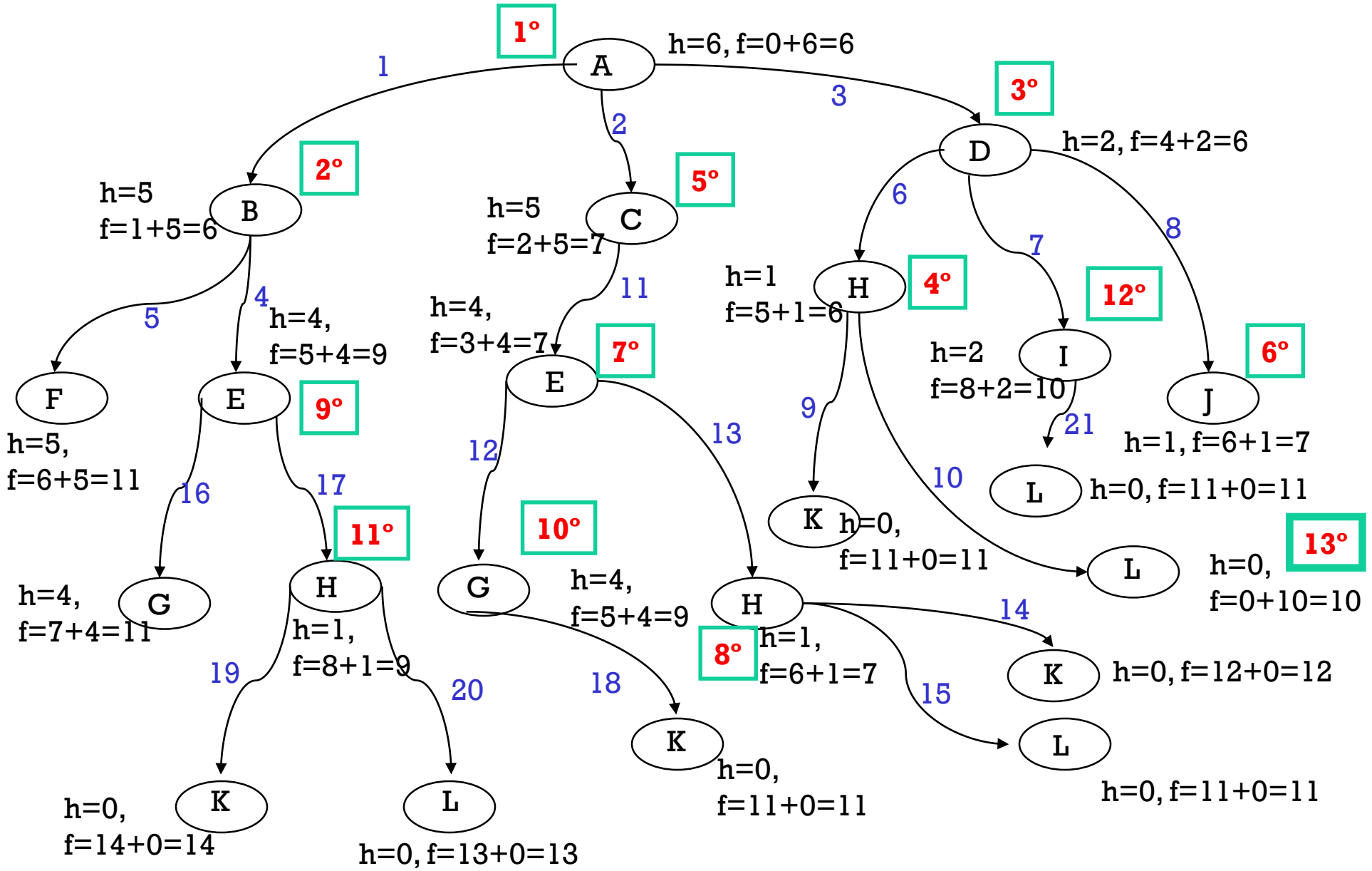
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



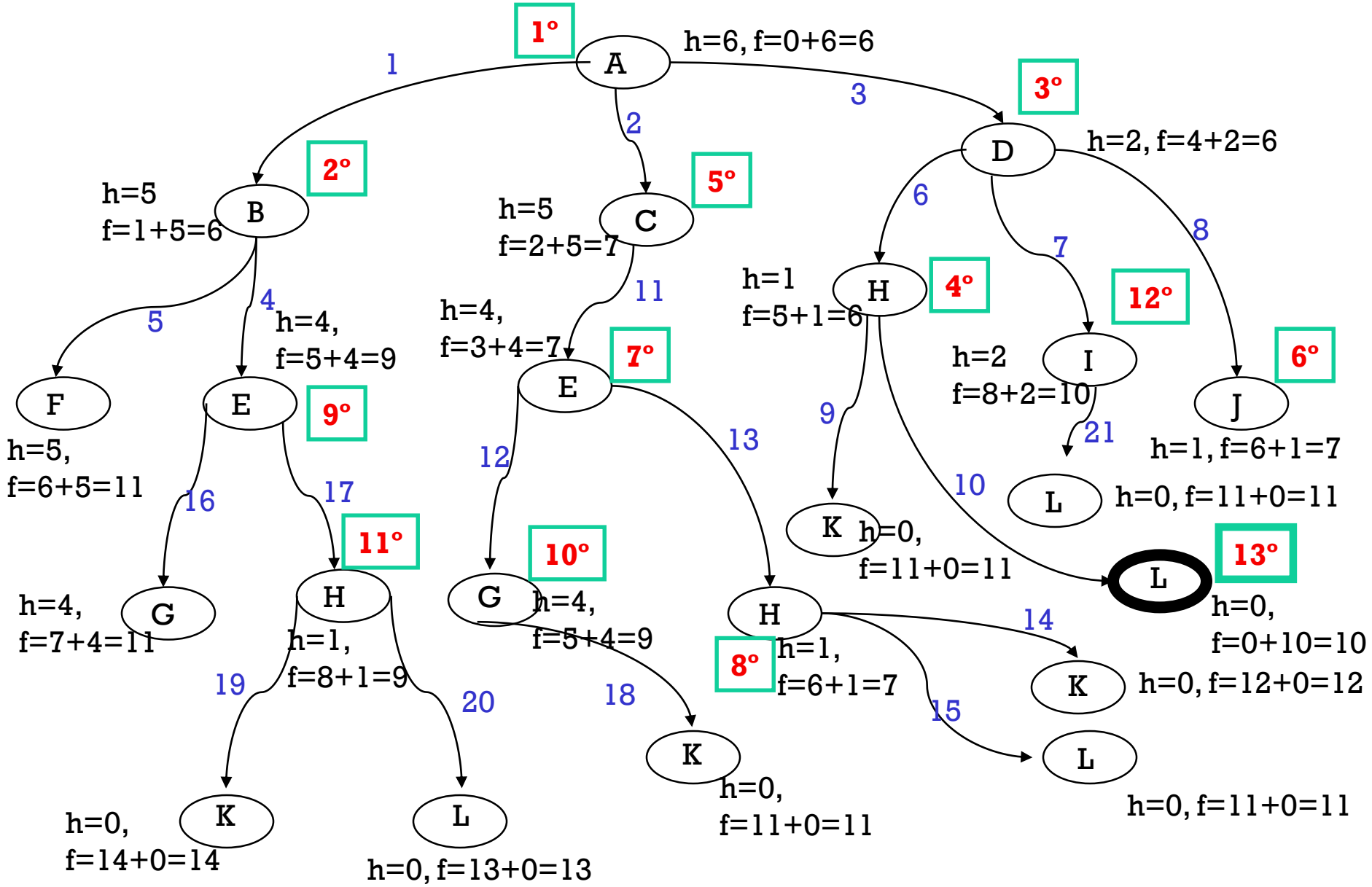
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



