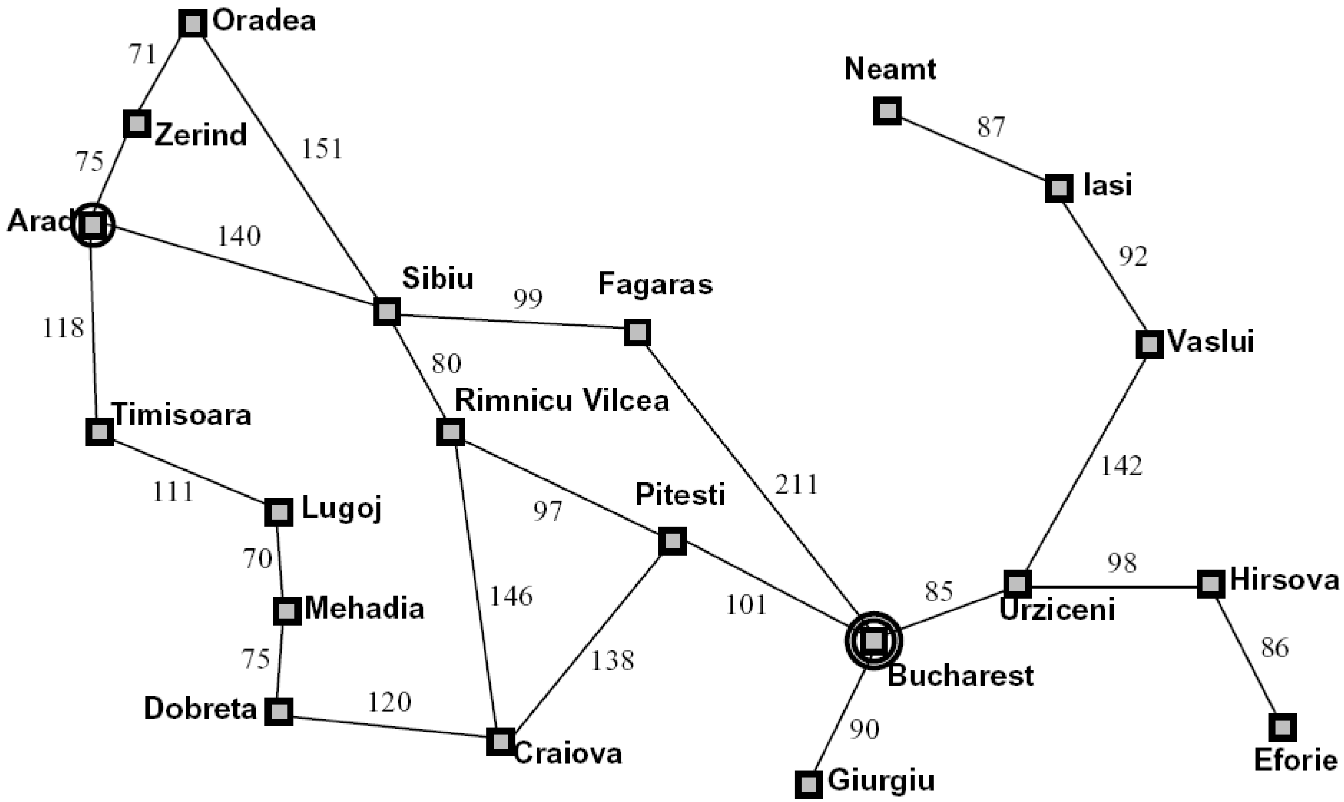
COGS44-COSC76/276 Artificial Intelligence Fall 2021 Uninformed search (Cont) & Informed Search

Soroush Vosoughi
Computer Science
Dartmouth College
Soroush@Dartmouth.edu

Reminders

- SA-1 due today Friday 11:59pm ET
- PA-1 due Sep 28th at 11:59pm ET

Recap: Search problem



- State space: cities
- Successor function: go to adjacent city
- Cost: distance between cities
- Start state: Arad
- Goal test: is state == Bucharest?

