COSC76/276 Artificial Intelligence
Fall 2022
Informed Search

Soroush Vosoughi

Computer Science

Dartmouth College

Soroush@Dartmouth.edu

Year 5783!

Survey

- Pace:
  - Shorter recaps
  - Questions in class
- Amount of work
- Short Assignments
- Office hours
- Grad vs Undergrad
- Blackboard recording
- Complete assignment schedule on Canvas

Reminders

- PA-1 due Sep 28th at 11:59pm ET
- SA-2 will be released today and it will due Oct 1st at 11:59pm ET
- Python OO refresher today during X-hours

Basic Search: Recap

- Modeling a real-world problem as a search problem to abstract away real-world details
  - State and action space, transition function
  - Planning is all “in simulation”
  - Model is a simplification of the world
- Search tree built on the fly to find a solution
  - Does not keep track of expanded nodes
- Variety of uninformed search (tree-search version) with different time and space complexity
  - BFS: expands shallowest node first
  - DFS: expands deepest node first
  - Limited DFS: DFS up to a given depth
  - Iterative DFS: run limited DFS with increasing depth limit until solution found

Keeping Track of History

- Graph-search (memoizing)
  – BFS
  – DFS
    - Path-checking DFS

Graph Search (memoizing)

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

- Tree-search does not keep track of the states already visited
- Graph-search keeps track of the states already visited does: memoizing – i.e., keeping track of the states already visited

Graph Search vs Tree Search

Graph Search vs Tree Search

Only Difference is Memoizing!

Is memoizing memory cost good for BFS and DFS?

- For BFS, memoizing memory cost is not so bad
  - Frontier is already big: $O(b^d)$
- For DFS, memoizing seems expensive
  - Frontier is tiny: $O(bm)$
- *Can we avoid building complete explored set for DFS?*

Path-checking DFS

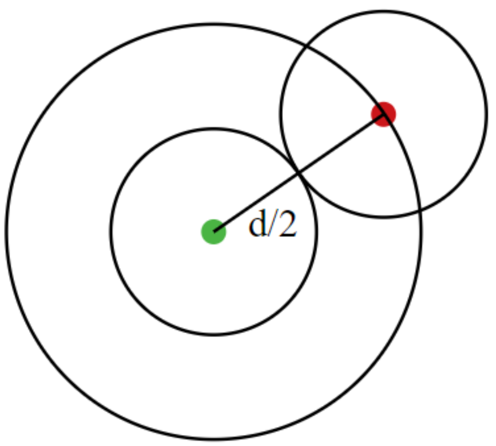
- Path-checking DFS keeps track of states on the current path only.

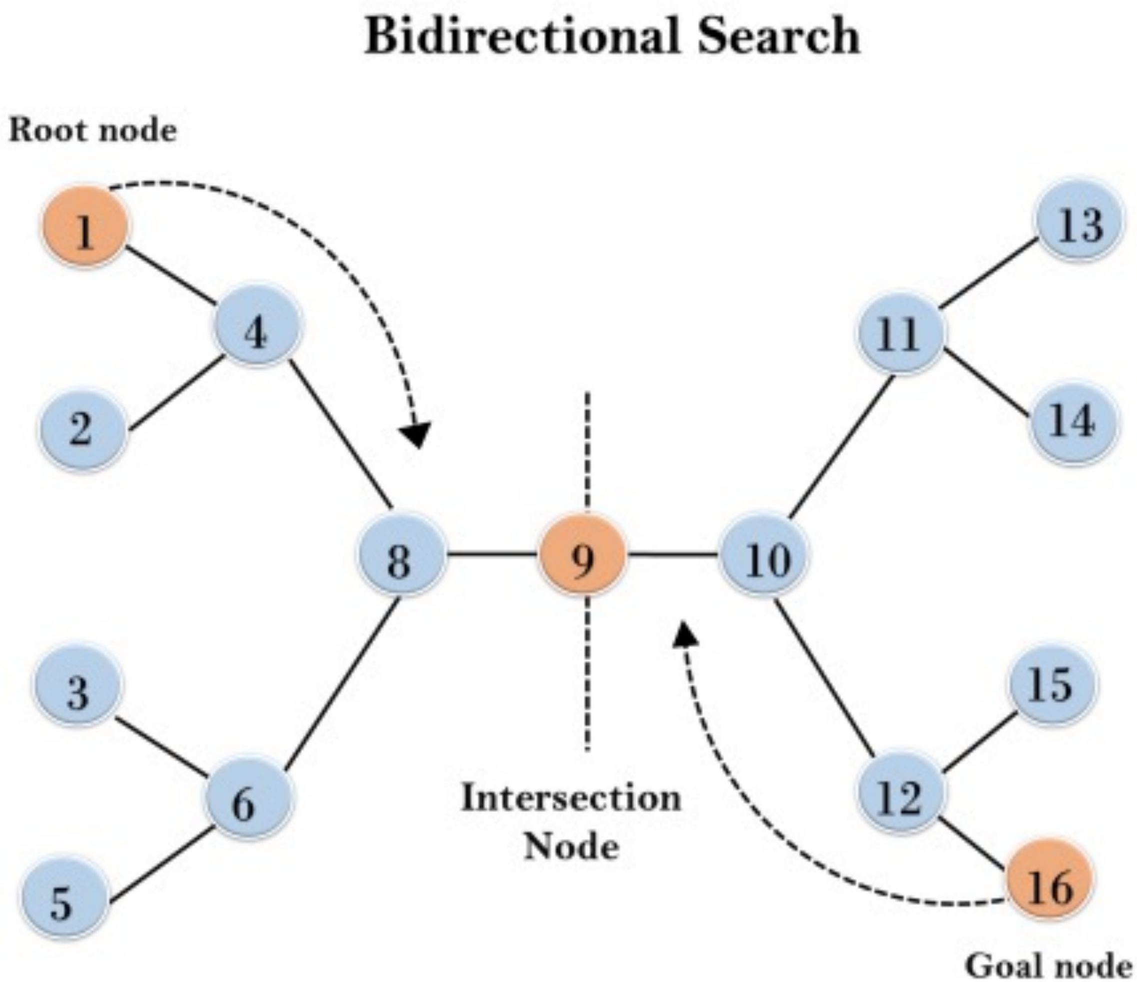
Graph Search (memoizing)