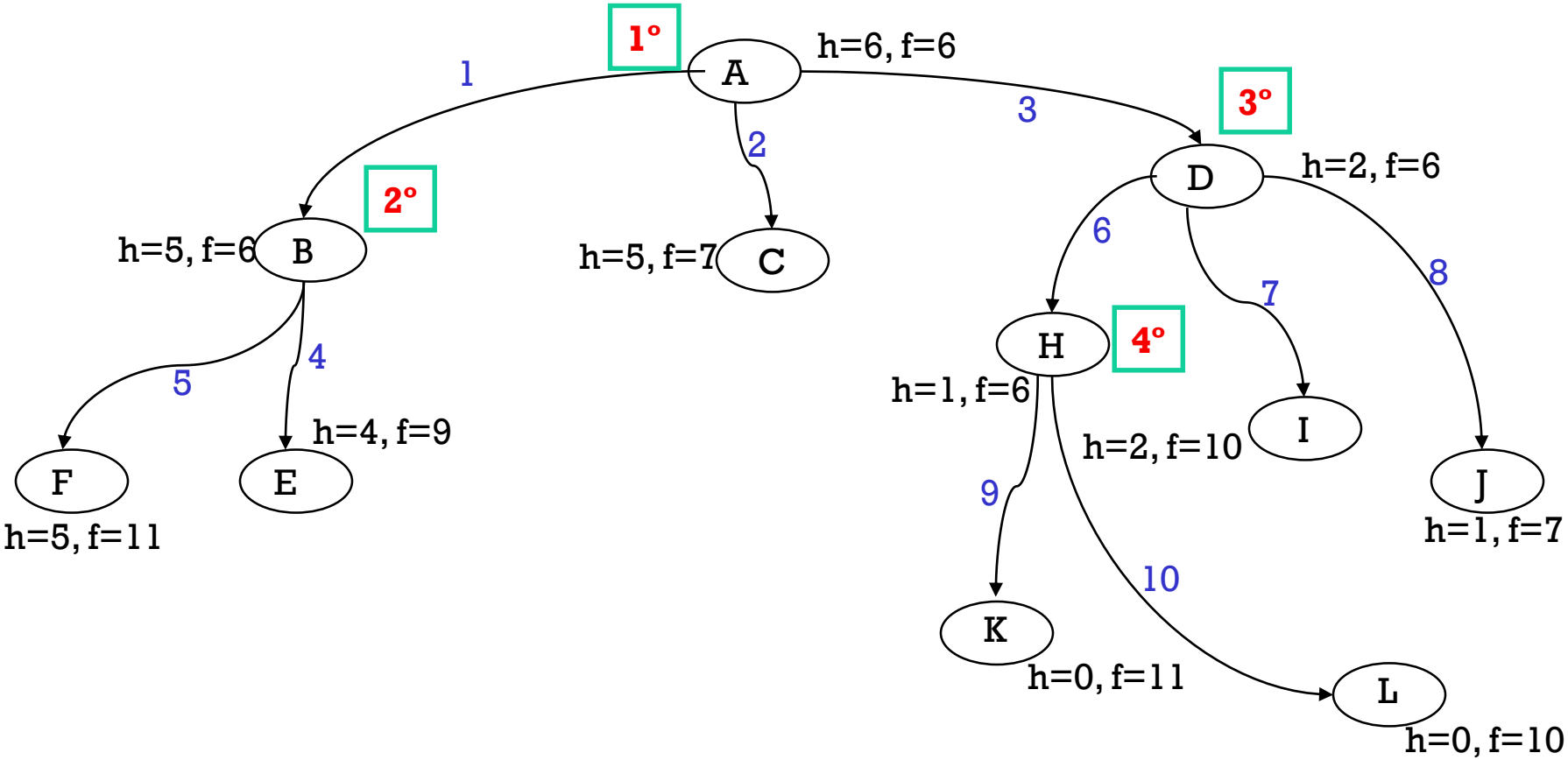
Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

A* + tree search



Remark: Ties are resolved by first expanding the oldest nodes (1) + alphabetical order (2)