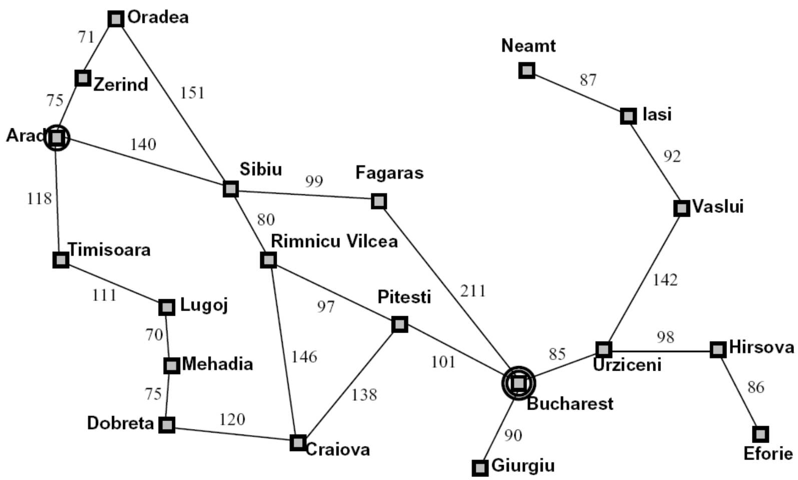
Recap: State space graph

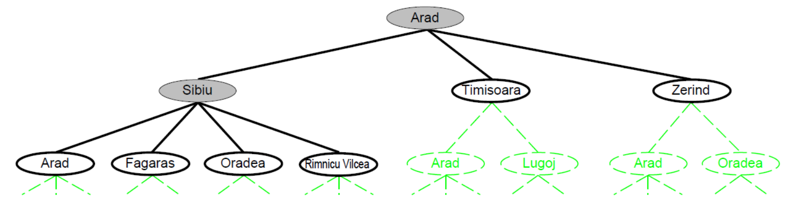
- State space graph: A mathematical representation of a search problem
  - States are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal states
- In a state space graph, each state occurs only once!
- The full graph is typically too big to store in memory

Recap: Search problems

- Element of search problems
  - A start state
  - A `goal_test` function that checks if a state is a goal state
  - A `get_actions` function that finds the legal actions from some state and a `transition` function that accepts a state, an action, and returns a new state, or alternatively, a `get_successors` function that returns a list of states given a starting state
  - A path_cost function that gives the cost of a path between a pair of states.
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

Recap: State space graphs vs search trees





- We construct both on demand – and we construct as little as possible

Recap: General tree search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy
- Does not keep track of expanded nodes

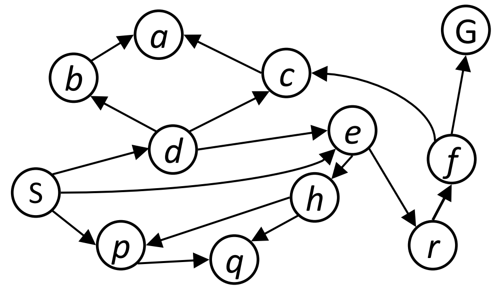
Is there any potential inefficiency here?

