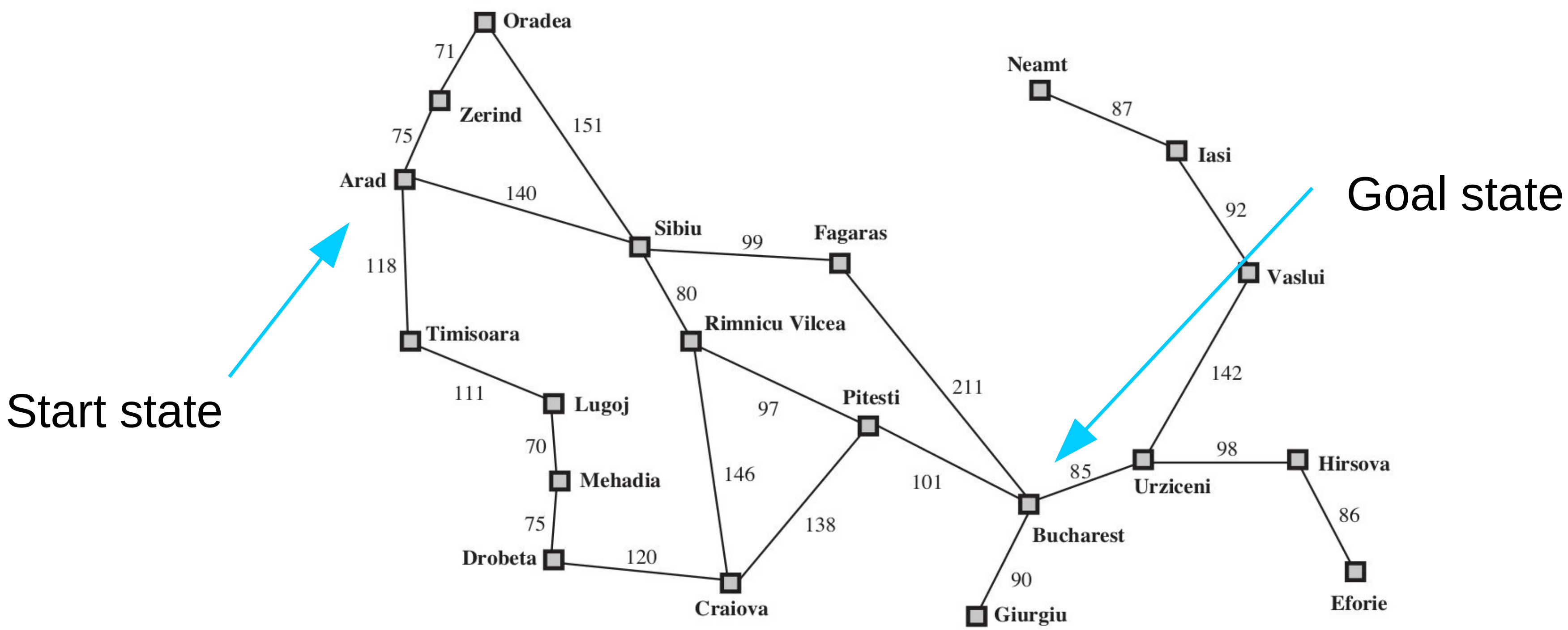
Breadth first search
Uniform cost search

Robert Platt
Northeastern University

Some images and slides are used from:
1. CS188 UC Berkeley
2. RN, AIMA

What is graph search?

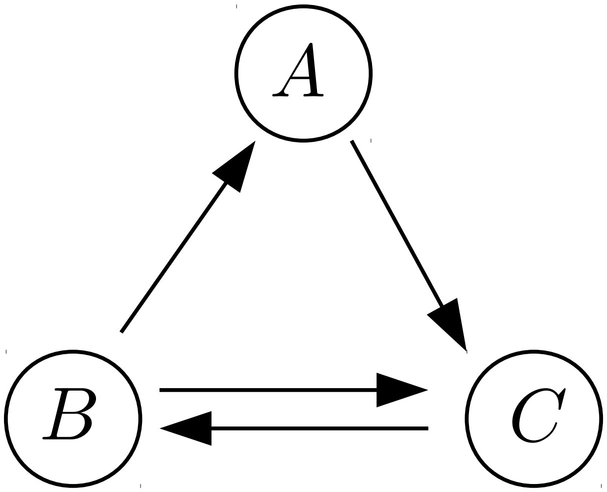


What is a graph?

Graph: $G = (V, E)$

Vertices: V

Edges: E



Directed graph

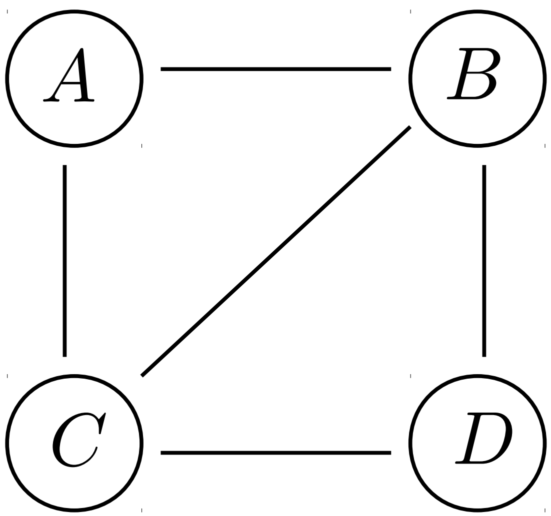
$$V = \{A, B, C\}$$
$$E = \{(B, A), (A, C), (B, C), (C, B)\}$$

What is a graph?