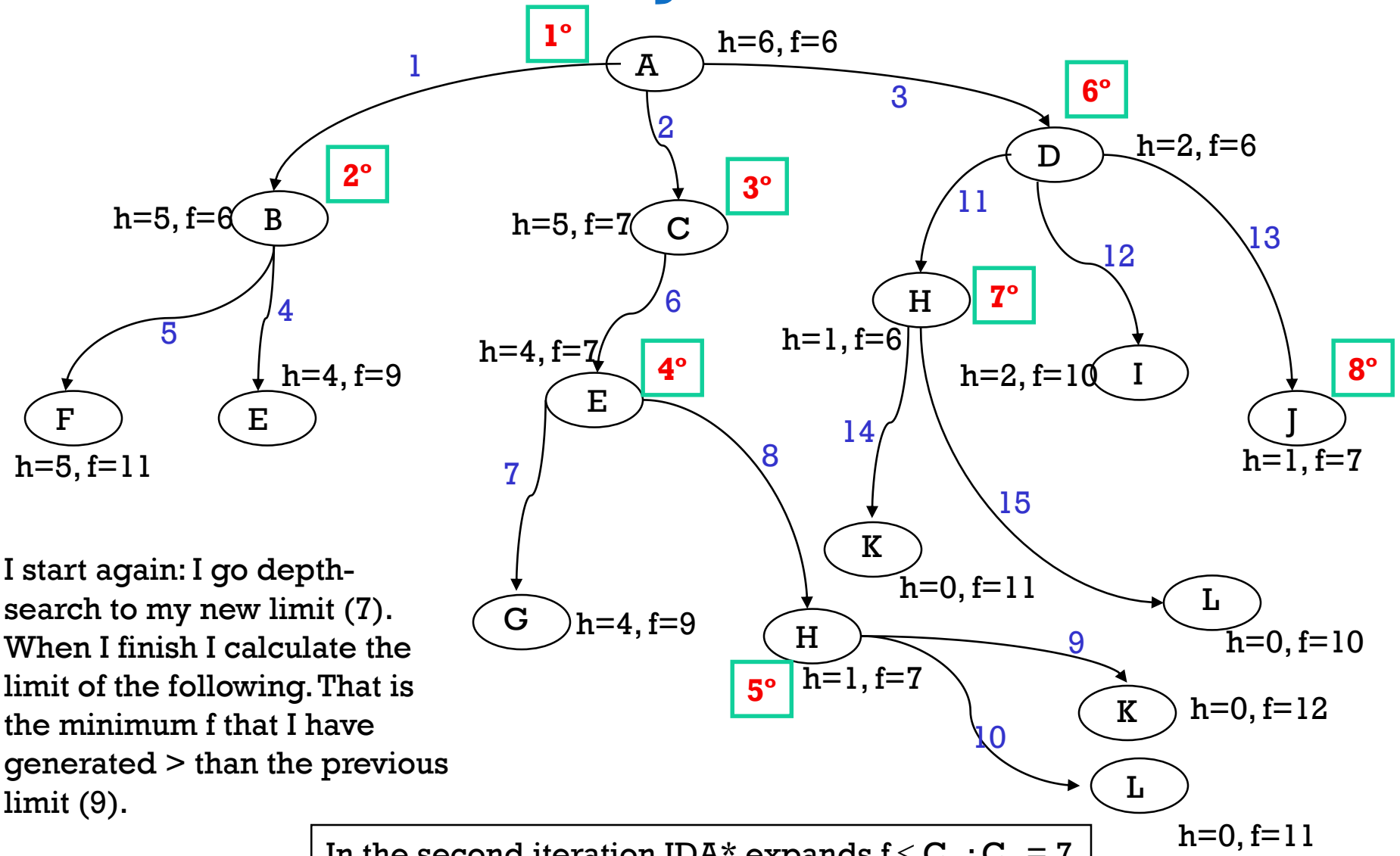
IDA* + tree search, I



I go depth-search to my limit, i.e. I don't expand anything > h = 6. When I finish I calculate the limit of the following. Which is the minimum f that I have generated > than the previous limit, i.e 7.