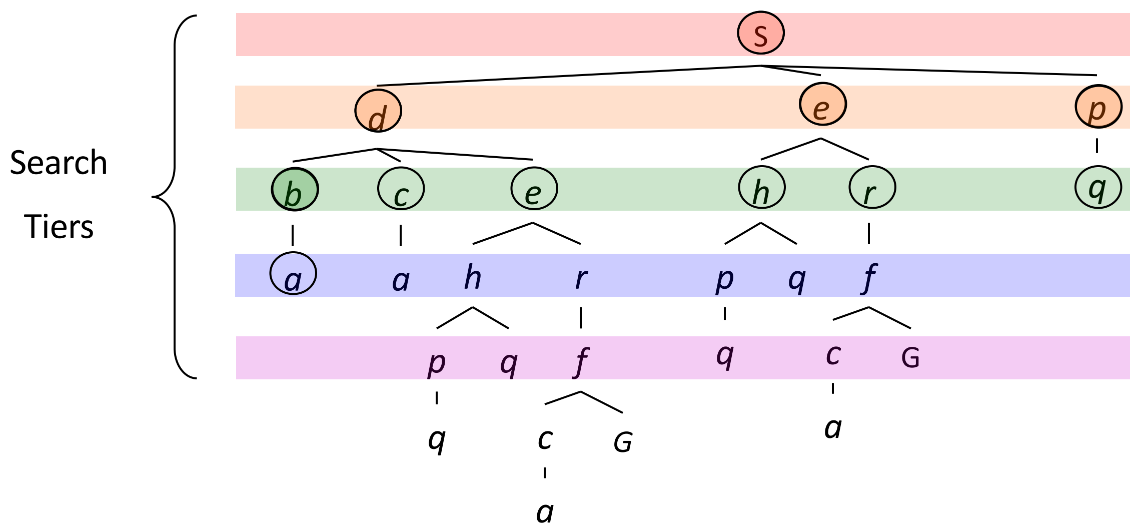
Recap: Breadth-First Search (BFS)

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue

Example

Example: Breadth-First Search (tree-search)





Properties of BFS

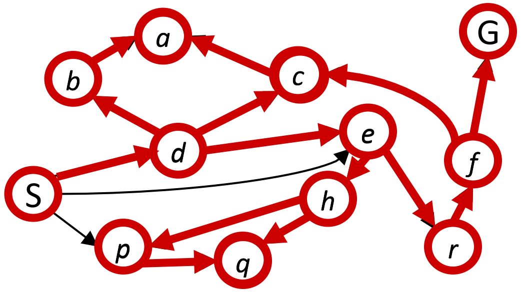
- Time:
  - $O(b^d)$
- Space:
  - $O(b^d)$
- Complete:
  - Yes if b is finite
- Optimal:
  - Yes, only if costs are all identical

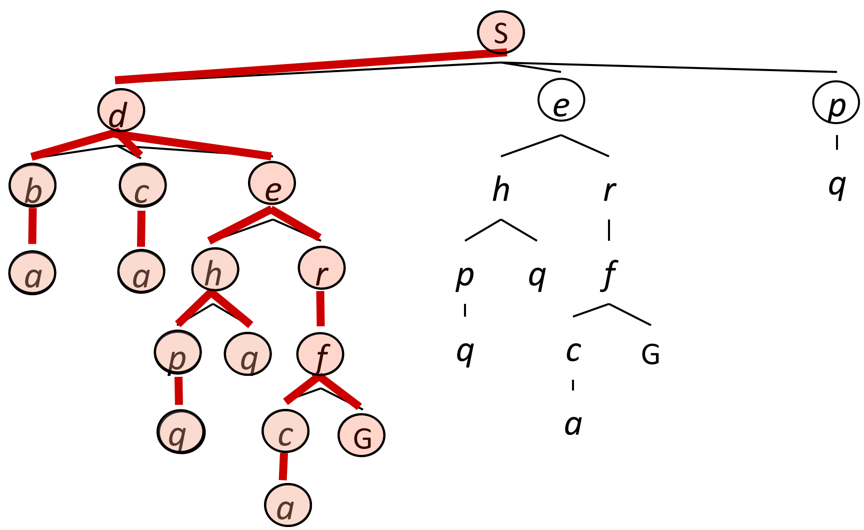
Depth-First Search (DFS)

- Expand deepest unexpanded nodes
- Implementation:
  - *fringe* is a LIFO stack

Example

Example: Depth-First Search (tree-search)





DFS Pseudocode (tree-search)

```
frontier = new stack
pack start_state into a node
add node to frontier

while frontier is not empty:
  get current_node from the frontier
  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:
    pack child state into node, add backpointer
    add the node to the frontier

return failure
```

Properties of DFS (tree-search)

- Time:
  - $O(b^m)$
- Space:
  - $O(bm)$
- Complete:
  - No
- Optimal:
  - No it finds the "leftmost" solution, regardless of depth or cost