Graph: $G = (V, E)$

Vertices: V

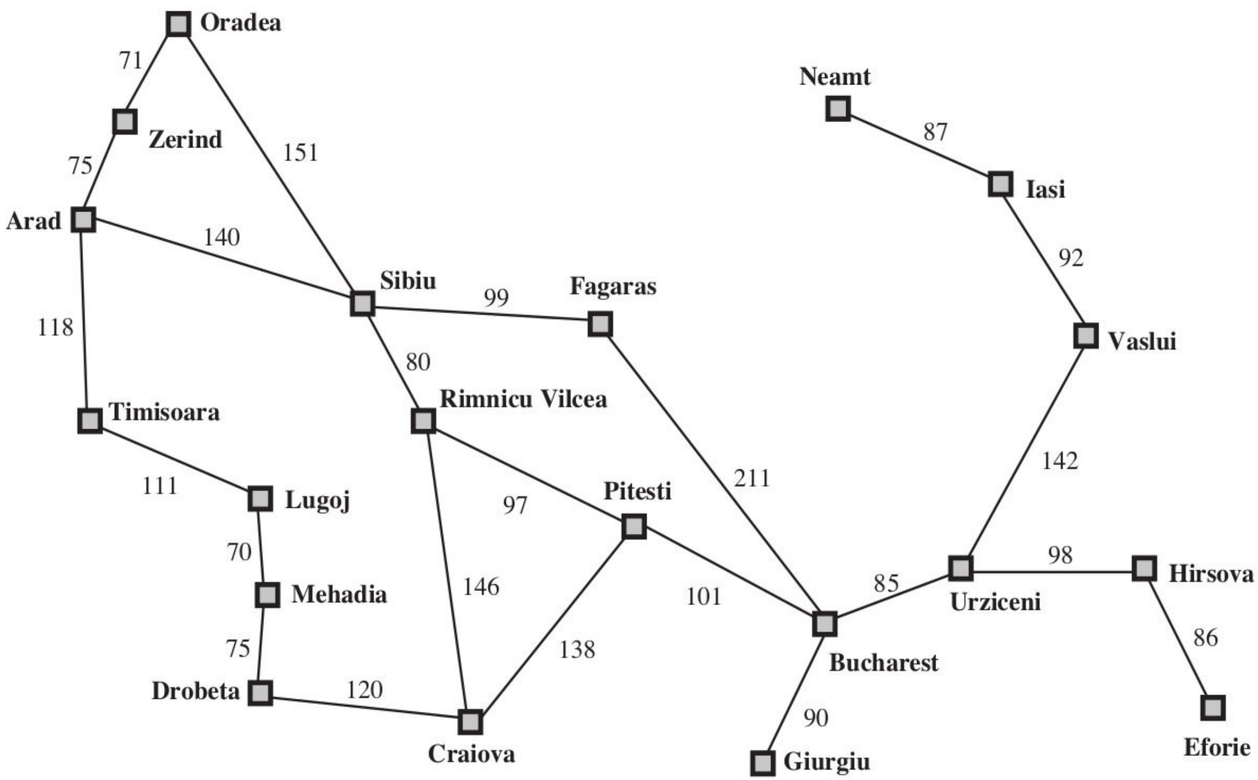
Edges: E



Undirected graph

$$V = \{A, B, C, D\}$$
$$E = \{\{A, C\}, \{A, B\}, \{C, D\}, \{B, D\}, \{C, B\}\}$$

Graph search



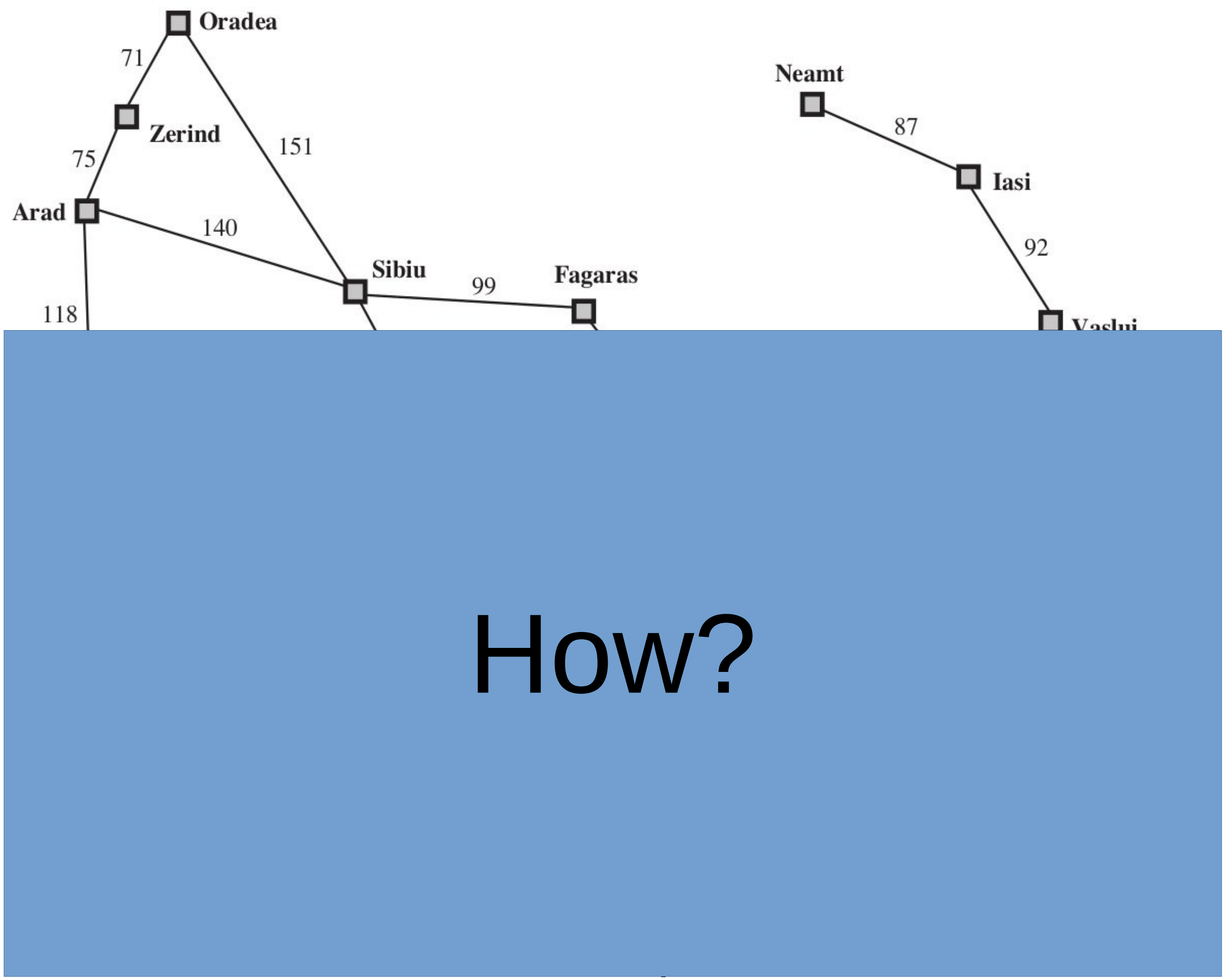
<u>Given</u>: a graph, *G*

<u>Problem</u>: find a path from A to B

– A: start state

– B: goal state

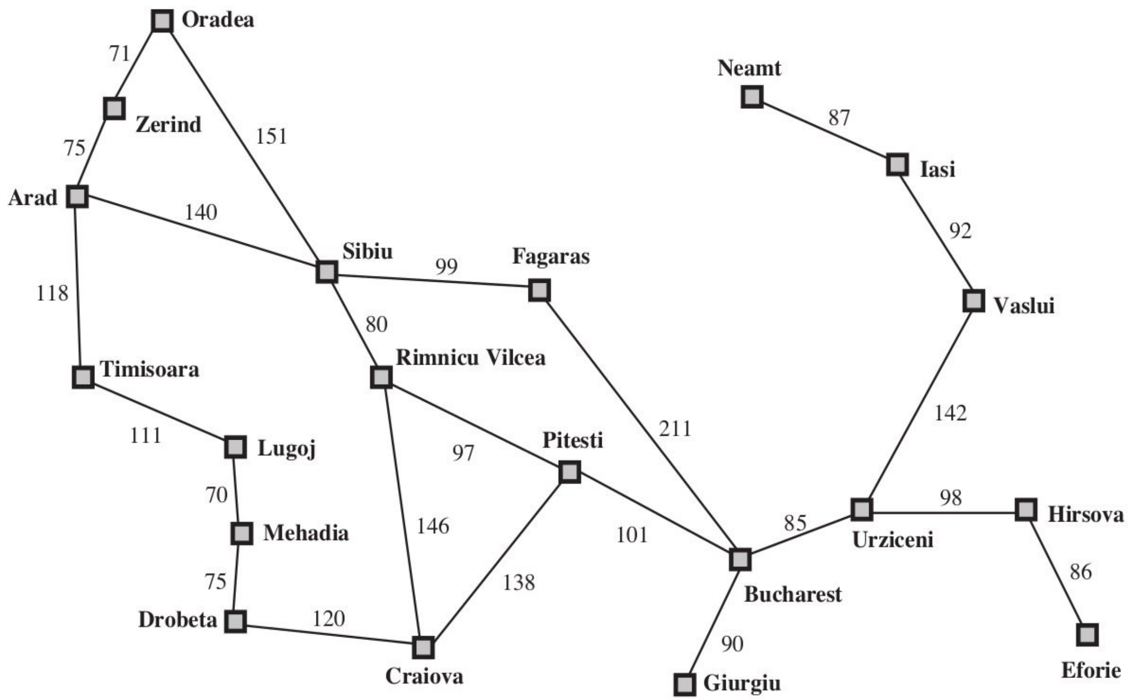
Graph search

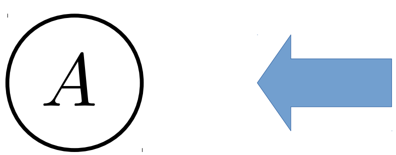


– A: start state

– B: goal state

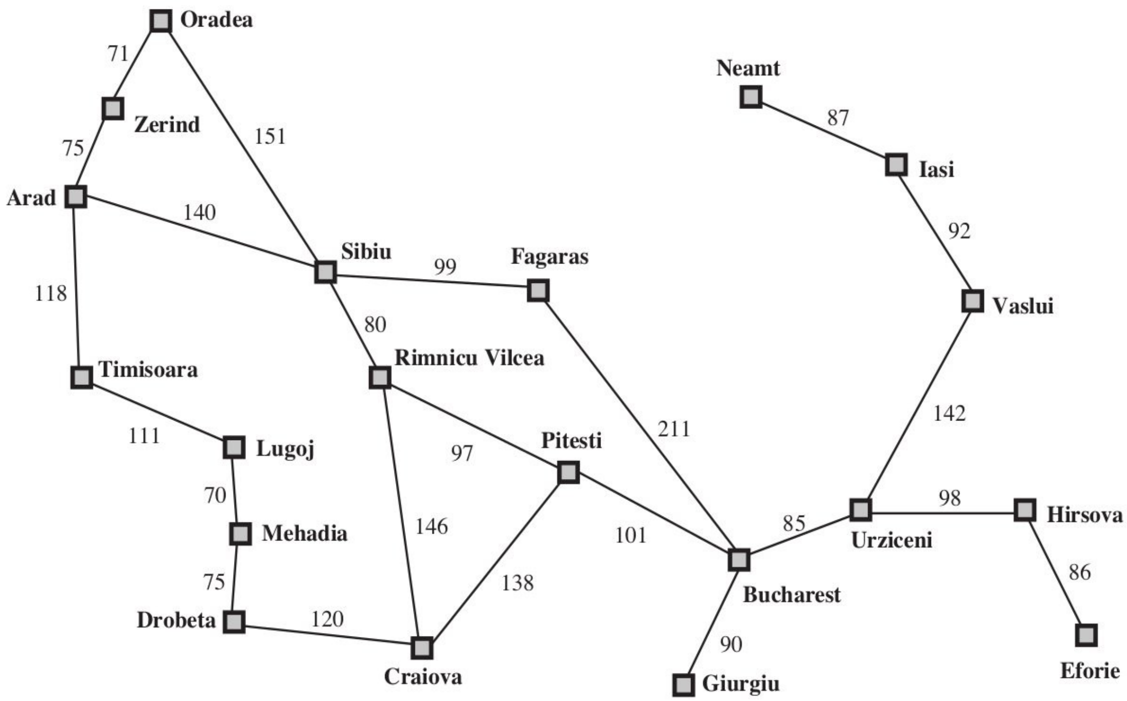
A search tree

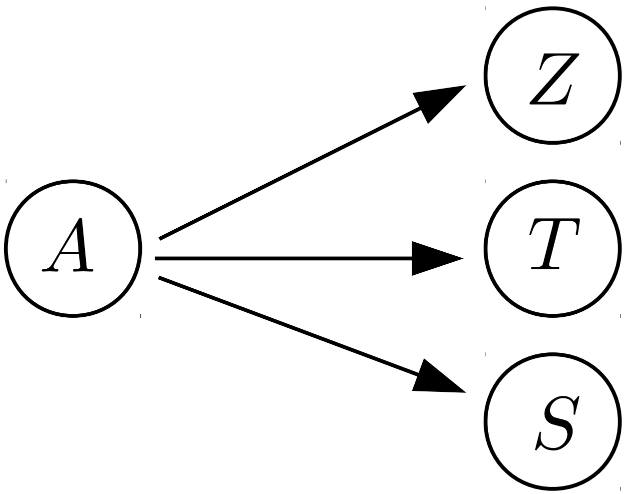




Start at *A*

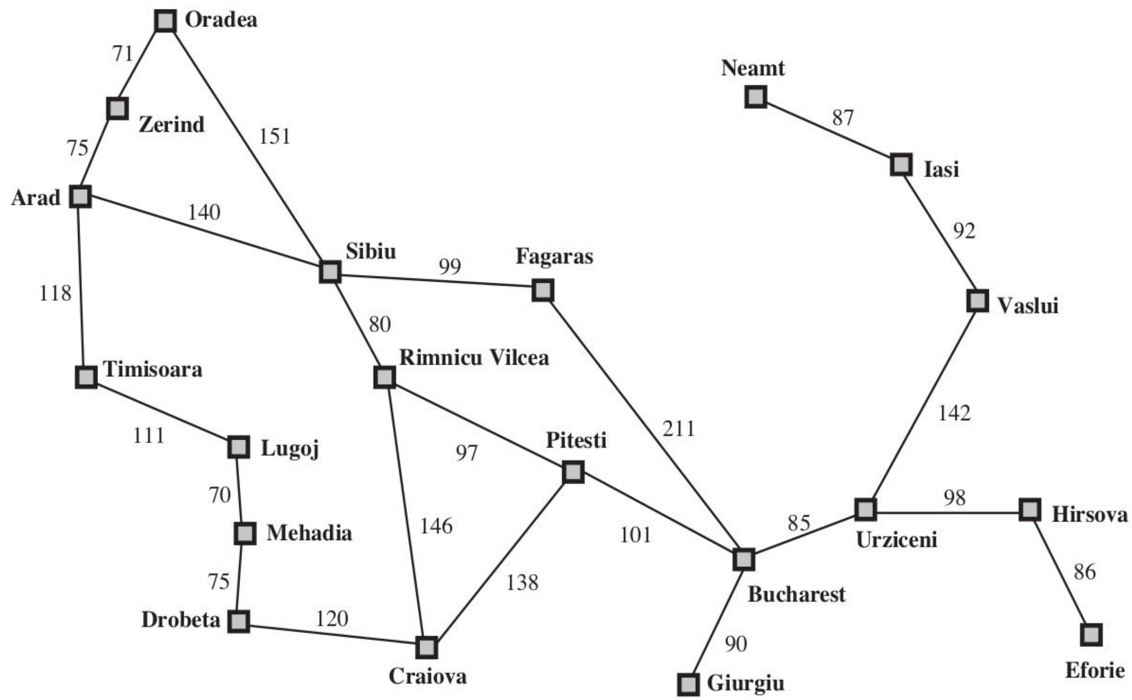
A search tree

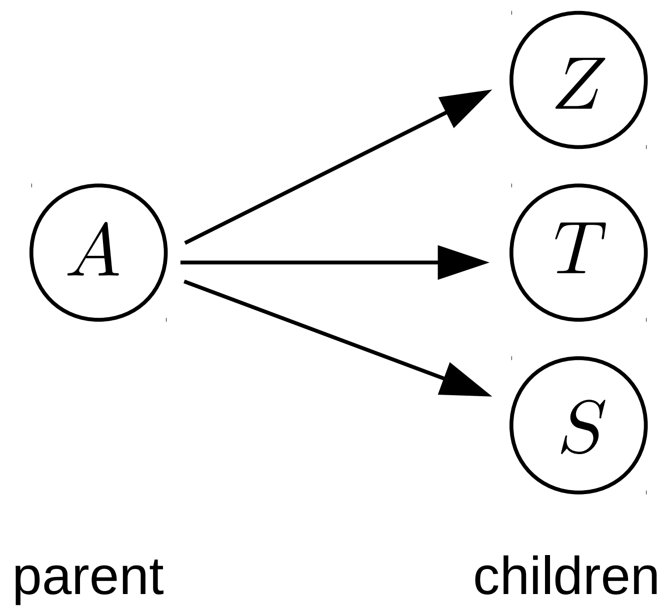




