

The background of the slide is a dark blue gradient with a complex pattern of glowing blue lines and dots, resembling a circuit board or a neural network. On the left side, there is a vertical rectangular area with a purple-to-blue gradient. Inside this area, the letters 'AI' are displayed in a large, white, sans-serif font. To the right of 'AI', there is a vertical line of small, light blue dots. Below the 'AI' text, there are two horizontal rows of small, light blue vertical bars, resembling a barcode or a data visualization.

# AI

# SEARCH STRATEGIES

Nguyễn Ngọc Thảo – Nguyễn Hải Minh  
{nnthao, nhminh}@fit.hcmus.edu.vn

# Outline

- Uninformed search
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
- Informed search
  - Greedy best-first search
  - A\* search



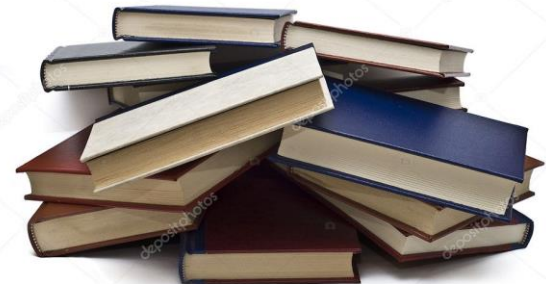
# Uninformed search



# Uninformed search strategies

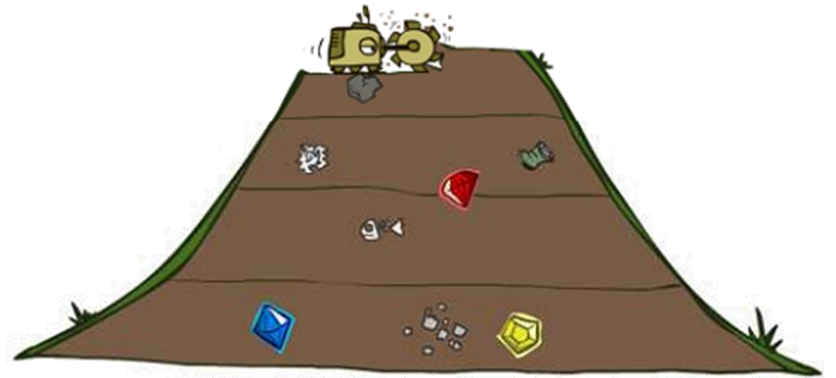
- An **uninformed search algorithm** is given no clue about **how close a state is to the goal(s)**.
- The agent can just **generate successors** and **distinguish a goal state** from a non-goal state.

Searching for a book in a disorderly stack is an instance of uninformed search, requiring the examination of each book until the desired one is found.



# Breadth-first search

---



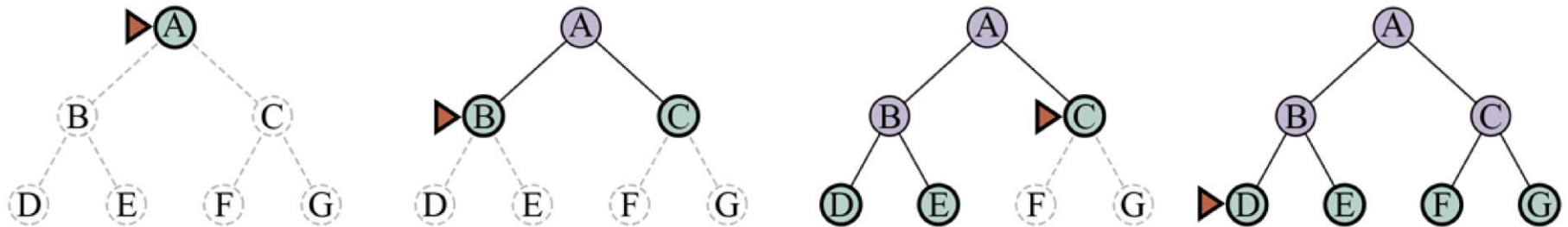
# Breadth-first search (BFS)

- **Idea:** Expand the root node first, then all the successors of the root node are next, then their successors, and so on.
- BFS is appropriate **when all actions have the same cost.**
- We implement BFS as a call to BEST-FIRST-SEARCH with the following evaluation function.

$$f(n) = \text{the depth of the node } n$$

- The depth of a node is the number of actions done to reach the node

# Breadth-first search: An example



BFS on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

# Breadth-first search: Implementation

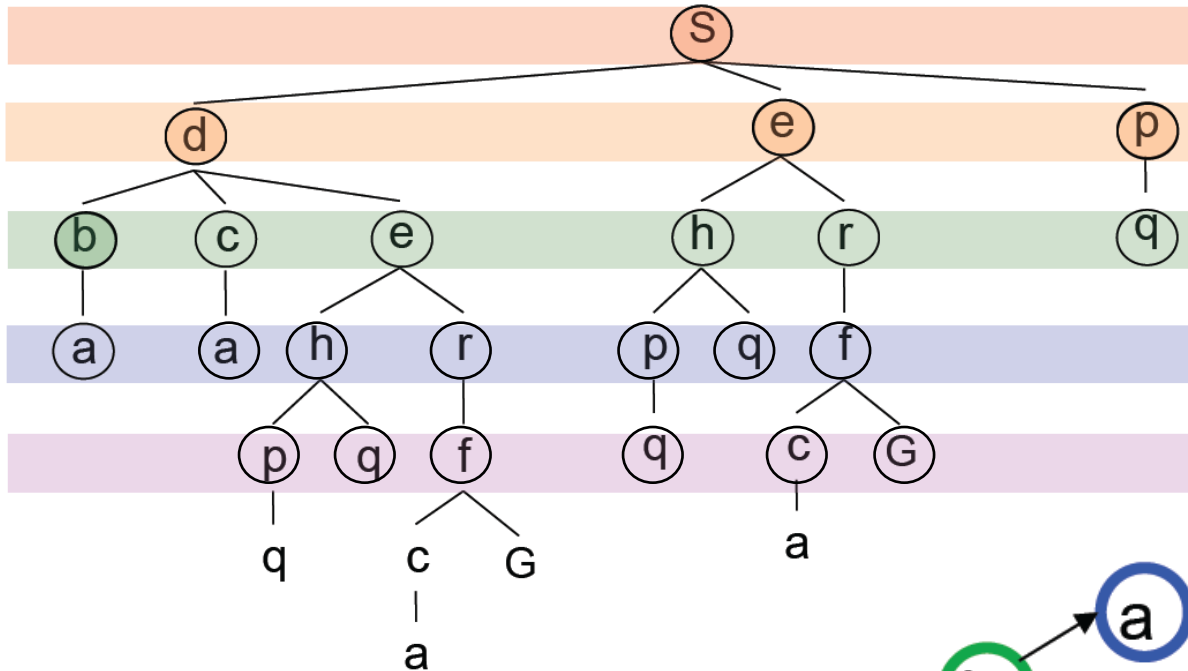
```
function BREADTH-FIRST-SEARCH(problem) returns a solution node, or failure  
  node  $\leftarrow$  NODE(problem.INITIAL)  
  if problem.IS-GOAL(node.STATE) then return node  
  frontier  $\leftarrow$  a FIFO queue, with node as an element  
  reached  $\leftarrow$  {problem.INITIAL}  
  while not IS-EMPTY(frontier) do  
    node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */  
    for each child in EXPAND(problem, node) do  
      s  $\leftarrow$  child.STATE  
      if problem.IS-GOAL(s) then return child  
      if s is not in reached then  
        add s to reached  
        add child to frontier  
  return failure
```



# BFS: Implementation tricks

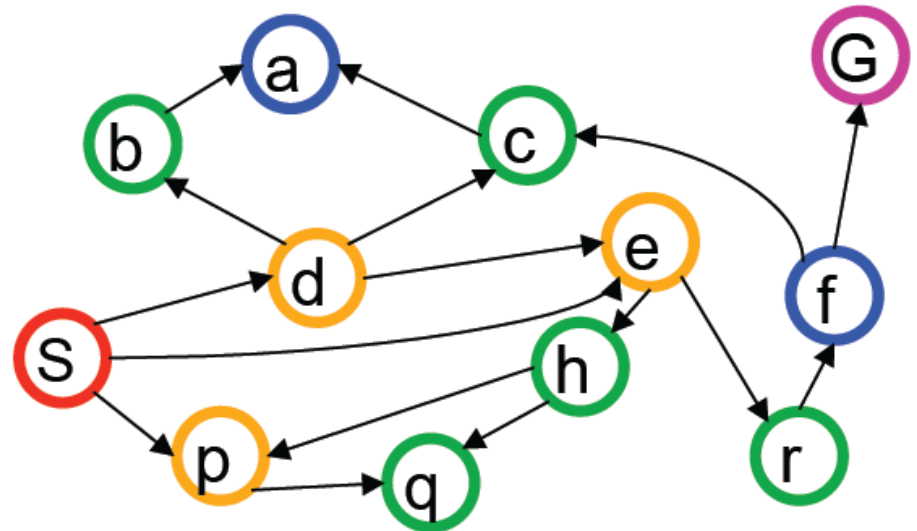
- A **FIFO queue will be faster** than a priority queue and **give the correct order of nodes**.
  - New nodes, which are always deeper than their parents, go to the back of the queue, and old nodes get expanded first.
- **Reached can be a set of states** rather than a mapping from states to nodes.
  - Once a state is reached, there is no better path to that state.
- BFS applies **early goal test** to check whether a node is a solution **as soon as it is generated**.
  - Best-first search use the **late goal test**—waiting until a node is popped off the queue.

# Breadth-first search: An example



Expansion order:

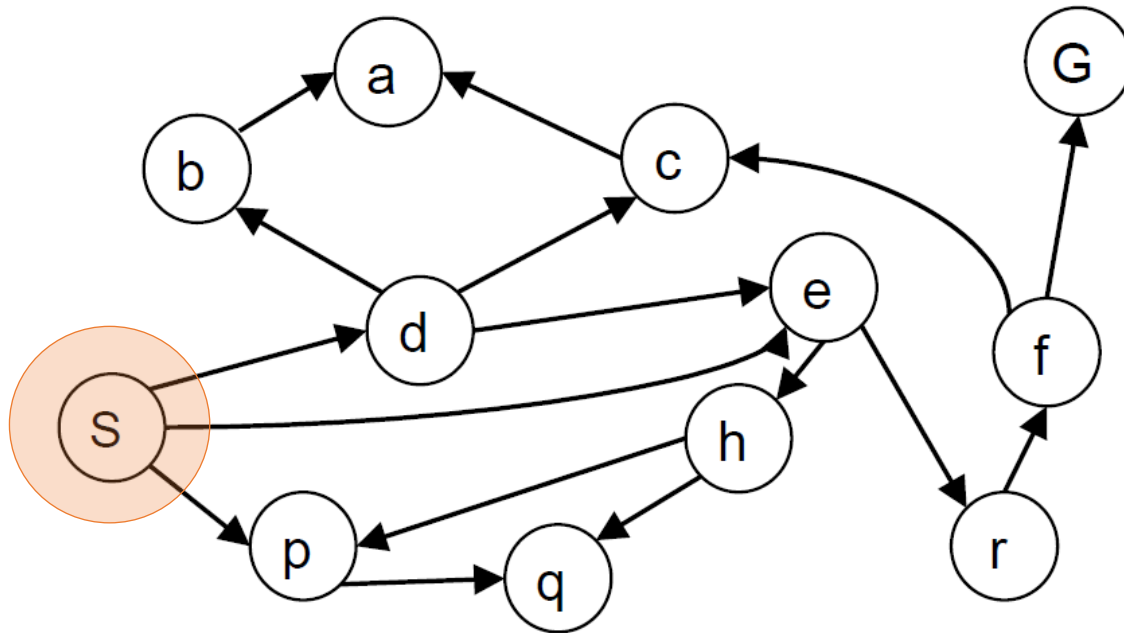
S, d, e, p, b, c, h, r, q, a, f, G



# Breadth-first search: An example

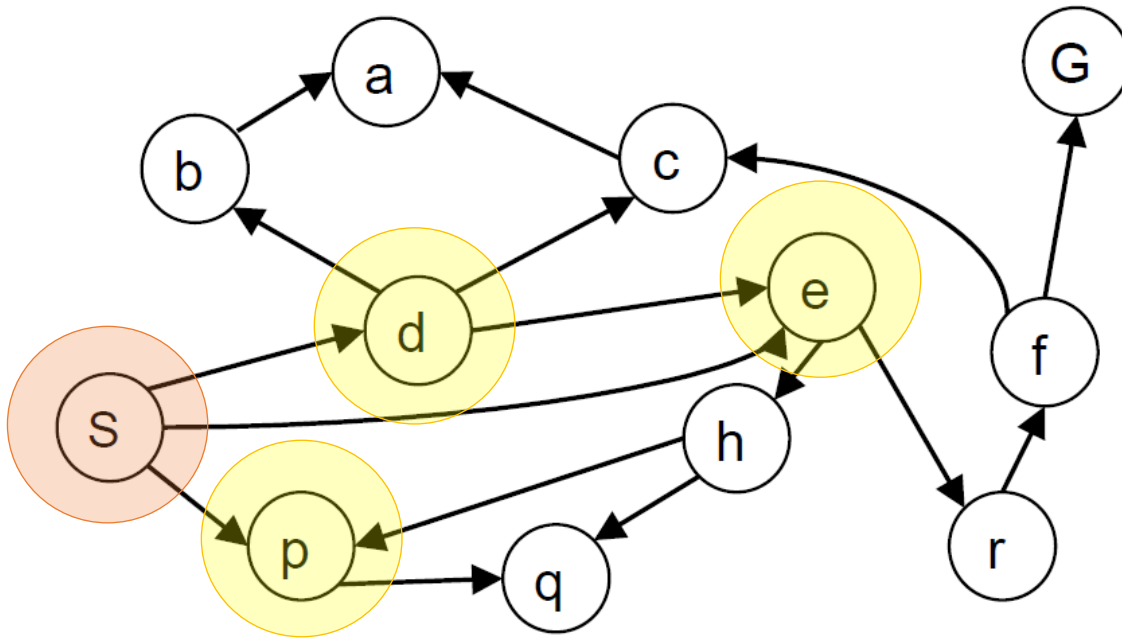
S

Search tree

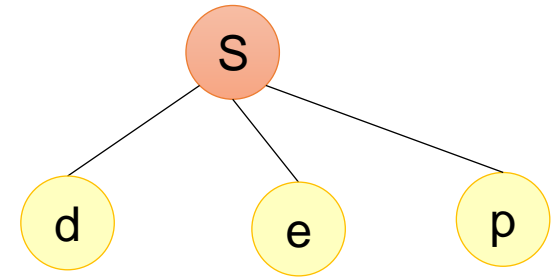


$d = 0$

# Breadth-first search: An example

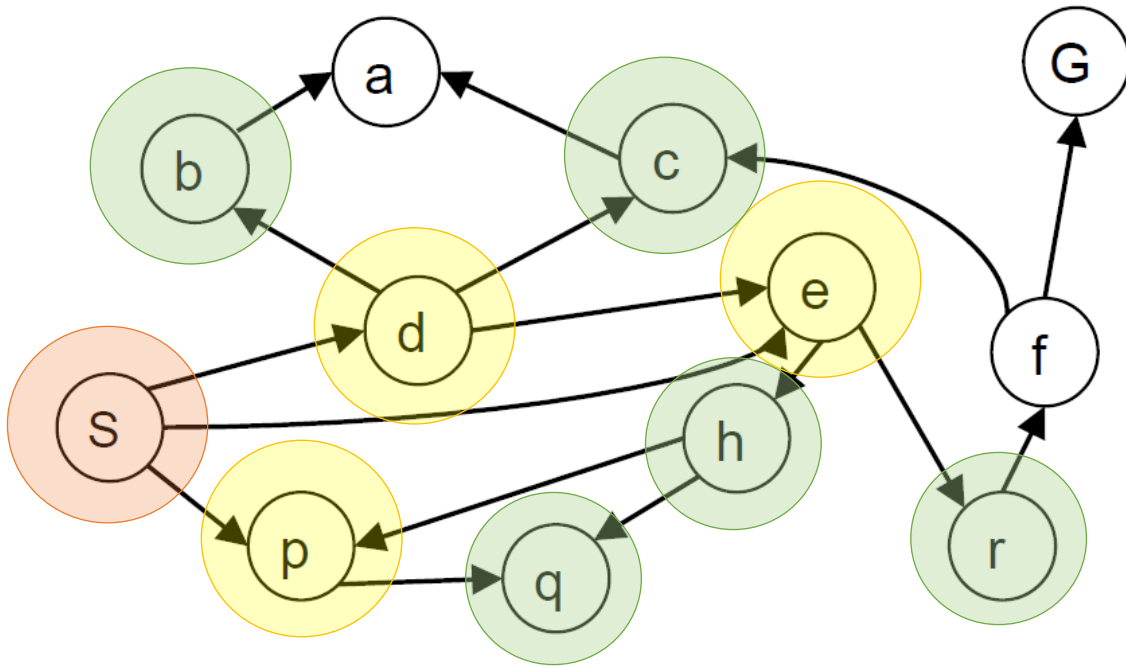


$d = 1$

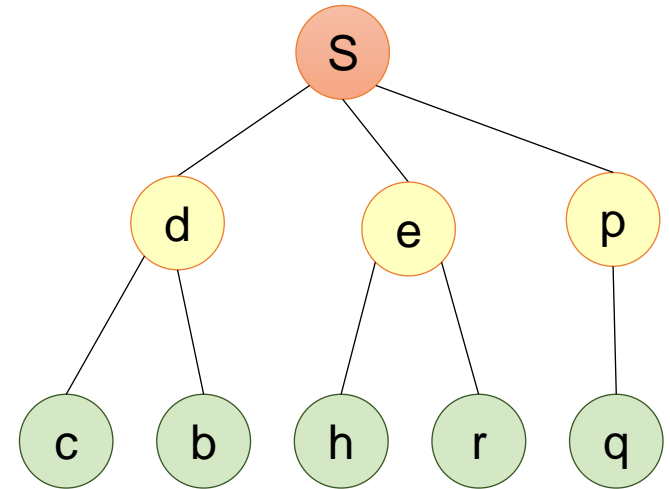


Search tree

# Breadth-first search: An example

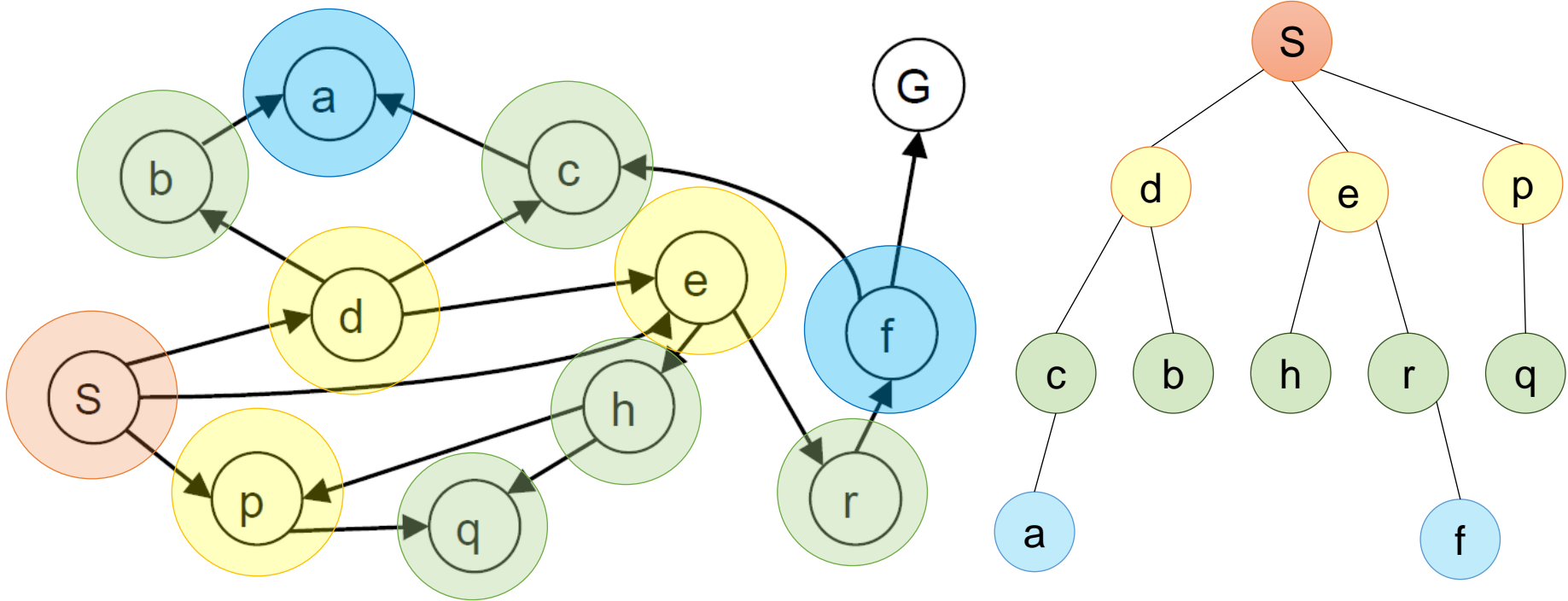


$d = 2$



Search tree

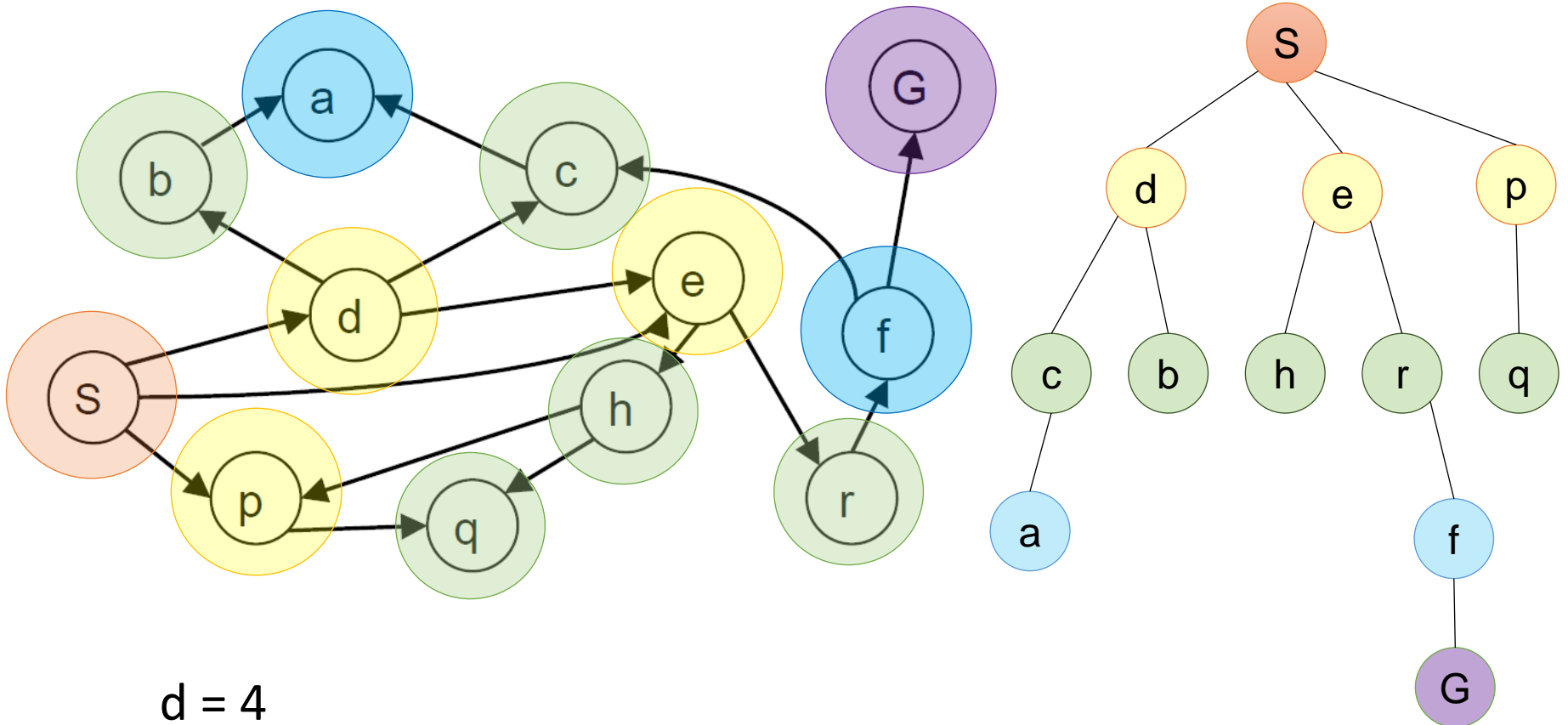
# Breadth-first search: An example



$d = 3$

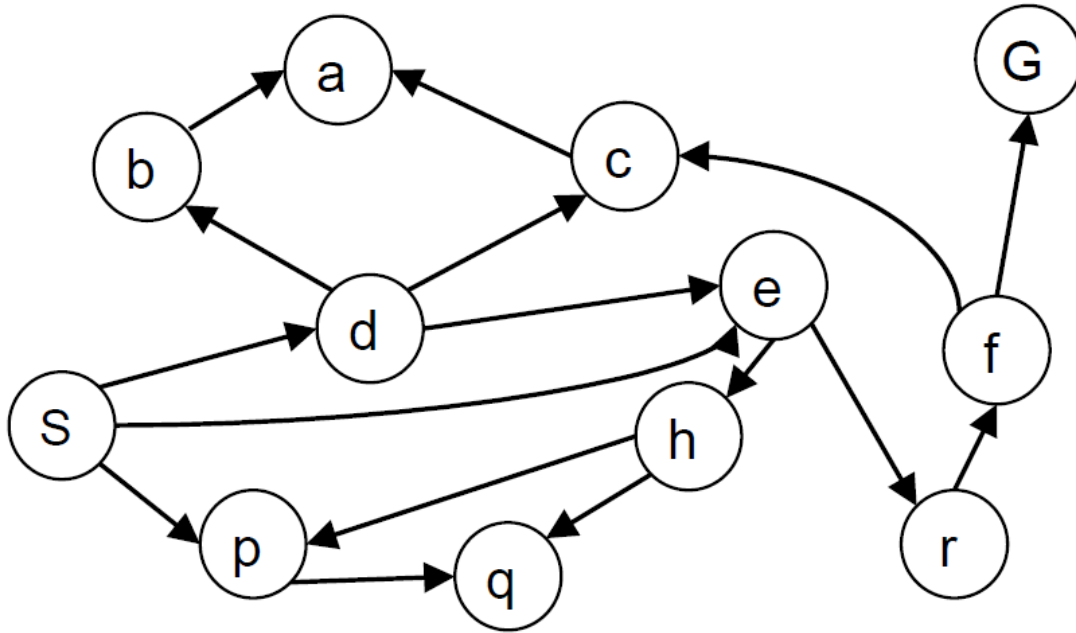
Search tree

# Breadth-first search: An example

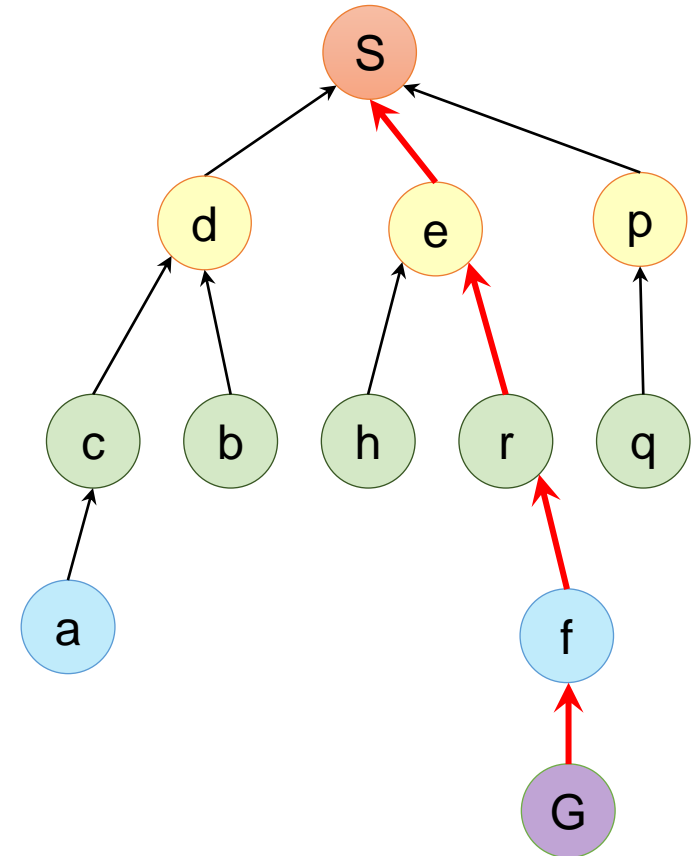


Search tree

# Breadth-first search: An example



Search path:  $S \rightarrow e \rightarrow r \rightarrow f \rightarrow G$

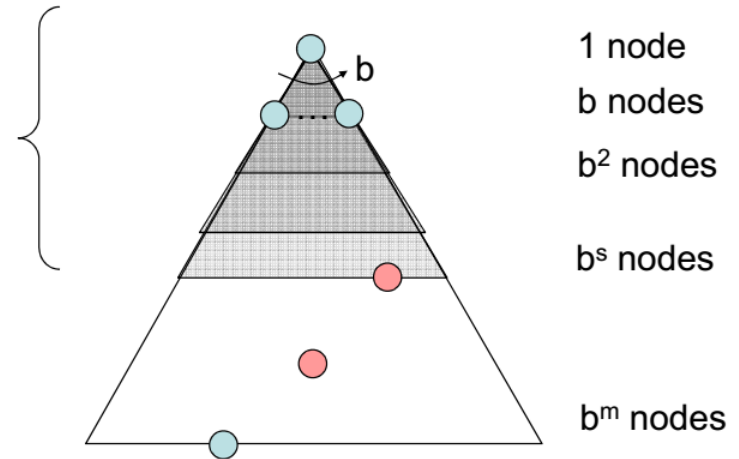


Search tree



# An evaluation of BFS

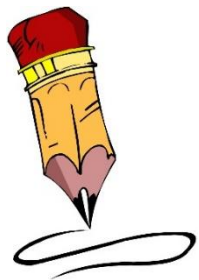
- **Completeness:** ✓
- **Optimality:** ✓ for problems where all actions have the same cost.
- Imagine searching a uniform tree where every state has  $b$  successors and the solution is at depth  $d$ .
- The total number of nodes generated is
$$1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$$
- All the nodes remain in memory, so both **time complexity** and **space complexity** are  $O(b^d)$ .



# BFS: A hypothetical experiment

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	$10^6$	1.1 seconds	1 gigabyte
8	$10^8$	2 minutes	103 gigabytes
10	$10^{10}$	3 hours	10 terabytes
12	$10^{12}$	13 days	1 petabyte
14	$10^{14}$	3.5 years	99 petabytes
16	$10^{16}$	350 years	10 exabytes

Time and memory requirements for BFS. The numbers shown assume branching factor  $b = 10$ ; 1 million nodes/second; 1000 bytes/node.

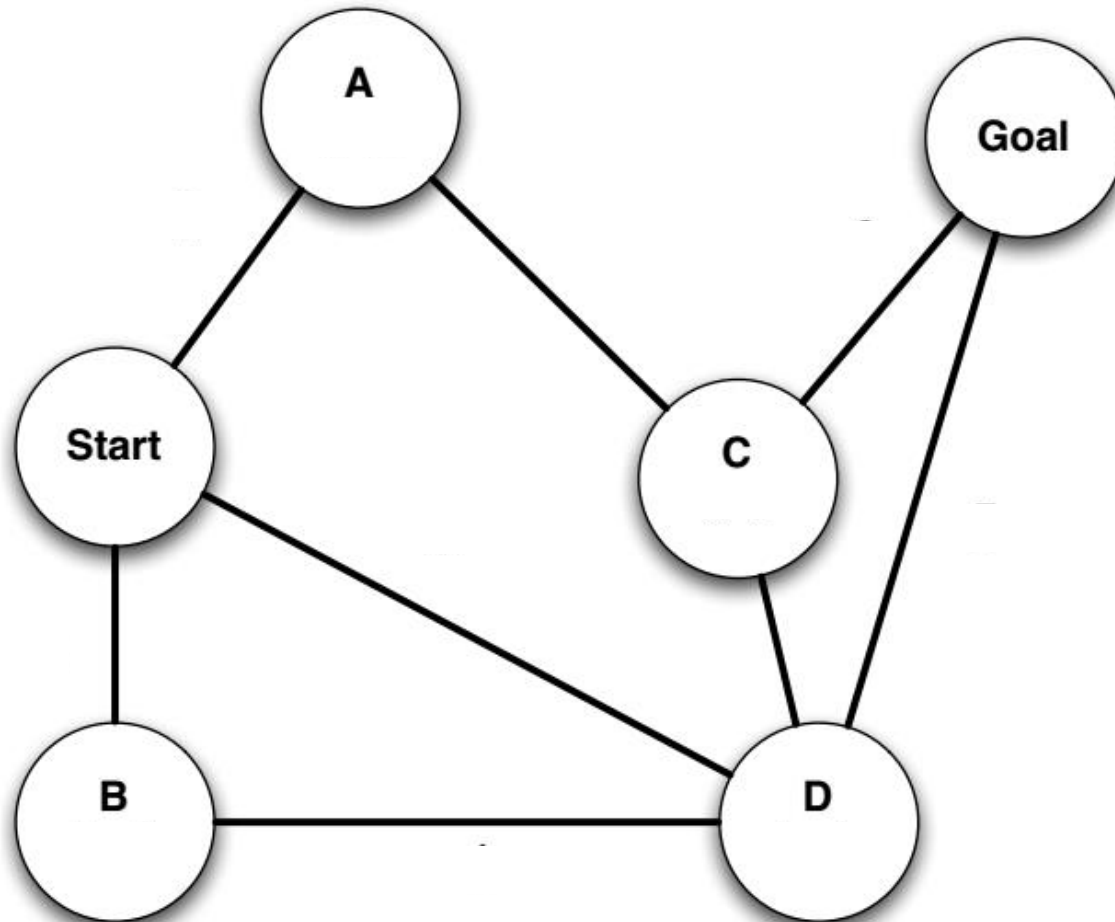


The memory requirements are a bigger problem for BFS than the execution time.

In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instance

# Quiz 01: Breadth-first search

Work out the order in which states are expanded and the path returned. Assume ties resolve in such a way that states with earlier alphabetical order are expanded first.



# Uniform-cost search

---



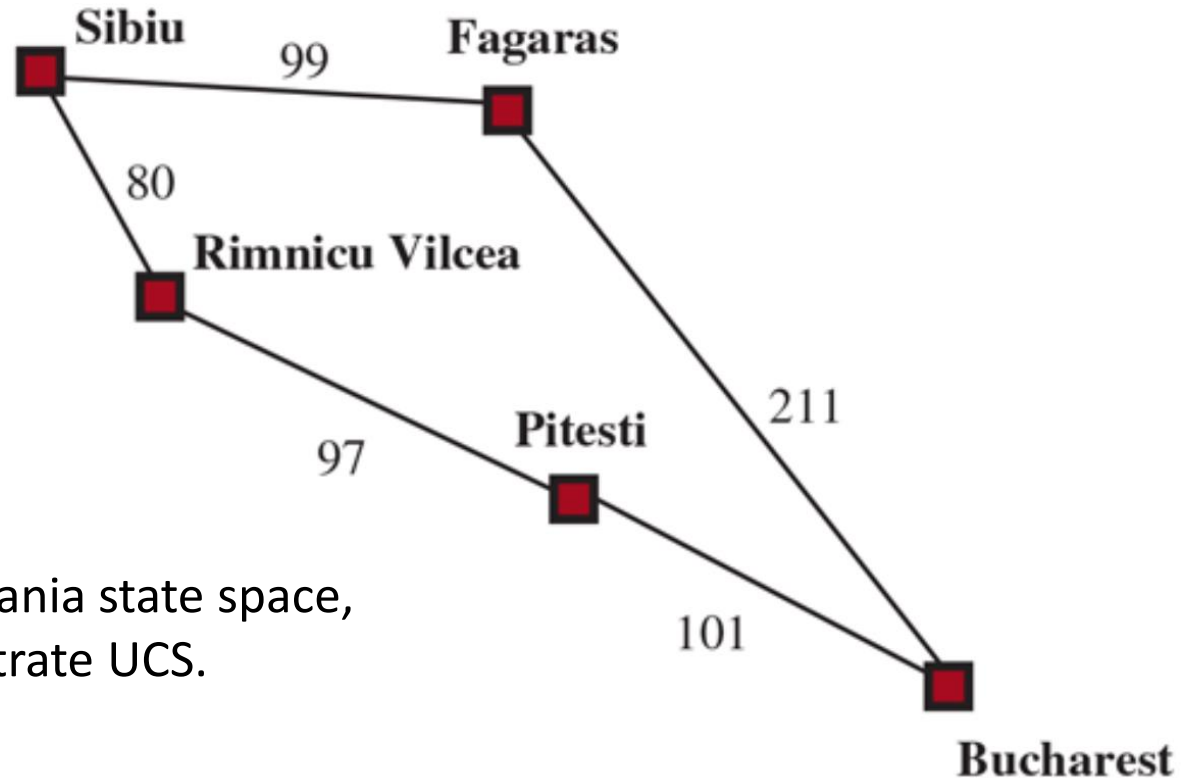
# Uniform-cost search (UCS)

- **Idea:** UCS resembles the mechanism of Dijkstra's algorithm.
- UCS is a good choice when the actions have different costs.
- This implements BEST-FIRST-SEARCH with the following evaluation function.

$$f(n) = \text{the cost of the path from the root to the current node}$$

- The path cost is usually denoted as  $g(n)$ .

# Uniform-cost search: An example



Part of the Romania state space,  
selected to illustrate UCS.

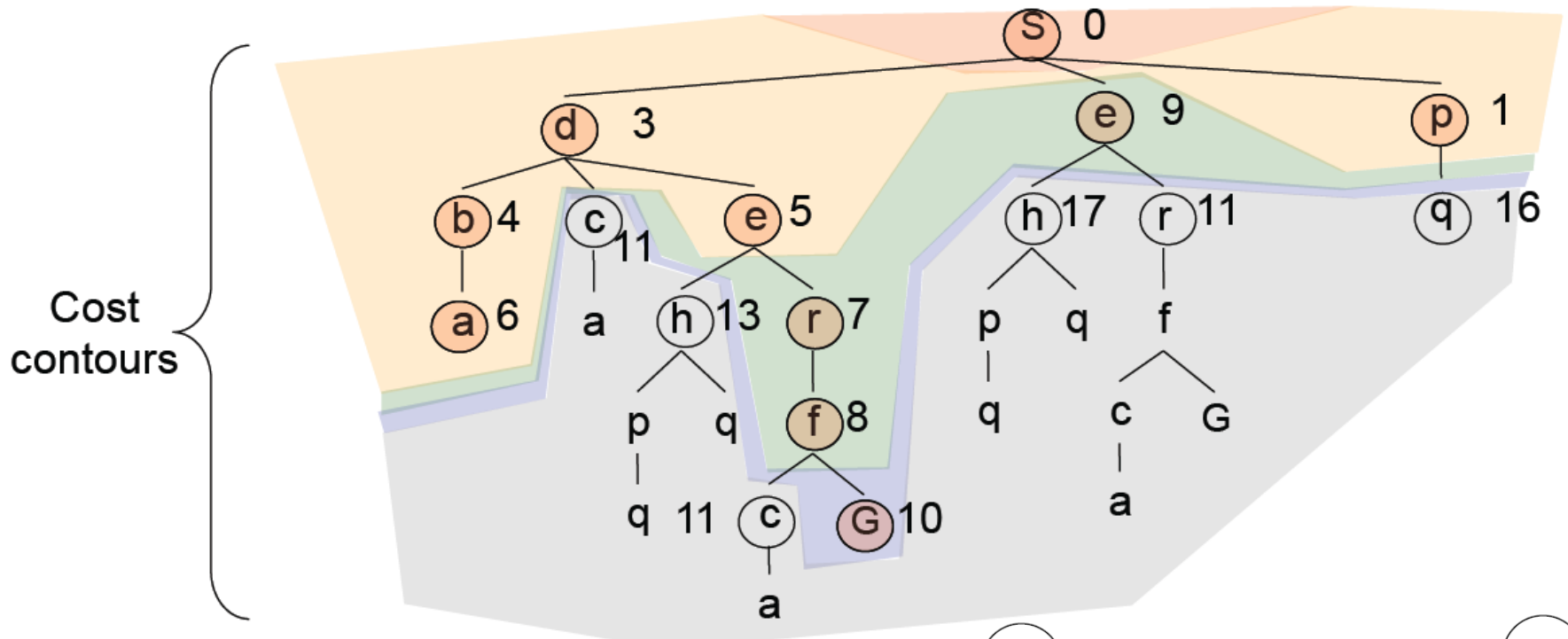
If we had checked for a goal upon generating a node rather than when expanding the lowest-cost node, we would have returned a higher-cost path.

# Uniform-cost search: Implementation

**function** UNIFORM-COST-SEARCH(*problem*) **returns** a solution node or *failure*  
**return** BEST-FIRST-SEARCH(*problem*, PATH-COST)

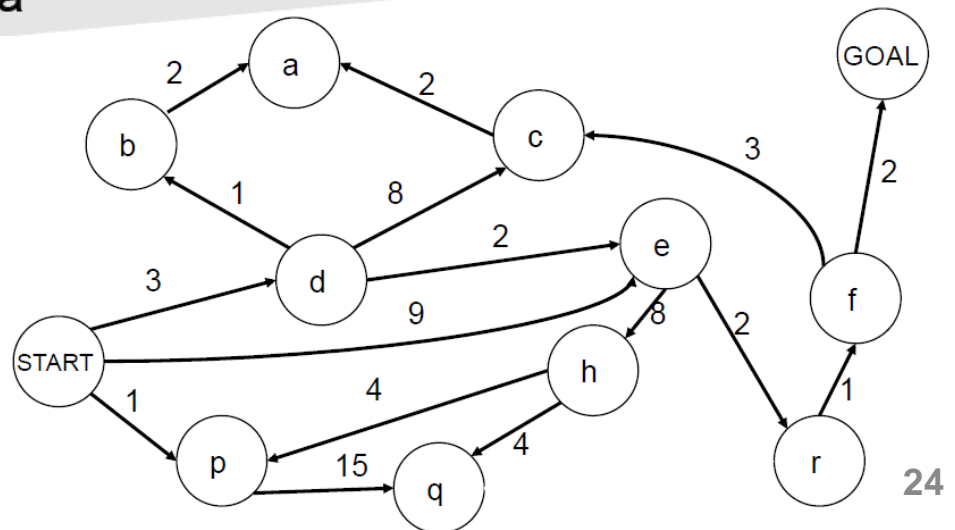
- UCS expands the node  $n$  with the **lowest** path cost  $g(n)$ .
- The **goal test** is applied to a node when it is **expanded**.
- There may be a **better path to a node currently on the frontier**, yet there is **no better path found to an expanded node**.