Discussion

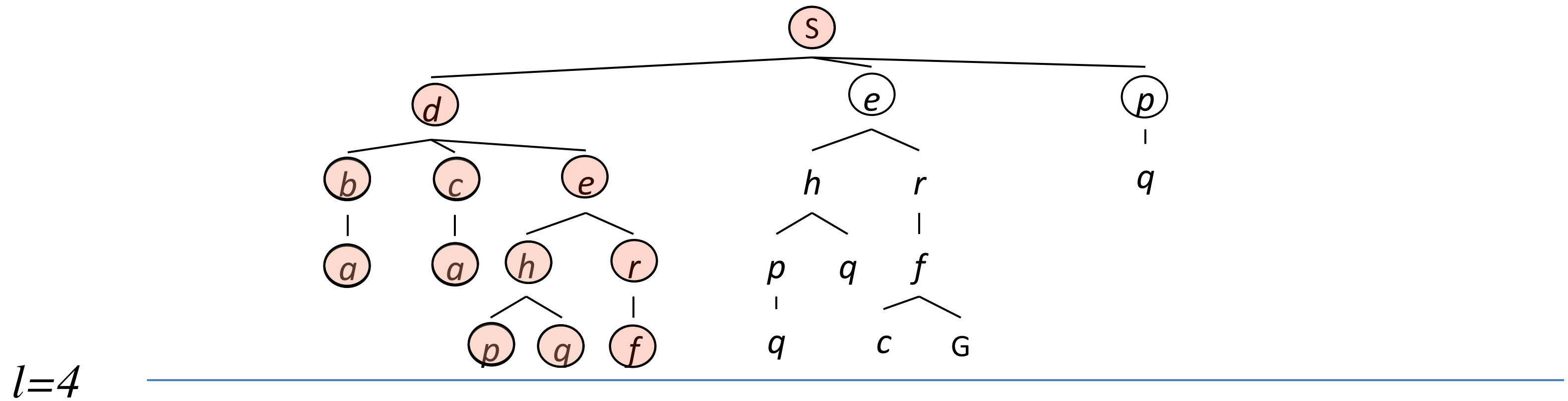
BFS vs DFS

- When will BFS outperform DFS?
  - Solutions not too far down
- When will DFS outperform BFS?
  - Solutions far at the bottom and memory constrained

Discussion

Depth-limited search

- DFS with depth limit l

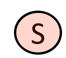


Properties of Depth limited

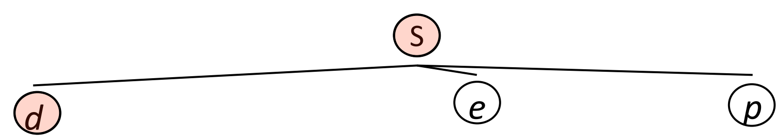
- Complete: No
- Time: $O(b^l)$
- Space: $O(bl)$
- Optimal: No

Iterative deepening

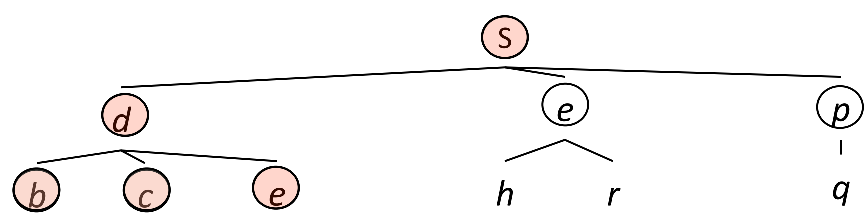
- Run Depth-limited search with increasing depth limit, i.e.,
  - $l=0$



  - $l=1$



  - $l=2$



  - ...

Properties of Iterative deepening

- Complete: Yes if b is finite
- Time: $O(b^d)$
- Space: $O(bd)$
- Optimal: only if costs are all identical

PA-1 - First programming assignment

- Includes modeling of real problem as search problem
- Apply uninformed search to find a solution to the problem
- You will find it on Canvas soon

Summary

- 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

Next

- Keeping history to avoid repetitions
- Can we do any better when searching for a solution than the algorithms we have seen so far?