Successors of *A*

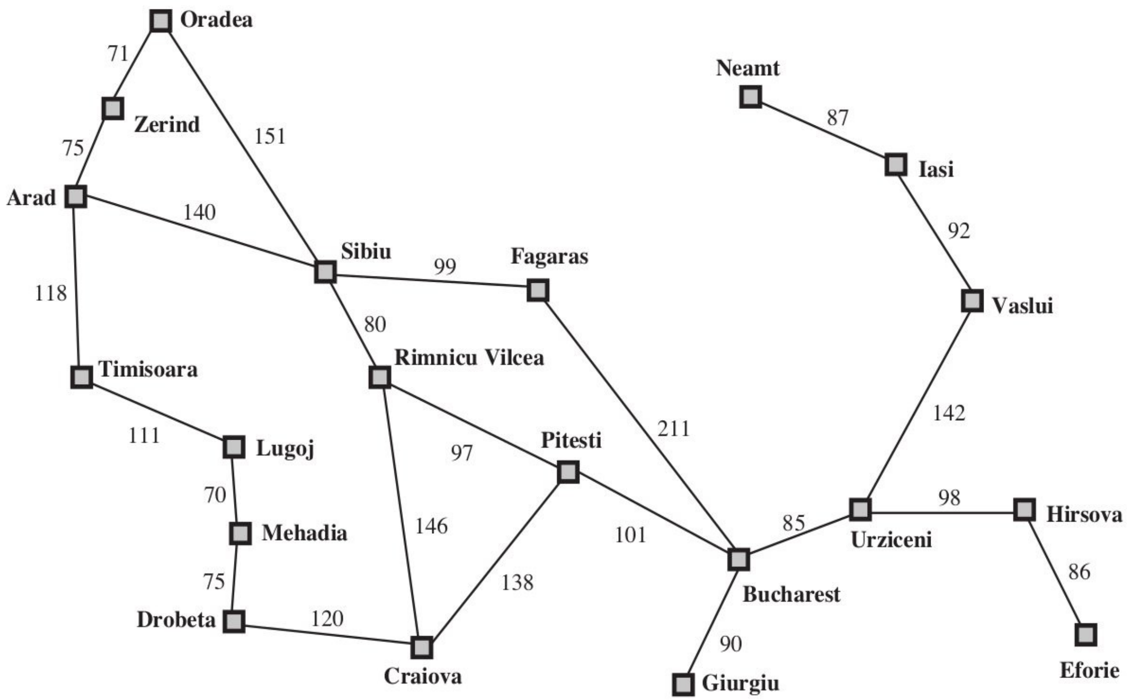
A search tree

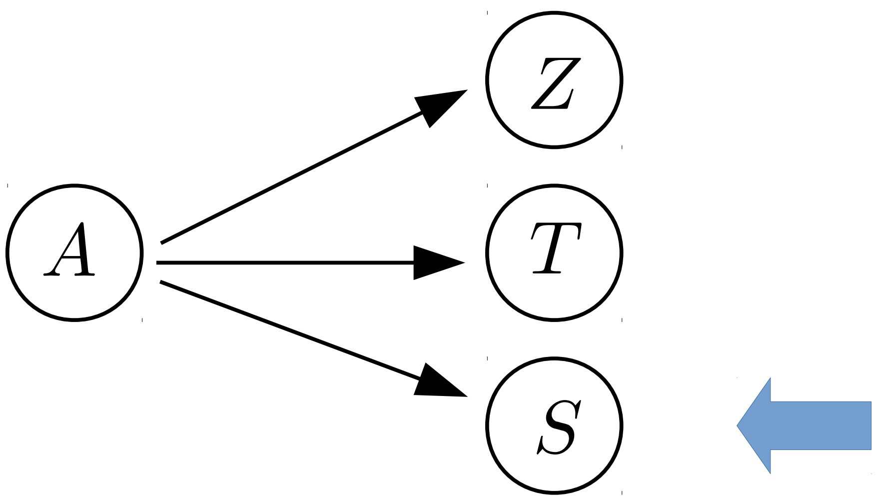




Successors of *A*

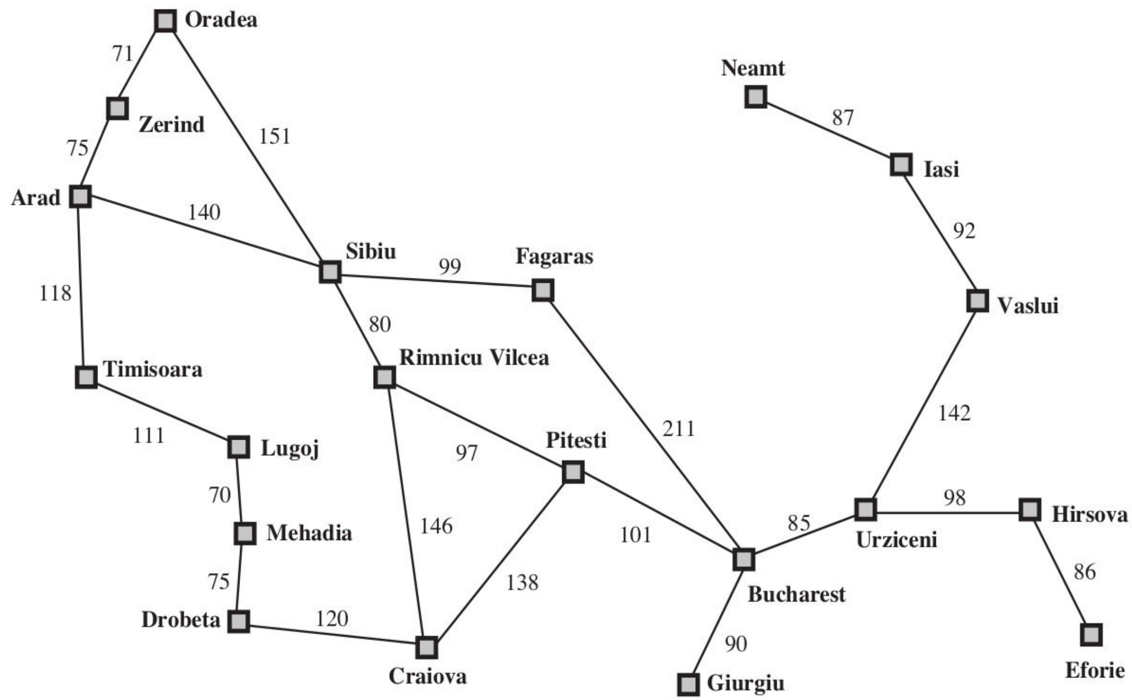
A search tree

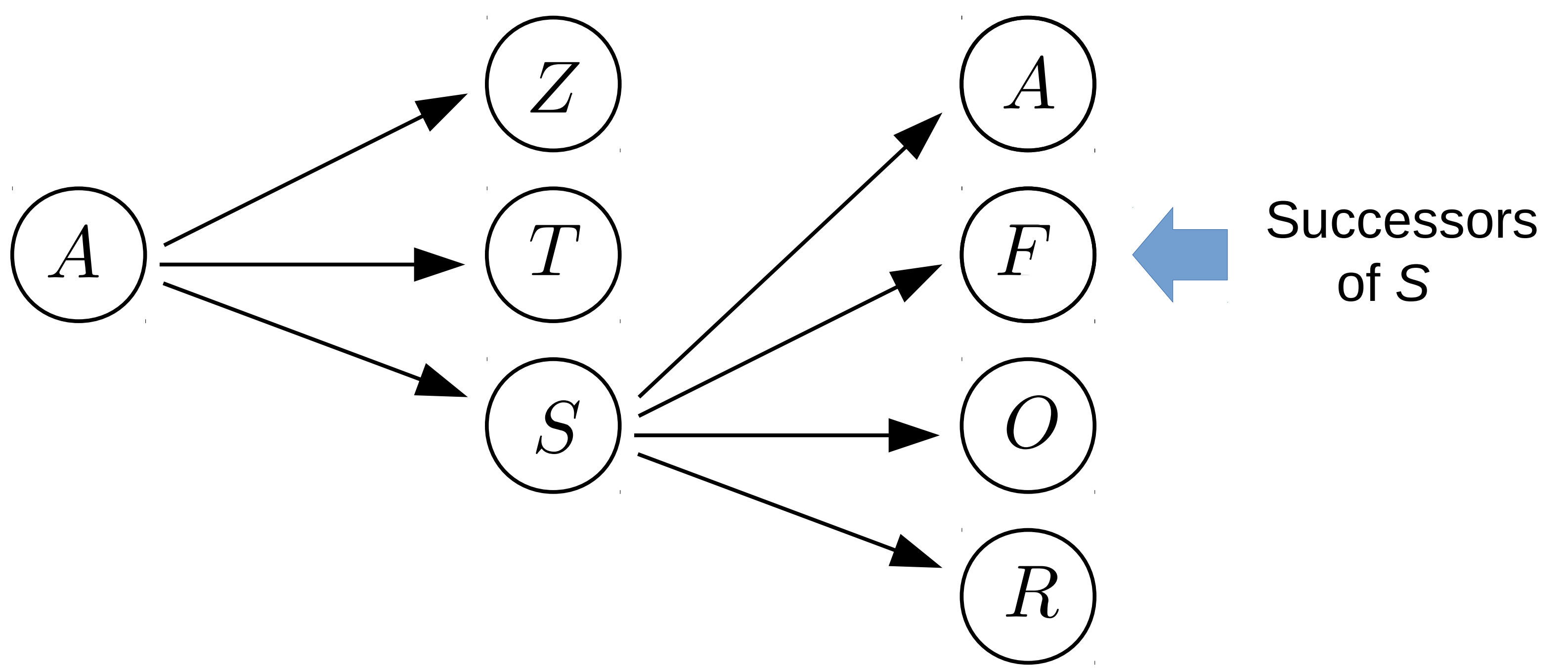




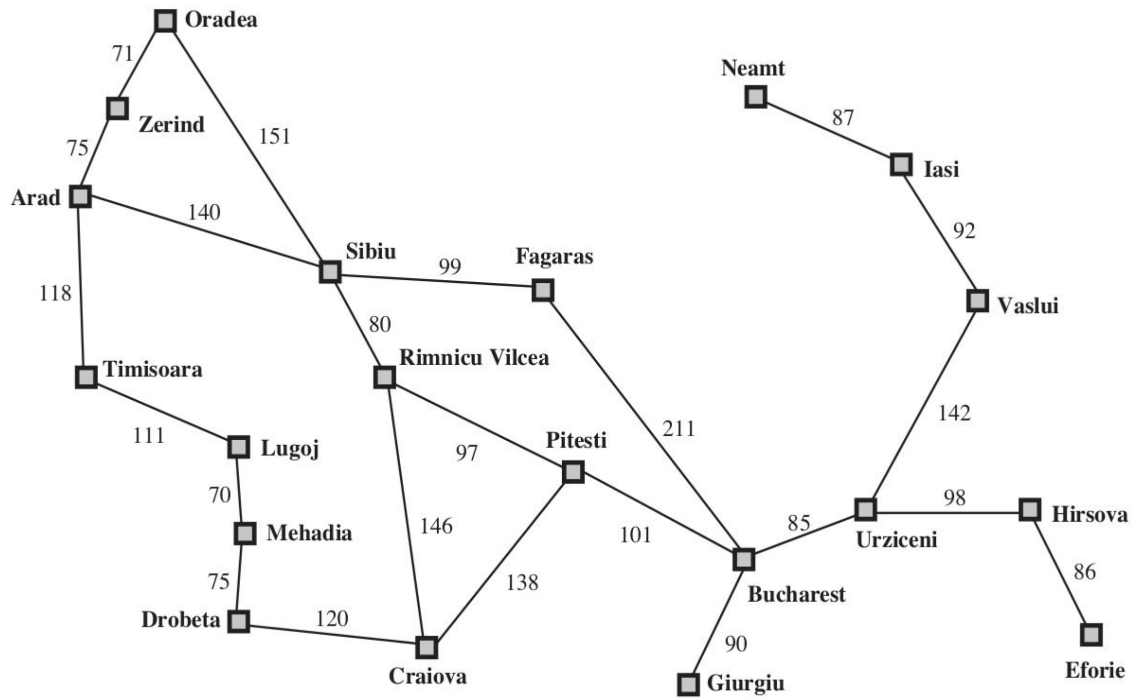
Let's expand S next

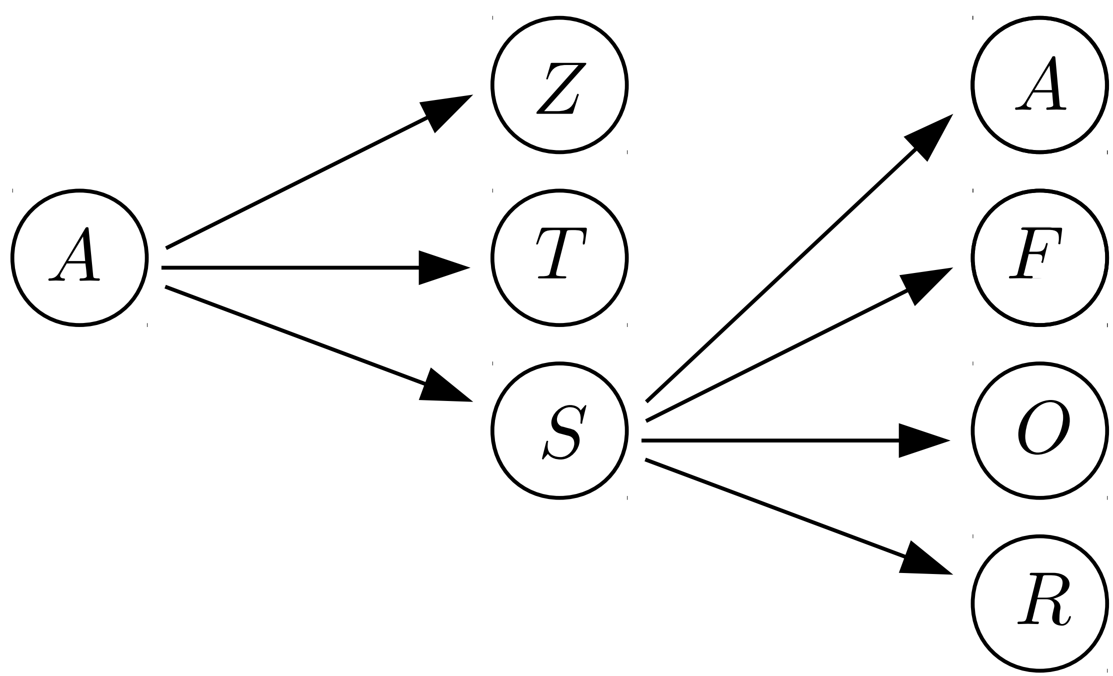
A search tree





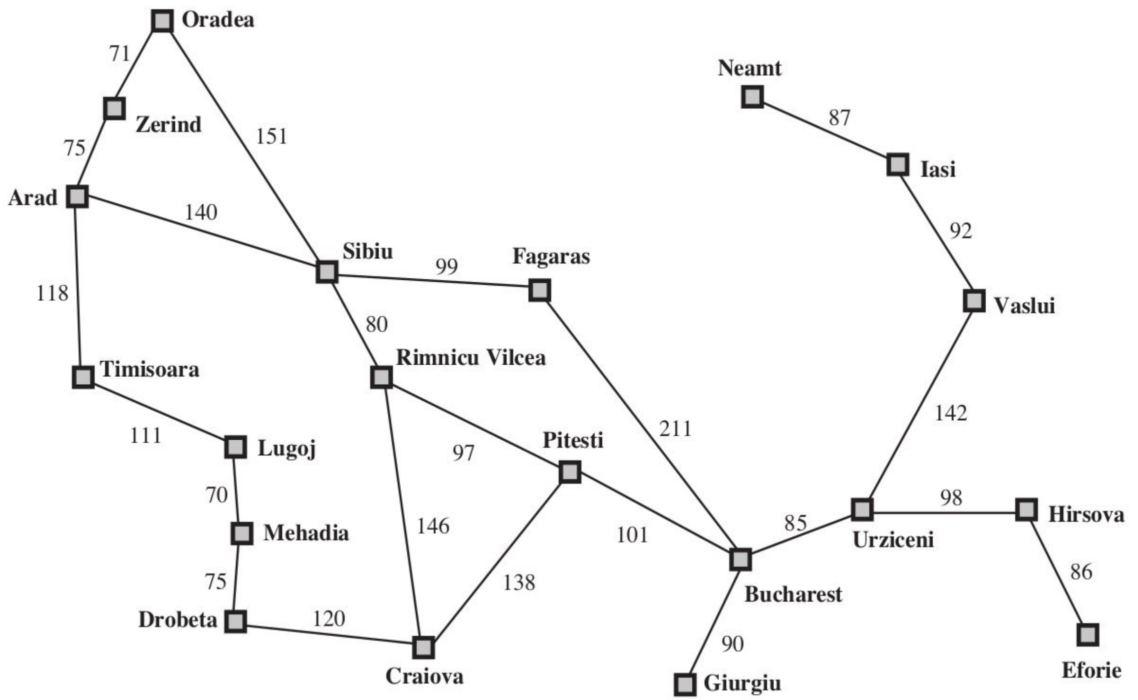
A search tree

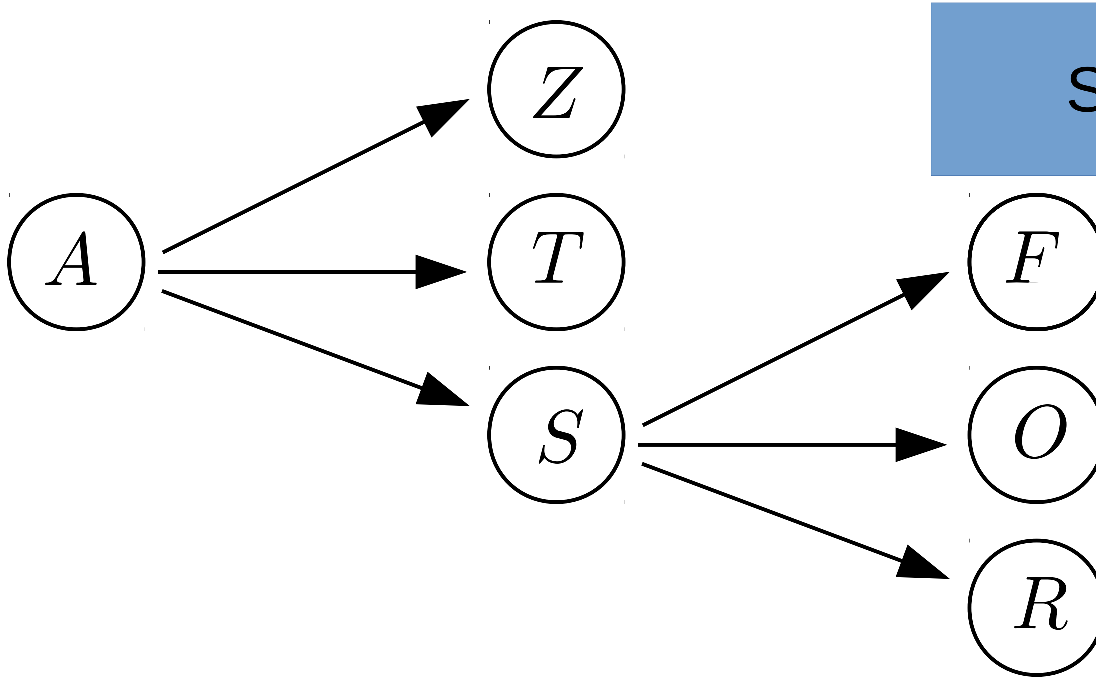




A was already visited!

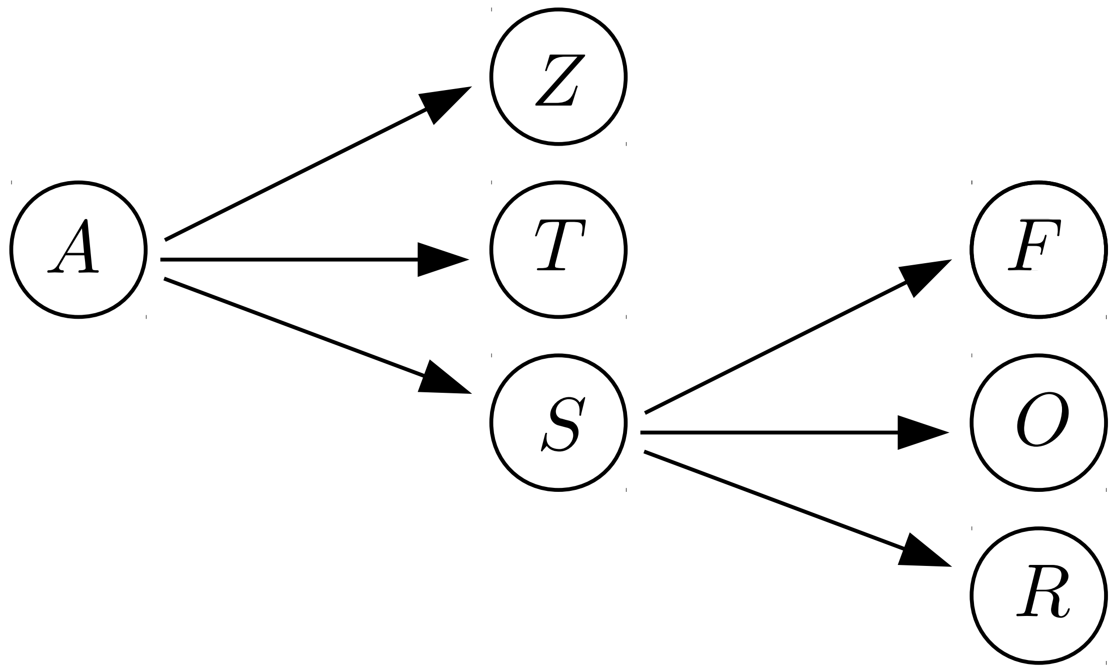
A search tree





So, prune it!

A search tree

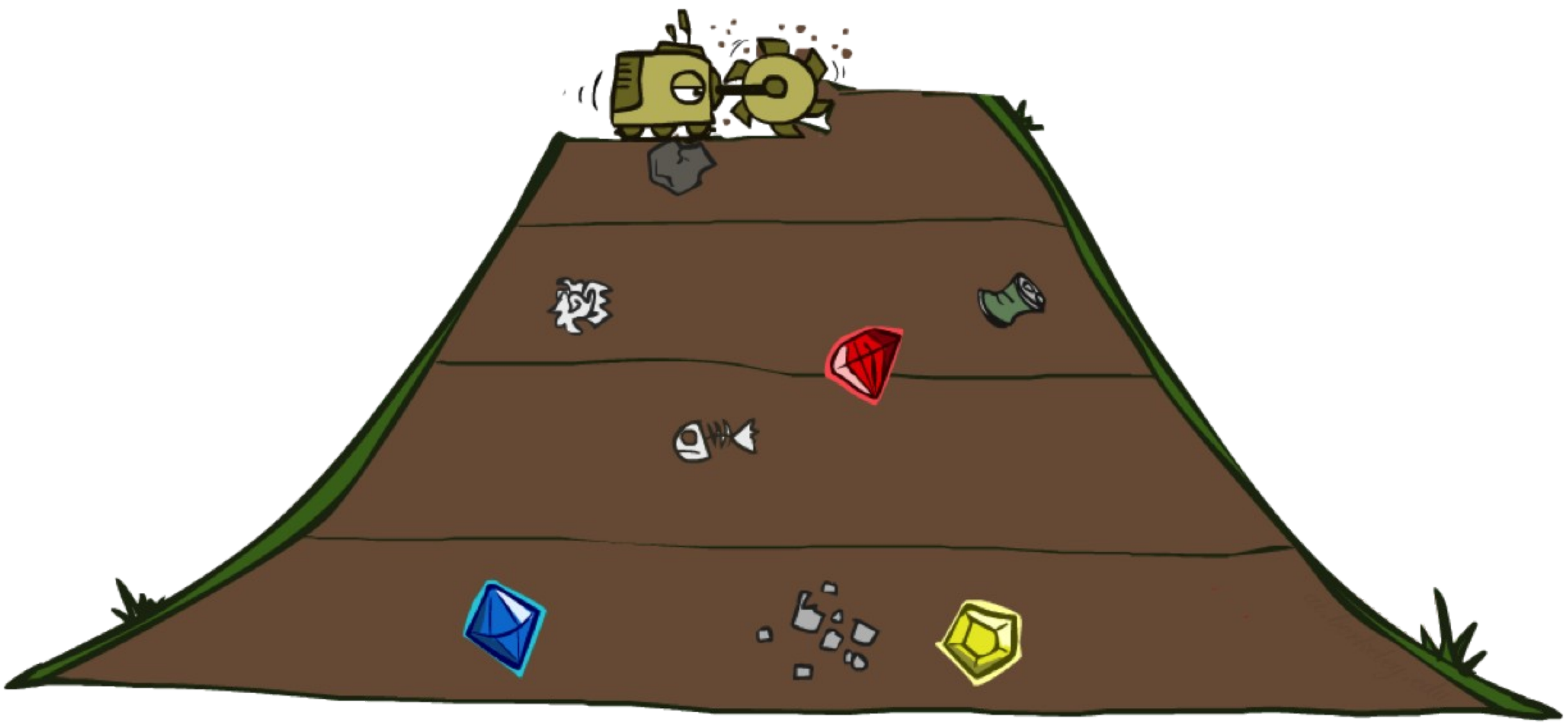


In what order should we expand states?