In the first iteration IDA* expands $f \leq C_0$; $C_0 = 6$
$C_1 = 7$

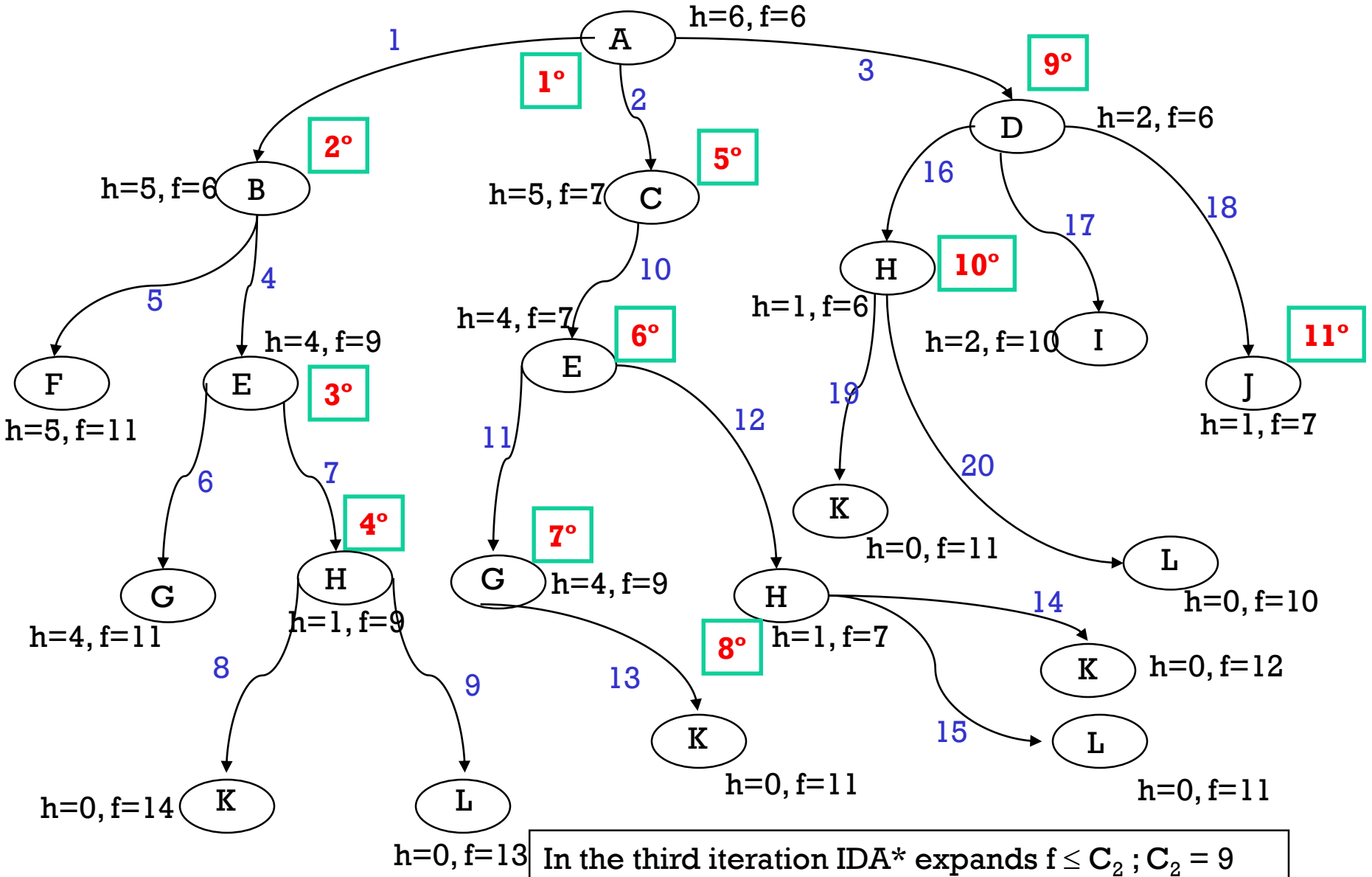
IDA* + tree search, II



I start again: I go depth-search to my new limit (7). When I finish I calculate the limit of the following. That is the minimum f that I have generated > than the previous limit (9).

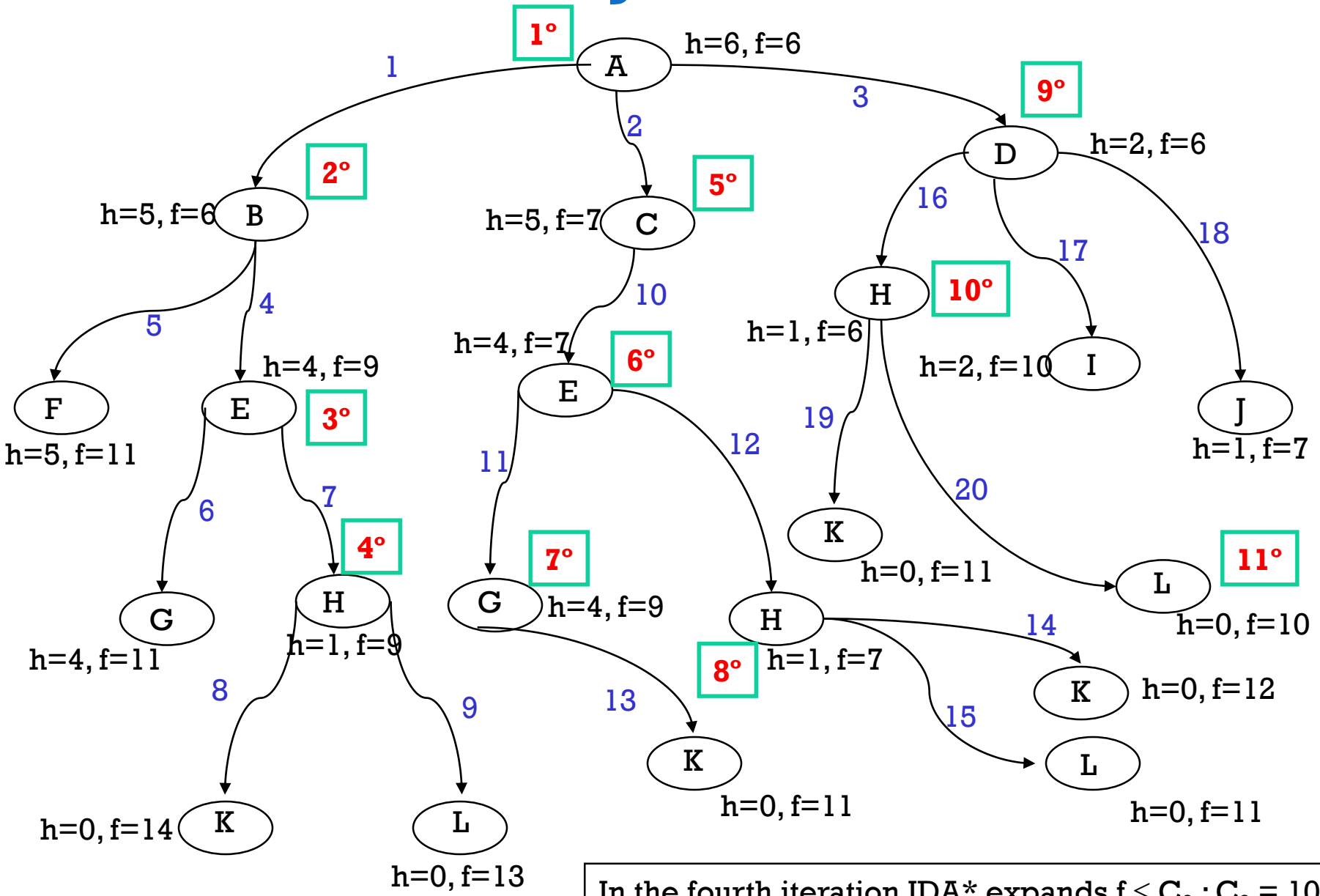
In the second iteration IDA* expands $f \leq C_1$; $C_1 = 7$
$C_2 = 9$