- Implement graph search methods (keeping track of history) for BFS and DFS

Outline

- Graph-search (memoizing)
  - BFS
  - DFS
    - Path-checking DFS
- Bi-directional search

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 does: memoizing – i.e., keeping track of the states already visited

BFS (graph) - pseudocode

```
frontier = new queue
pack start_state into a node
add node to frontier

explored = new set
add start_state to explored

while frontier is not empty:
  get current_node from the frontier
  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

return failure
```

DFS (graph) - pseudocode

```
frontier = new stack
pack start_state into a node
add node to frontier

explored = new set
add start_state to explored

while frontier is not empty:
    get current_node from the frontier
    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 node, add backpointer
        add the node to the frontier

return failure
```

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?*

Discussion. Write down on paper first

Path-checking DFS

- Path-checking DFS keeps track of states on the current path only, and doesn't loop
- Does not eliminate redundant paths

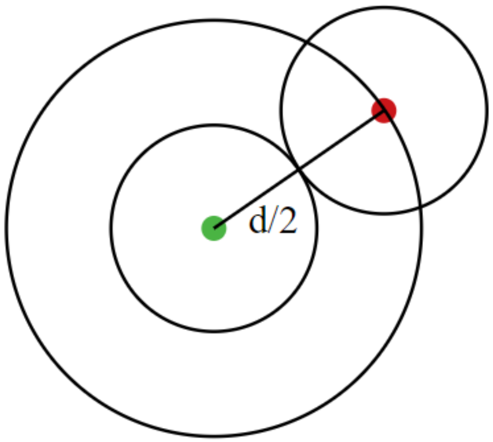
Comparing uninformed graph search

Algorithm	Time	Memory	Complete	Optimal
BFS (graph)	$O(min(n,b^d))$	$O(min(n,b^d))$	y	y*
DFS (memoizing)	$O(min(n,b^m))$	$O(min(n,b^m))$	y	n
DFS (path-checking)	$O(min(m^n,mb^m))$	$O(m)$	y	n
Iterative deepening (path-checking)	$O(min(d^n,db^d))$	$O(d)$	y	y*

With state space size n

Bi-directional search

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



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

- Implement cost-sensitive search
- Implement informed search methods

Outline

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

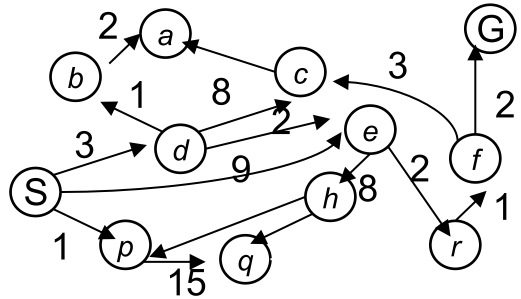
Uniform Cost Search (UCS)

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

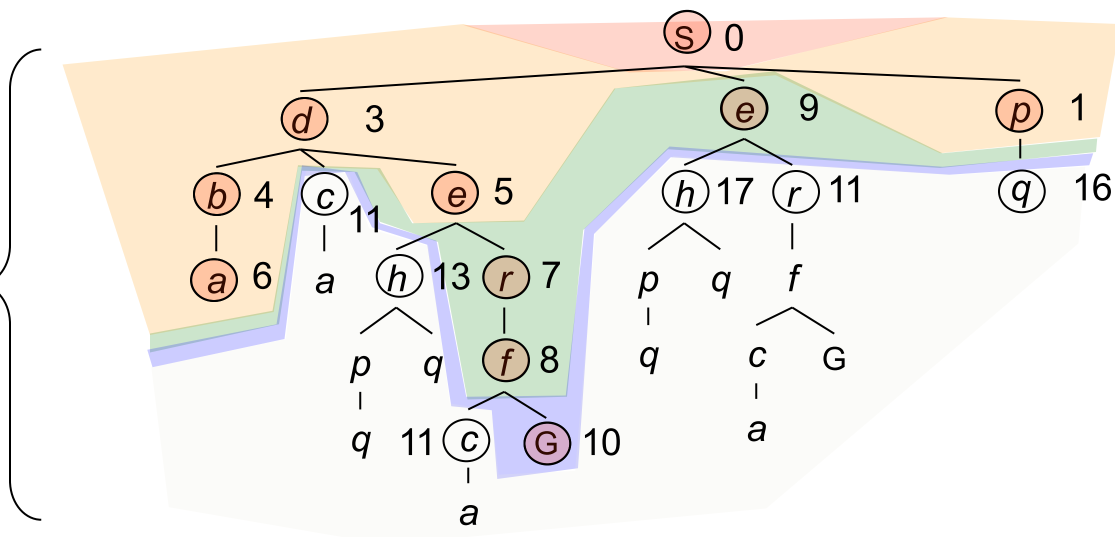
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)