- Makes all search algorithms complete as it avoids loops.

Bi-directional search

- Sometimes you can search backwards:
  - a single identifiable goal
  - **inverse transition function** available (i.e., get-predecessors)
- Bi-directional search (e.g., BFS)
  - – Time $b^{d/2} + b^{d/2} < b^d$
  - – Complete and Optimal: y if BFS (same caveats)





Javatpoint.com

Summary

- Graph search to avoid repetitions
  - BFS, DFS (memoizing or path checking)
  - Trade-offs with memory use
- Bi-directional search: apply search from start and goal

Next

- Can we use cost and information about the goal to guide the search?
  - Uniform cost search
  - Informed search

Outline

- Uniform cost search
- Informed search methods
  - Heuristics
  - Greedy search
  - A* search

Uniform Cost Search (UCS)

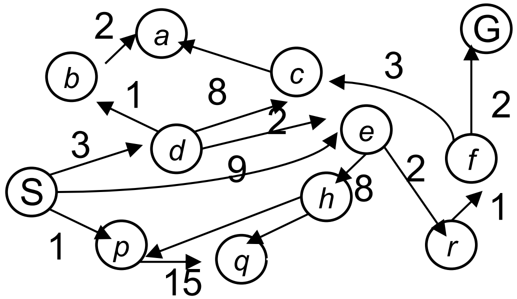
- *Strategy: expand the cheapest node first:*
- *Fringe is a priority queue (priority: cumulative cost)*

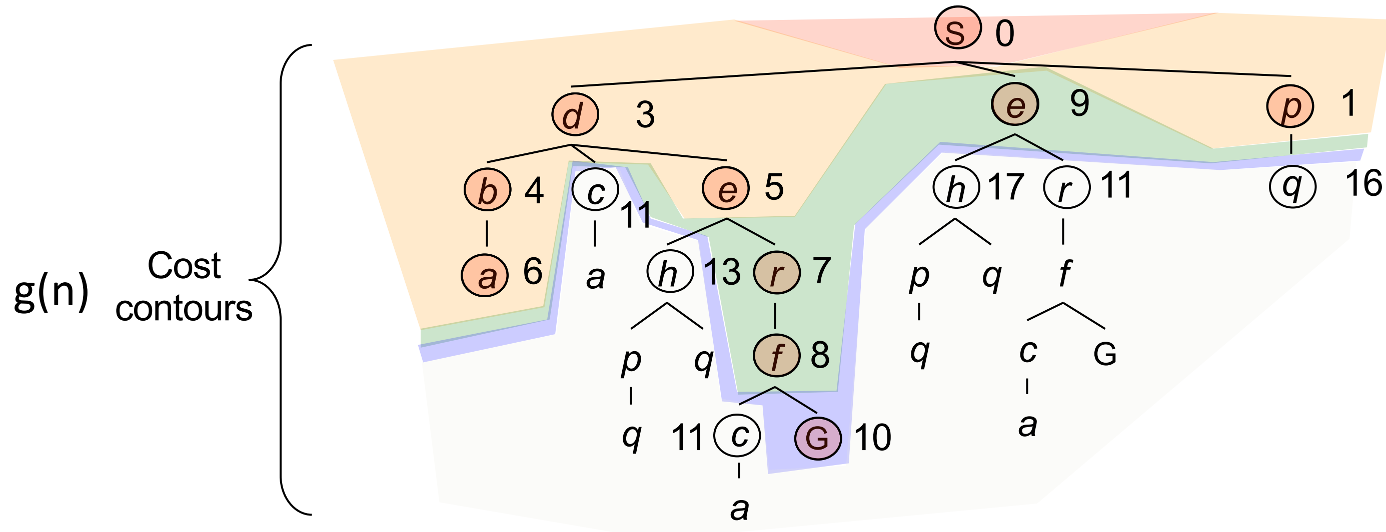
How would you modify BFS to incorporate cost?

Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)





UCS (graph) - pseudocode

```
frontier = new priority queue (ordered by path cost)
pack start_state into a node
add node to frontier

explored = new set (for memoizing)
add start_state to explored

while frontier is not empty:
  get current_node from the frontier # chooses lowest-cost node
  get current_state from current_node

  if current_state is the goal:
    backchain from current_node and return solution

  for each child of current_state:
    if child not in explored:
      add child to explored
      pack child state into a node, with backpointer to current_node
      add the node to the frontier
    else if child is in frontier with higher path cost
      replace that frontier node with child node

return failure
```

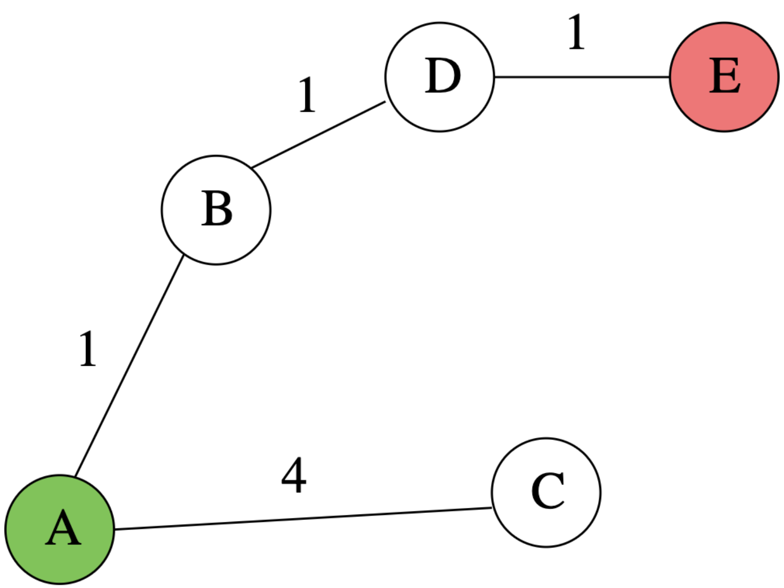
Properties of UCS

- Complete: Assuming best solution has a finite cost and minimum cost is positive, yes
- Time: $O(b^{C^*/\varepsilon})$
- Space: $O(b^{C^*/\varepsilon})$
- Optimal: Yes

C^* cost of optimal solution

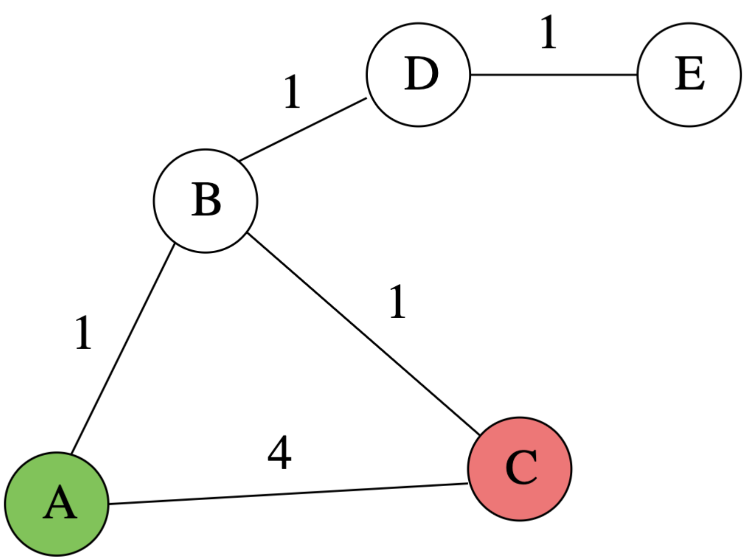
ε Smallest step cost

UCS – priority queue



- **Priority queue:**
- A0
- (pop A0, push B1, C4)
- B1 C4
- (pop B1, push D2)
- D2 C4
- (pop D2, push E3)
- E3 C4