# Uniform-cost search: An example



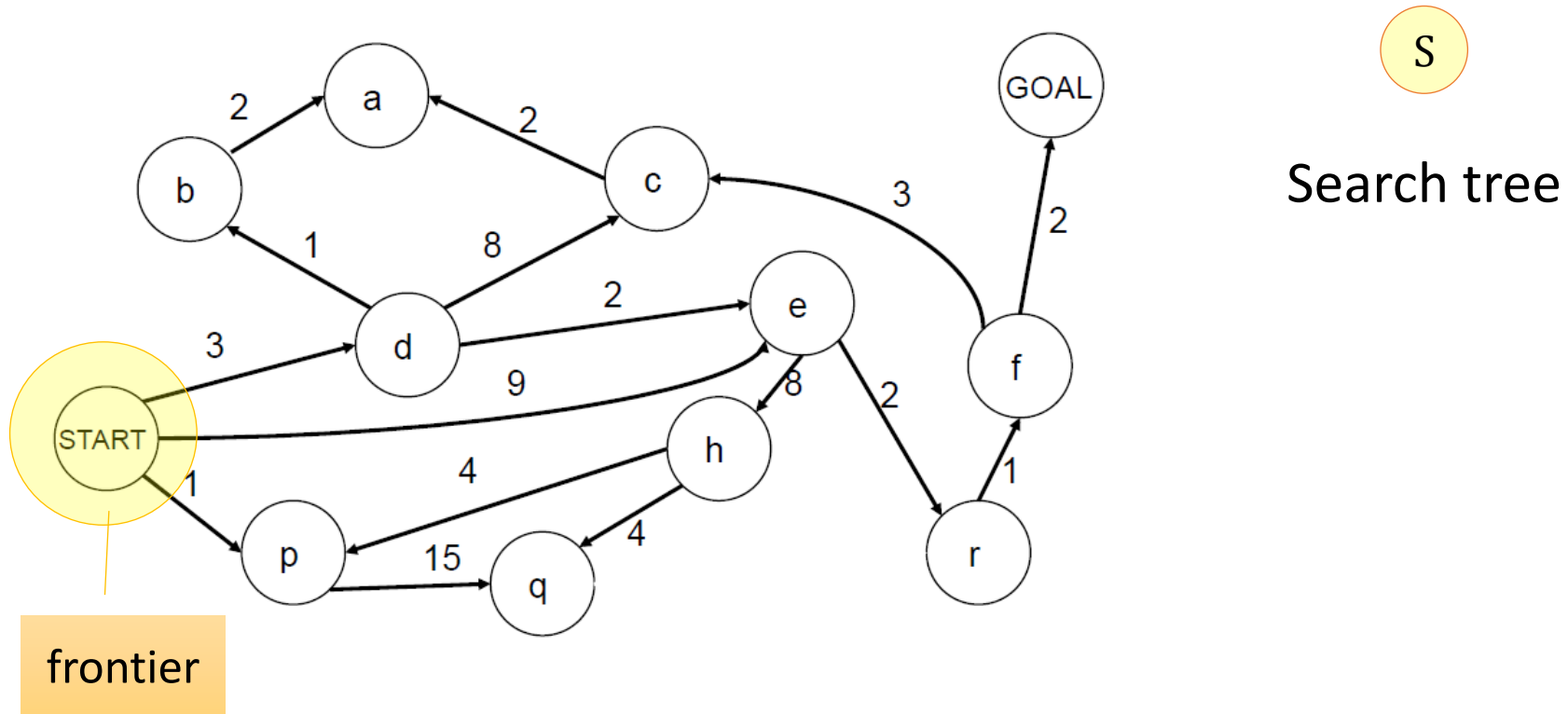
Expansion order:

S, p, d, b, e, a, r, f, e, G



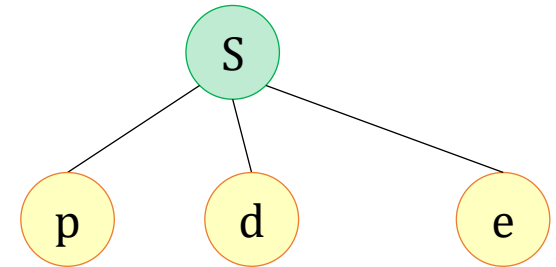
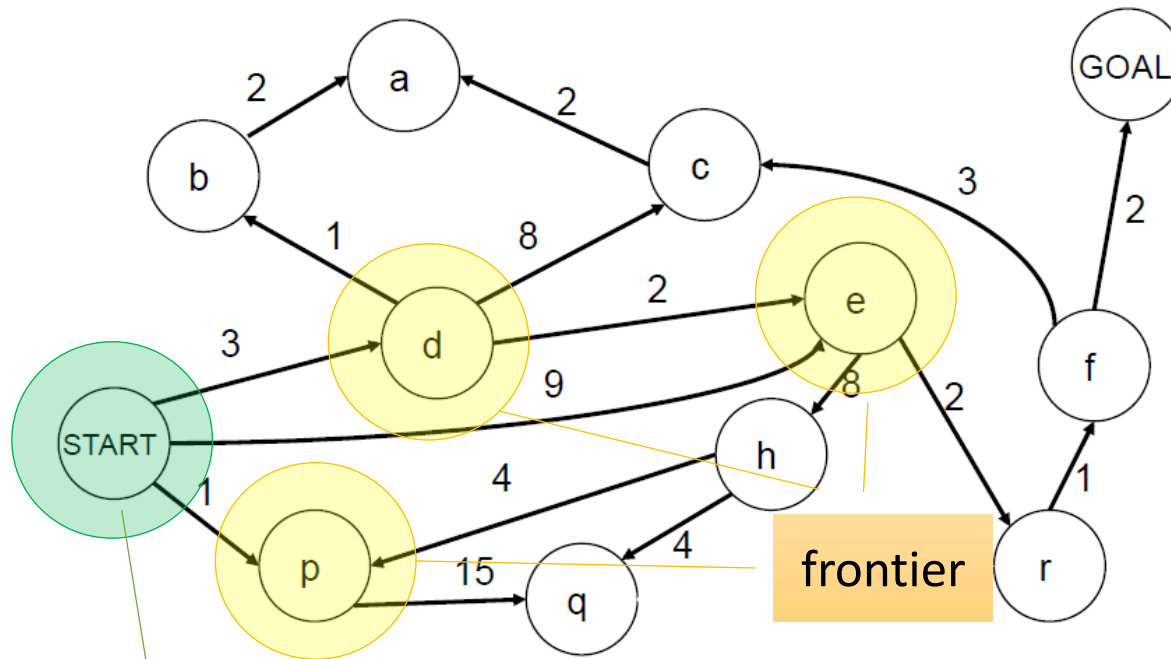