Cost contours



Example

UCS (graph) - pseudocode

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

explored = new set
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
```

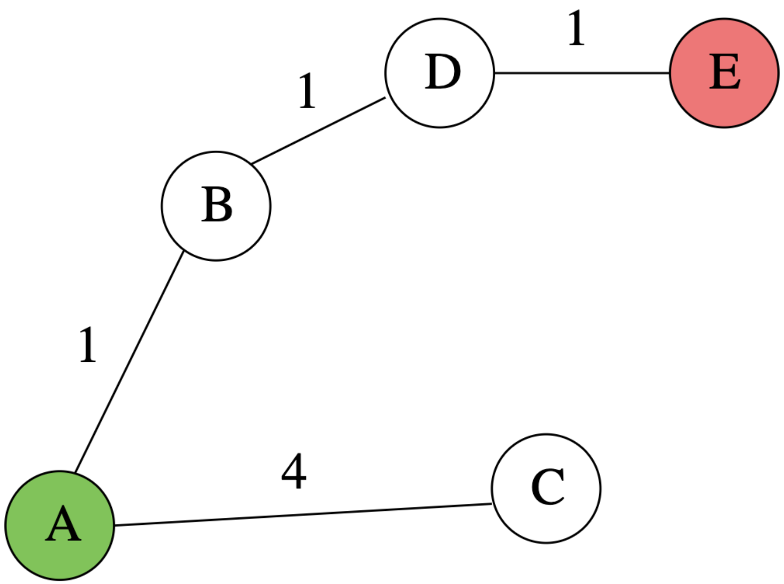
Properties of UCS

- Complete: Assuming best solution has a finite cost and minimum arc cost is positive, yes
- Time: # of nodes with $g <=$ cost of optimal solution, $O(b^{C^*/\varepsilon})$
- Space: # of nodes with $g <=$ cost of optimal solution, $O(b^{C^*/\varepsilon})$
- Optimal: Yes

C^* cost of optimal solution

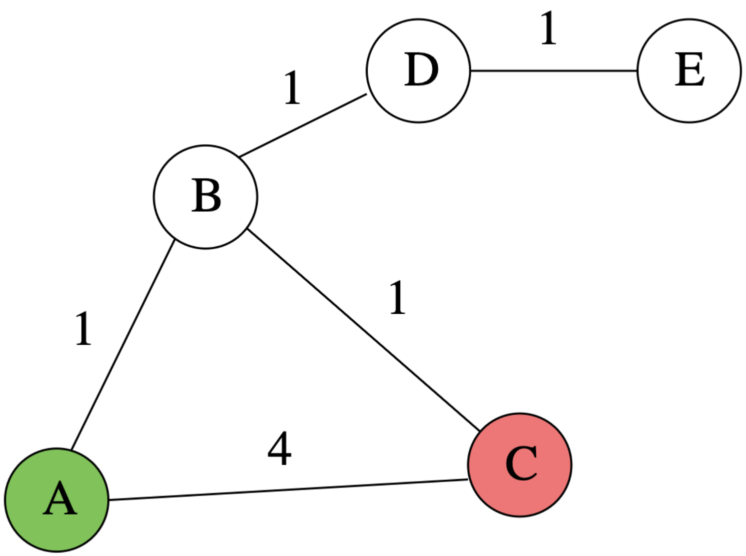
ε Smallest step cost

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