IDA* + tree search, III



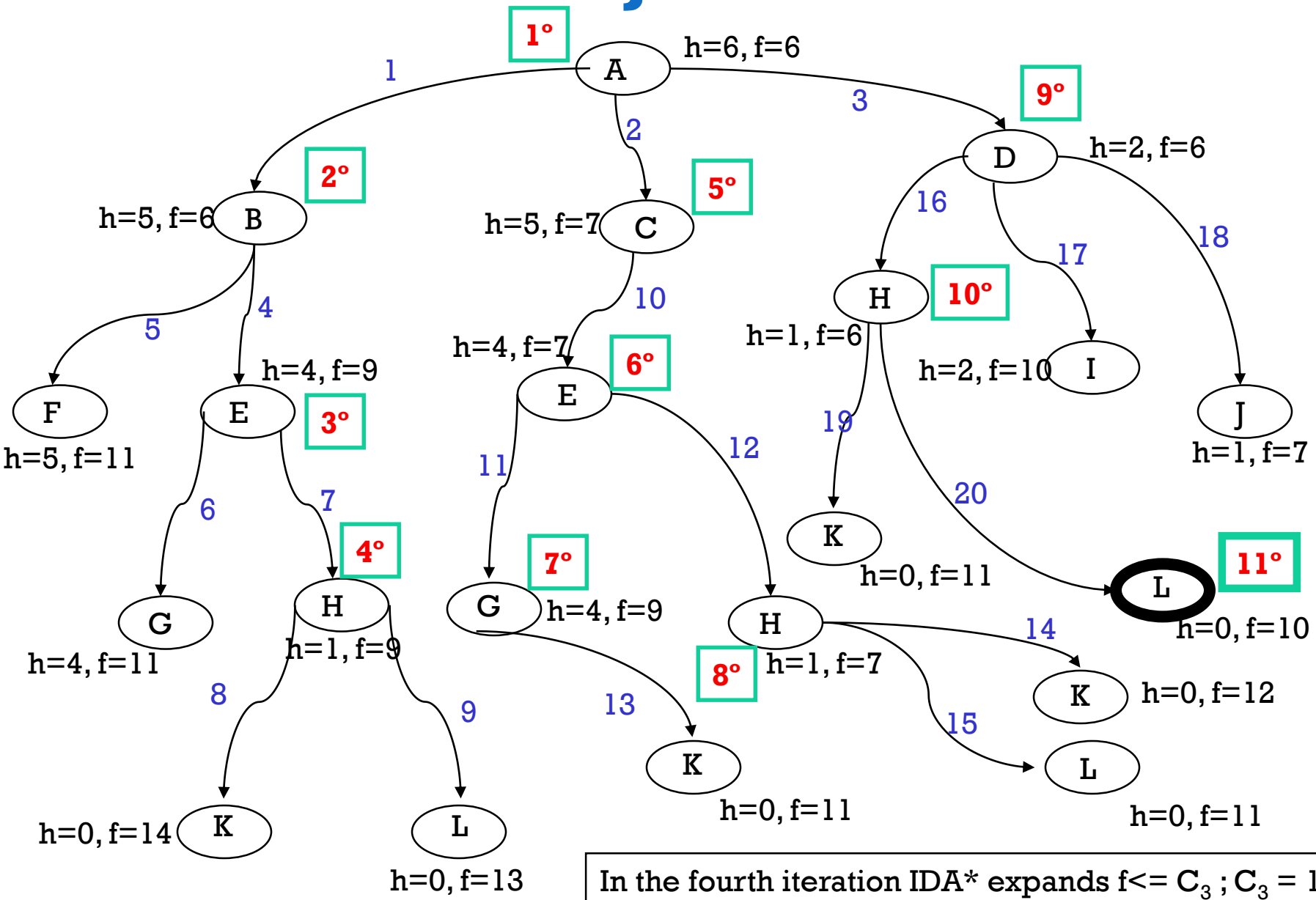
In the third iteration IDA* expands $f \leq C_2$; $C_2 = 9$
$C_3 = 10$

IDA* + tree search, IV



In the fourth iteration IDA* expands $f \leq C_3$; $C_3 = 10$
Target found

IDA* + tree search, IV



In the fourth iteration IDA* expands f<= C_3 ; C_3 = 10
Target found

3. Comparison of the heuristics' quality

A) - Ad-hoc evaluation: What to assess?

- ❑Wide range (number of possible values)
- ❑Different values for each immediate successor to a node
- ❑That it is possible to apply it to all allowed states
- ❑Let 0 for target states. Different from 0 for non-target states
- ❑Based on dynamic characteristics of the problem,
  - ❑Assign different weights according to relevance,
  - ❑Use separate terms for each important characteristic.
- ❑(… If it can be verified: admissible and consistent)

B) - Compare the quality of two heuristics

- ❑By demonstration of dominance (theoretical method)
  - ❑The more dominant is better (more precision)
- ❑By effective branching factor **b *** (experimental method)
  - ❑The one with the least b * is better

Effective branching factor b *

- If **N** is the number of nodes expanded by a search algorithm (e.g. A *) for a particular problem and the depth of the solution is d,
  - then **b *** is the branching factor that a **fictitious uniform** tree of depth **d** should have to contain **N** nodes
- Is fulfilled: $N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$
  - $52 = 1 + 1.91 + (1.91)^2 + (1.91)^3 + (1.91)^4 + (1.91)^5 \rightarrow b^* = 1.91$

$$b^* = \sqrt[d]{N} \rightarrow 2{,}2$$

- Knowing **d** and **N**, then **N = O(b*)**, approximately
- b * is practically constant from certain depths
  - The experimental measures of b * on a set of randomly generated cases provide a good guide to the full usefulness of a heuristic
- A good heuristic will have a value of b * close to 1
- Example: 8-puzzle with d = 12
  - Blind search (iterative depth): N = 3.644.035, b * = 2.78
  - *$A^*(h'_b)$: N = 227, b^* = 1.42* — *nº misplaced*
  - *$A^*(h'_a)$: N = 73, b^* = 1.24* — *Manhattan*

Evaluating heuristics

- Comparison of admissible heuristics h_1, h_2 :
  - h_2 dominates h_1 , if $\forall n: h_2(n) \geq h_1(n)$
  - If we use A* search and h_2 dominates h_1
    - h_2 never expands more nodes than h_1
    - Generally, $b_2^* \leq b_1^*$
  - Use h_2 if h_2 dominates h_1 and the costs of computing the heuristics are comparable

$$N = b^* + (b^*)^2 + \ldots + (b^*)^d = b^* \frac{(b^*)^d - 1}{b^* - 1}$$

Heuristic Generation: Three Methods

- Three methods to define or generate heuristics of a problem:
  1. **Relaxation**: Relax the restrictions or rules of the problem. The exact solutions of the relaxed problem are used as heuristics for the original.
  2. **Abstraction**: Define problems derived from the original. Its exact solutions are stored and used as heuristics of the original problem.
  3. **Learning**: Solve many problems of the given type and draw conclusions about their behaviour
     - Eg. AlphaStar learns to evaluate StarCraft 2 screens using a deep neural network.