# Uniform-cost search: An example



PQ = { (S:0) }

# Uniform-cost search: An example

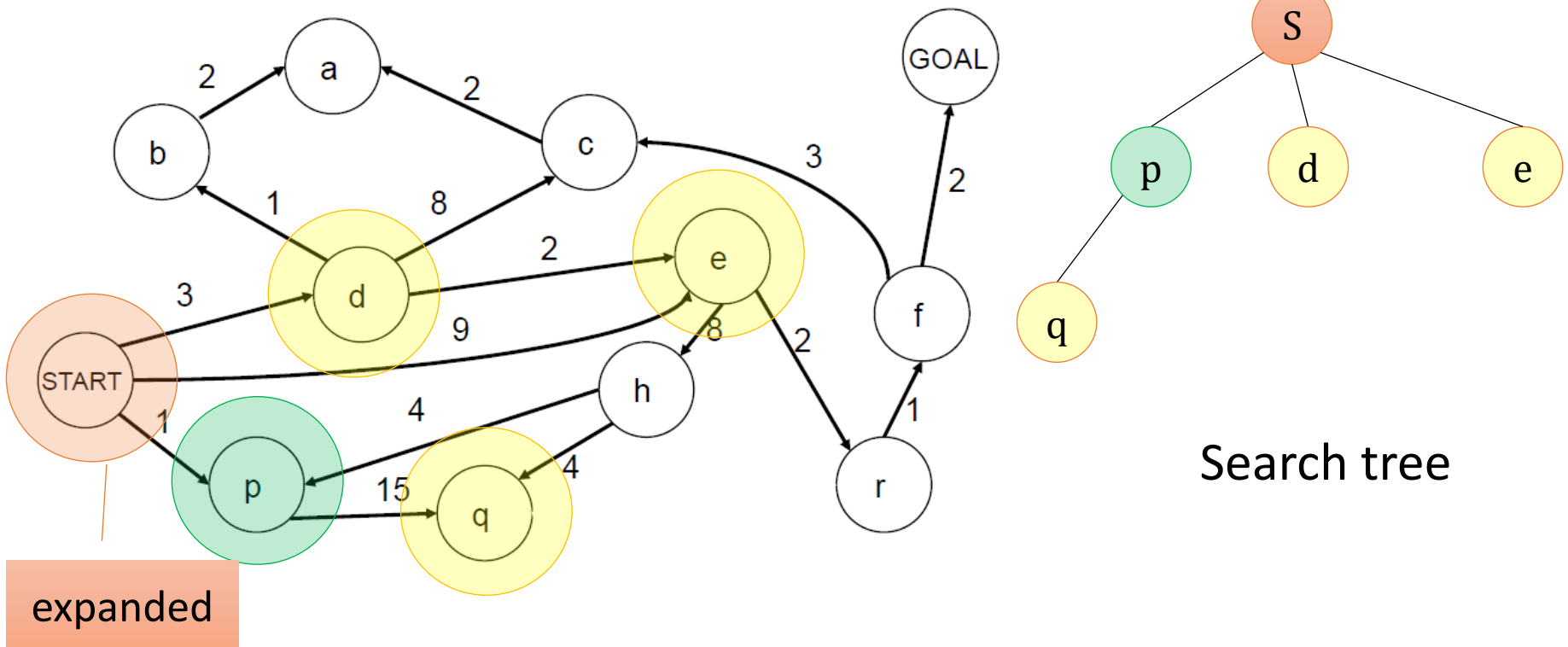


Search tree

Selected for  
expansion

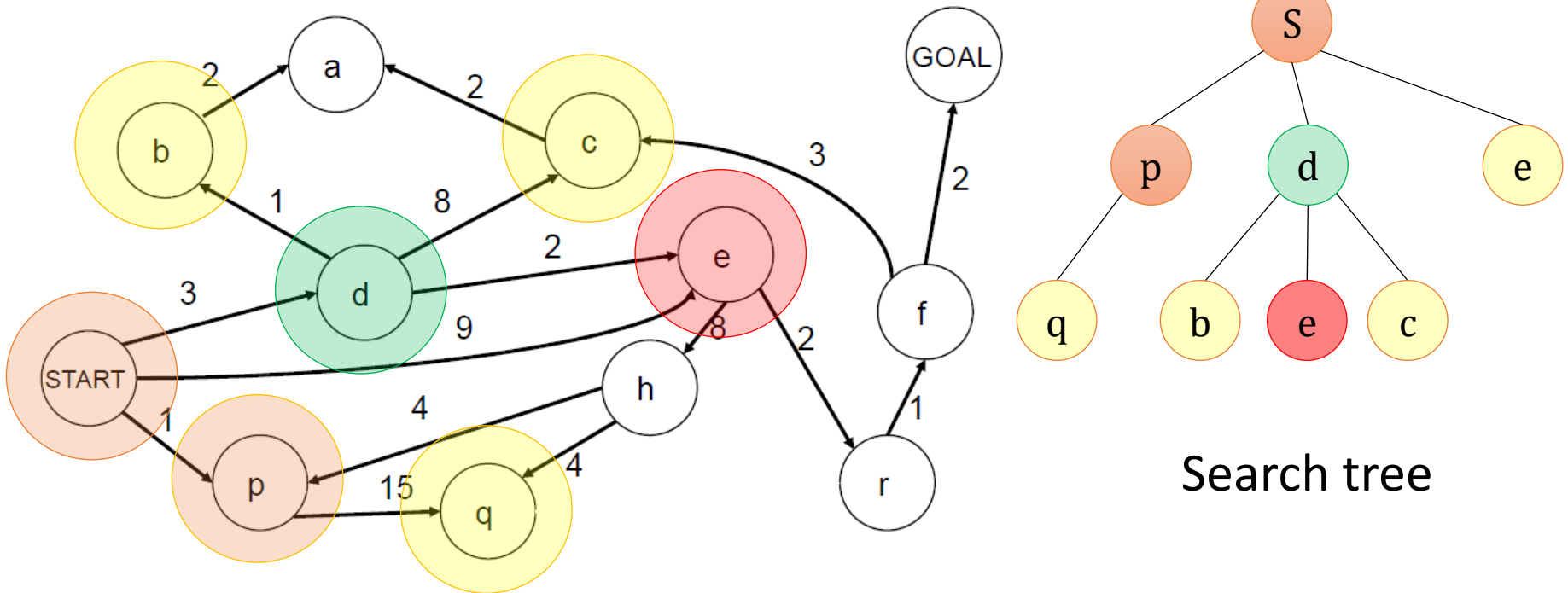
$$PQ = \{ (p:1), (d:3), (e:9) \}$$

# Uniform-cost search: An example



PQ = { (d:3), (e:9), (q:16) }

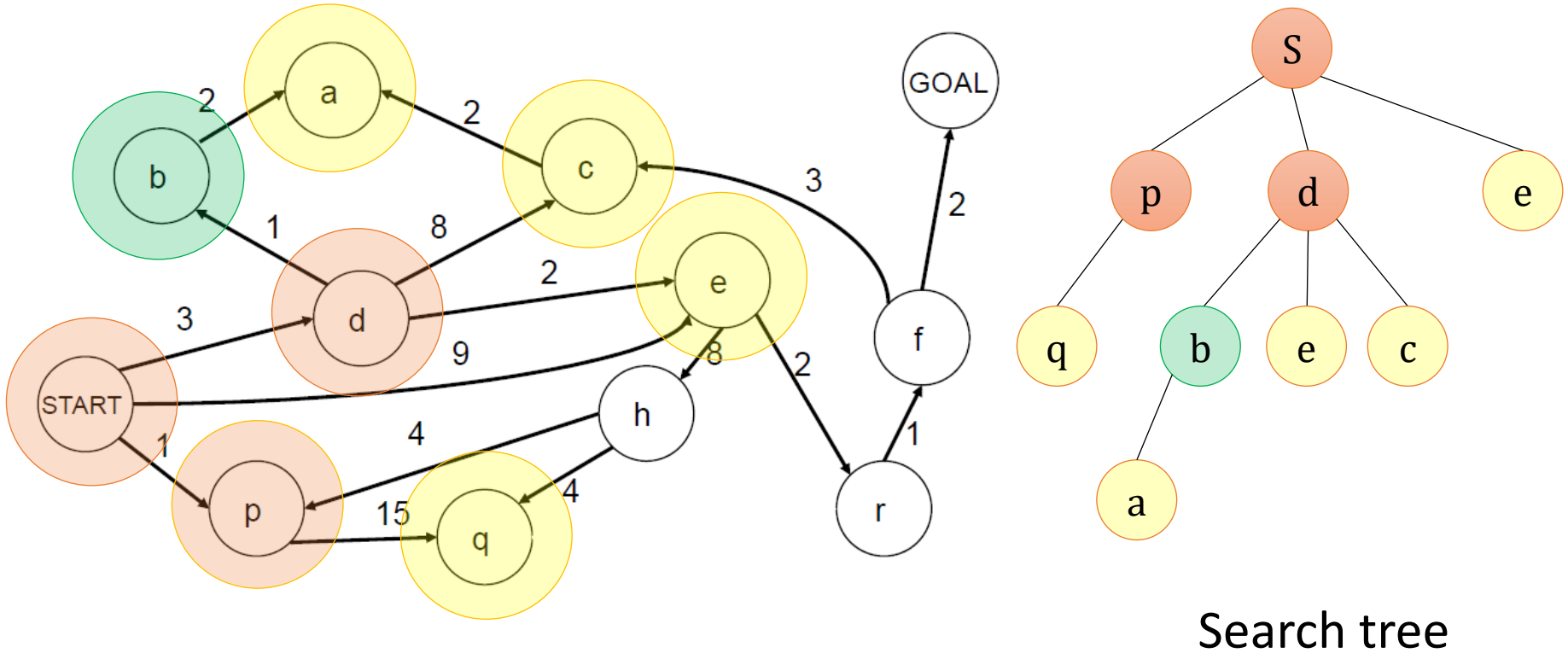
# Uniform-cost search: An example



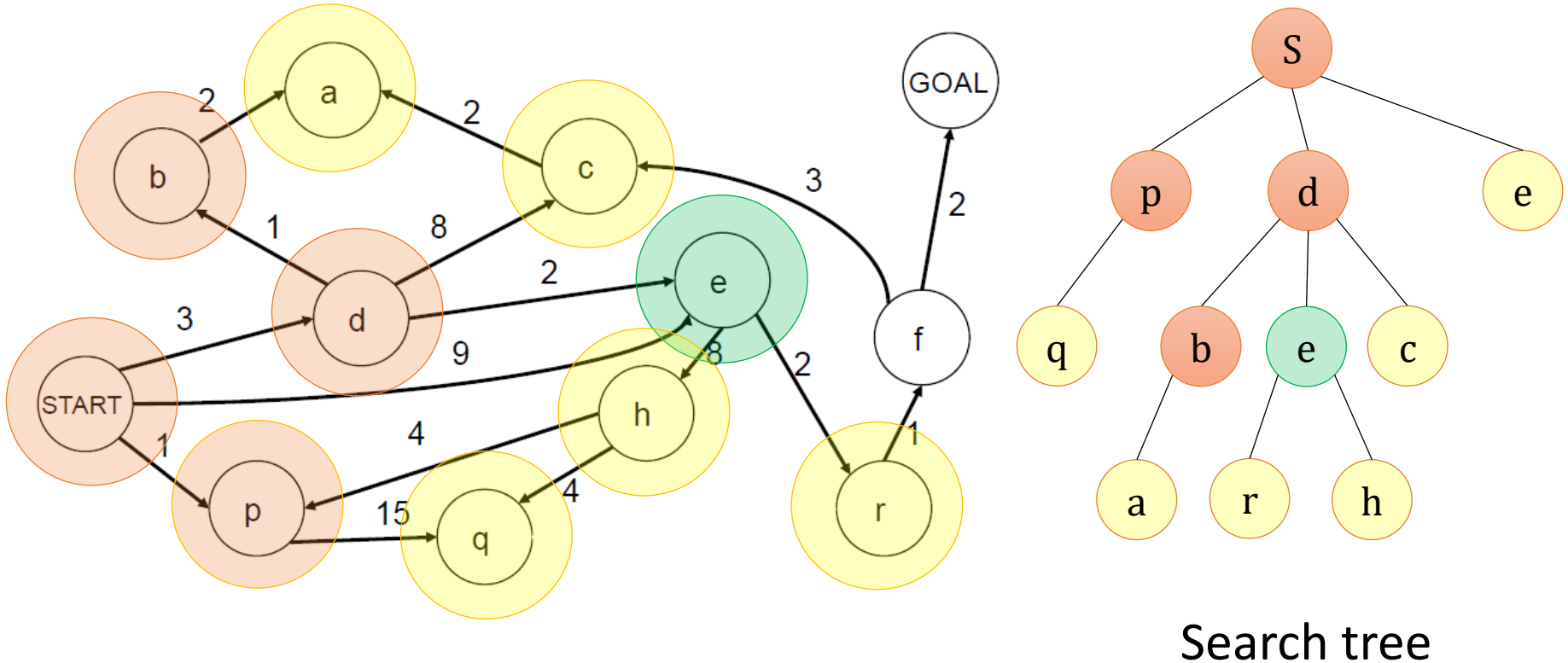
PQ = { (b:4), (e:5), (c:11), (q:16) }

Update path cost of e

# Uniform-cost search: An example

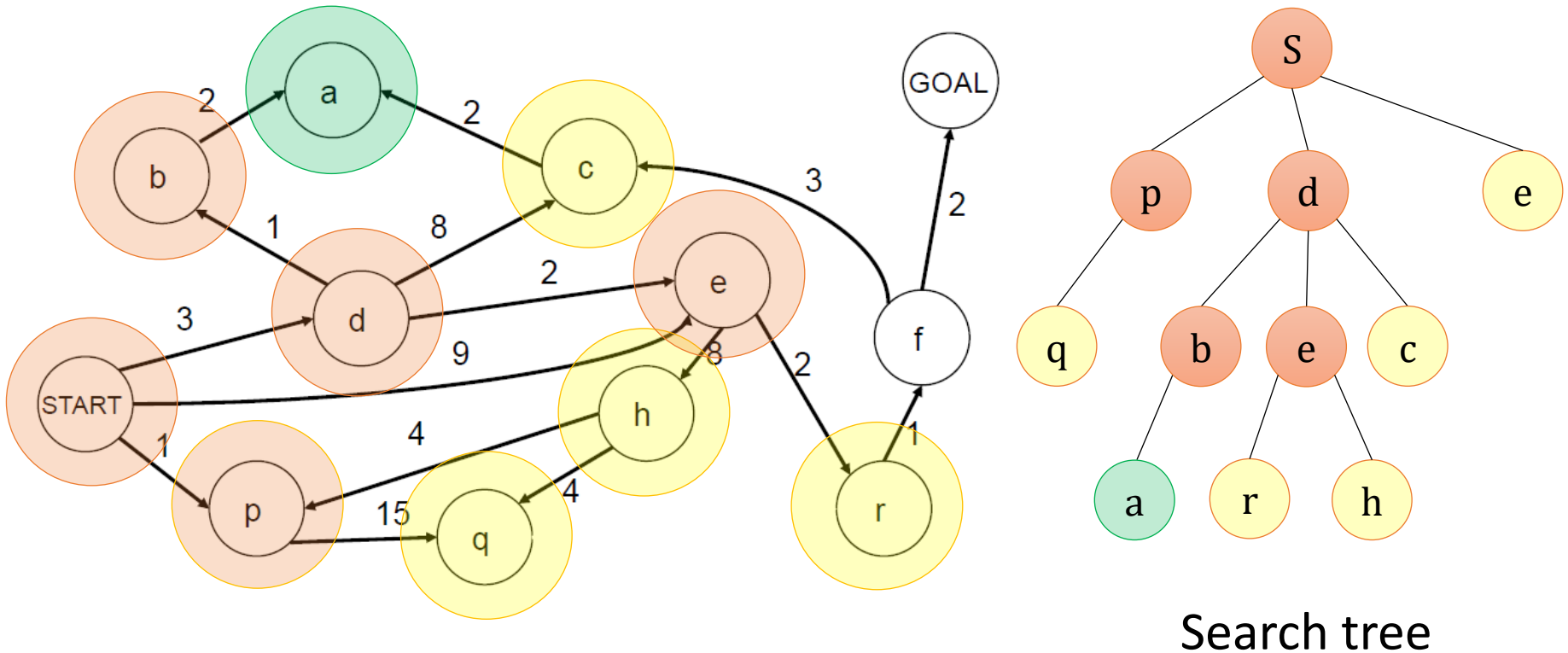


# Uniform-cost search: An example



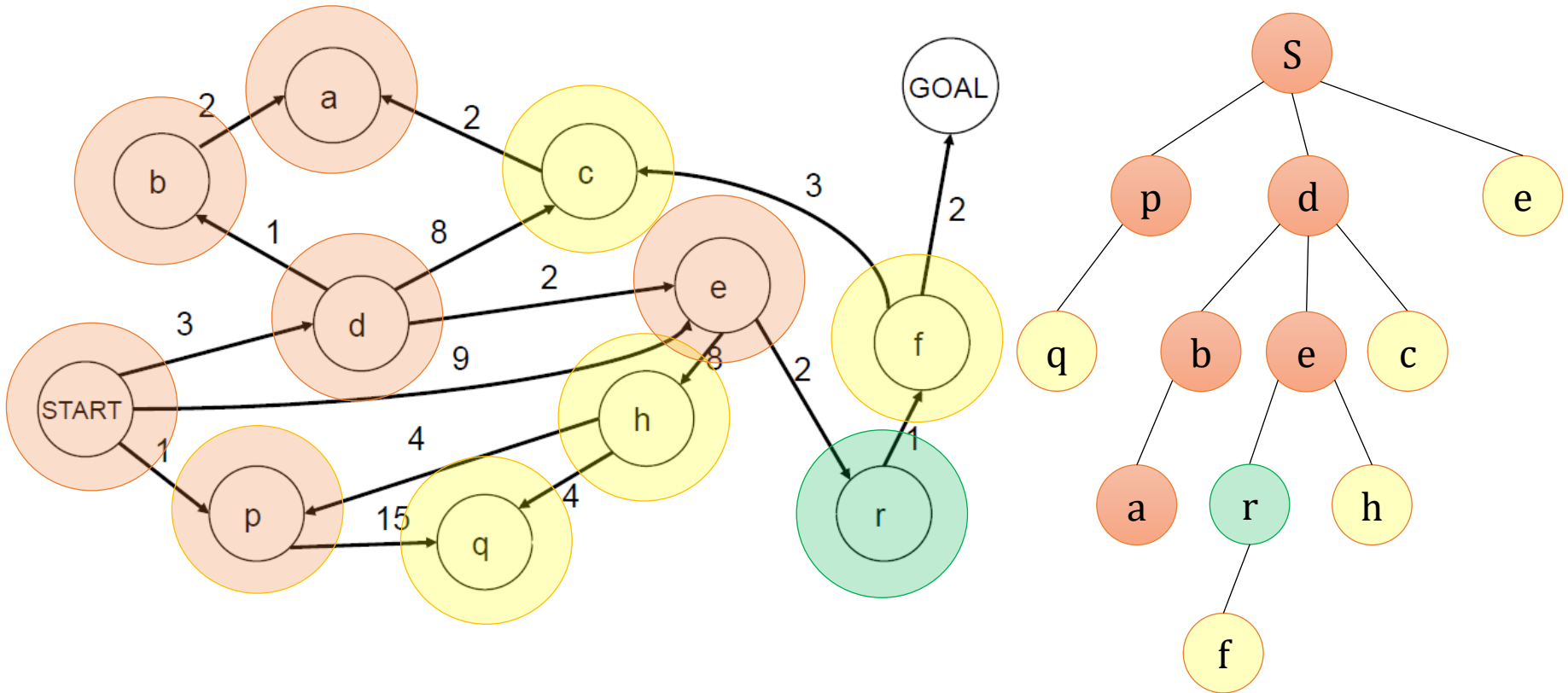
PQ = { (a:6), (r:7), (c:11), (h:13), (q:16) }

# Uniform-cost search: An example



PQ = { (r:7), (c:11), (h:13), (q:16) }

# Uniform-cost search: An example

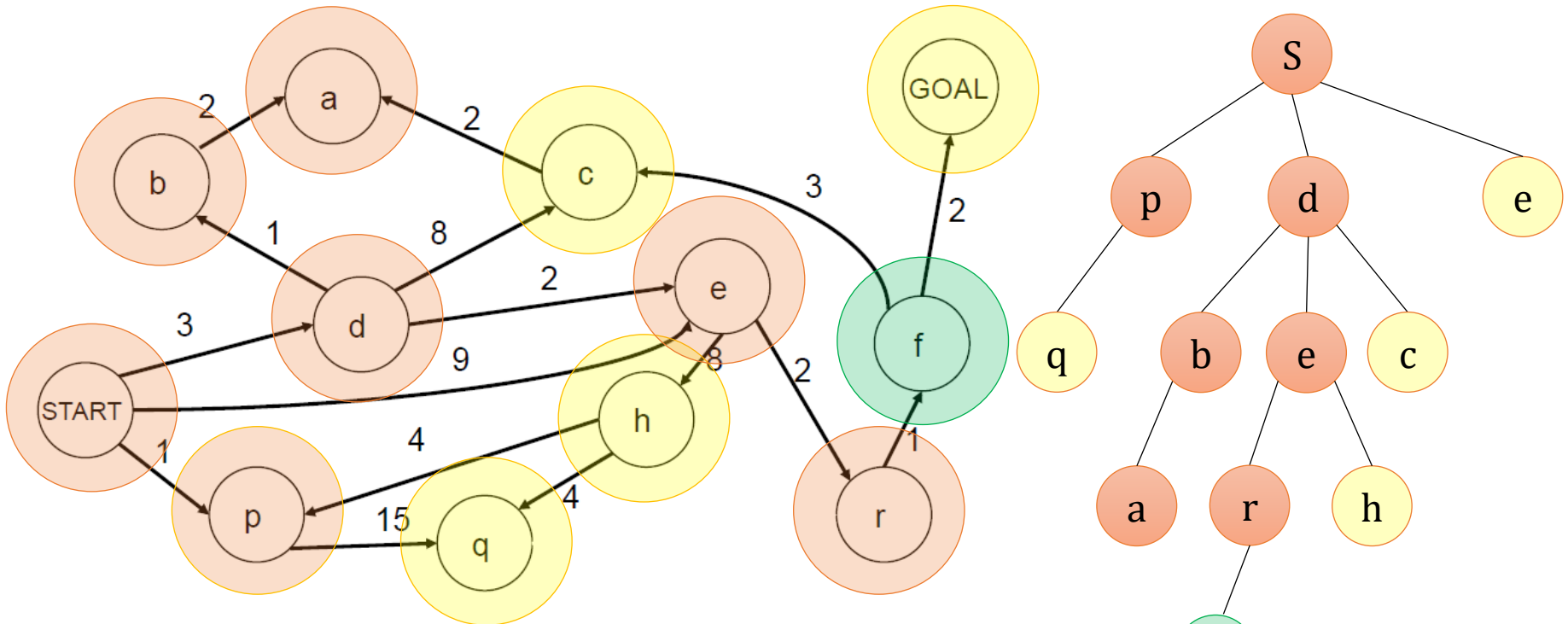


PQ = { (f:8), (c:11), (h:13), (q:16) }

Search tree



# Uniform-cost search: An example



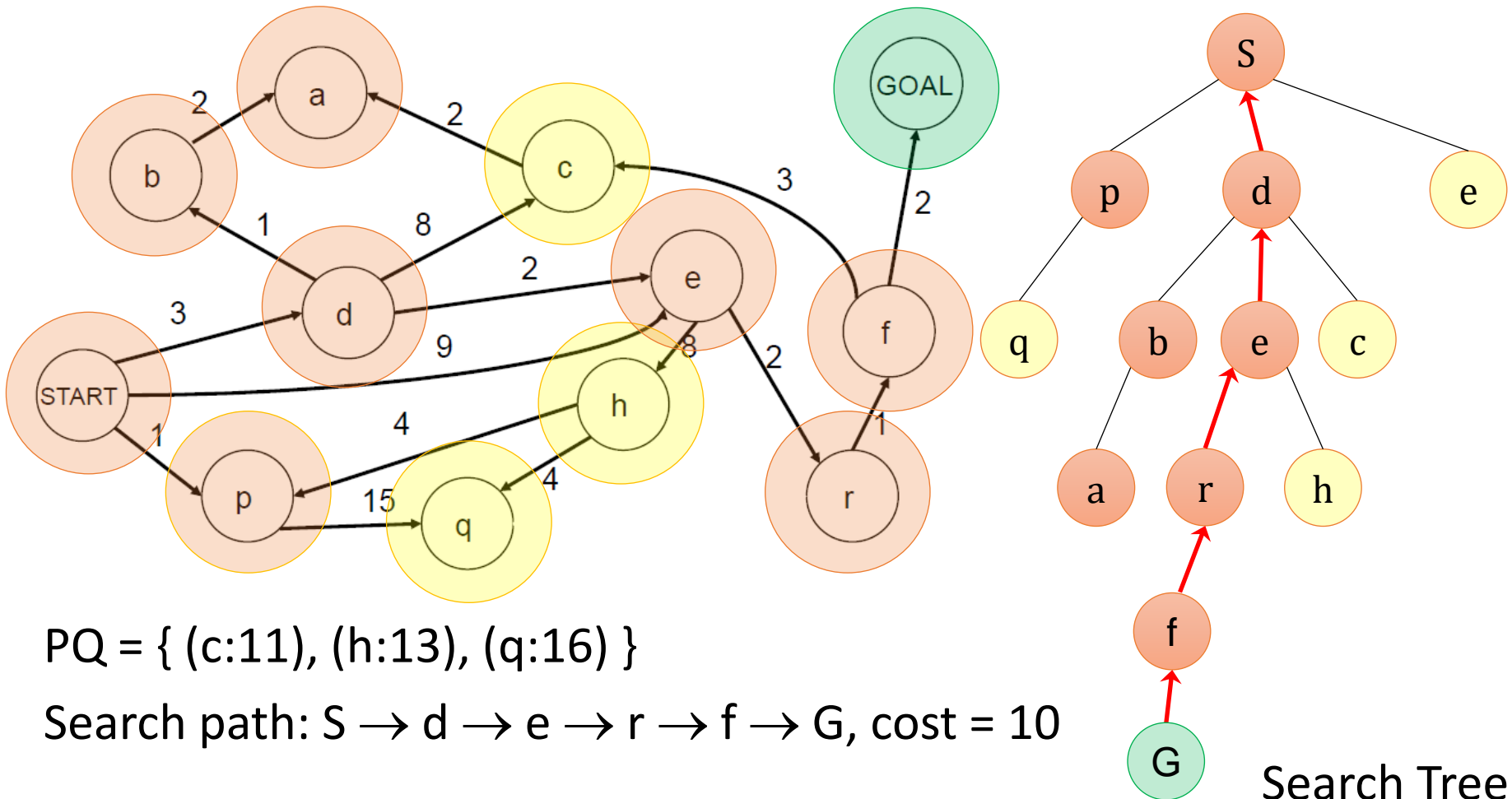
PQ = { (G:10), (c:11), (h:13), (q:16) }

Not update path cost of c

Should we STOP here?

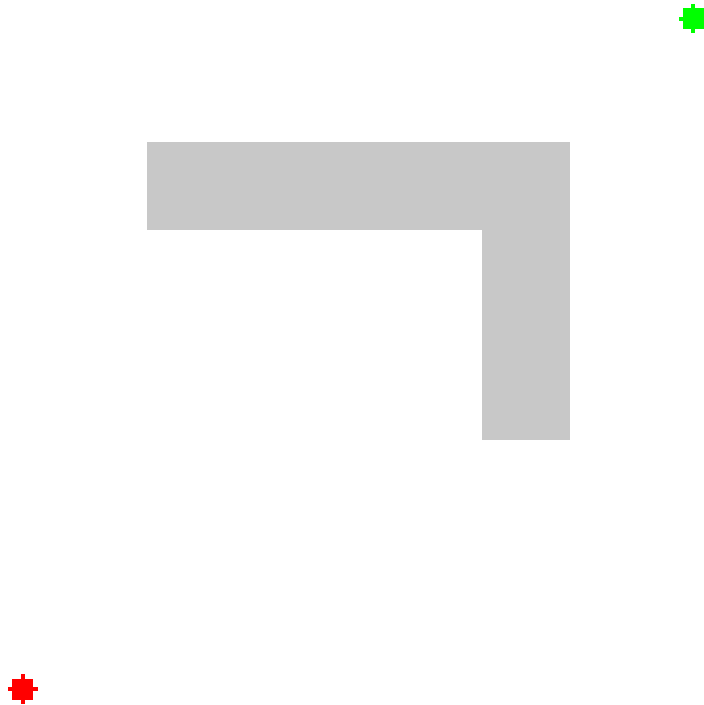
Search tree

# Uniform-cost search: An example



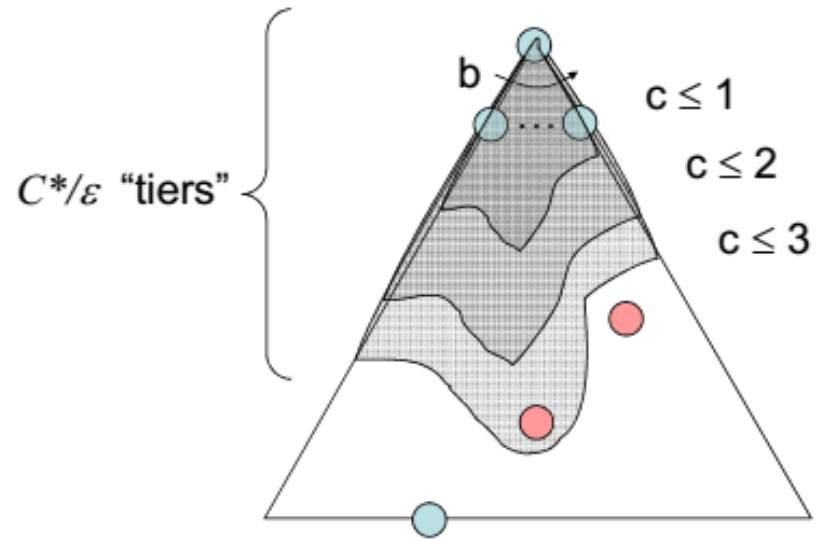
NO! Only stop when GOAL is POPPED off PQ

# Uniform-cost search: A demonstration



# An evaluation of UCS

- Let  $C^*$  be the cost of the optimal solution, and  $\epsilon$  be the lower bound of the cost of each action ( $\epsilon > 0$ ).
- Completeness: ✓
- Optimality: ✓
- Both **time complexity** and **space complexity** are  $O(b^{1+\lceil C^*/\epsilon \rceil})$ , which can be greater than  $O(b^d)$ .
  - UCS may explore large trees of small steps before exploring paths involving large and perhaps useful action.
- When all step costs are equal,  $O(b^{1+\lceil C^*/\epsilon \rceil})$  is just  $O(b^{d+1})$ .
  - UCS unnecessarily expands nodes at depth  $d$ , while BFS stops as soon as it generates a goal.

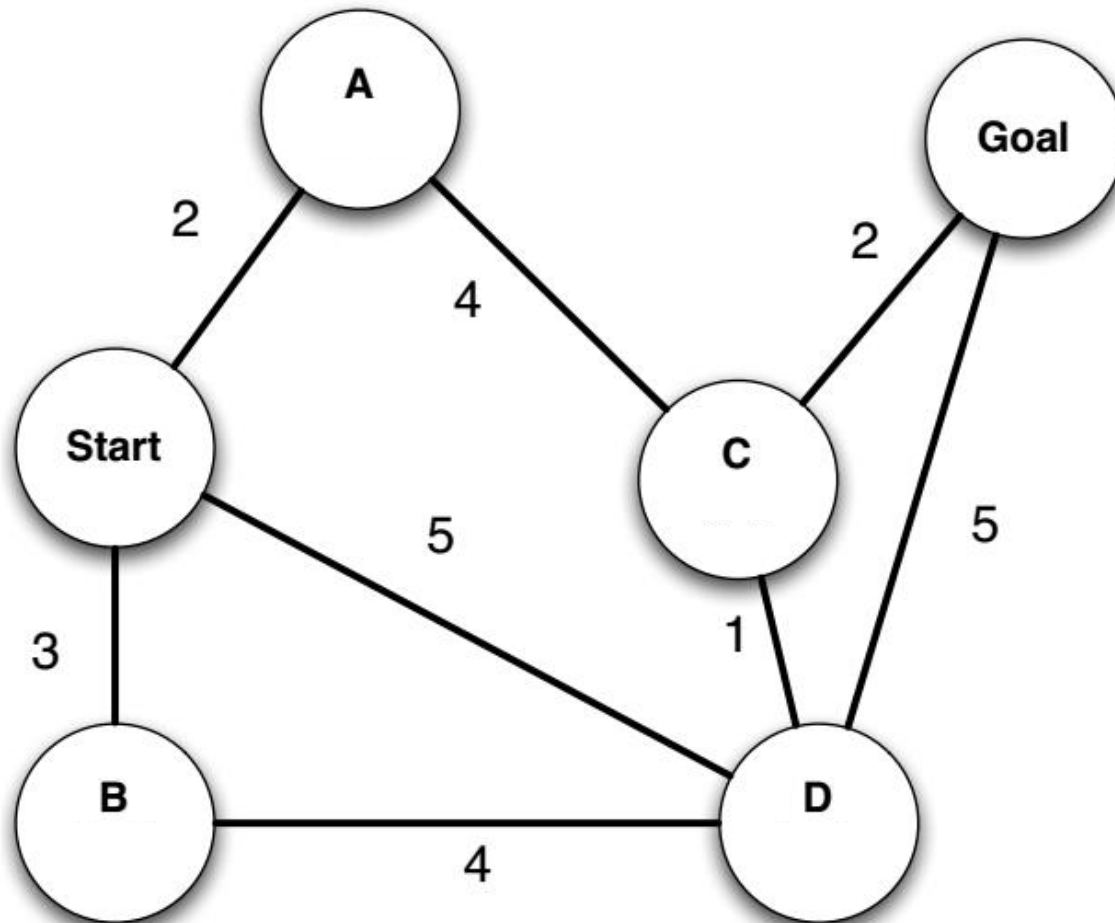


# Uniform-cost search: Optimality

- **Graph separation property:** Every path from the initial state to an unexplored state must pass through a state on the frontier.
  - Proved inductively
- **Optimality of UCS:** proof by contradiction
  - Suppose UCS terminates at goal state  $n$  with path cost  $g(n) = C$  but there exists another goal state  $n'$  with  $g(n') < C$
  - There must exist a node  $n''$  on the frontier that is on the optimal path to  $n'$ .
  - Since  $g(n'') < g(n') < g(n)$ ,  $n''$  should have been expanded first!
- **UCS expands nodes in order of their optimal path cost.**

# Quiz 02: Uniform-cost search

Work out the order in which states are expanded and the path returned. Assume ties resolve in such a way that states with earlier alphabetical order are expanded first.