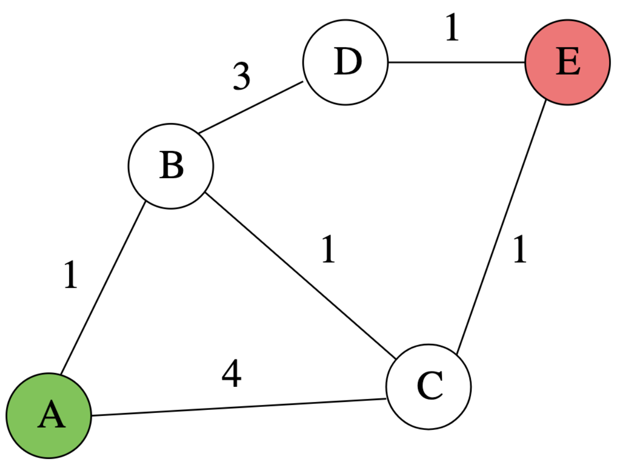
UCS – priority queue



- **Priority queue:**
- A0
- (pop A0, push B1, *C4*)
- B1 C4
- (pop B1, push C2)
- C2
- (pop C2, goal found!)

Modifying priorities in UCS

```
…
else if child is in frontier with higher path cost
      replace that frontier node with child node
…
```



- **Priority queue:**
- A0
- B1 *C4*
- *C2* D4
- E3 D4

Discussion

How do you efficiently check if a node is in a priority queue, or replace it efficiently?

Modifying priorities in UCS

```
…
else if child is in frontier with higher path cost
      replace that frontier node with child node
…
```

Operation	Heap	Unsorted ArrayList	Sorted ArrayList
isEmpty	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insert	$O(\log_2 n)$	$\Theta(1)$	$O(n)$
minimum	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
extractMin	$O(\log_2 n)$	$\Theta(n)$	$\Theta(1)$

Refresher from CS10

Modifying priorities in UCS

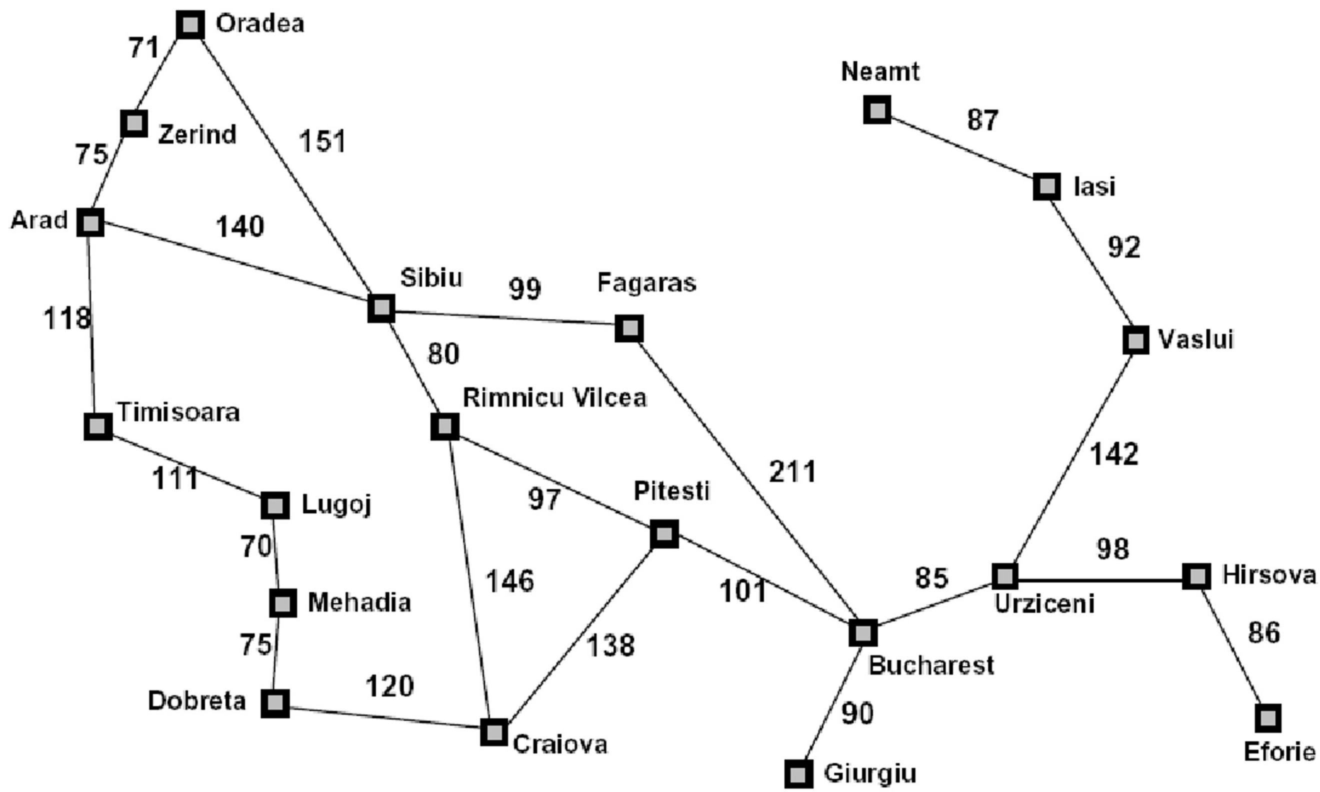
```
…
else if child is in frontier with higher path cost
      replace that frontier node with child node
…
```