       https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii

Finding heuristic functions

- If several admissible heuristics are available ($h_1, h_2, \ldots h_K$), the heuristic
  - $h_{max}(n) = \max\{h_1(n), h_2(n), \ldots h_K(n)\}$ for each n is admissible and dominates $h_1, h_2, \ldots h_K$.
- **Relaxation method**:
  - Define a **relaxed problem** by eliminating some restrictions of the original problem.
  - Use the **optimal cost function of the relaxed problem** as a heuristic for the original problem.
  - The obtained **heuristic** is **admissible and monotonic** (since it is an optimal cost function)
  - 8-puzzle problem:
    - Original problem: The tile in position A can move to B if A is adjacent to B and B is empty.
    - Relaxed problems:

      The tile in position A can move to B, even if B is occupied:

      1. No restrictions (h1: # tiles whose placement is incorrect)
      2. If A adjacent to B (h2: Manhattan distance or block distance)

    h2 dominates h1

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Heuristic functions: Steps to design h

1. Analyse the problem: choose which characteristics are relevant for *h*
   - ❑ Characteristics
     - ❑ Dynamic: what changes between states
     - ❑ Static: stay the same between states
     - ❑ If we relax them, what characteristics simplify the problem? → Relaxation
   - ❑ Study the restrictions of the problem (eg: the rules of the game) → Relaxation
     - ❑ If we relax them, which ones simplify the problem?
   - ❑ Generate partial state trees: random nodes + multiple levels
     - ❑ What difficulties appear to reach the solution?
       - ❑ We can remove details and make a simpler problem → Abstraction
       - ❑ what can we solve and calculate the cost to use as an heuristic?
     - ❑ Are there difficult states that need some special relaxation?
       - ❑ There are interactions between the elements that hinder the progress
2. Create the h with several terms: several *h* combined: $3 * h_a(n) + 5 * h_c(n)$
   - ❑ What features and restrictions should participate in h (are the terms)
     - ❑ Choose those characteristics, restrictions and interactions that affect to a greater degree the resolution of the problem
     - ❑ The greater or lesser degree decides the weight of each term: the value that multiplies

Summary Quality criteria: Does h meet most?

1. h is applicable on all valid states?
2. h guide to the target?
    1. Is the range of values generated by h wide?
    2. The neighbouring / child states have different values of h?
3. h (target states) = 0?
4. h is consistent? (or admissible at least)
5. If you can find another function that "dominates" h: use it
6. If you can do experiments, study that b * is close to 1

Problem solving using informed search

- Introduction
- Best-first search
- Iterative improvement algorithms
  - Introduction
  - Hill Climbing
  - Maximum slope: Gradient Ascent
  - Simulated annealing

Iterative improvement algorithms: Local Search

- In many problems the path to the target is irrelevant
  - 8-queens what matters is the final configuration
  - Domains:
    - IC design, floor layout, work planning, automatic scheduling, network optimization, portfolio management ...
- A different class of algorithms that don't care about the paths
  - ignore the cost of the path, in particular
- Local search algorithms work with a single current state and generally
  - they move only to the neighbours of the state
    - Not like A * or greedy who jump in search space, guided by f
- Although they are not systematic, they have two key advantages:
  - They use very little memory
    - The paths followed by the search are rarely held back
  - They find acceptable solutions in large state spaces

Informed local optimization methods

- Local search algorithms solve pure optimization problems
  - The goal is to find the best state according to a certain goal function
- Informed Local Optimization Methods: Some Optimization Issues
  - The solution itself has an associated cost that you want to optimize
    - Eg: in the backpack problem we optimize the solution
    - But the cost of the path is indifferent
    - Eg: Machine Learning methods, we need to optimize the result, minimize the error (more on lesson 5)
  - Usual approach as a search in the **solution space**
  - Can be extrapolated to state space search (using heuristics)
- One type is Climbing Algorithms
  - They consume few resources
  - But they can get stuck in a local optimum
  - Constant complexity in space: open-list never has more than one node
    - Irrevocable: a single path remains in expectation (without going back)
  - Neither optimal nor complete
  - They significantly prune the search space but without offering guarantees

Hill climbing

- The name comes from using higher values for better nodes
  - It's like climbing a mountain
  - We use the other version: the lower the value the better
- At each step, the current node is attempted to be replaced by the first best neighbour
  - First successor with a heuristic measure lower than the current node
- Try to move in the direction of a better value
  - downhill, look for a decreasing value
- Ends when
  - It finds a solution or
  - reaches a node where no neighbour has a lower value
- It does not keep a tree, but only the current node
  - the state and its value according to the goal function to optimize (just use h)
- Does not look forward beyond the immediate neighbour of the current state
- It's like "generate and test“
  - Nuanced by a heuristic function (gives you domain knowledge)
  - Use it only if you have a good h and no other useful knowledge