– here, we expanded *S*, but we could also have expanded *Z* or *T*

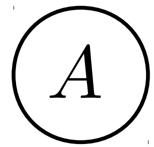
– different search algorithms expand in different orders

Breadth first search (BFS)

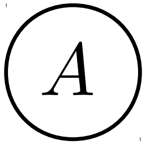


Slide: Adapted from Berkeley CS188 course notes (downloaded Summer 2015)

Breadth first search (BFS)

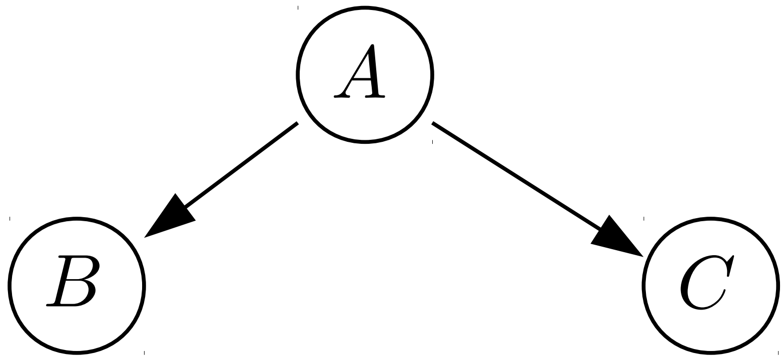


Breadth first search (BFS)

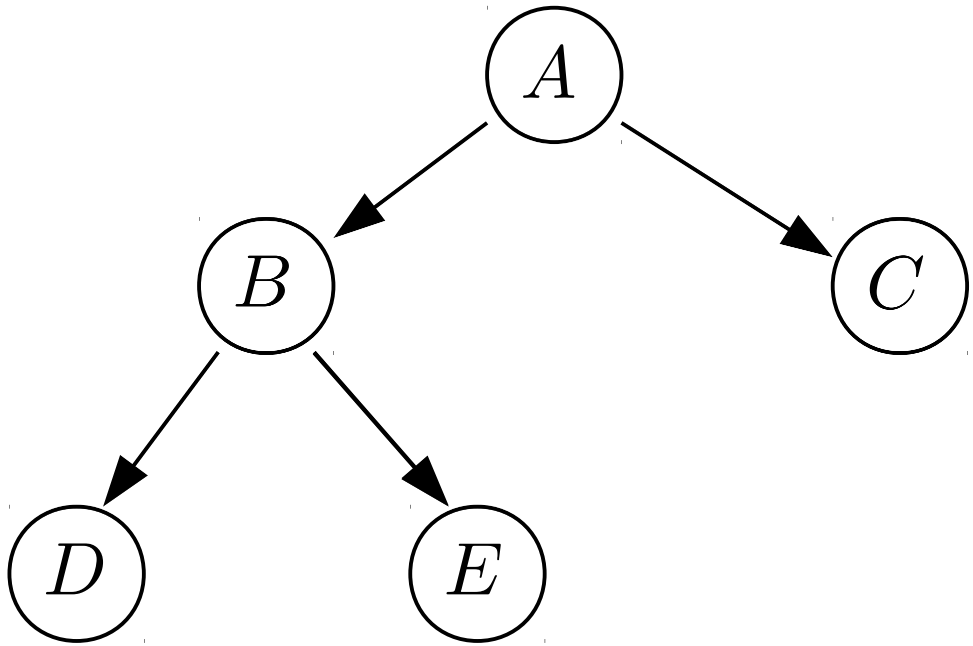


Start node

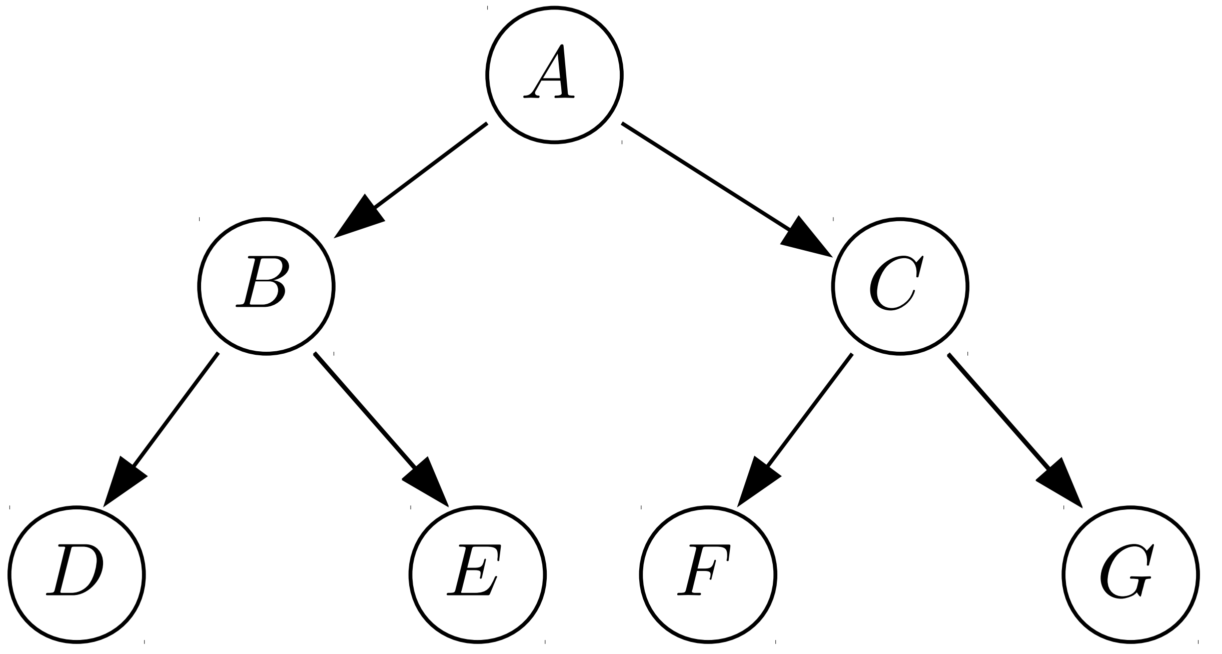
Breadth first search (BFS)



Breadth first search (BFS)



Breadth first search (BFS)



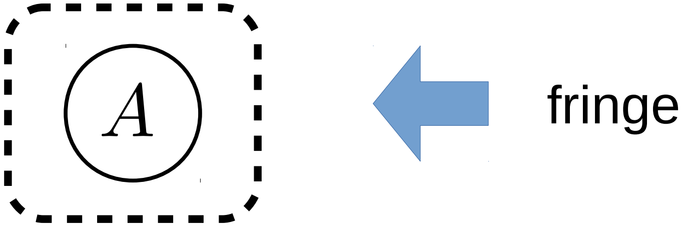
Breadth first search (BFS)

<u>Fringe</u>

We're going to maintain a queue called the <u>fringe</u>

– initialize the fringe as an empty queue

Breadth first search (BFS)

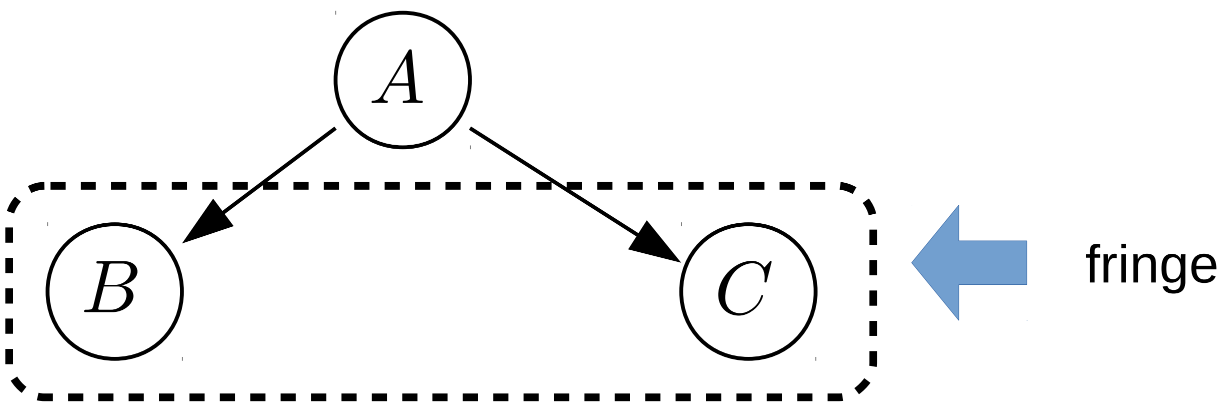


<u>Fringe</u>
A

– add *A* to the fringe

Breadth first search (BFS)

<u>Fringe</u>
B
C

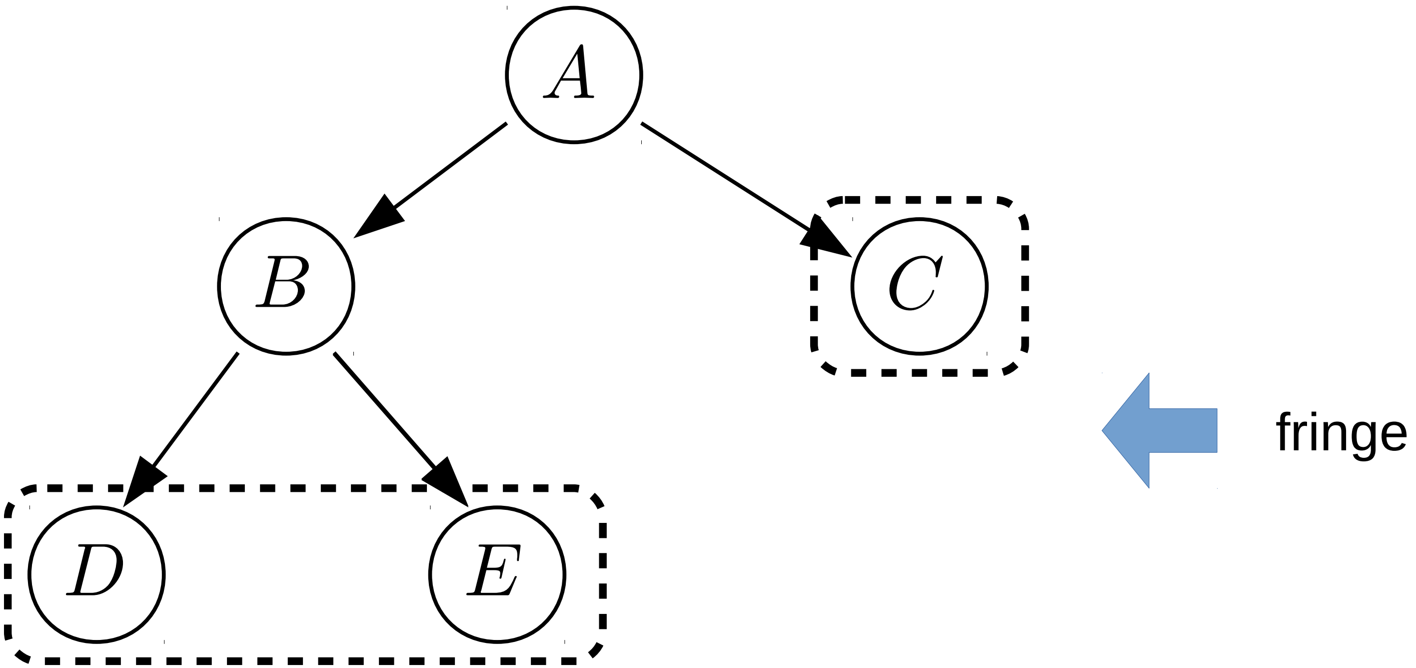


-- remove *A* from the fringe

-- add successors of *A* to the fringe

Breadth first search (BFS)

<u>Fringe</u>
C
D
E

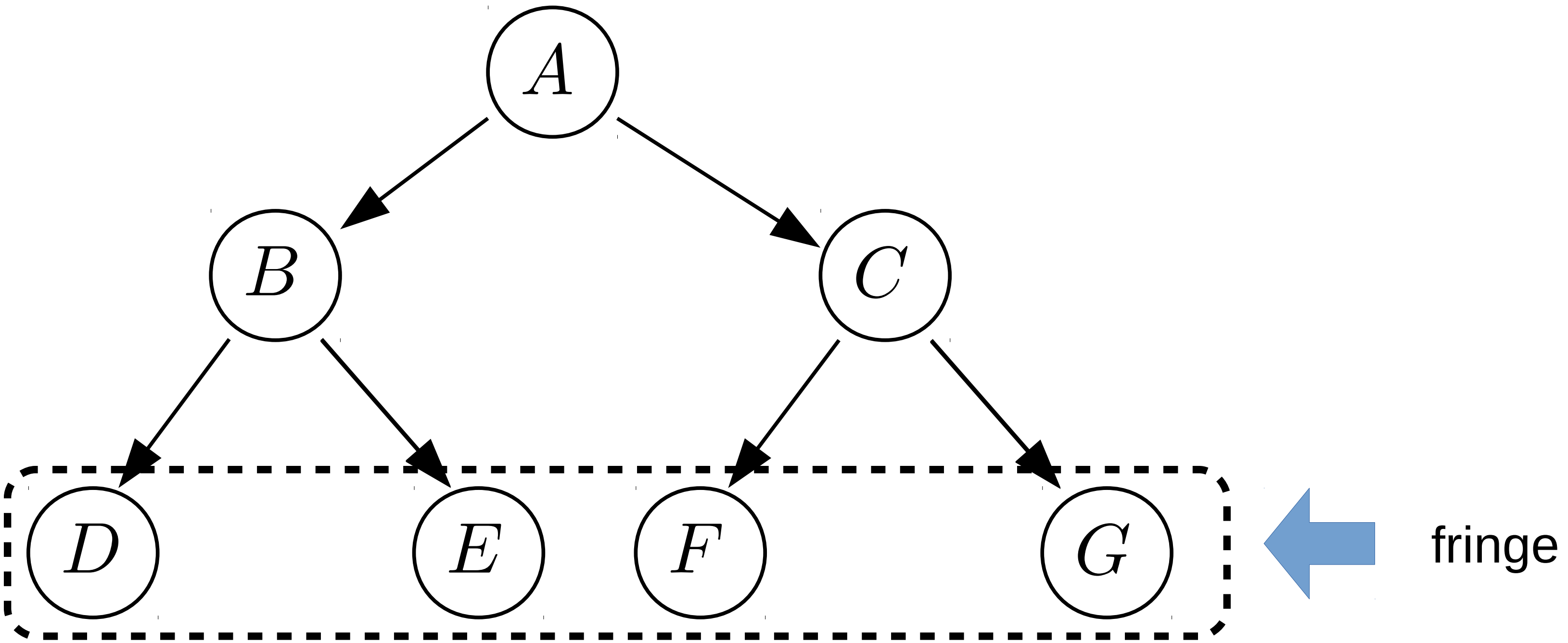


-- remove *B* from the fringe

-- add successors of *B* to the fringe

Breadth first search (BFS)