# Depth-first search

---



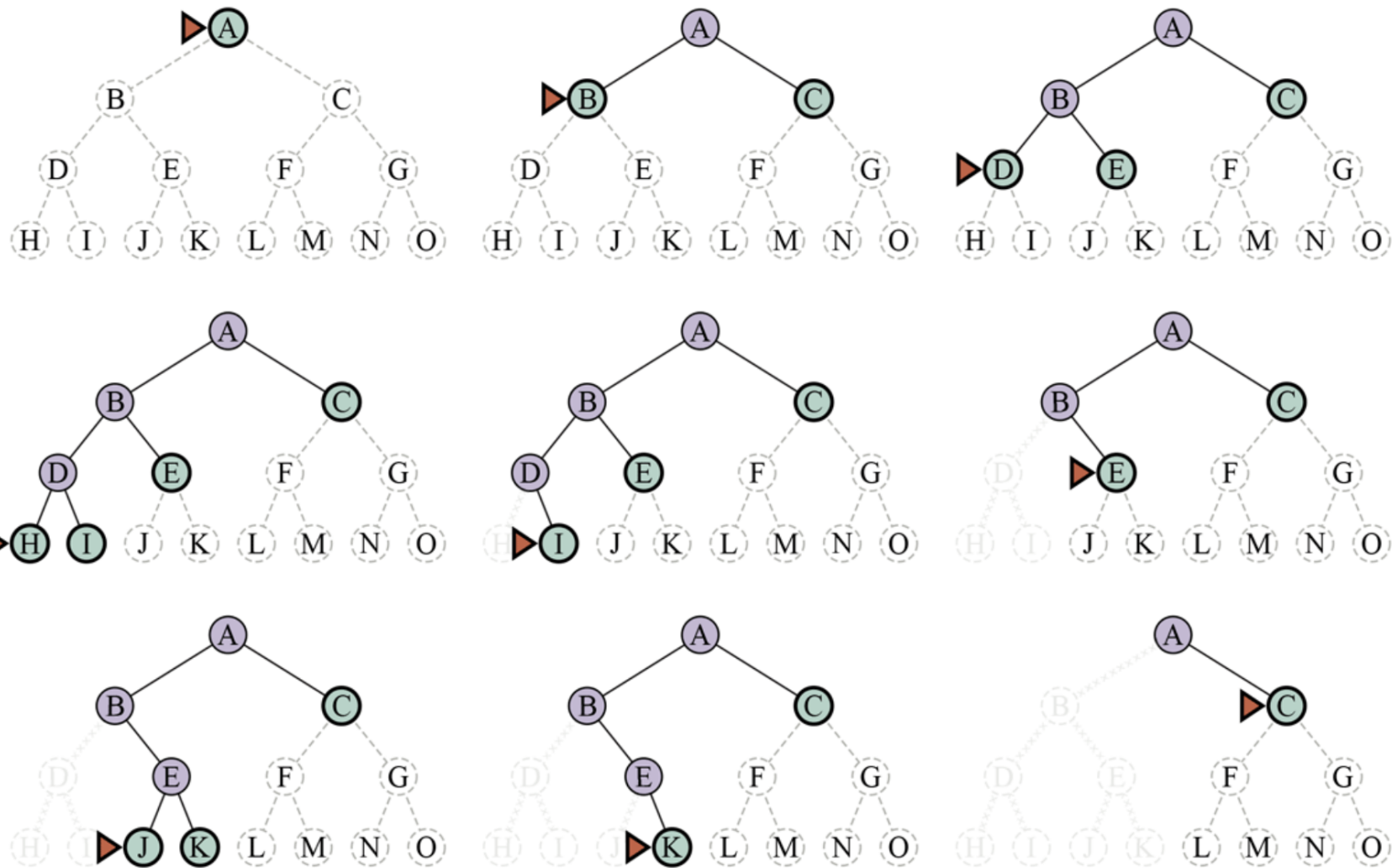
# Depth-cost search (DFS)

- **Idea:** Always expands the deepest node in the frontier first.
- It makes a call to BEST-FIRST-SEARCH with the following evaluation function.

$$f(n) = \text{the negative of the depth}$$

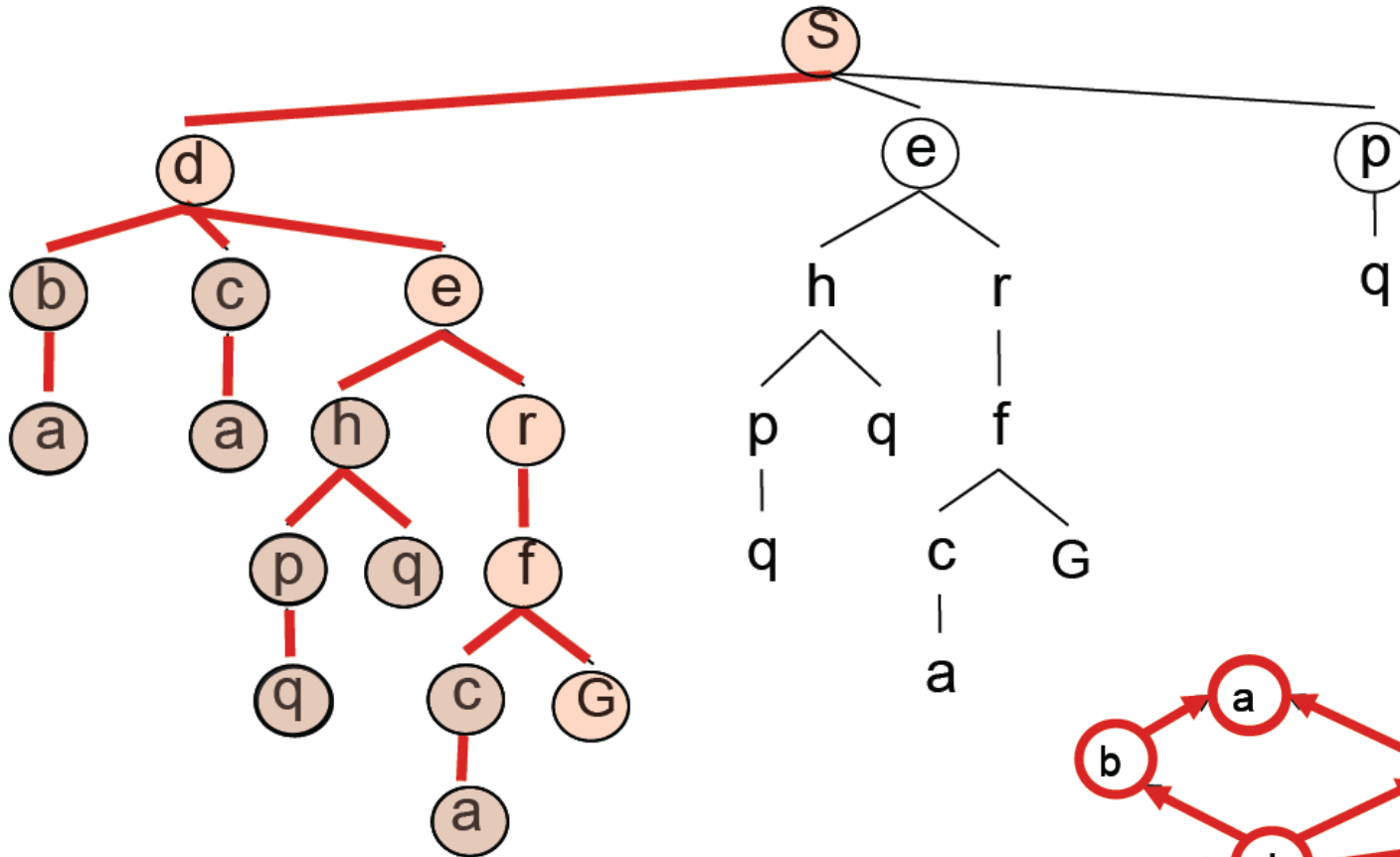
- DFS is usually implemented as a tree-search procedure that does not keep a table of reached states.





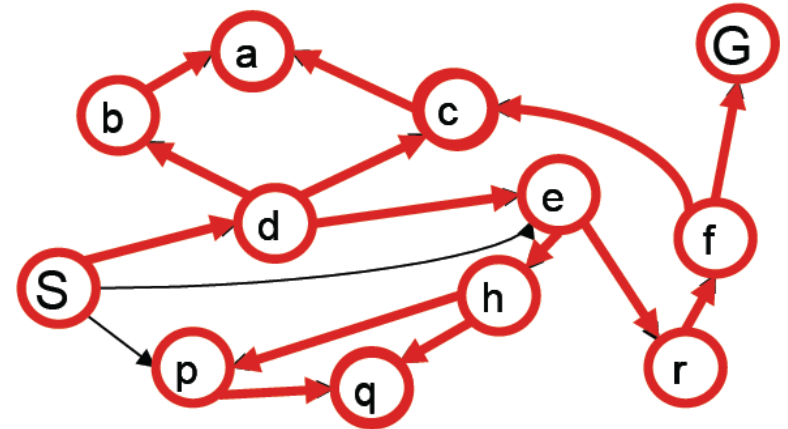
A dozen steps (left to right, top to bottom) in the progress of a DFS on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded

# Depth-first search: An example



Expansion order:

S, d, b, a, c, a, e, h, p, q, q, r, f, c, a, G

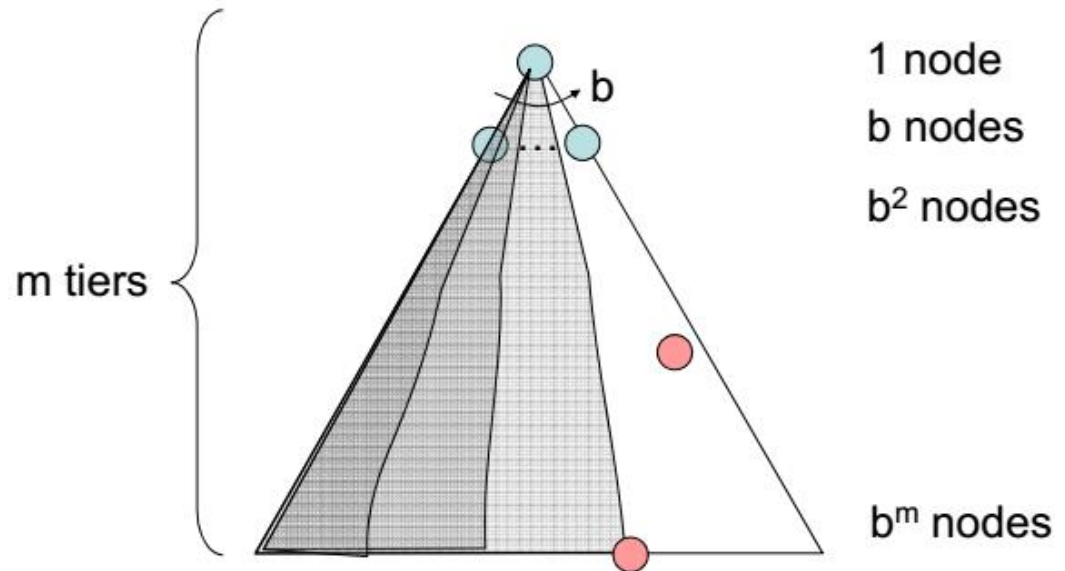


# Tree-search DFS: Completeness

- **Finite tree-shaped state spaces:** DFS is efficient and complete
- **Acyclic state spaces:** It can revisit the same state many times via different paths yet still explore the entire space.
- **Cyclic state spaces:** It can get stuck in an infinite loop.
  - A mechanism to check each new node for cycles is required.
- **Infinite state spaces:** DFS can get stuck going down an infinite path, even if there are no cycles.
  - E.g., the Knuth's 4 problem → keep applying the factorial operator

# An evaluation of tree-search DFS

- Completeness: ❌
- Optimality: ❌
- Time complexity:  $O(b^m)$
- Space complexity:  $O(bm)$



- Only siblings on path to root are tracked.

- Tree-search DFS loses the reached table, and the frontier is very small.
- Hence, it has been the basic workhorse of many AI tasks, like constraint satisfaction, propositional satisfiability, and logic programming.

# Depth-first search in use

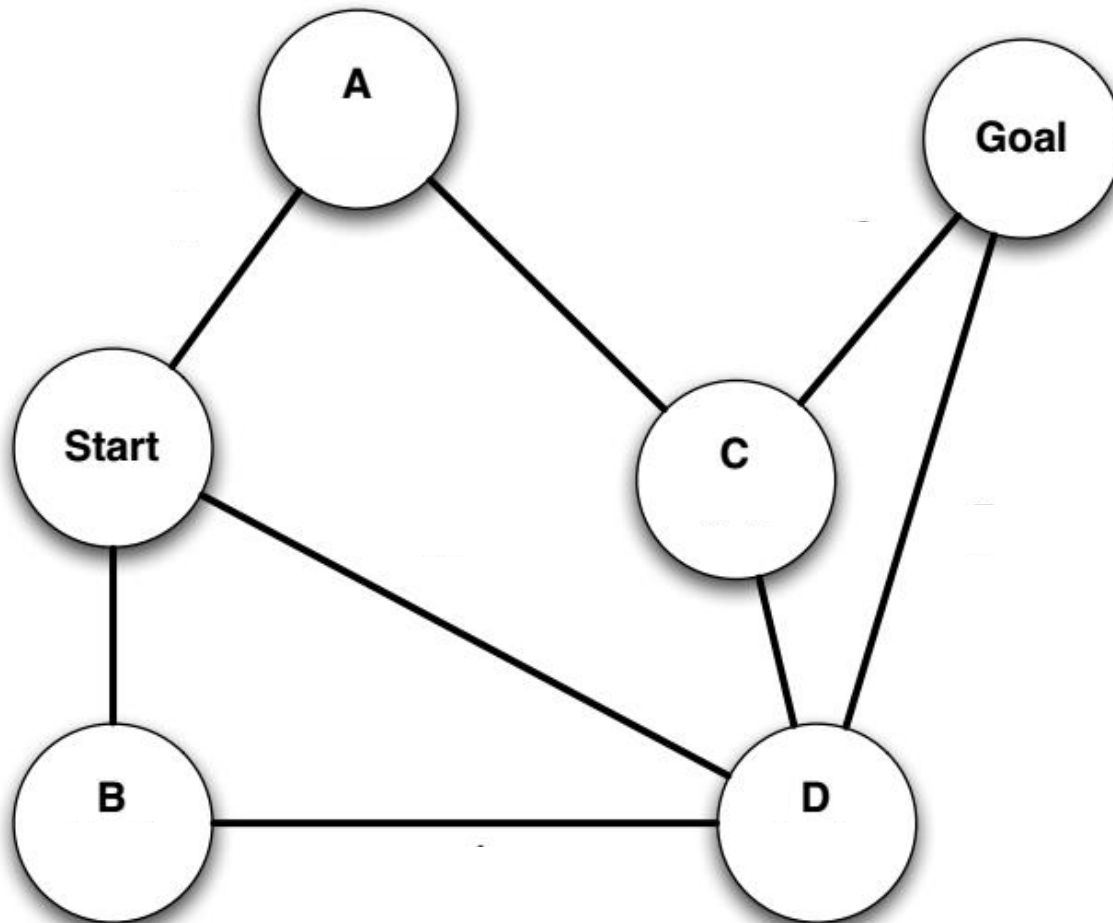
- We use **tree-search DFS** and **apply early goal test** to check whether a node is a solution as soon as it is generated.
- **Avoid cycles** by **following up the chain of parents** to see if the state at the end of the path has appeared earlier in the same path.

# Backtracking search



- Backtracking is critical to problems with large state descriptions (e.g., robotic assembly), due to the **significant memory saving over DFS**.
- Only **one successor is generated at a time** (rather than all successors).
- Each partially expanded node tracks which successor to generate next.
- Successors are generated by **modifying the current state description**, instead of allocating memory for a brand-new state.
- An efficient set data structure for the states allows checking for a cyclic path in time  $O(1)$  rather than  $O(m)$ .
- It **must be able to undo each action** to facilitate backtracking.

# Quiz 03: Depth-first search

Work out the order in which states are expanded and the path returned. Assume ties resolve in such a way that states with earlier alphabetical order are expanded first.

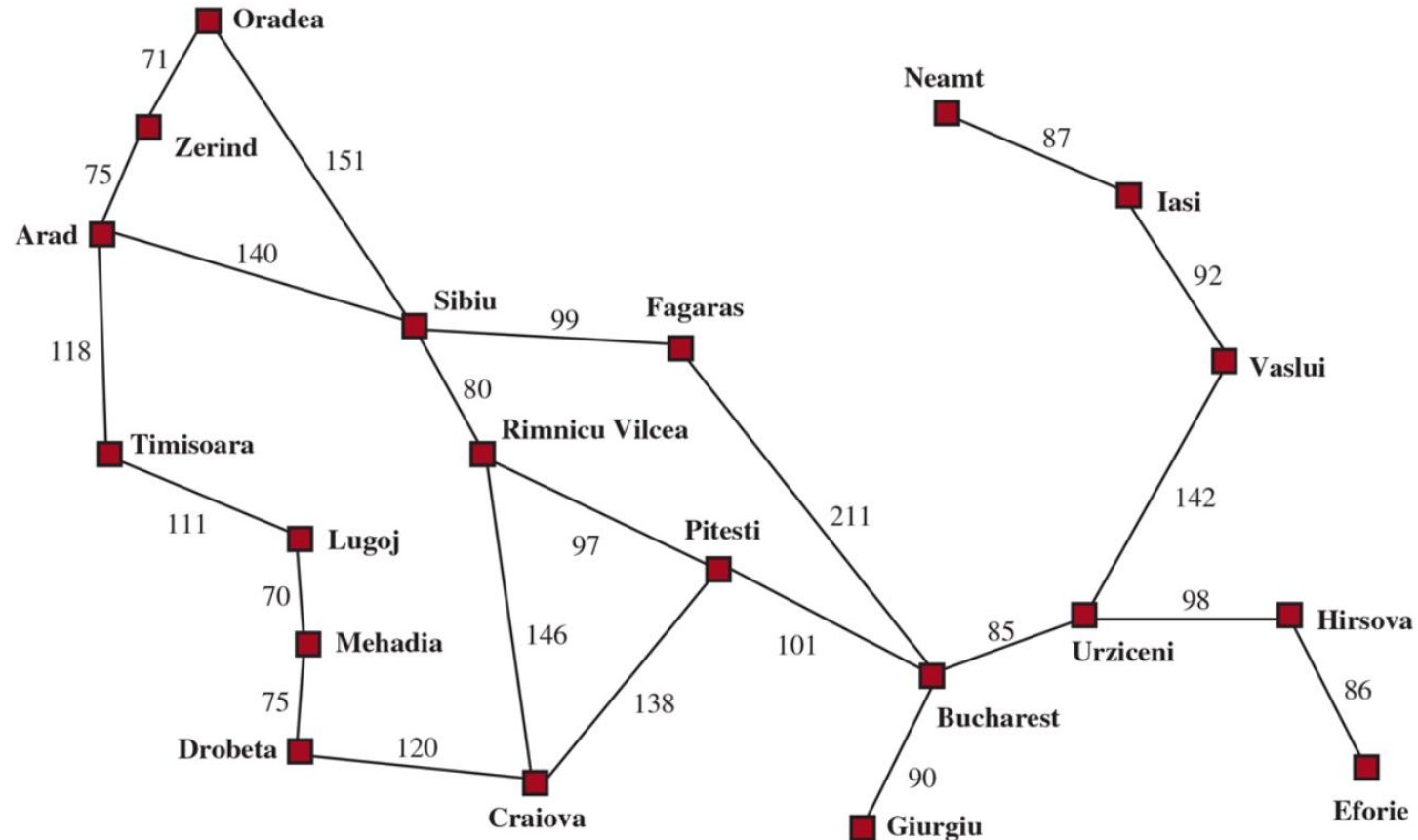


# Depth-limited search (DLS)

- **Idea:** We supply a **depth limit  $l$**  to **keep DFS** from wandering down an infinite path.
- All the **nodes at depth  $l$**  behave as if **they had no successors**.
- Depth limits can be based on knowledge of the problem.
  - E.g., diameter of state space
- However, it is typically unknown ahead of time in practice.
- **Completeness:**  if  $l < d$
- **Time complexity:**  $O(b^l)$
- **Optimality:** 
- **Space complexity:**  $O(bl)$



# Depth-limited search: An example



There are 20 cities in the Romania map  $\rightarrow l = 19$

Still, any city is reached from other cities in at most 9 steps  $\rightarrow l = 9$  is better.

```

function DEPTH-LIMITED-SEARCH(problem, l)
    returns a node or failure or cutoff
    frontier  $\leftarrow$  a LIFO queue (stack), with NODE(problem.INITIAL) as an element
    result  $\leftarrow$  failure
    while not IS-EMPTY(frontier) do
        node  $\leftarrow$  POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        if DEPTH(node) > l then
            result  $\leftarrow$  cutoff
        else if not IS-CYCLE(node) do
            add child to frontier
    return result

```

DLS returns one of three different values: either a solution node; or failure, when it has exhausted all nodes and proved there is no solution at any depth; or cutoff, to mean there might be a solution at a deeper depth than  $l$ . This is a tree-search algorithm that does not keep track of reached states, and thus uses much less memory, but runs the risk of visiting the same state multiple times on different paths. If the IS-CYCLE check does not check all cycles, the algorithm may get caught in a loop.