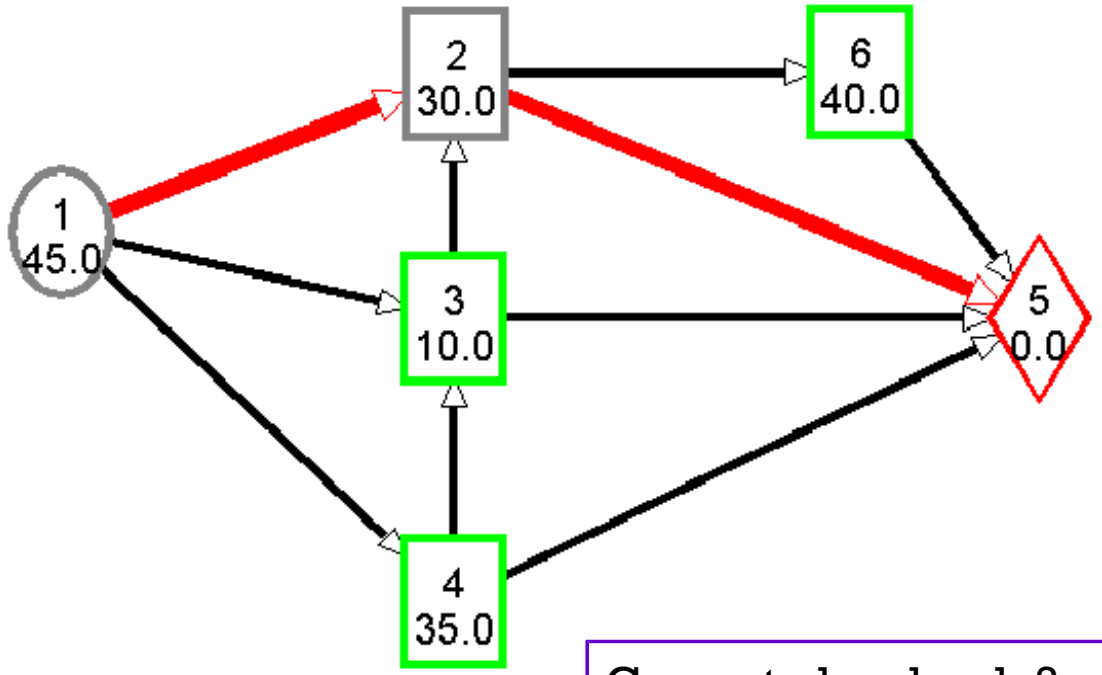
Hill climbing

- As in depth-search but guided by h, it only descends if it improves
- Very dependent on the children generation order

```
evaluate initial state
if is a target state then return and stop
  if no ACTUAL := initial
while aplicable operators to ACTUAL and solution not found do
    Select not applied operator to ACTUAL  apply operator and
    generate NEW_STATE
    evaluate NEW_STATE
    if is a target state then return and stop
      if no
          if NEW_STATE has better h' tan ACTUAL then  ACTUAL := NEW_STATE
```

Hill Climbing solution

Children generation order: increasing order



Generated nodes: 1, 2 and 5

solution: 1-2-5 Cost (not taken into account: 200+30 = 230

Hill Climbing by maximum slope: Gradient Ascent

- Variant: study all neighbours of the current node
- At each step, the current node is replaced by the best neighbour
  - The Neighbour with the best value of h, is the Successor with lower heuristic measure
    - The one that supposes the steepest descent of h,
    - with what goes down the path of maximum slope
- It moves in the direction of decreasing value, that is, downhill
- It ends when it finds
  - a solution or
  - reaches node where no neighbour has lower value
- It does not look forward beyond the immediate neighbours of the current state
- Sometimes called greedy local search
  - because it takes the best neighbour state without thinking where it will go next
- It progresses very fast towards a solution,
  - but it tends to get stuck for various reasons in local minimums

Continuous optimization

- **Gradient ascent** (greedy local search in a continuous space): Move along the direction of the gradient of the objective function

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$$

  - The direction of the gradient of a function is the direction of greatest local variation.
  - α is the climbing rate (a small value)
  - If the analytical form of the gradient is not available, numerical estimates of the gradient can be used:

$$\frac{\partial}{\partial x_i} f(\mathbf{x}) \approx \frac{f(\mathbf{x}+\mathbf{h}_i) - f(\mathbf{x}-\mathbf{h}_i)}{2h_i} \quad i = 1,2,\ldots,D$$

$$\mathbf{h}_i = \begin{pmatrix} 0 \\ \vdots \\ h_i \\ \vdots \\ 0 \end{pmatrix} \leftarrow \text{component number i} \quad h_i \to 0$$

- **Newton-Raphson**

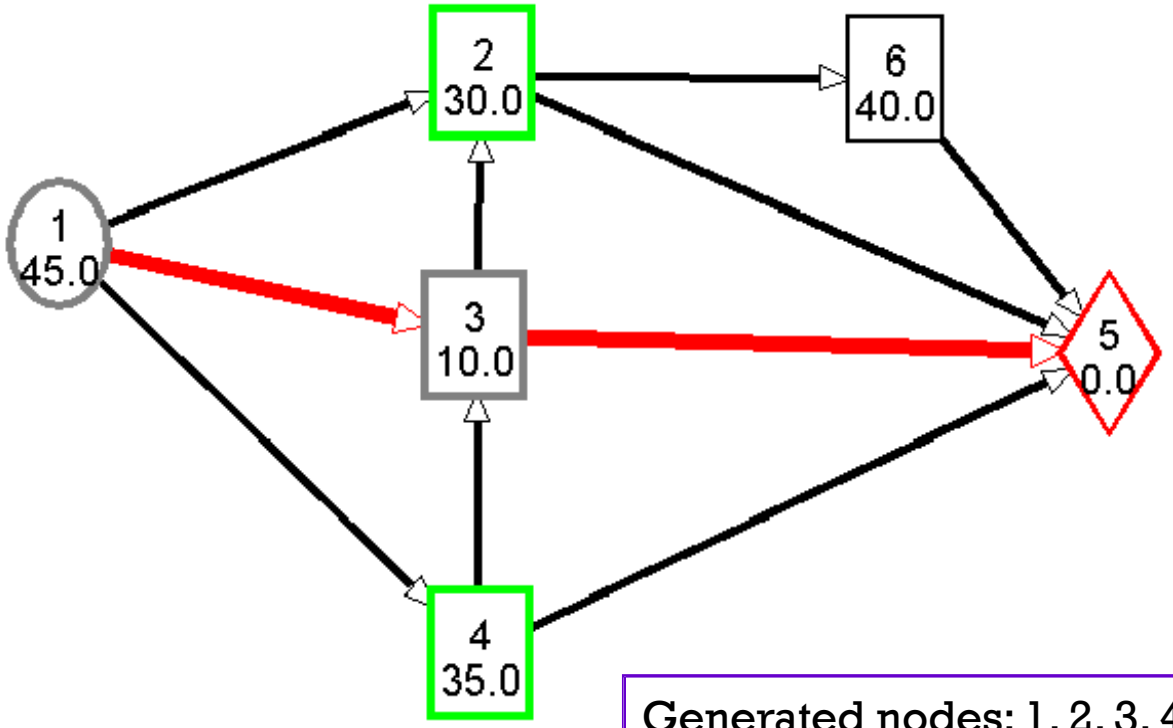
$$\mathbf{x} \leftarrow \mathbf{x} + \mathbf{H}^{-1}(\mathbf{x}) \cdot \nabla f(\mathbf{x}); H_{ij} = \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} \text{ (Hessian matrix)}$$

- **Cuasi-Newton**
- **Conjugated gradients**
- **<u>Constrained optimization</u>**: linear, quadratic, non-linear programming.