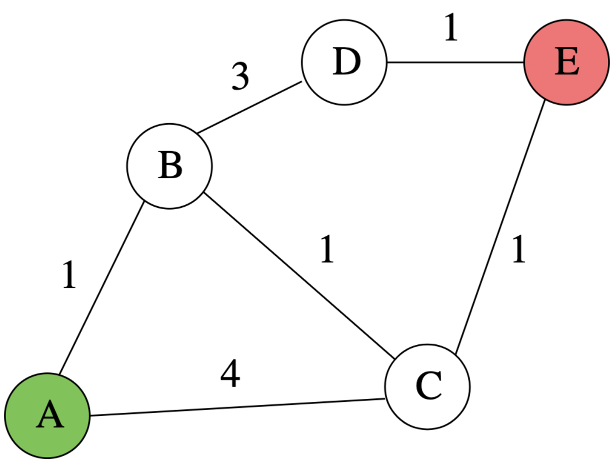
UCS – priority queue



- **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

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

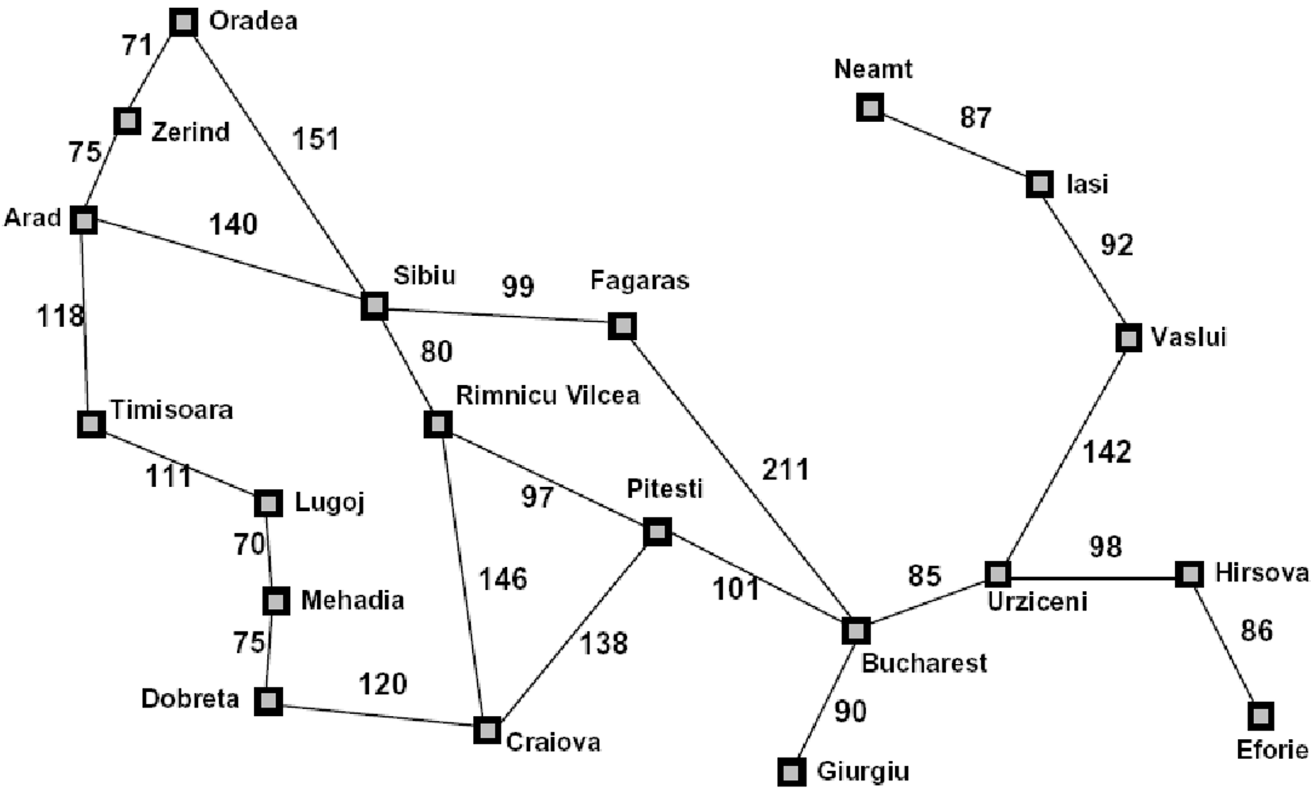
```
…
else if child is in frontier with higher path cost
      replace that frontier node with child node
…
```

- Fibonacci heap (will be in CS31)
- Mark expensive item as removed, add cheaper item
  - Replace explored set with explored dictionary, storing cost of least-expensive route to node as value.
  - If a state is being considered for addition to the frontier, check if it is in explored already.
    - If not, add to frontier.
    - If so,