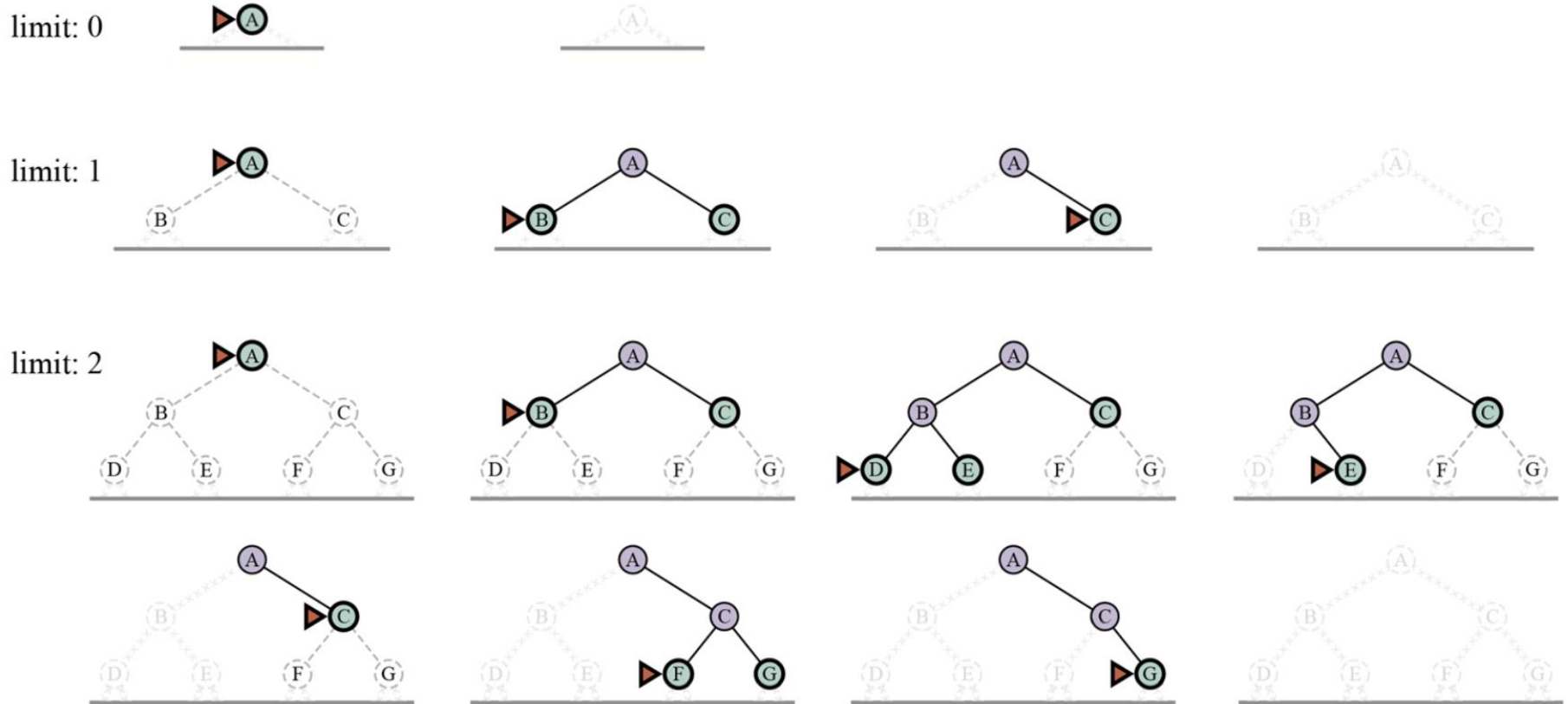
# Iterative deepening search (IDS)

- **Idea:** Gradually increase the limit until a goal is found or a failure value is returned from DLS.

```
function ITERATIVE-DEEPENING-SEARCH(problem)  
    returns a solution node, or failure  
  
    for depth = 0 to  $\infty$  do  
        result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)  
        if result  $\neq$  cutoff then return result
```

- It is preferred when the state space exceeds the available memory capacity, and the depth of the solution is unknown.

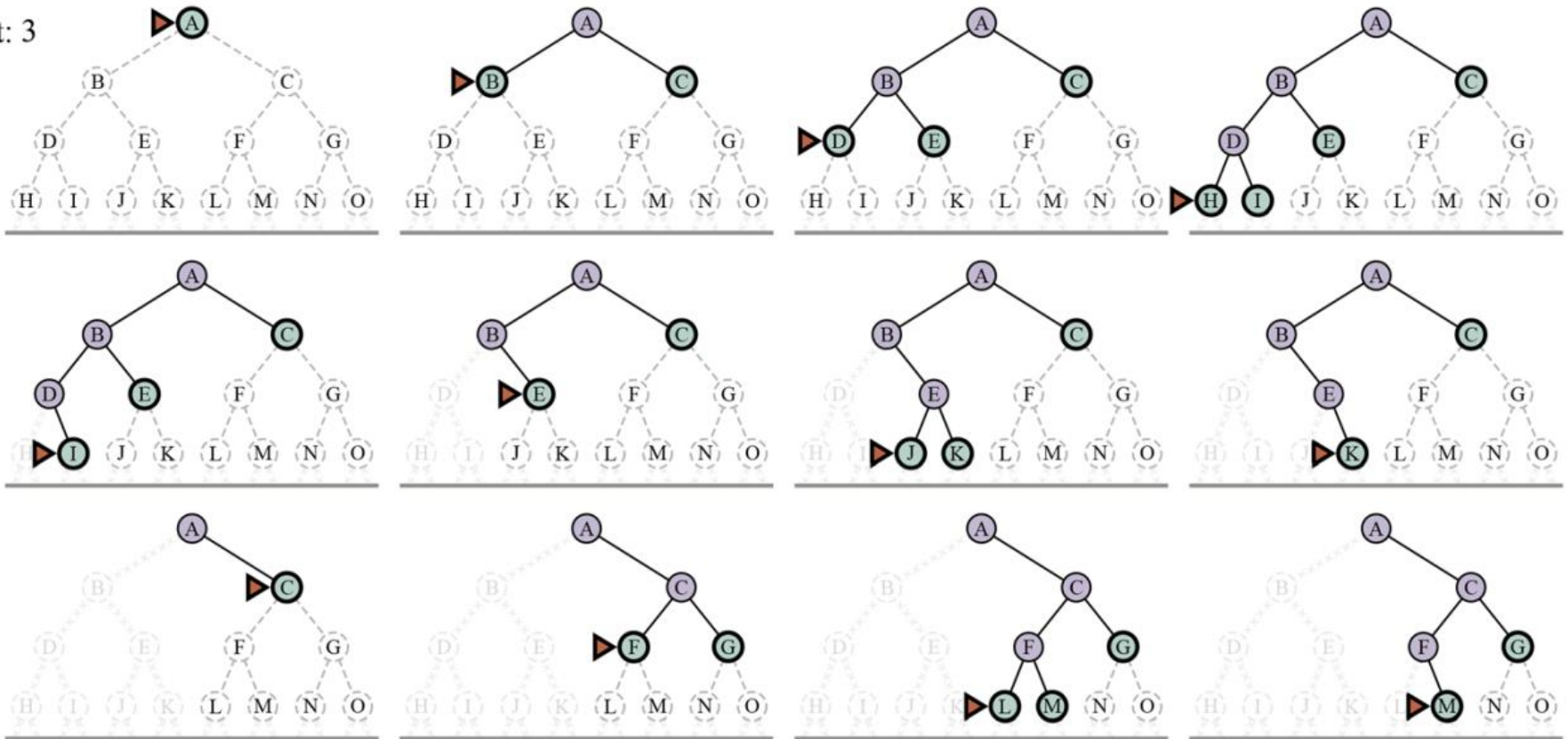
# Iterative deepening search: An example



Four iterations of IDS for goal **M** on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.



# Iterative deepening search: An example

limit: 3



Four iterations of IDS for goal **M** on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

# An evaluation of IDS

- IDS leverages the benefits of both DFS and BFS.
- **Completeness:**  on finite acyclic state spaces, or on any finite state space when we check nodes for cycles all the way up the path.
- **Optimality:**  for problems where all actions have the same cost.
- **Space complexity:**  $O(bd)$  when there is a solution, or  $O(bm)$  on finite state spaces with no solution.
- **Time complexity:**  $O(b^d)$  if there is a solution, or  $O(b^m)$  if there is none.

# An evaluation of IDS

- IDS may appear inefficient as it regenerates states near the top of the search tree multiple times.
- However, for state spaces where most nodes are located at the bottom level, **this inefficiency is less significant.**

- The total number of nodes generated in the worst case is:

$$N(IDS) = (d)b^1 + (d-1)b^2 + (d-2)b^3 + \dots + b^d$$

which gives a time complexity of  $O(b^d)$ .

- For example, with  $b = 10$  and  $d = 5$ 
  - $N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$
  - $N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$ .

# Bidirectional search

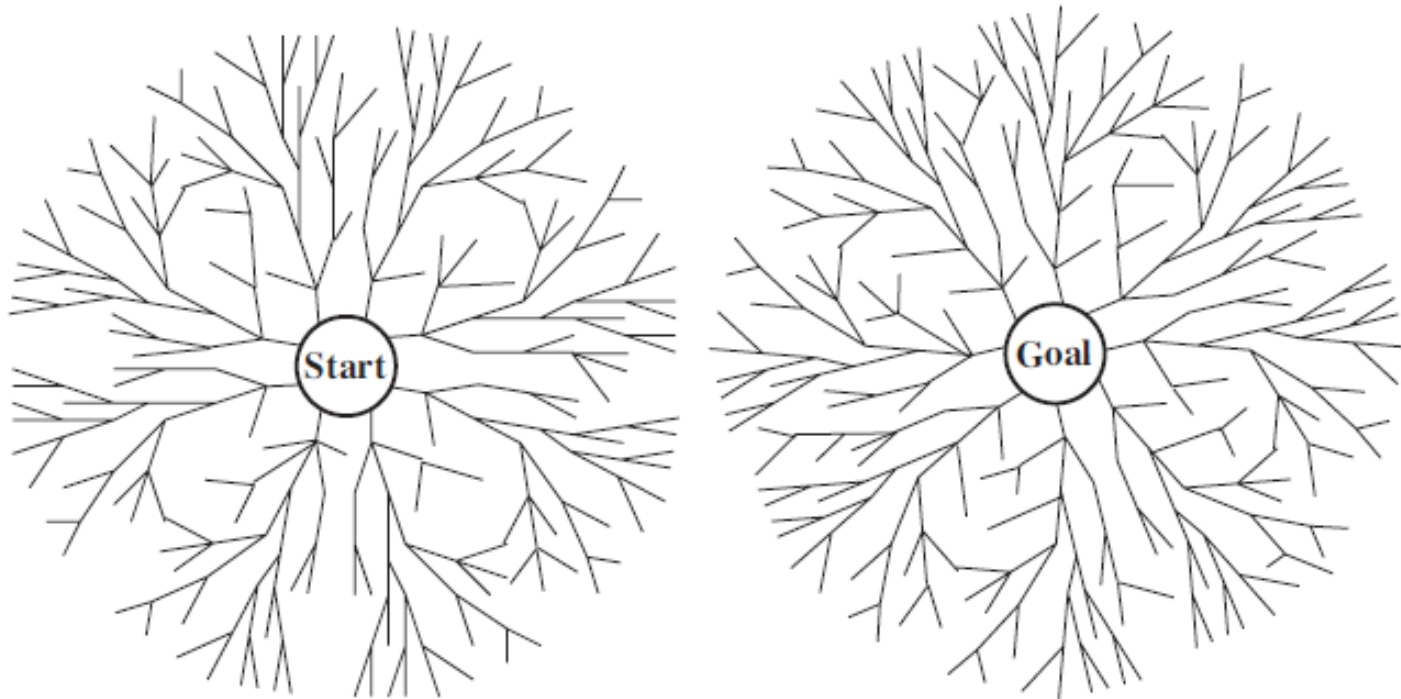
---





# Bidirectional search

- **Idea:** Simultaneously search forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet.



# Bidirectional search

- **Motivation:**  $b^{1/2} + b^{1/2}$  is that is much less than  $b^d$ .
  - E.g., it is 50,000 times less when  $b = d = 10$ .
- It requires to **keep track of two frontiers and two tables of reached states**.
- Furthermore, it must be able to **reason backwards**.
  - If state  $s'$  is a successor of  $s$  in the forward direction, we need to know that is  $s$  a successor of  $s'$  in the backward direction.
- We have **a solution** when **the two frontiers collide**.
- There are **many different versions of bidirectional search** according to the choice of the evaluation function.

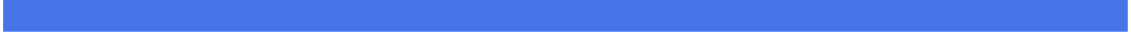
# Uninformed search algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes <sup>1</sup>	Yes <sup>1,2</sup>	No	No	Yes <sup>1</sup>	Yes <sup>1,4</sup>
Optimal cost?	Yes <sup>3</sup>	Yes	No	No	Yes <sup>3</sup>	Yes <sup>3,4</sup>
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Evaluation of search algorithms.  $b$  is the branching factor;  $m$  is the maximum depth of the search tree;  $d$  is the depth of the shallowest solution or is  $m$  when there is no solution;  $\ell$  is the depth limit. Superscript caveats are as follows: <sup>1</sup> complete if  $b$  is finite, and the state space either has a solution or is finite; <sup>2</sup> complete if all action costs are  $\geq \epsilon > 0$ ; <sup>3</sup> cost-optimal if action costs are all identical; <sup>4</sup> if both directions are breadth-first or uniform-cost.



# Informed (Heuristic) search



# Greedy best-first search

---



# Informed search strategies

- An **informed search algorithm** uses **domain-specific hints** about the **location of goals** to find solutions **more efficiently**.
- The hints come in the form of a **heuristic function**.

$h(n)$  = *estimated cost of the cheapest path  
from the state at node  $n$  to a goal state*



While relying solely on a compass may not lead to precise destination arrival, it can provide a general estimation of the direction you should head.

# Heuristics: An explanation

- A **heuristic** is any **practical approach** to problem solving sufficient for reaching an **immediate goal** where an optimal solution is usually impossible.
- It is not guaranteed to be optimal, perfect, logical, or rational.
- It can speed up the process of finding a satisfactory solution, while easing the cognitive load of decision-making

**Representativeness heuristic:** estimate the likelihood of an event by comparing it to a prototype already exists in mind.



Who is warm and caring with a great love of children?

# Heuristic function

- The heuristic function  $h(n)$  takes a node as input and it depends only on the state at that node.
  - The path cost  $g(n)$ , on the other hand, is accumulative from the root.
- Arbitrary, nonnegative, problem-specific
- If node  $n$  represents the goal state, it must be  $h(n) = 0$ .



# Greedy best-first search (GBFS)

- **Idea:** Expand first the node with the lowest  $h(n)$  value—the node that appears to be closest to the goal.
- GBFS calls BEST-FIRST-SEARCH with the evaluation function

$$f(n) = h(n)$$

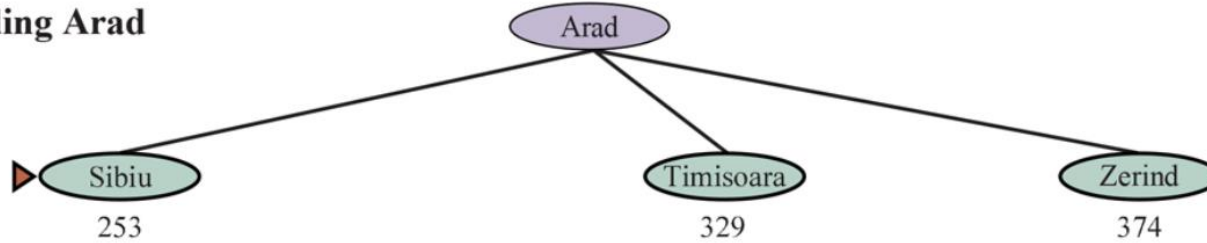
In the touring holiday example, the straight-line distances to Bucharest are heuristic values.

<b>Arad</b>	366	<b>Mehadia</b>	241
<b>Bucharest</b>	0	<b>Neamt</b>	234
<b>Craiova</b>	160	<b>Oradea</b>	380
<b>Drobeta</b>	242	<b>Pitesti</b>	100
<b>Eforie</b>	161	<b>Rimnicu Vilcea</b>	193
<b>Fagaras</b>	176	<b>Sibiu</b>	253
<b>Giurgiu</b>	77	<b>Timisoara</b>	329
<b>Hirsova</b>	151	<b>Urziceni</b>	80
<b>Iasi</b>	226	<b>Vaslui</b>	199
<b>Lugoj</b>	244	<b>Zerind</b>	374

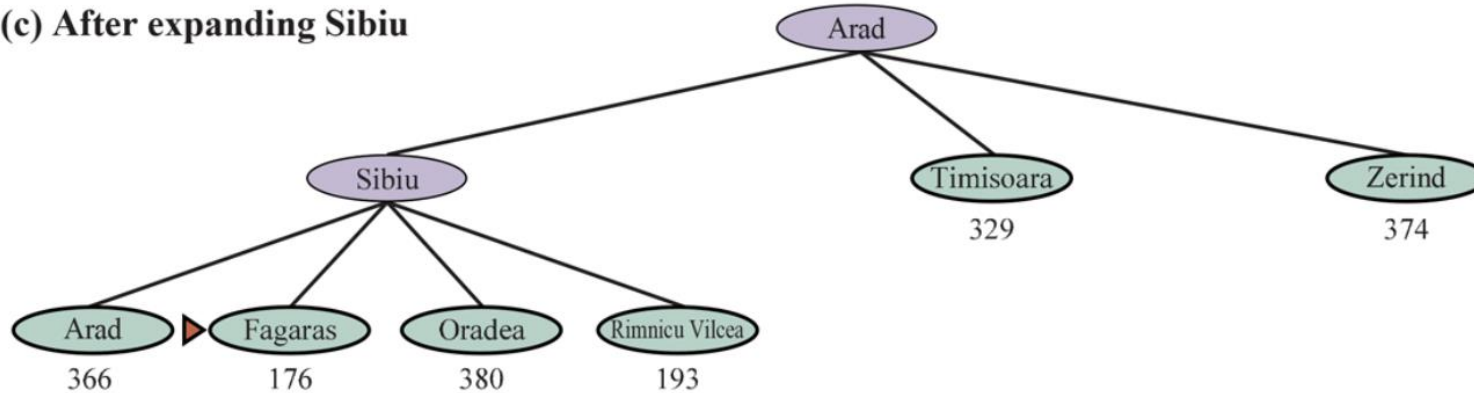
**(a) The initial state**



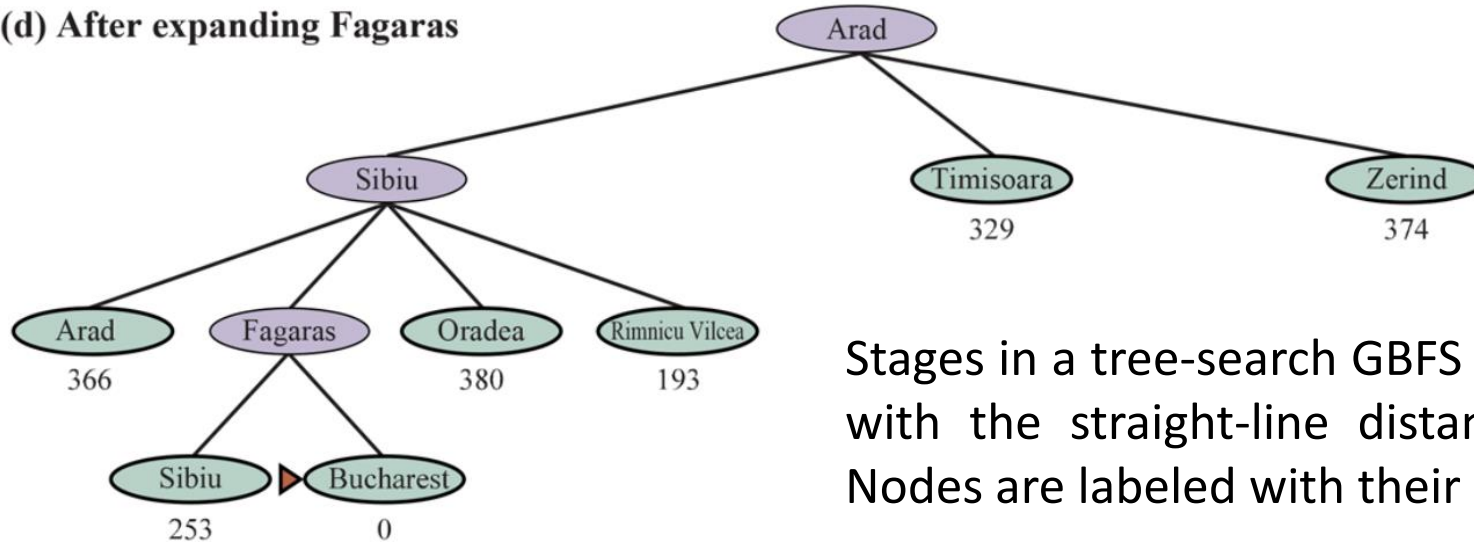
**(b) After expanding Arad**



**(c) After expanding Sibiu**






**(d) After expanding Fagaras**



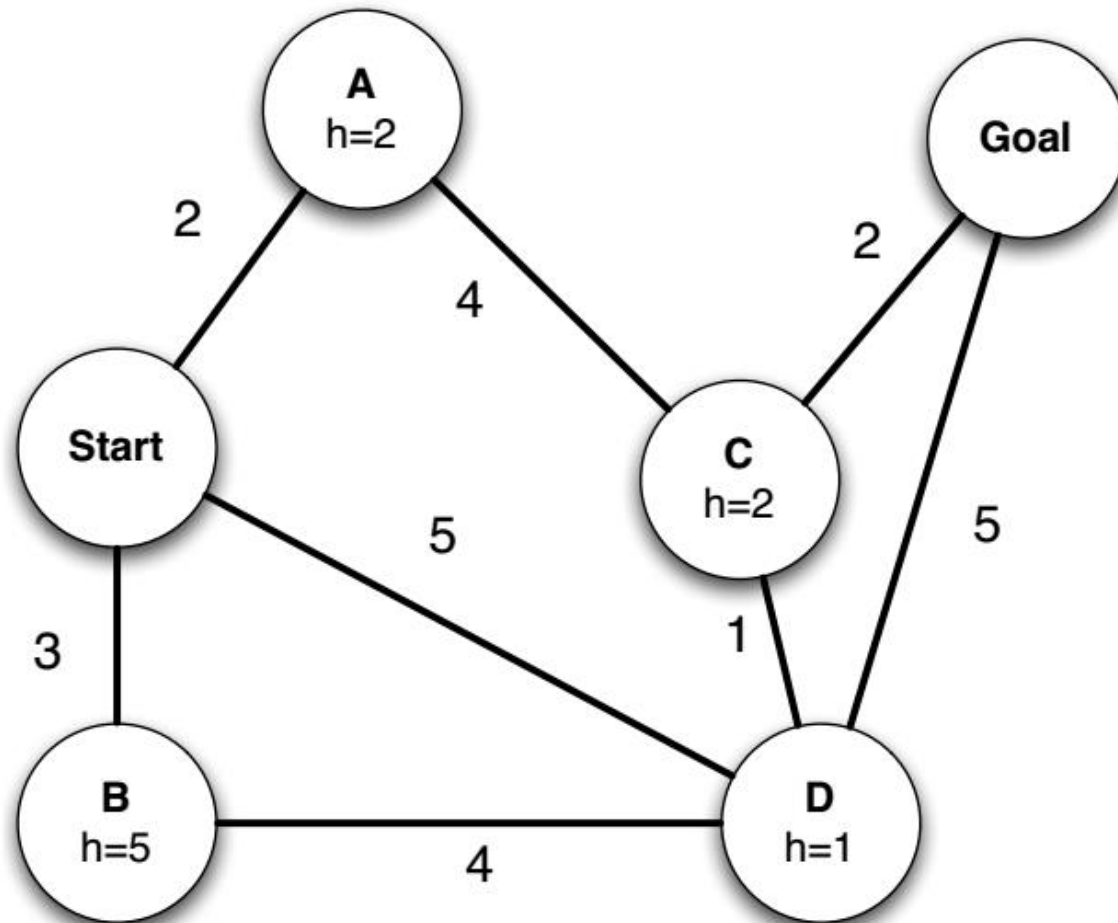
Stages in a tree-search GBFS for Bucharest with the straight-line distance heuristic. Nodes are labeled with their h-values. 66

# An evaluation of GBFS

- Consider the graph-search greedy best-first search algorithm
- Completeness:  in finite state spaces,  in infinite ones
- Optimality: 
- Time complexity:  $O(b^m)$ , reduced substantially with a good heuristic, on certain problems reaching  $O(bm)$ .
- Space complexity:  $O(b^m)$ , all nodes are kept in memory.