Fringe
D
E
F
G

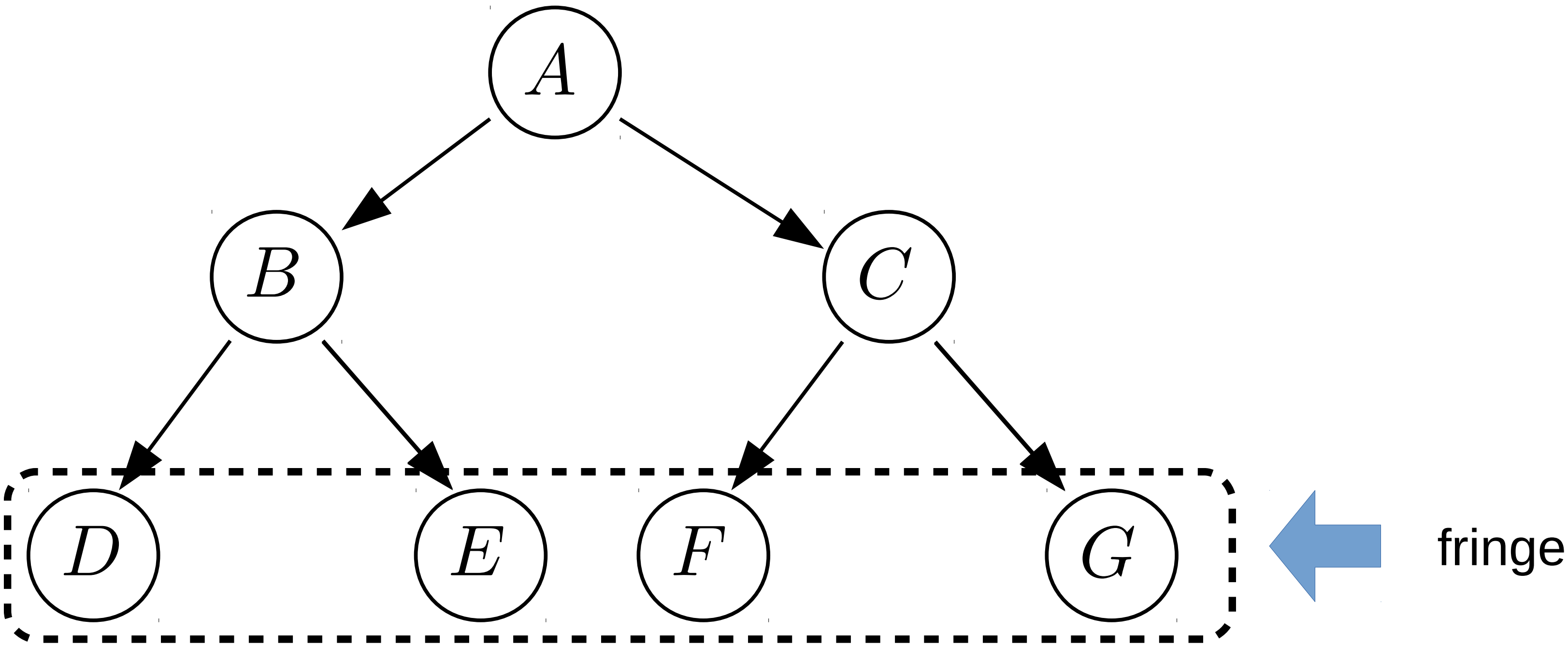


-- remove C from the fringe

-- add successors of C to the fringe

Breadth first search (BFS)

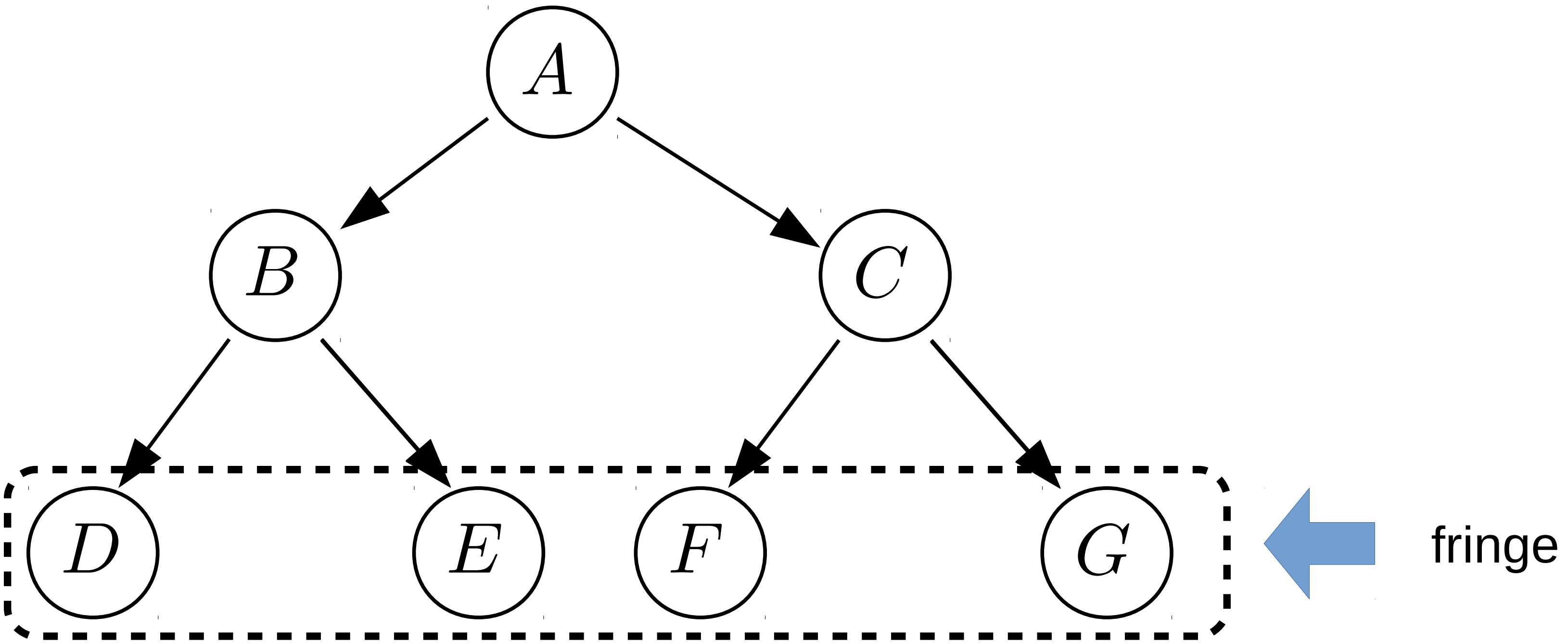
Fringe
D
E
F
G



Which state gets removed next from the fringe?

Breadth first search (BFS)

<u>Fringe</u>
D
E
F
G

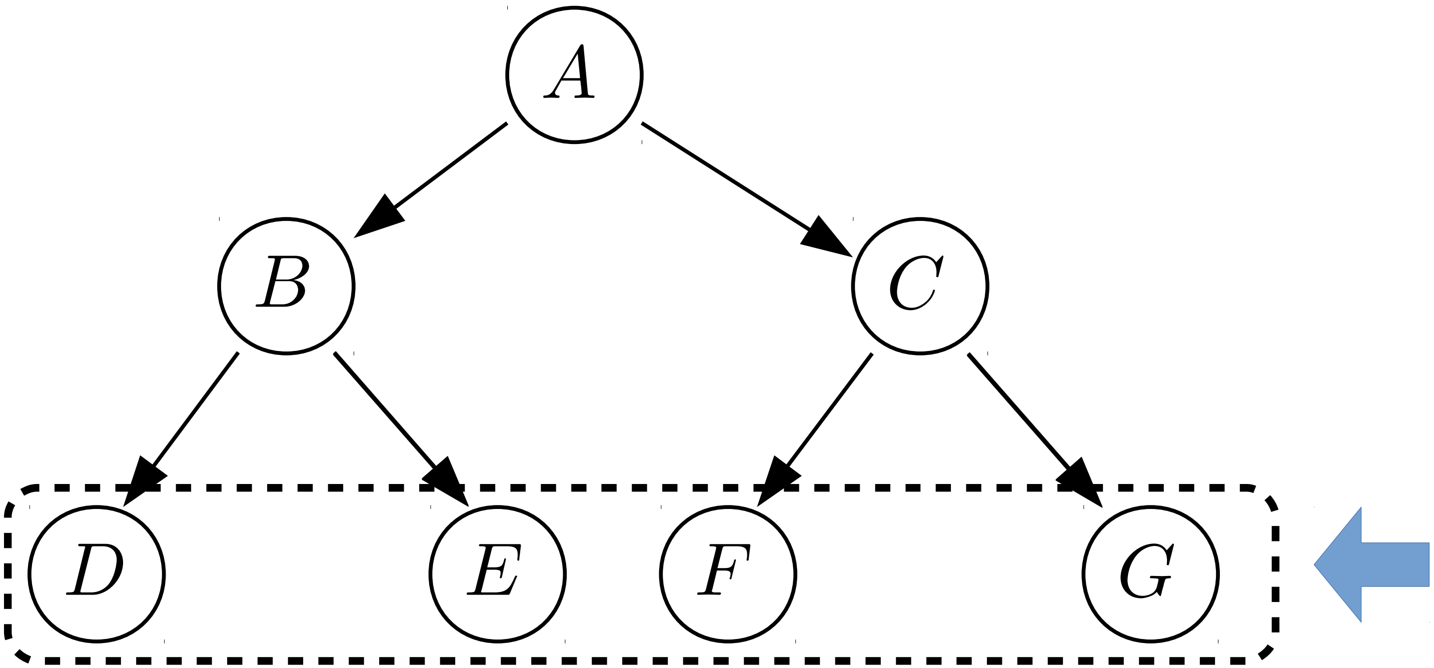


Which state gets removed next from the fringe?

What kind of a queue is this?

Breadth first search (BFS)

<u>Fringe</u>
D
E
F
G



fringe

Which state gets removed next from the fringe?

What kind of a queue is this?

FIFO Queue!
(first in first out)

Breadth first search (BFS)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Breadth first search (BFS)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

What is the purpose of the *explored* set?

BFS Properties

Is BFS <u>complete</u>?
– is it guaranteed to find a solution if one exists?

BFS Properties

Is BFS complete?
– is it guaranteed to find a solution if one exists?

What is the time complexity of BFS?
– how many states are expanded before finding a sol'n?
  - – b: branching factor
  - – d: depth of shallowest solution
  - – complexity = ???

BFS Properties

Is BFS <u>complete</u>?
– is it guaranteed to find a solution if one exists?

What is the <u>time complexity</u> of BFS?
– how many states are expanded before finding a sol'n?
  – b: branching factor
  – d: depth of shallowest solution
  – complexity = $O(b^d)$

BFS Properties

Is BFS complete?
– is it guaranteed to find a solution if one exists?

What is the time complexity of BFS?
– how many states are expanded before finding a sol'n?
  – b: branching factor
  – d: depth of shallowest solution
  – complexity = $O(b^d)$

What is the space complexity of BFS?
– how much memory is required?
  – complexity = ???

BFS Properties

Is BFS <u>complete</u>?
– is it guaranteed to find a solution if one exists?

What is the <u>time complexity</u> of BFS?
– how many states are expanded before finding a sol'n?
 – b: branching factor
 – d: depth of shallowest solution
 – complexity = $O(b^d)$

What is the <u>space complexity</u> of BFS?
– how much memory is required?
 – complexity = $O(b^d)$

BFS Properties

Is BFS complete?
– is it guaranteed to find a solution if one exists?

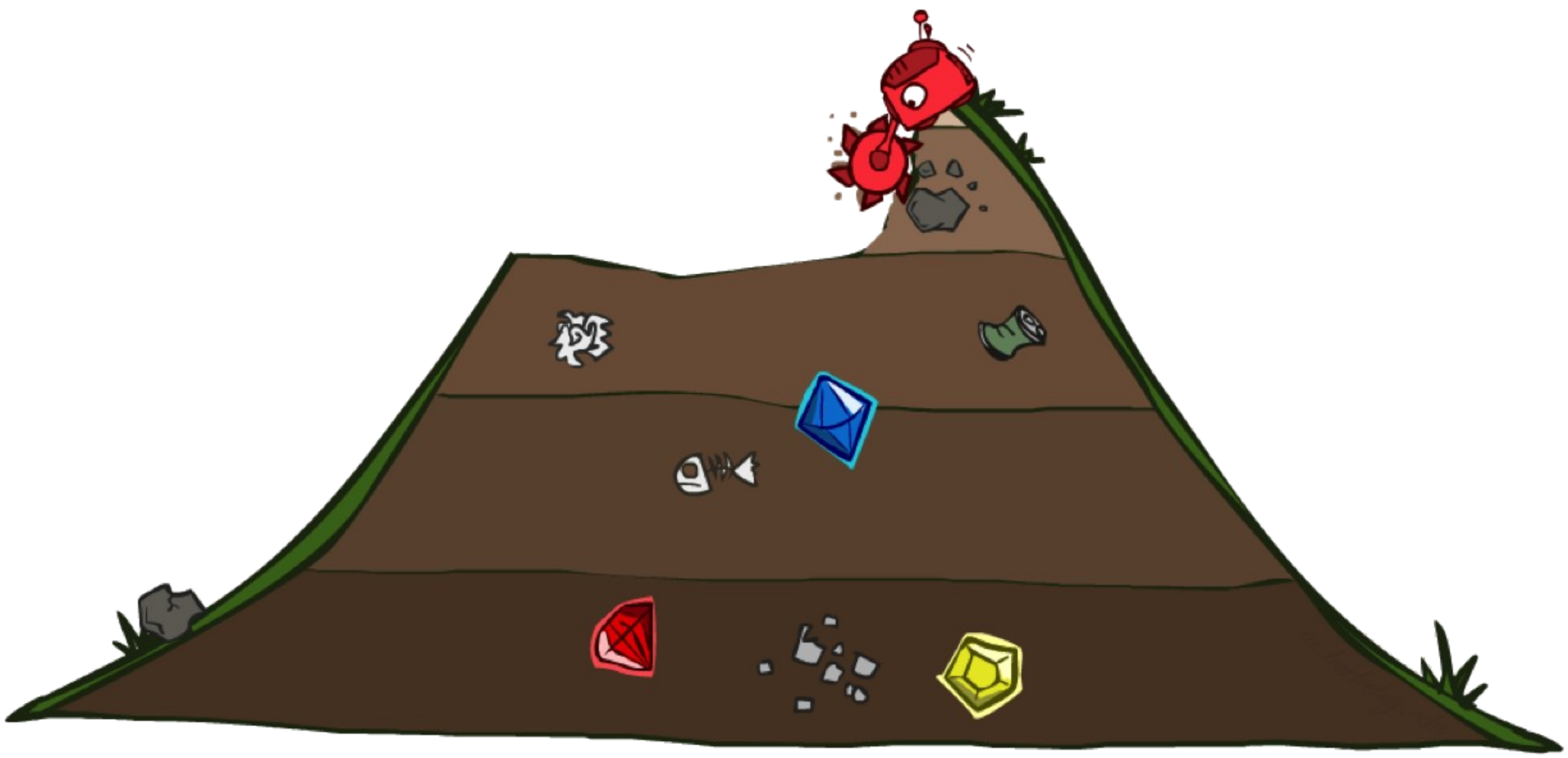
What is the time complexity of BFS?
– how many states are expanded before finding a sol'n?
– b: branching factor
– d: depth of shallowest solution
– complexity = $O(b^d)$

What is the space complexity of BFS?
– how much memory is required?
– complexity = $O(b^d)$

Is BFS optimal?
– is it guaranteed to find the best solution (shortest path)?