- Fibonacci heap (will be in CS31)
- There will be instructions https://docs.python.org/3/library/heapq.html#priority-queue-implementation-notes

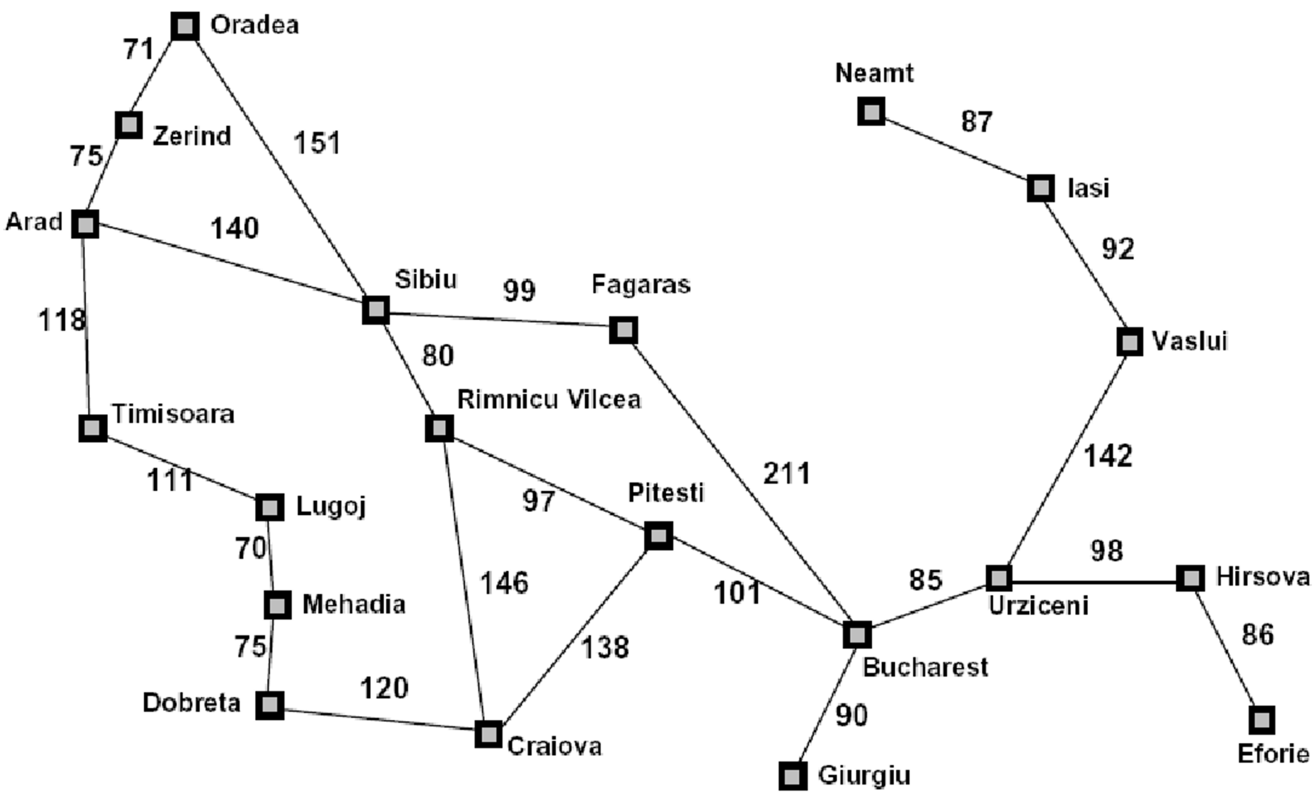
Informed search algorithms

- Sometimes we could estimate how close a state is to a goal
  - This function is called **heuristic function** **(*h(n)*)**

Heuristic example



Heuristic example

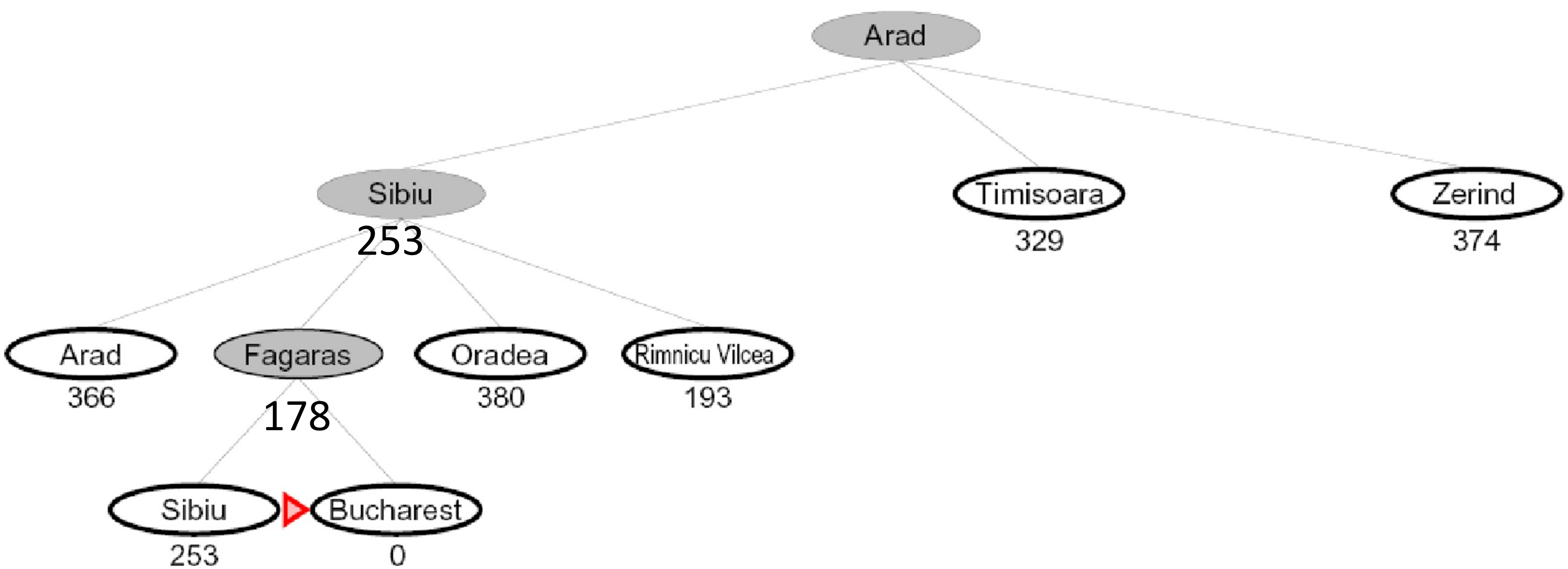


Straight−line distance to Bucharest

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

Greedy search

- Expand the node with minimum heuristic



Properties of greedy search (aka, best-first search)

- Complete: Complete in finite space with memoizing
- Time: $O(b^m)$
- Space: keeps all nodes in memory, $O(b^m)$
- Optimal: No

Discussion: UCS vs. Greedy

Discussion.

Discussion: UCS vs. Greedy

- UCS:
  - Disadvantage: Time
  - Advantage: Optimal*
  - Takes past cost into account.
- Greedy:
  - Disadvantage: Not optimal
  - Advantage: Time
  - Only concerned with the cost at the current step.

Discussion.