Hill Climbing by maximum slope

```
evaluate initial state
if is a target  state then return and stop
 if no ACTUAL := initial
while not stop and not solution found do
    SIG := worst node than any successor to ACTUAL
     for each operator applicable to ACTUAL
          apply operator and generate NEW_STATE
            evaluate NEW_STATE
          if is a target then return and stop
            if no
                if NEW_STATE is better than SIG then  SIG := NEW_STATE
     if SIG is better than ACTUAL then ACTUAL := SIG  if no stop
```

Hill Climbing by maximum slope solution



Generated nodes: 1, 2, 3, 4 y 5

Solution: 1-3-5 Cost(not taken into account): 60+200 = 260

Climbing algorithm problems

- They may not find a solution:
  - state that is not target and has no better neighbours.
- This will happen if the algorithm has reached:
  - A local minimum (or local optimum)
    - A state better than its neighbours but worse than other states further afield
  - A plateau
    - All neighbouring states have the same heuristic value
    - It is impossible to determine the best move: it would be blind search
  - A ridge: Mix of the above
    - region in which h does not lead to any target state.
    - May end at a local minimum or have a plateau-like effect

Variants of climbing algorithms: improvements

- Variations improve by trying to resolve blockages
- Depth + scaling: nodes of equal depth are ordered by putting the most promising ones at the beginning and backtracking is allowed
  - Recovers completeness and exhaustiveness
- Restart all or part of the search
- Go one step further: generate successors of successors and see what happens
  - If a local optimum appears, go back to a previous node and try a different direction
  - If a plateau appears, make a big jump to get off the plateau
    - How to get away?
      - Random restart: start search from different randomly chosen points,
        - saving the best solution found so far
      - Not applicable to default initial state issues

Simulated annealing

- Success is highly dependent on the shape of the state space
  - NP-hard problems: they usually have an exponential number of local optimum
    - AI: to solve them in an acceptable time and in an approximate way
- A- A hill climbing algorithm that never makes reverse movements towards "worse" states is necessarily incomplete
  - Often it is a good idea to get a little worse to get better later
- B- A purely random algorithm is complete but very inefficient
- C- Simulated annealing
  - Combine climbing with random choice of paths
  - Produces both efficiency and completeness
    - In metallurgy, this process is followed to temper metals and crystals
      - heating them to a high temperature and then gradually cooling them, for the material to solidify into a low energy crystalline state

Simulated annealing

- It is like a minimization problem: the function to optimize is the energy E
  - In the algorithm, the E is a function to define: evaluate (state)
  - When starting, the temperature T is high
    - movements contrary to the optimization criterion are allowed: worsen
  - At the end of the process, when T is low,
    - behaves like a simple climbing algorithm
  - The temperature T is a function of the number of cycles already executed
  - The planning of the cooling (variation of T) is determined empirically and is previously fixed (another function to be defined)
- If the cooling (decrease T) is slow enough
  - a global optimum is reached with probability close to 1
- Used a lot in design
  - VLSI, in factory planning, in large-scale optimization tasks
  - It seems to be the most widely used informed search strategy

Simulated annealing

- Maximizes an "energy function" according to the following scheme

```
function SIMULATED-ANNEALING (problem, program) returns a solution state
  inputs:       problem, a problem
                program: a correspondence between iteration and "temperature"
  local variables:
                current, next: nodes
                T, "temperature" that controls the probability of going downhill

current ← GENERATE-NODE ( INITIAL-STATE[problem] )
for t ← 1 to ∞ do
                T ← program[T]
                if T=0 then return current
                next ← a successor of current randomly chosen
                ΔE ← VALUE[next] – VALUE [current]
                if ΔE > 0 then current ← next
                else current ← next only with probability exp(ΔE/T)
```

- Randomly select a neighbor of the current state
  - If $\Delta E > 0$ accept.
  - if $\Delta E \leq 0$ acceptar only with probability $p = \exp(\Delta E/T)$
- $T = 0 \Rightarrow$ Climbing.
- $T = \infty \Rightarrow$ Stochastic search.

- Geometric annealing program
  - Keep T constant for a number of iterations (an "epoch")
  - After each epoch, reduce T by multiplying it by $\beta<1$

Simulated annealing

```
evaluate(INITAL)
if initial is solution then return and stop
  if no
          ACTUAL := initial  BEST_FOR_NOW := ACTUAL

          T := INITIAL_TEMPERATURE        starts high: allows "bad" moves
while there are applicable operators to ACTUAL and solution not found
    Select randomly operator not applied to ACTUAL
                                  {choose move randomly (not the best)}
    apply operator and obtain NEWSTATE
    calculate ΔE := evaluate(NEWSTATE) - evaluate(ACTUAL)
    if NEWSTATE is solution then return and stop
          if no . . .
```

Simulated annealing (continues)

[. . .if not] if NEWSTATE is better than ACTUAL {if situation improves}

ACTUAL := NEWSTATE{movement is accepted}

if NEWSTATE is better than BEST_FOR_NOW

then BEST_FOR_NOW := NEWSTATE

if not {if it does not improve the situation, it is accepted with prob. < 1}

Compute P':= $e^{-\Delta E/T}$

{probability of going to a worse state: decreases exponentially with the "badness" of the movement, and when the temperature T drops}

Obtain N {nº Random in intervalo [0,1]}

decides randomly if to keep the movement

if N < P' {movement is accepted}

then ACTUAL := NEWSTATE

actualizeT according to cooling planned

planning is decided a priori: how quickly do we want to decrease T

return BEST_FOR_NOW as solution

Parallel local search

- Local beam search: Parallel stochastic searches in k states
  1. Randomly select k initial states.
  2. Generate all the successors of the k states.
  3. Select from these successors the best k states.
  4. Repeat step 2 with the chosen set, until the convergence criterion is satisfied.

- Genetic Algorithms: Use artificial evolution to maximize fitness function
  - Problem coding:
    - An individual corresponds to a possible solution to the problem.
    - Each individual is represented by a **chromosome**: Fixed-length string that completely codes the individual (for example, using a string of {0,1}).
  - Example of genetic algorithm:
    1. Randomly initialize a population of M individuals
    2. **Select** parents **according to fitness**.
    3. Generate a new population of M children through **crosses** between the elected parents.
    4. Introduce variations in individuals using **mutation**.
    5. Repeat from step 2 until the convergence criterion is satisfied.