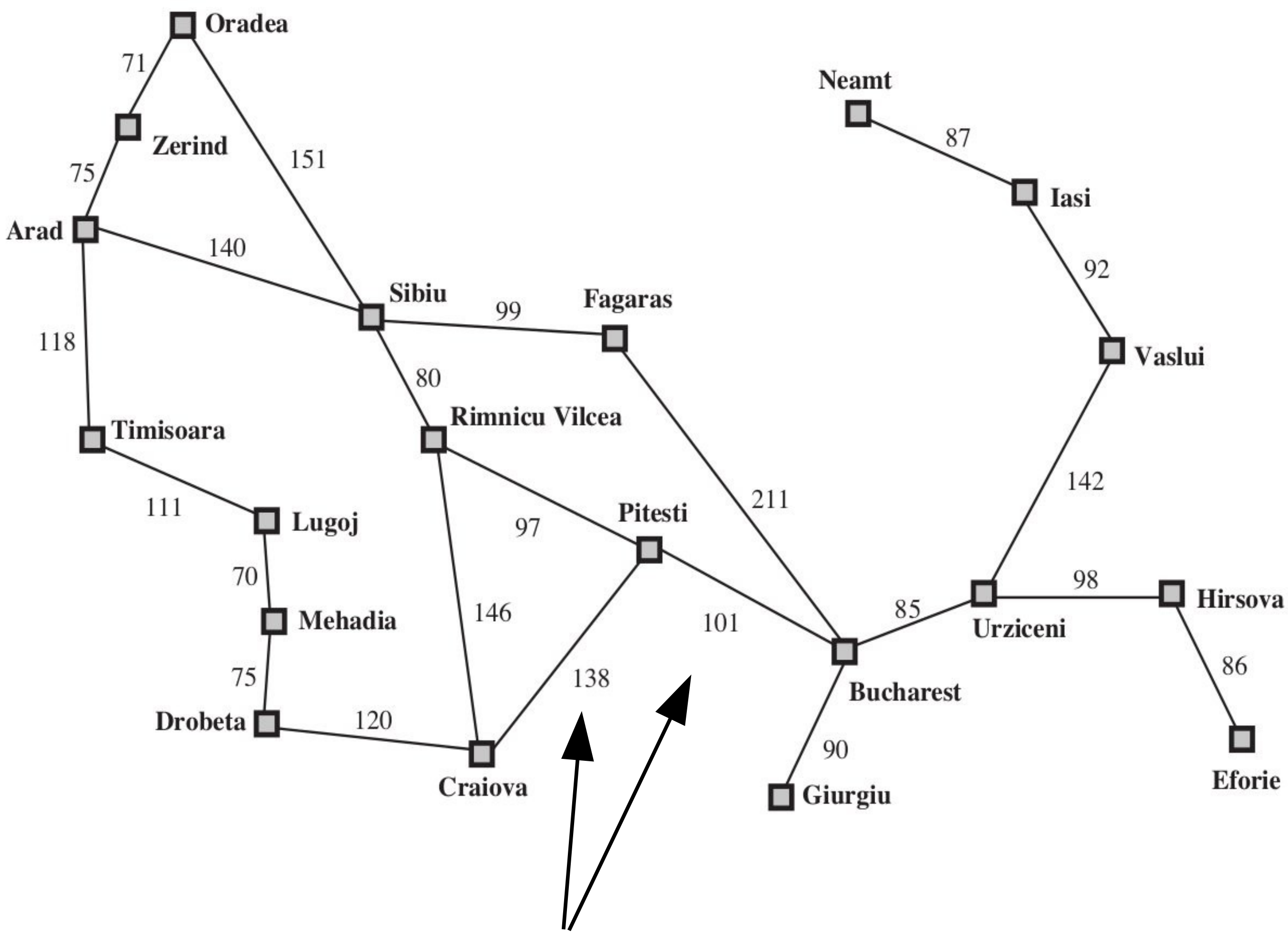
Another BFS example...

Uniform Cost Search (UCS)



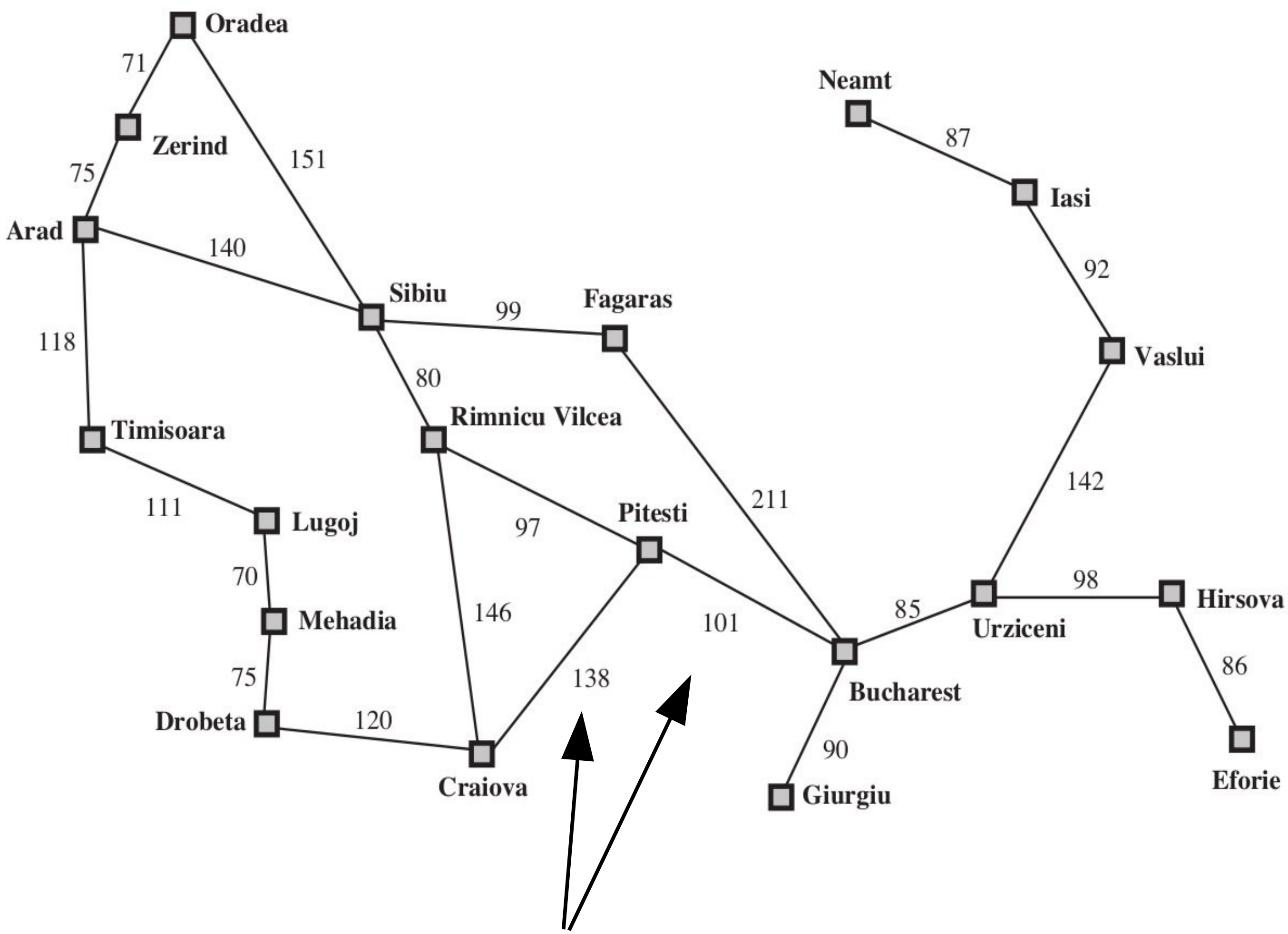
Slide: Adapted from Berkeley CS188 course notes (downloaded Summer 2015)

Uniform Cost Search (UCS)



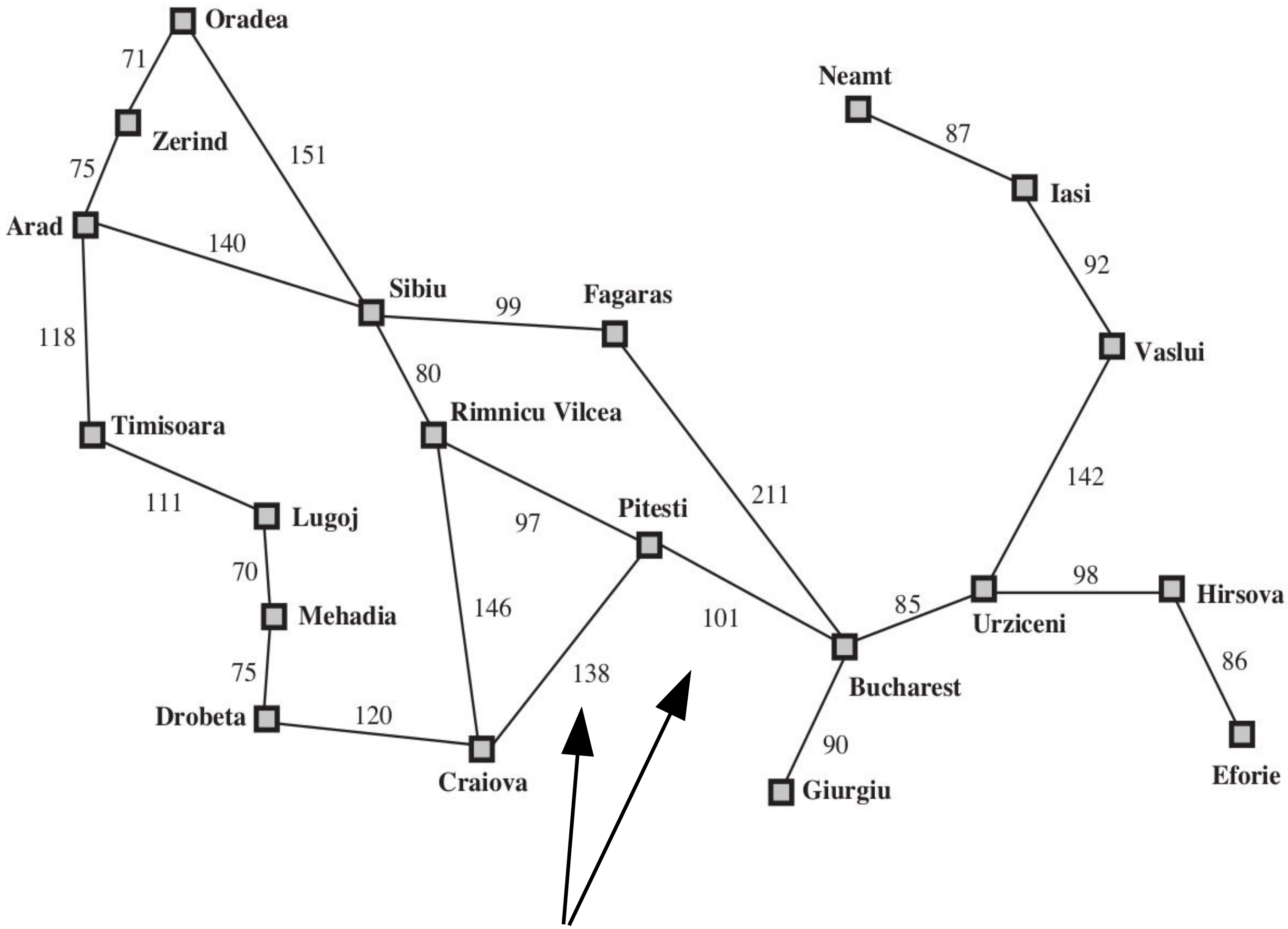
Notice the distances between cities

Uniform Cost Search (UCS)



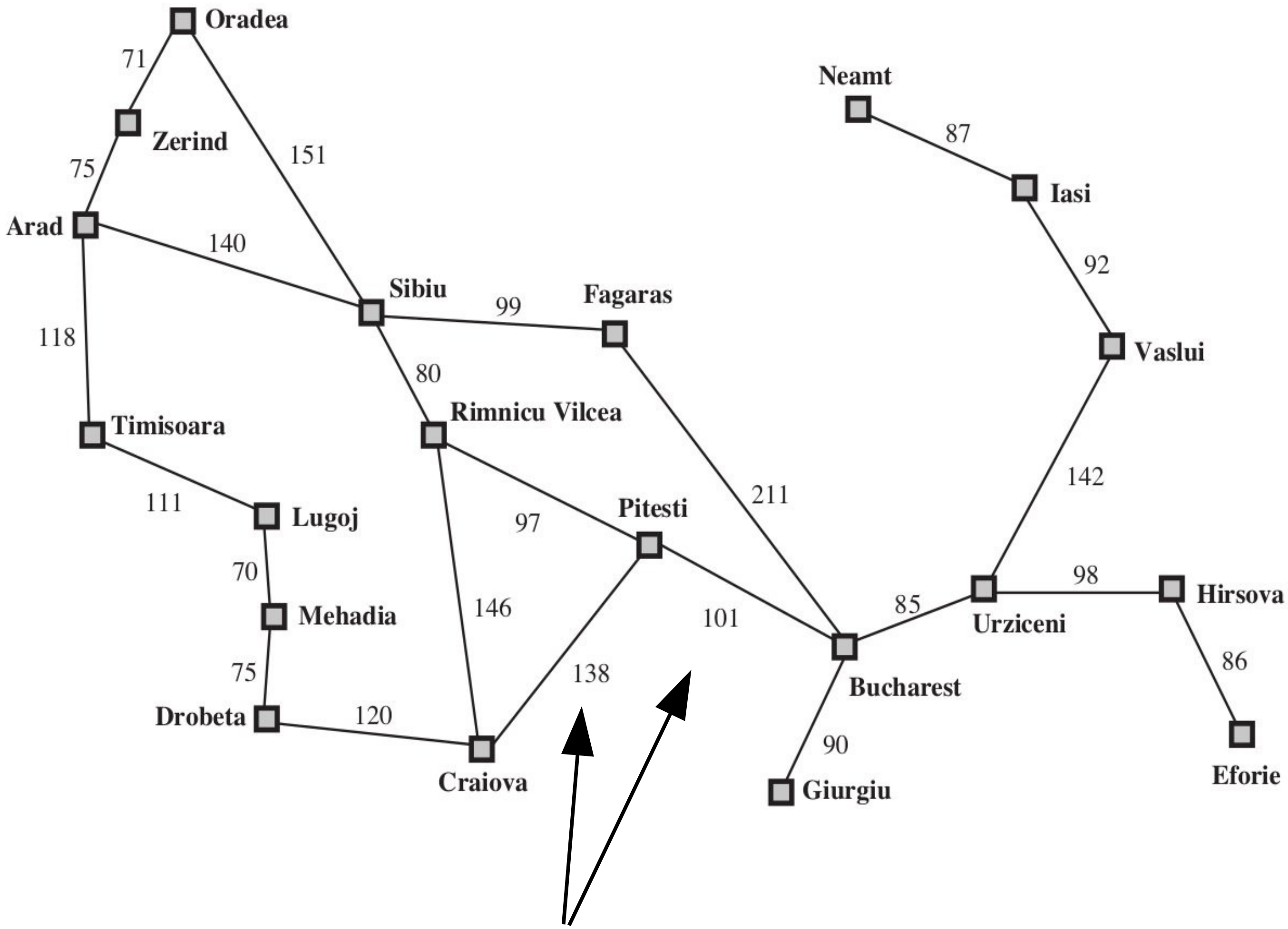
Notice the distances between cities
– does BFS take these distances into account?

Uniform Cost Search (UCS)



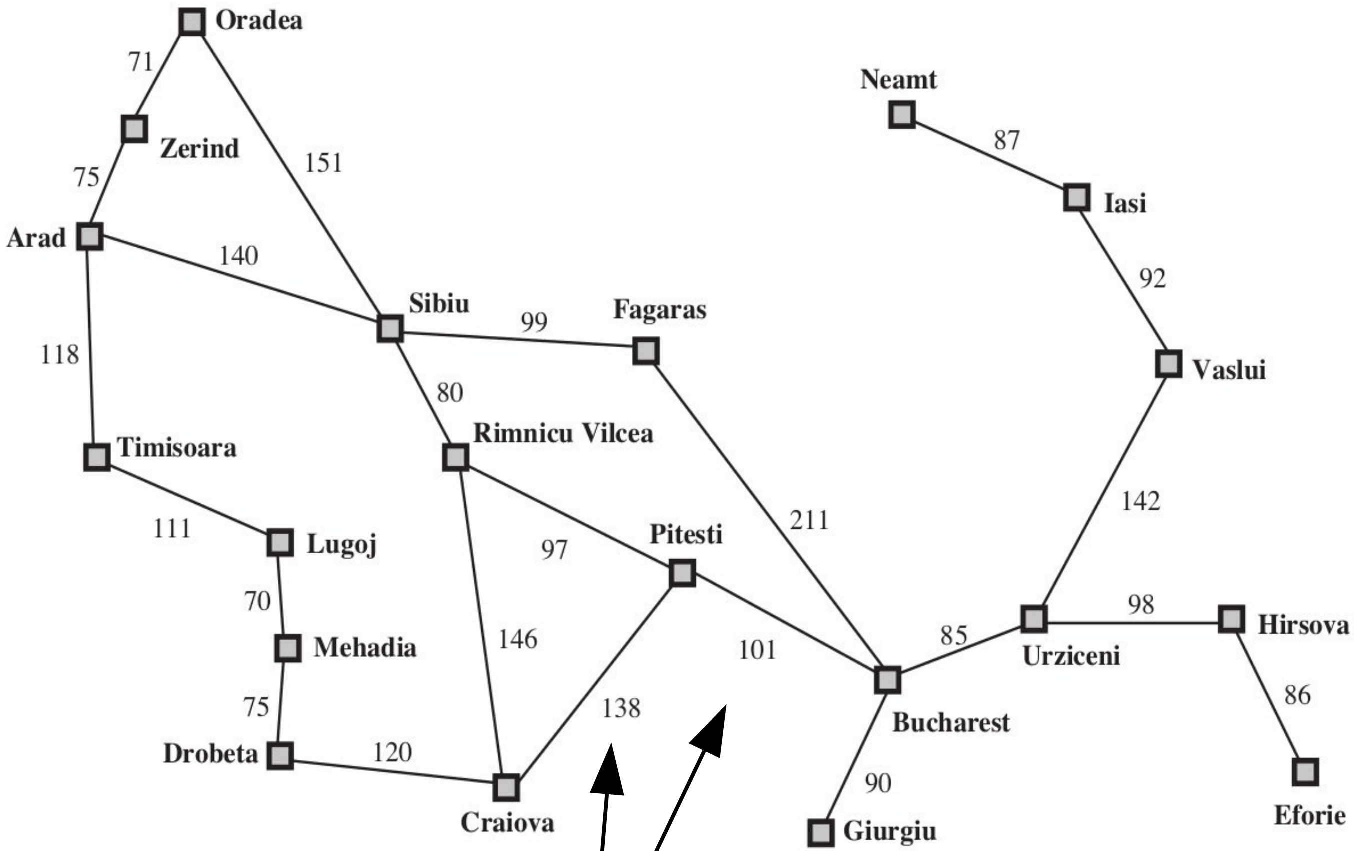
Notice the distances between cities
– does BFS take these distances into account?
– does BFS find the path w/ shortest milage?