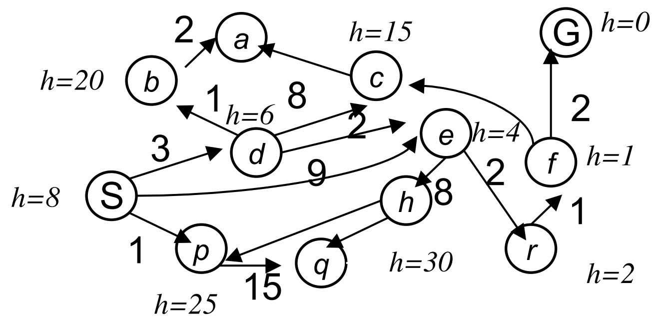
A* search

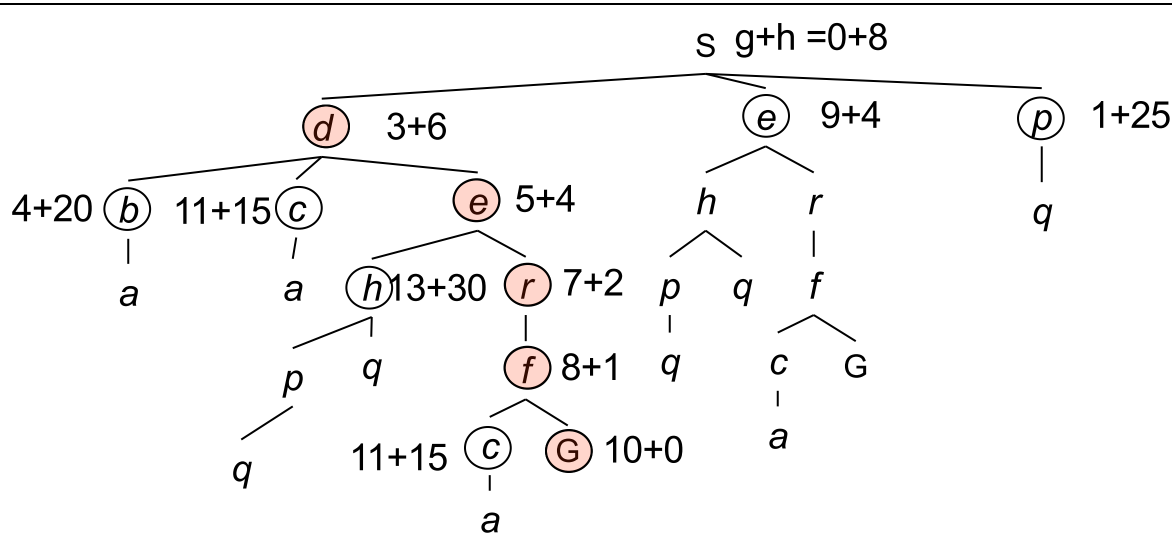
- Combine UCS and greedy
- Evaluation function (cost + heuristic)
  - $f(n) = g(n)+h(n)$

A* example

Strategy: expand a node with lowest f value (g+h)

Fringe is a priority queue (priority: f)

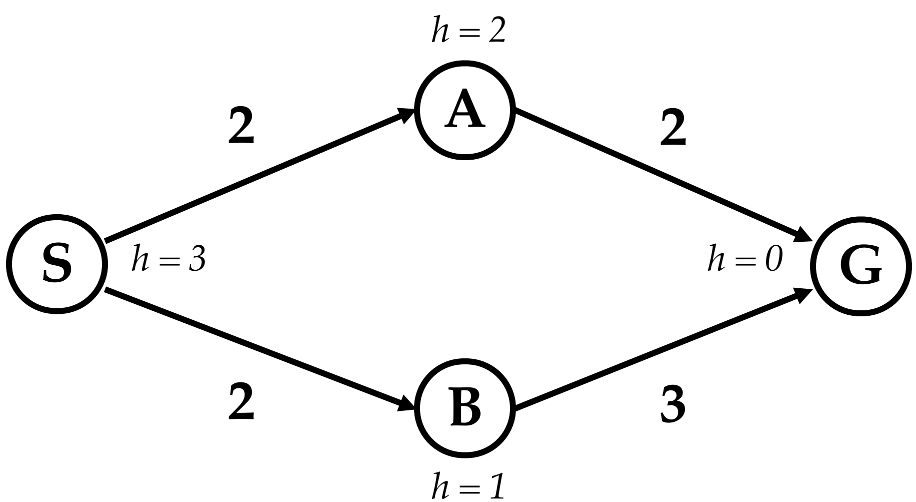




Discussion.

Discussion on A* termination

- Should we stop when we enqueue the goal?