      - but the new node has a less expensive path cost, change the cost in explored and add the new node anyway.
      - Otherwise, do not add the node to the frontier.

Informed search algorithms

- Sometimes we could estimate how close a state is to a goal
  - This function is called heuristic function

Heuristic example

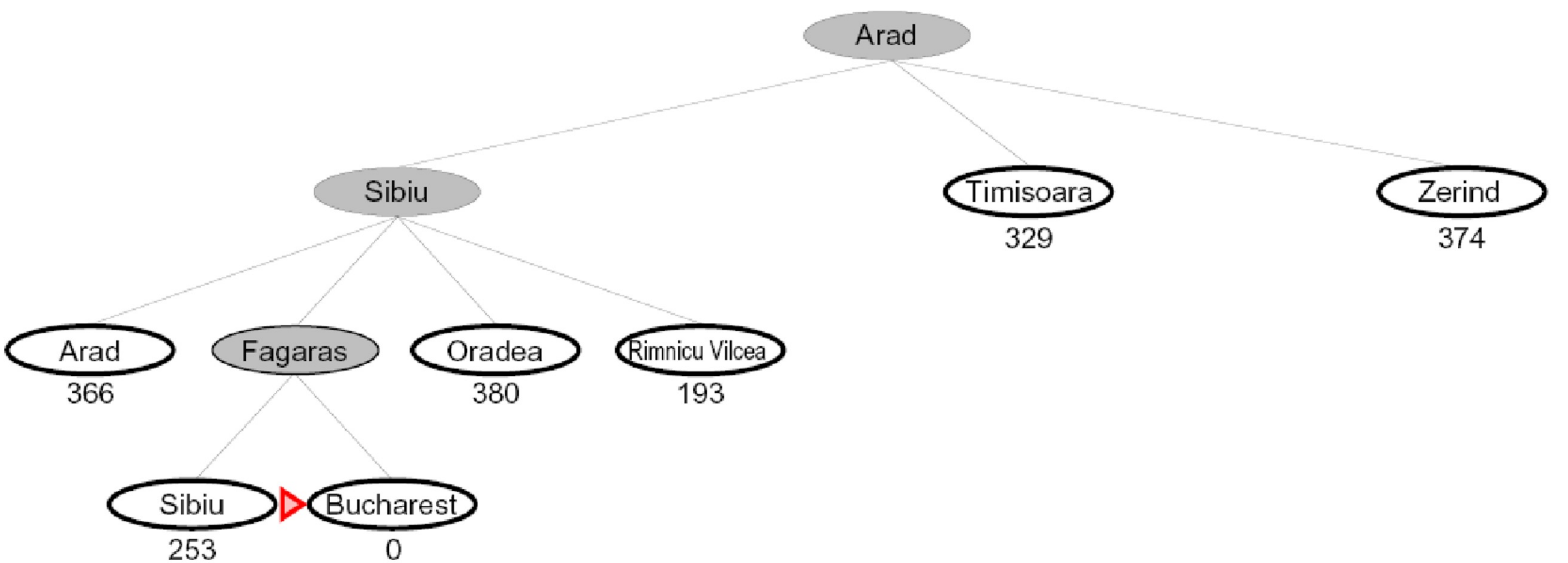


Straight−line distance to Bucharest

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

- Complete: Can get stuck in loops, but 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

- When UCS finds a solution faster than greedy?
- When greedy finds a solution faster than UCS?

Discussion. Write down on paper first