Uniform Cost Search (UCS)



Notice the distances between cities
– does BFS take these distances into account?
– does BFS find the path w/ shortest milage?
– compare S-F-B with S-R-P-B. Which costs less?

Uniform Cost Search (UCS)



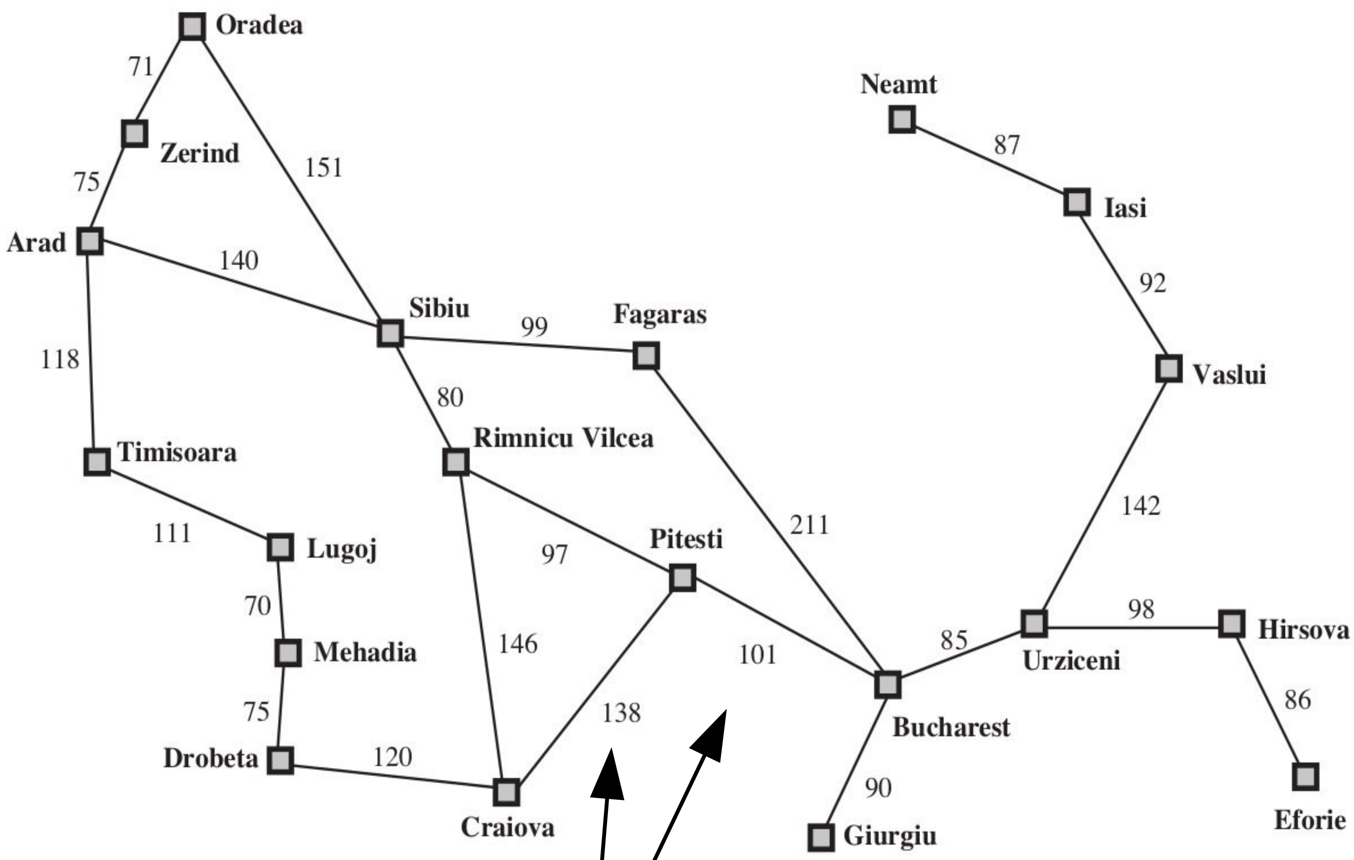
Notice
– doe
– doe
– com

How do we fix this?

nt?

less?

Uniform Cost Search (UCS)



Notice
– doe nt?
– doe
– com less?

How do we fix this?
UCS!

Uniform Cost Search (UCS)

Same as BFS except: expand node w/ smallest <u>path cost</u>

Length of path

Uniform Cost Search (UCS)

Same as BFS except: expand node w/ smallest path cost

Length of path

Cost of going from state *A* to *B*: $c(A, B)$

Minimum cost of path going from start state to *B*: $g(B)$

Uniform Cost Search (UCS)

Same as BFS except: expand node w/ smallest <u>path cost</u>

Length of path

Cost of going from state *A* to *B:* $c(A, B)$

Minimum cost of path going from start state to *B:* $g(B)$

BFS: expands states in order of hops from start

UCS: expands states in order of $g(s)$

Uniform Cost Search (UCS)

Same as BFS except: expand node w/ smallest path cost

Length of path

Cost of going from state *A* to *B:* $c(A, B)$

Minimum cost of path going from start state to *B:* $g(B)$

BFS: exp

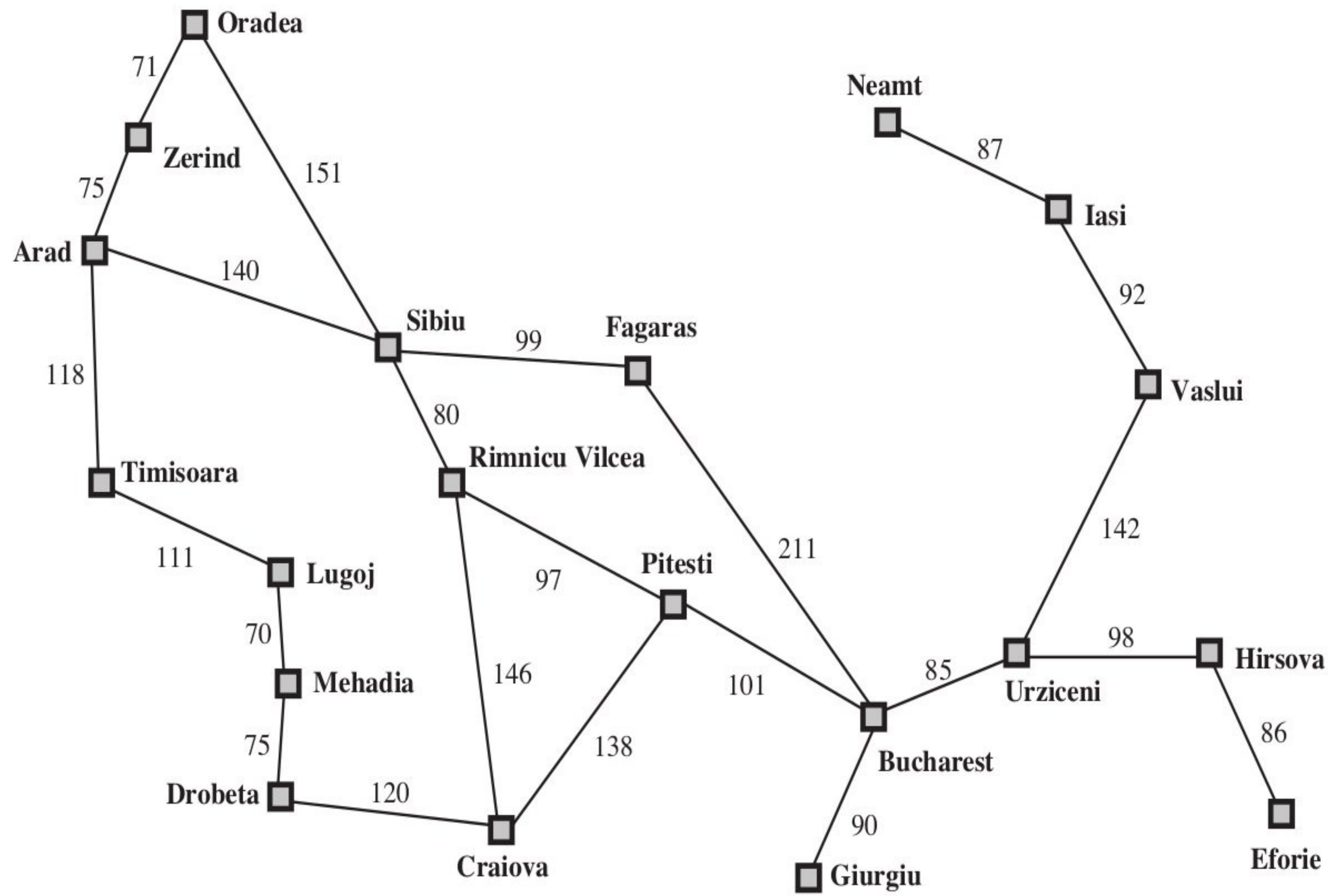
UCS: ex

How?

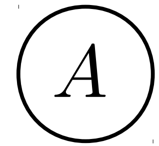
Uniform Cost Search (UCS)

Simple answer: change the FIFO to a priority queue
– the priority of each element in the queue is its path cost.

Uniform Cost Search (UCS)



UCS

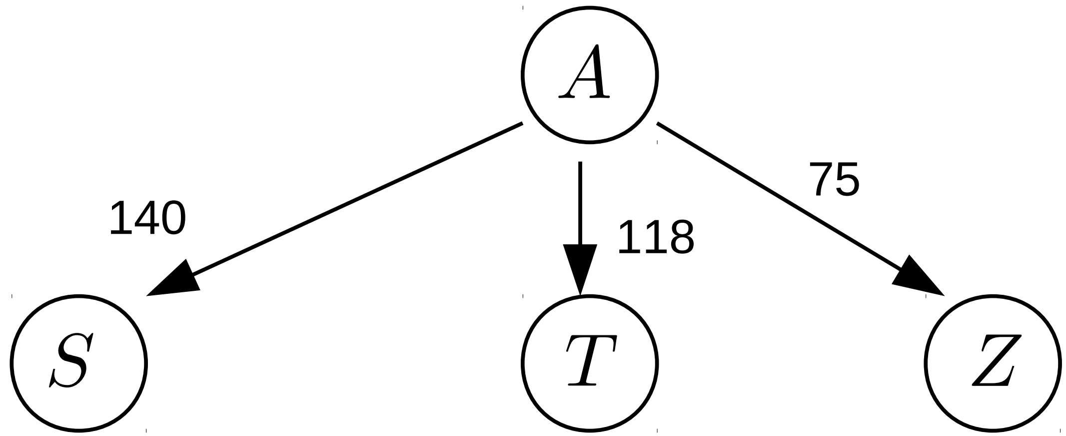


Fringe	Path Cost
A	0

Explored set:

UCS

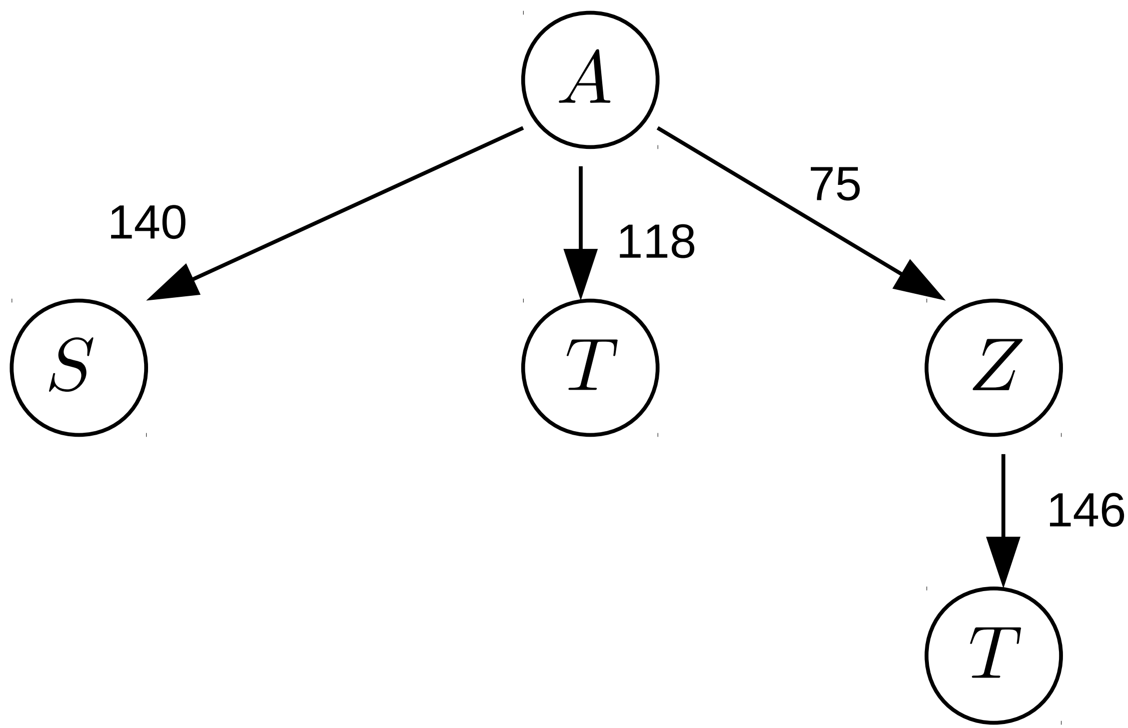
<u>Fringe</u>	<u>Path Cost</u>
~~A~~	~~0~~
S	140
T	118
Z	75