# Quiz 04: Greedy best-first search

Work out the order in which states are expanded and the path returned. Assume ties resolve in such a way that states with earlier alphabetical order are expanded first.



# A\*search

---



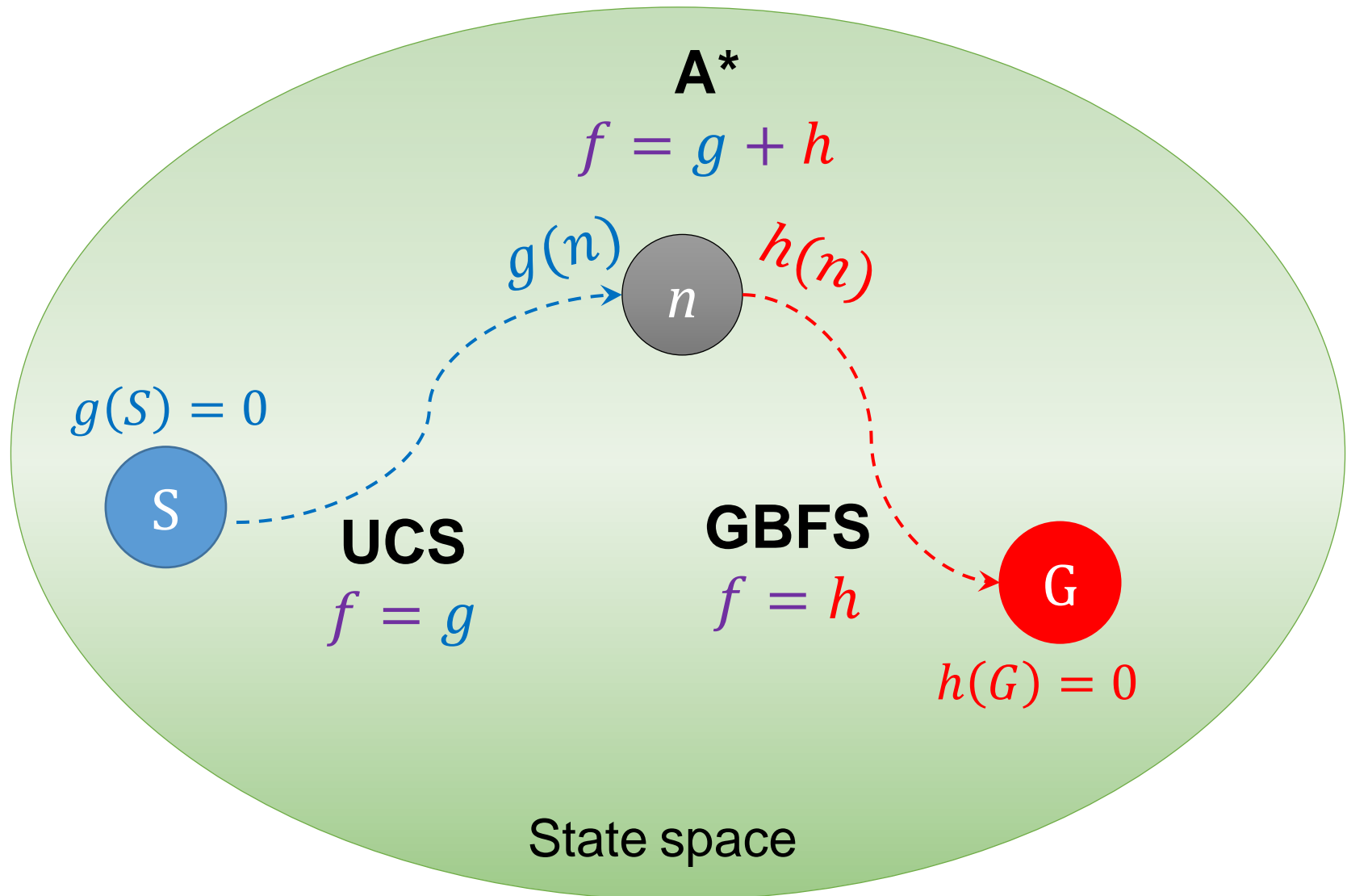
# A\* search

- This is the **most common informed search** algorithm.
- A\* calls BEST-FIRST-SEARCH with the evaluation function

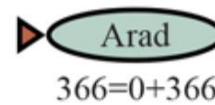
$$f(n) = h(n) + g(n)$$

- $g(n)$  is the path cost from the initial state to node  $n$  and  $h(n)$  is the estimated cost of the shortest path from  $n$  to a goal state.
- Thus,  $f(n)$  expresses the estimated cost of the best path that continues from  $n$  to a goal.

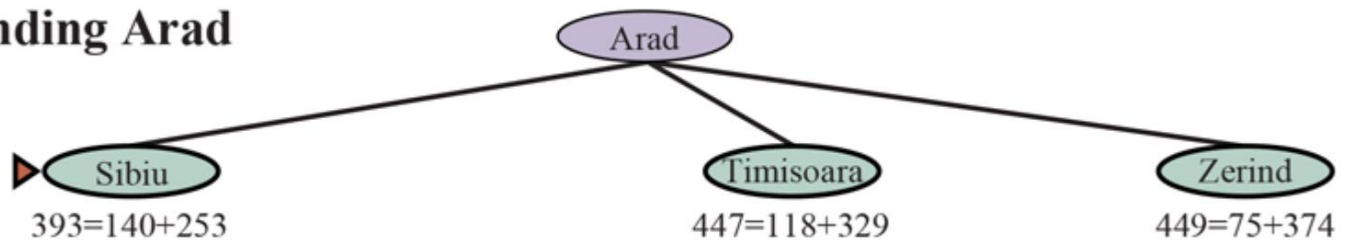
# A\* search vs. GBFS vs. UCS



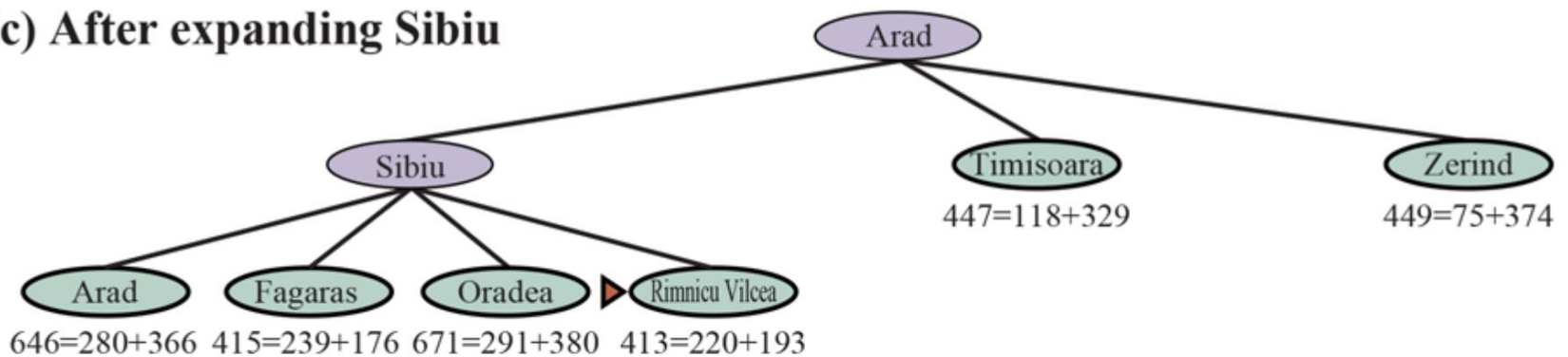
**(a) The initial state**



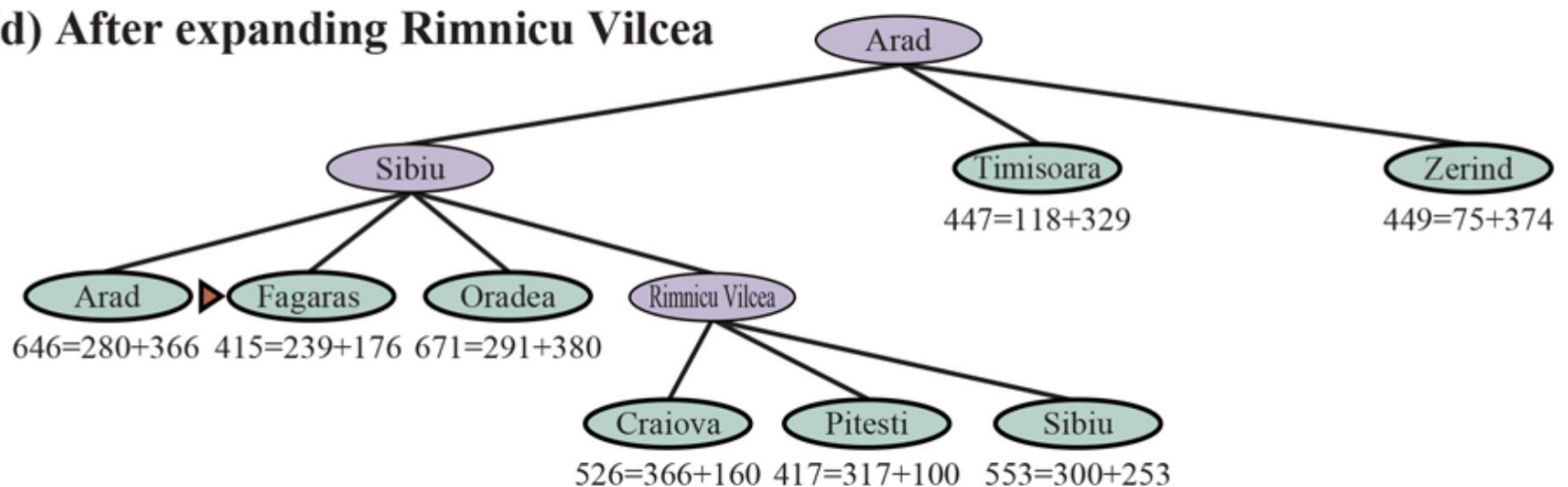
**(b) After expanding Arad**



**(c) After expanding Sibiu**

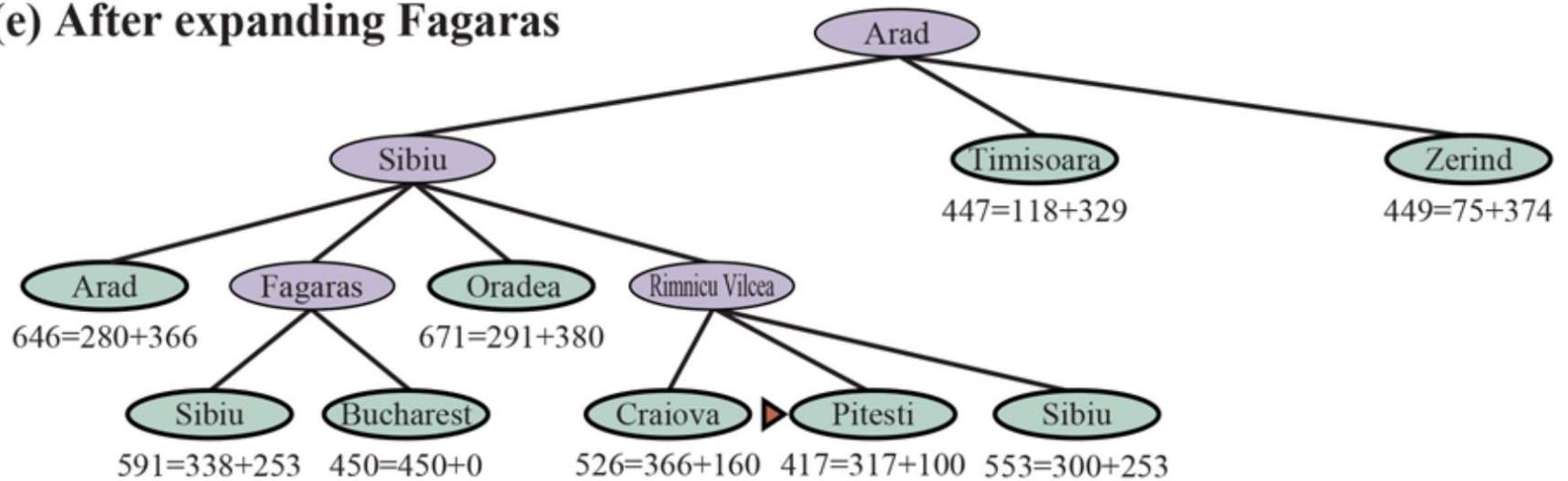


**(d) After expanding Rimnicu Vilcea**

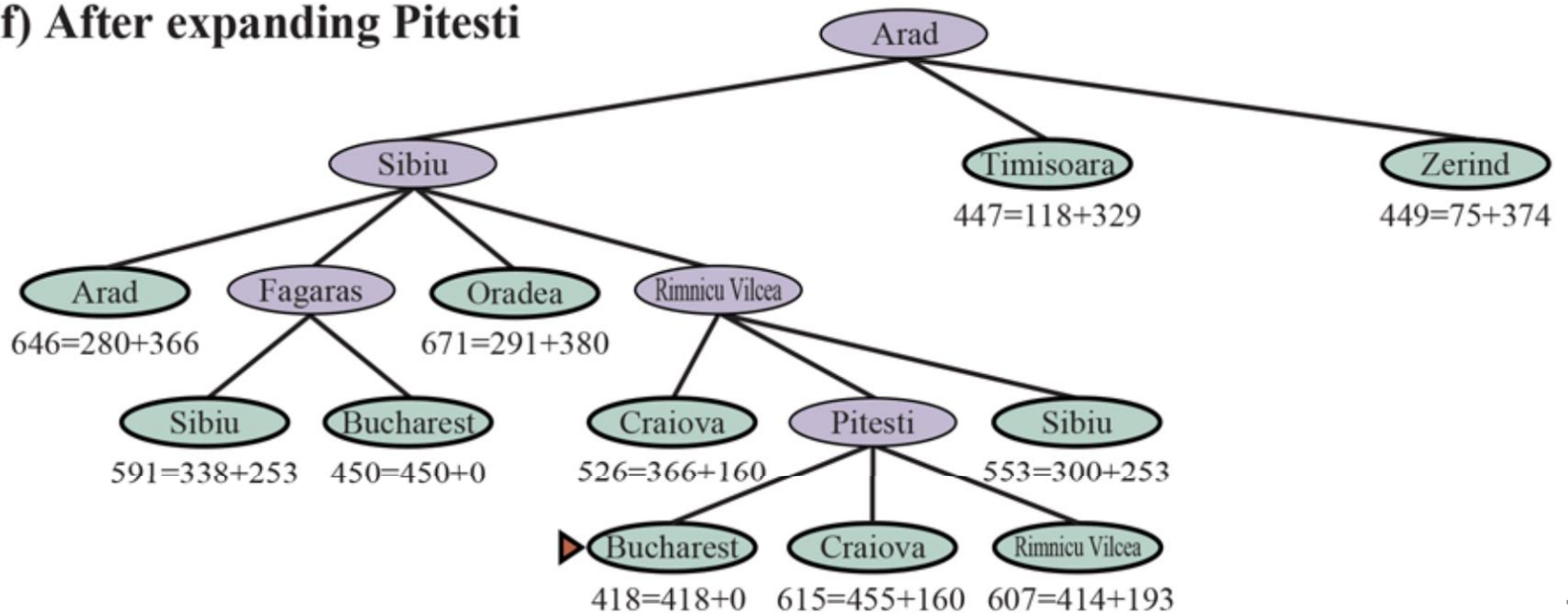




**(e) After expanding Fagaras**

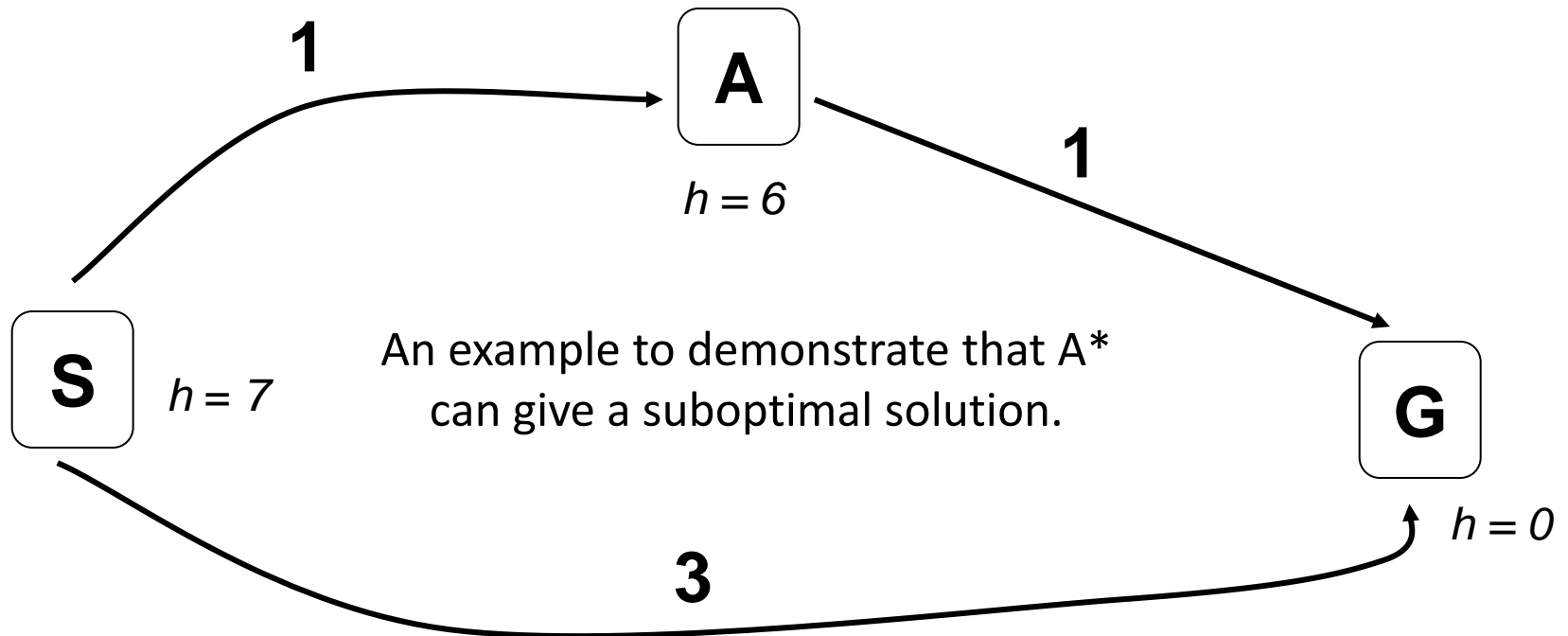


**(f) After expanding Pitesti**



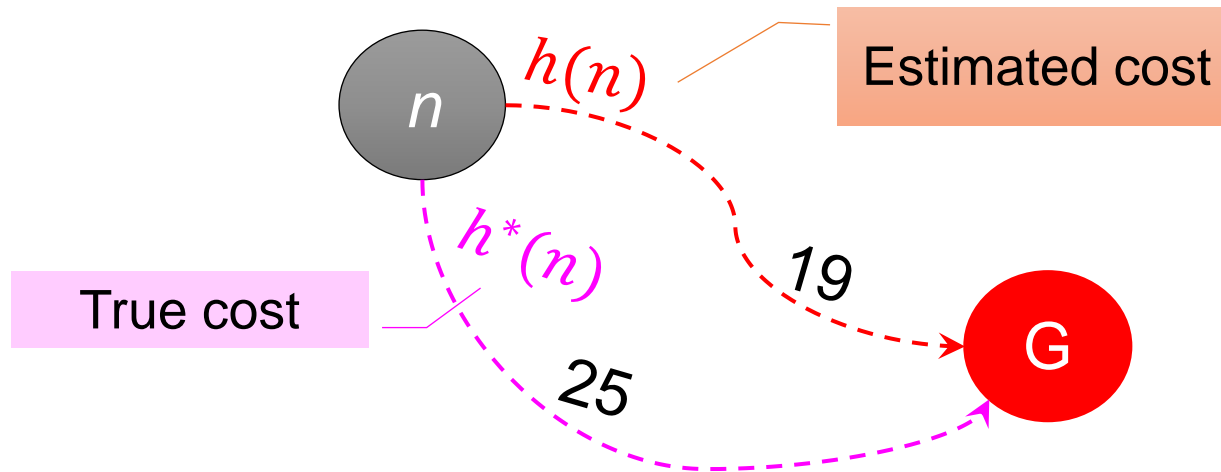
# A\* search: Completeness and Optimality

- Assume that all action costs are at least  $\epsilon > 0$ , and the state space either has a solution or is finite.
- A\* is **complete**, yet its **optimality depends** on the properties of the heuristic.