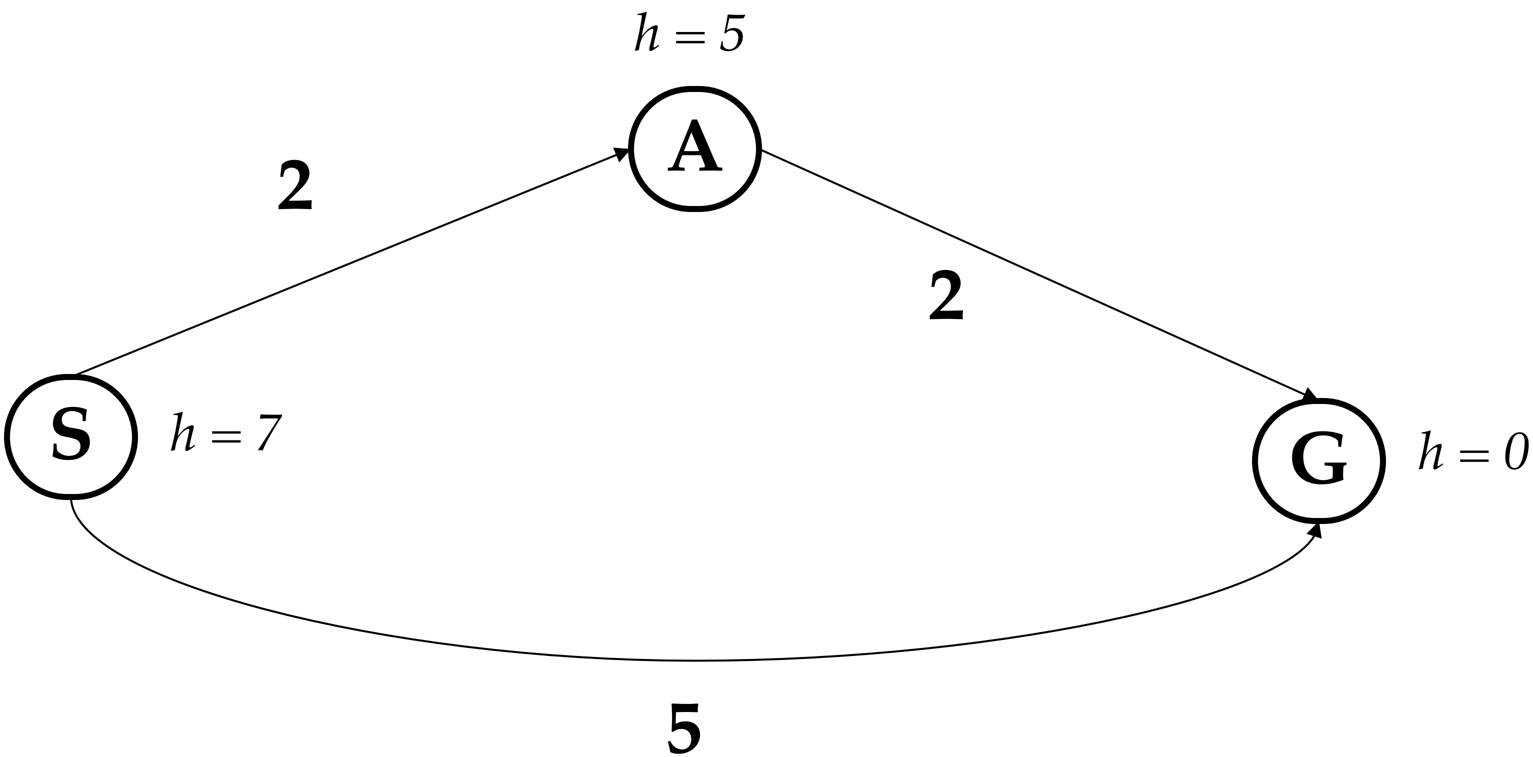


- No, stop when we dequeue a goal

- **Priority queue:**
- S3
- (pop S3, push B3, A4)
- B3 A4
- (pop B3, push G5 (if stopped when enqueuing not optimal path))
- A4 G5
- (pop A4, push G4)
- G4, G5
- (pop G4, done!)

Discussion.

Optimality of A*



- Which solution will be found by A*?
  - S,G, with cost 5 instead of S,A,G with cost 4, because of overestimated heuristic

Optimality of A* for tree-search

- A* *tree* search produces shortest paths if the heuristic is **optimistic** (also called **admissible**): it underestimates cost of path to goal from any node on the tree.

$$0 <= h(n) <= h^*(n)$$

where $h^*(n)$ is the true cost from node n.