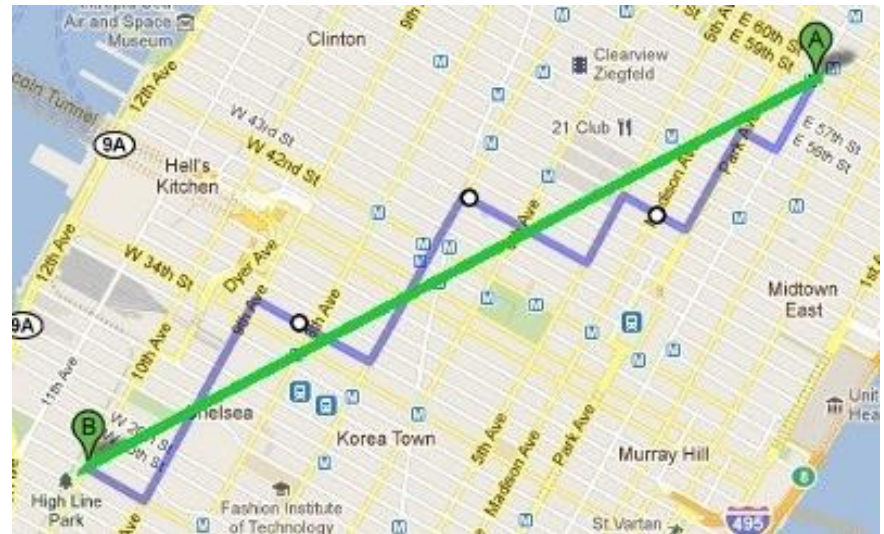
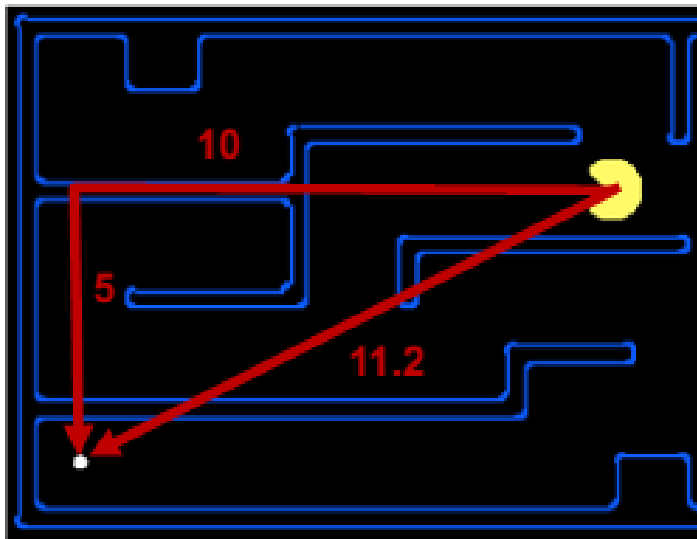
# Heuristic function: Admissibility

- An **admissible heuristic** is the one that **never overestimates the optimal cost to reach a goal**.
- Let  $h^*(n)$  be the **true cost** to reach the goal state from  $n$ .
- **$h(n) \leq h^*(n)$**  for every node  $n$ .



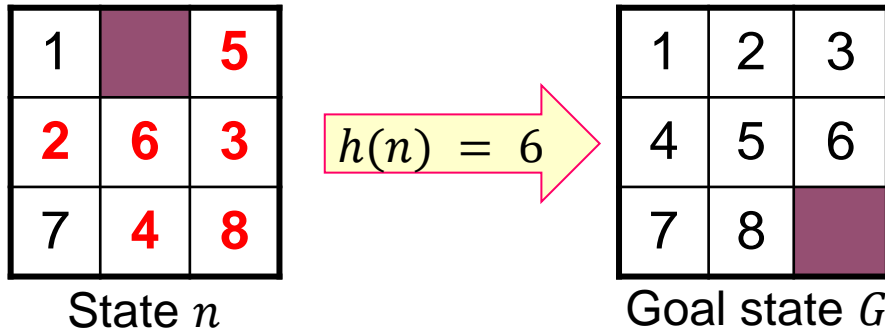
# Admissible heuristic: Examples

- The straight-line distance  $h_{SLD}$  for route-finding problems.

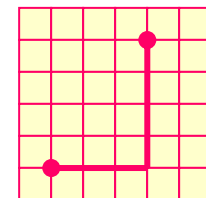
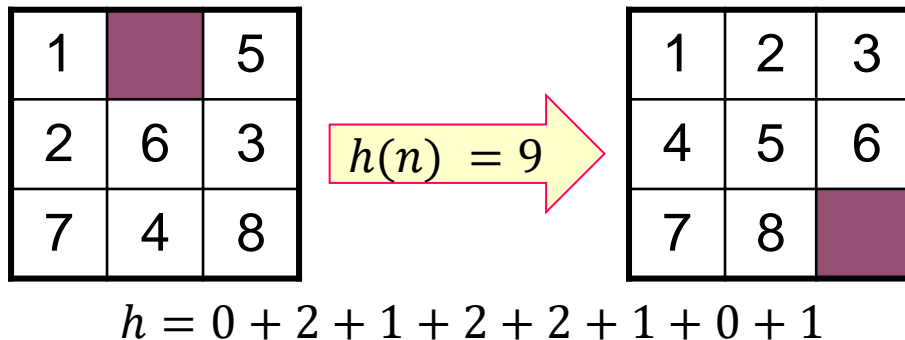


# Admissible heuristic: Examples

- $h(n)$  = number of misplaced numbered tiles



- $h(n)$  = sum of the (Manhattan) distance of every numbered tile to its goal position

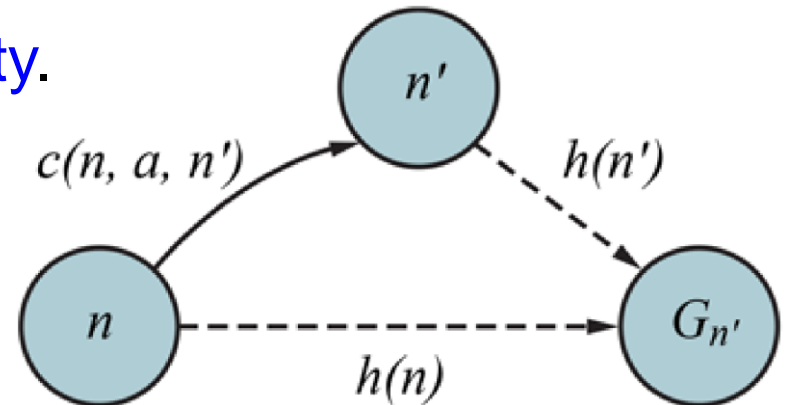


$$d = dx + dy$$

# Heuristic function: Consistency

- A **consistent heuristic**  $h(n)$  satisfies the following inequality
$$h(n) \leq c(n, a, n') + h(n')$$
  - for every node  $n$  and successor  $n'$  of  $n$  generated by any action  $a$ .

- This is a form of **triangle inequality**.



- Every consistent heuristic is **admissible** (but not vice versa).

# Consistent heuristic

- A **consistent heuristic** ensures that the initial encounter with a state occurs on an optimal path.
- This eliminates the need to re-add a state to the frontier or modify an entry in the reached set.
- **Graph-search A\*** secures optimality with consistent heuristic.
- An **inconsistent heuristic** can lead to **redundant paths to the same state**.
- This results in multiple nodes with lower path costs in the frontier, causing inefficiencies in both time and space.

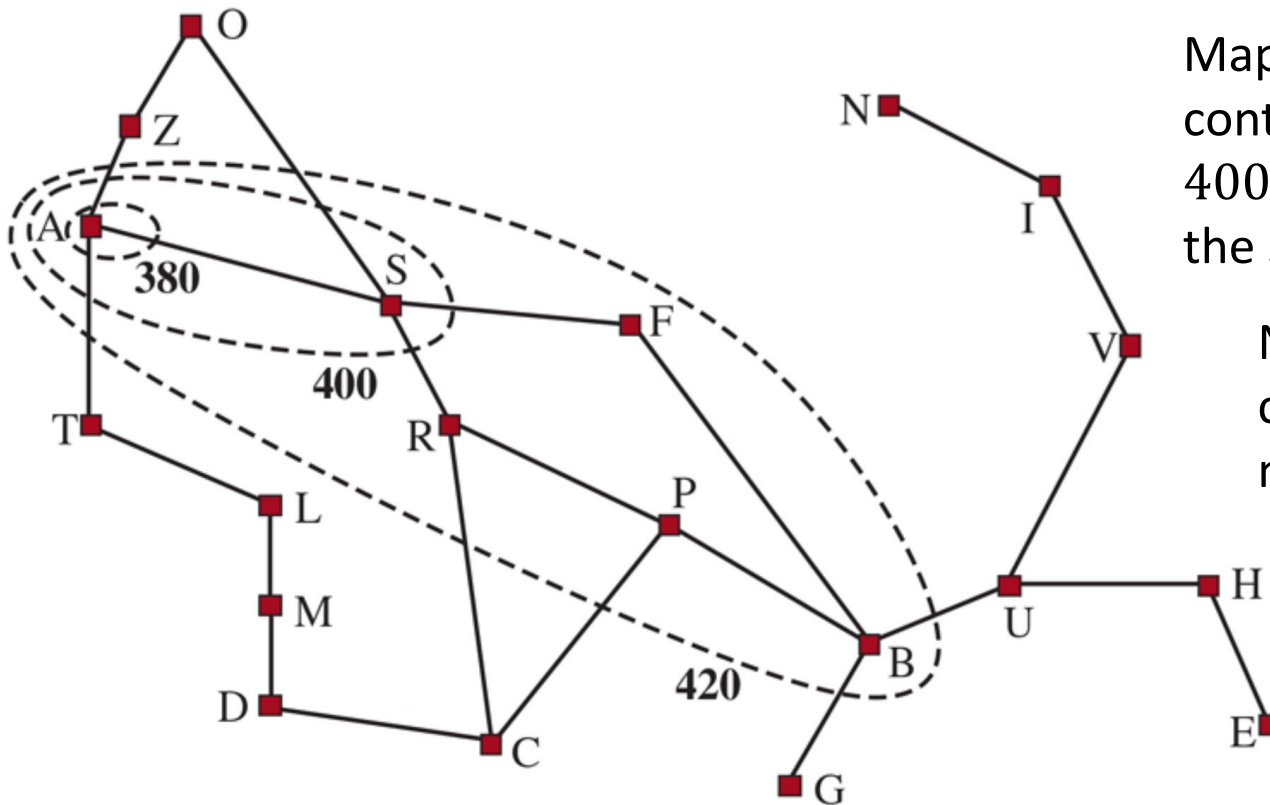
# Admissible heuristic

- Tree-search  $A^*$  assures optimality with admissible heuristic.
- $A^*$  may or may not be optimal with an inadmissible heuristic.
- It still returns cost-optimal solution in the below two cases.
  1. All nodes on a cost-optimal path have admissible heuristic values, regardless of values for states outside the path.
  2. The optimal solution has cost  $C^*$ , the second-best has cost, and  $h(n)$  consistently overestimates costs by no more than  $C_2 - C^*$ .



# Search contours

- Search contours are concentric bands in which contour  $i$  has all nodes with  $f = f_i$  where  $f_i < f_{i+1}$ .

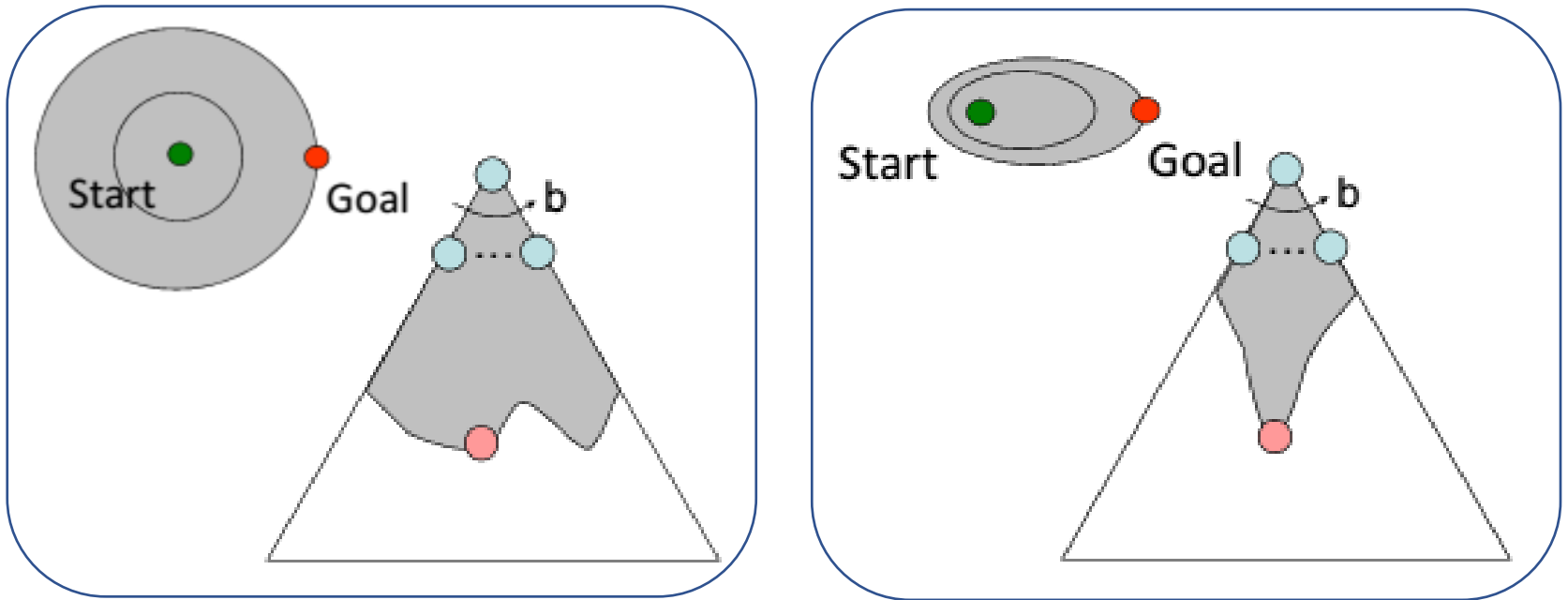


Map of Romania showing contours at  $f = 380$ ,  $f = 400$ , and  $f = 420$ , with the start state Arad.

Nodes inside a given contour have costs at most the contour value.

# A\* contours vs. UCS contours

- The bands of UCS are “circular” around the start state.



- The bands of A\*, with more accurate heuristics, stretch toward the goal state and become more narrowly focused to the optimal path.

# An evaluation of $A^*$

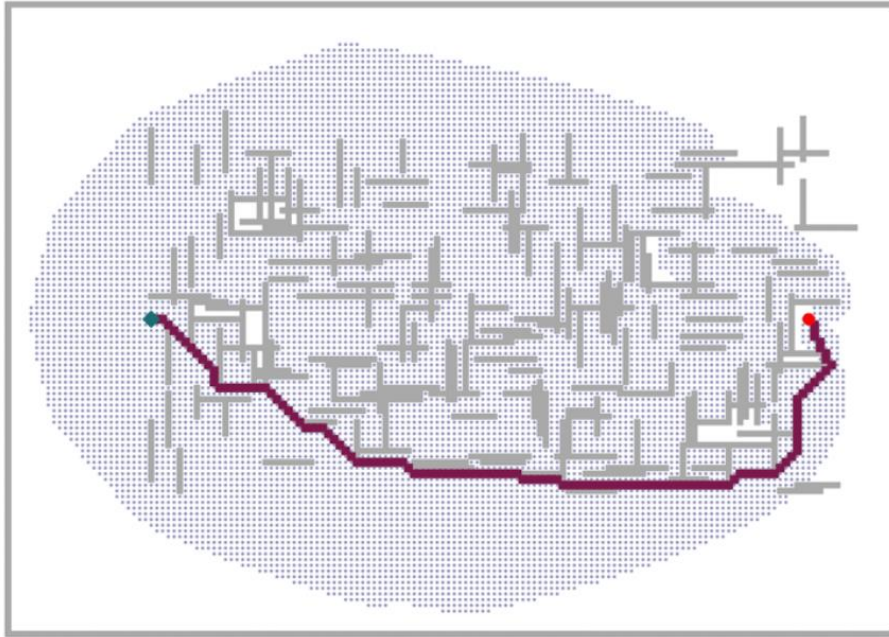
- $A^*$  surely expands nodes reached from the initial state on a path where every node on the path has  $f(n) < C^*$ .
- It might then expand some of the nodes right on the “goal contour” (where  $f(n) = C^*$ ) before selecting a goal node.
- $A^*$  expands no nodes with  $f(n) > C^*$ .
- $A^*$  search is complete, cost-optimal, and optimally efficient among all such algorithms.
  - No other optimal algorithm is guaranteed to expand fewer nodes.

# Weighted A\* search

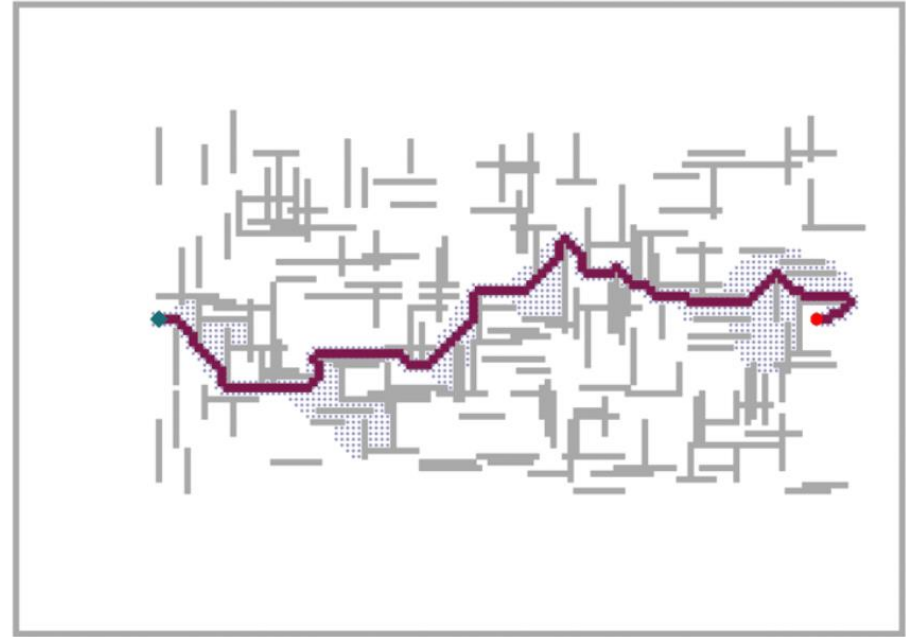
- **Detour index** (in road engineering): A multiplier that adjusts the straight-line distance for the typical curvature of roads.
  - E.g., a value of 1.3 means that if two cities are 10 miles apart in straight-line distance, the best path between them is 13 miles.
  - For most localities, the detour index ranges between 1.2 and 1.6.
- **Weighted A\* search** weighs the **heuristic value more heavily**, giving us the evaluation function

$$f(n) = g(n) + W \times h(n) \quad (1 < W < \infty)$$

# Weighted $A^*$ : An example



(a)



(b)

Two searches on the same grid: (a) an A\* search and (b) a weighted A\* search with weight  $W = 2$ . The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A\* explores 7 times fewer states and finds a path that is 5% more costly.

# Memory bounded search

- A\* usually **runs out of space** before it runs out of time.
- **Idea:** Try something like DFS, but not forget everything about the branches we have partially explored.

Beam search

Iterative-deepening A\*  
search (IDA\*)

Recursive best-first  
search (RBFS)

Memory-bounded A\*  
(MA\*)

Simplified Memory-  
bounded A\* (SMA\*)

# Quiz 02: Graph-search A\*

Work out the order in which states are expanded and the path returned. Assume ties resolve in such a way that states with earlier alphabetical order are expanded first.