Explored set: A

UCS

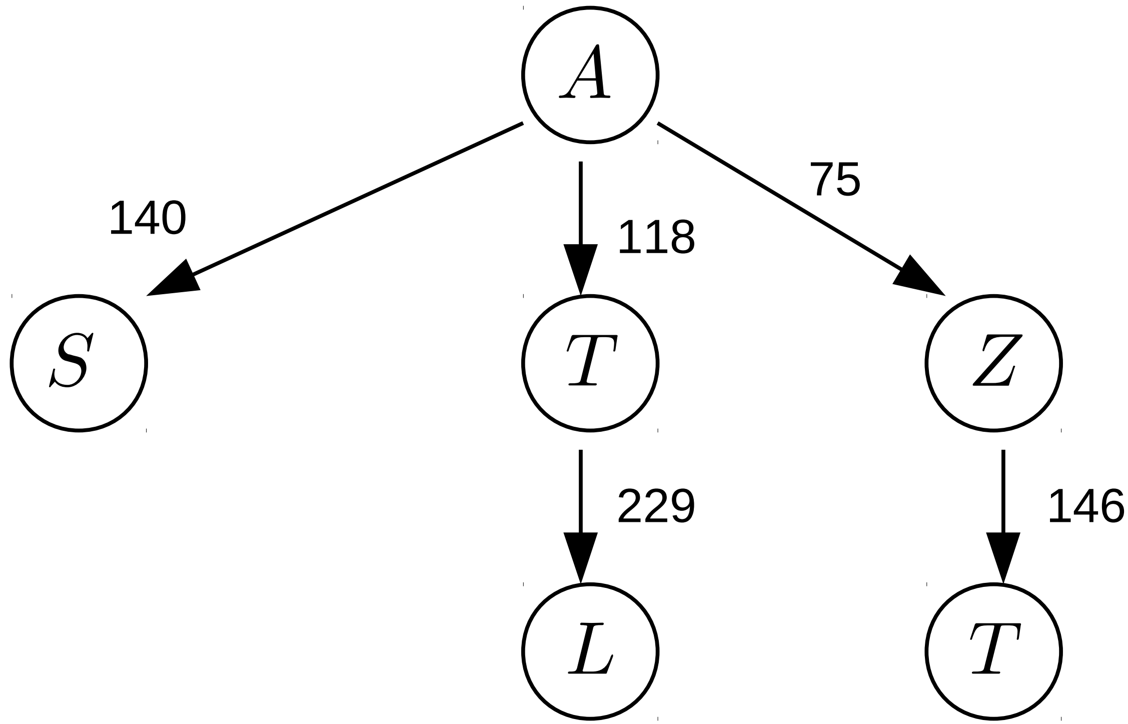
Fringe	Path Cost
~~A~~	~~0~~
S	140
T	118
~~Z~~	~~75~~
T	146



Explored set: A, Z

UCS

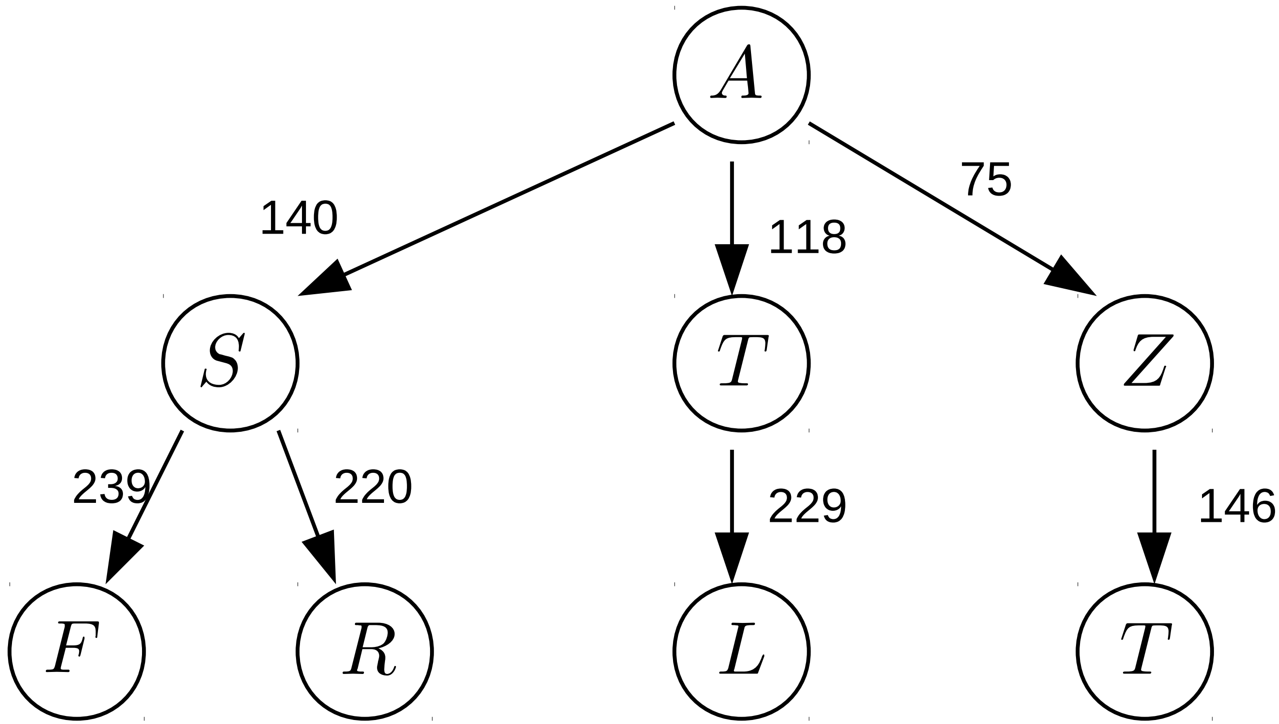
Fringe	Path Cost
~~A~~	~~0~~
S	140
~~T~~	~~118~~
~~Z~~	~~75~~
T	146
L	229



Explored set: A, Z, T

UCS

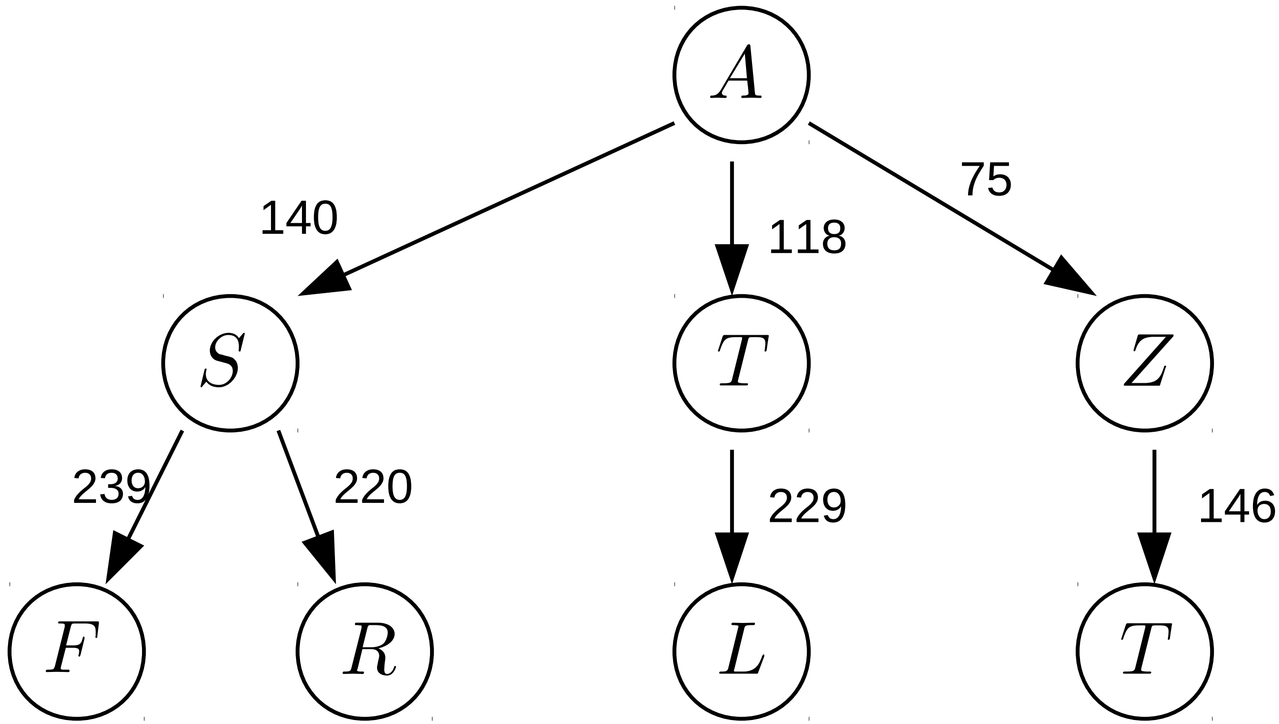
Fringe	Path Cost
~~A~~	~~0~~
~~S~~	~~140~~
~~T~~	~~118~~
~~Z~~	~~75~~
T	146
L	229
F	239
R	220



Explored set: A, Z, T, S

UCS

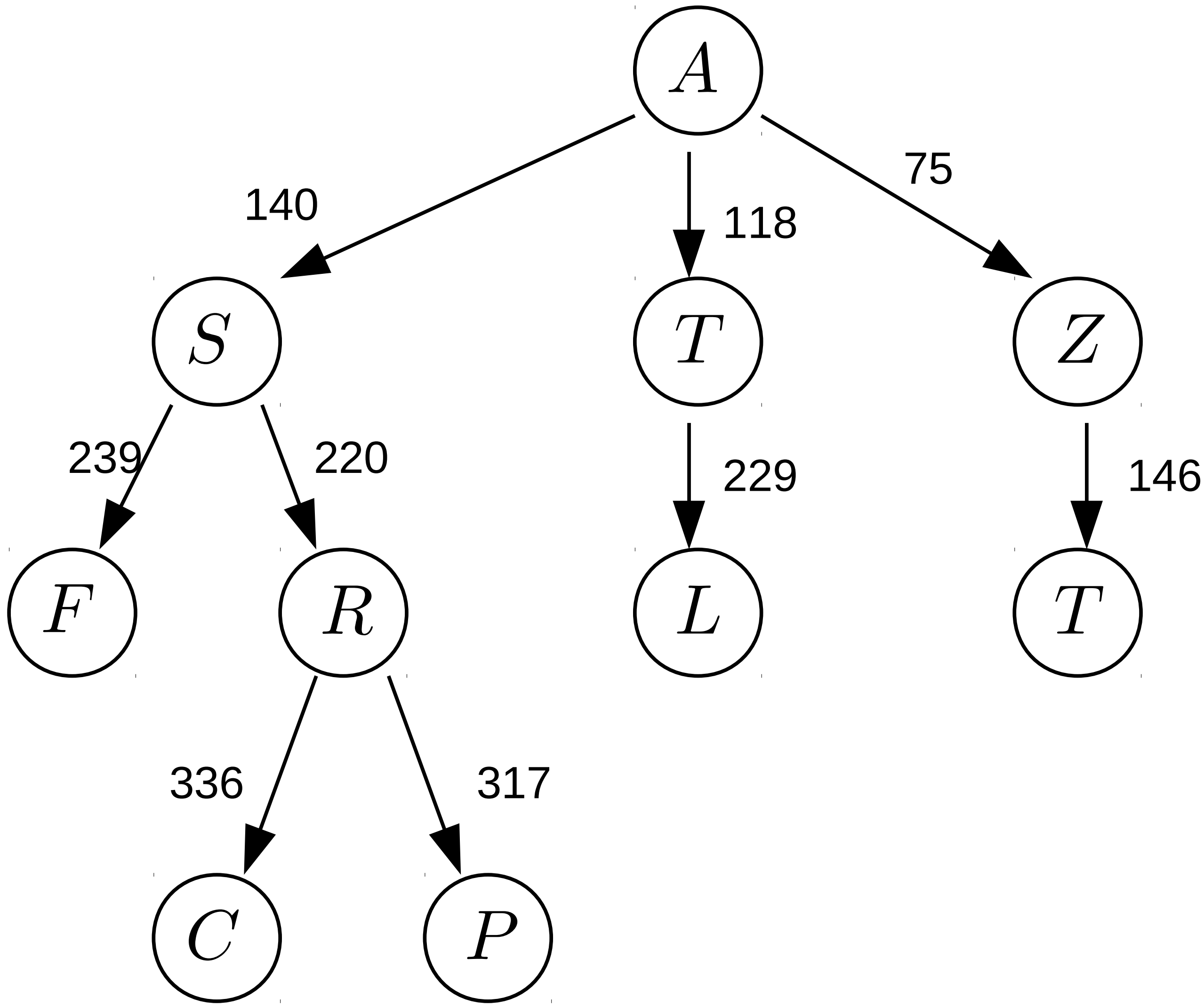
Fringe	Path Cost
~~A~~	~~0~~
~~S~~	~~140~~
~~T~~	~~118~~
~~Z~~	~~75~~
~~T~~	~~146~~
L	229
F	239
R	220



Explored set: A, Z, T, S

UCS

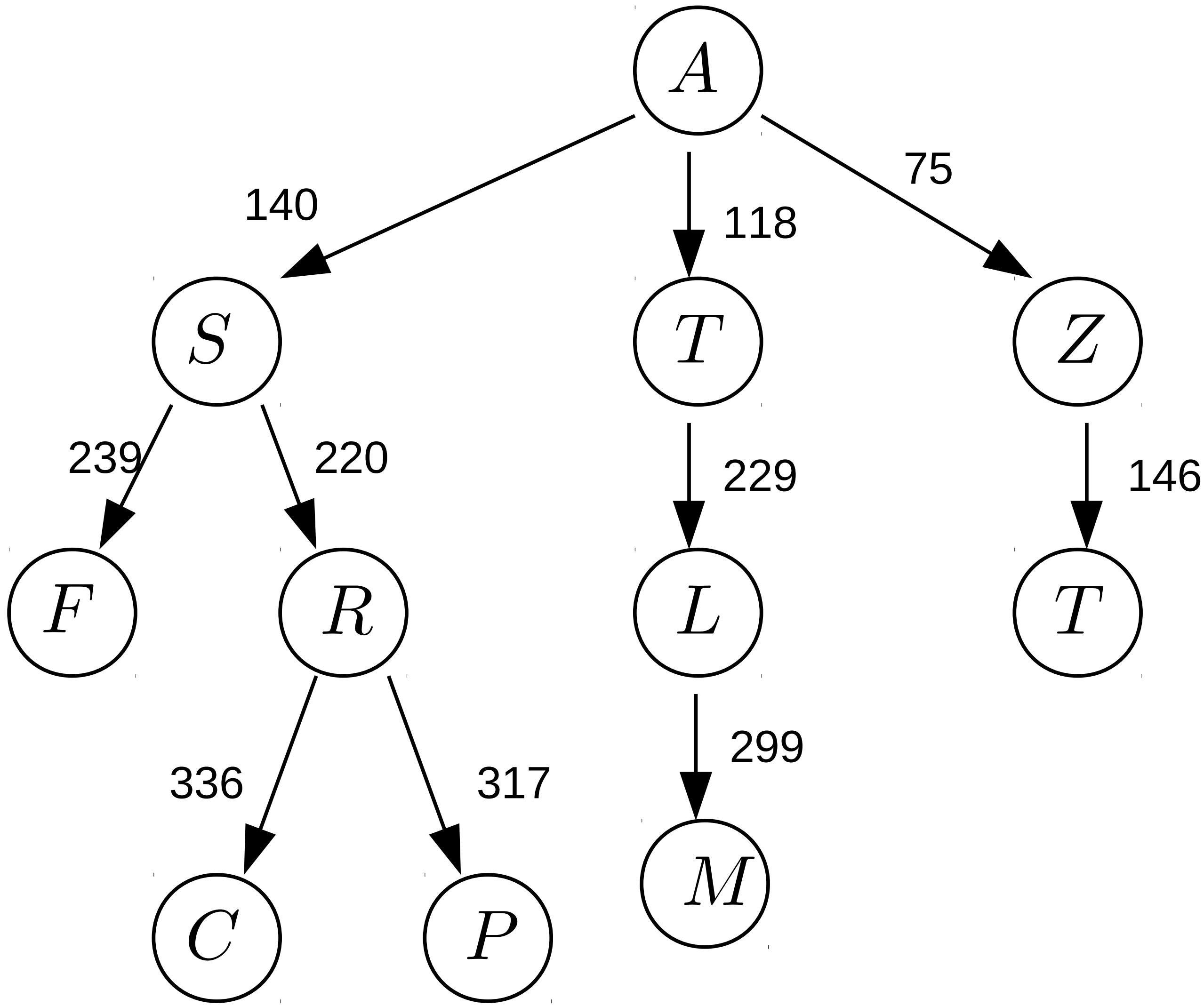
Fringe	Path Cost
~~A~~	~~0~~
~~S~~	~~140~~
~~T~~	~~118~~
~~Z~~	~~75~~
~~T~~	~~146~~
L	229
F	239
~~R~~	~~220~~
C	336
P	317



Explored set: A, Z, T, S, R

UCS

Fringe	Path Cost
~~A~~	~~0~~
~~S~~	~~140~~
~~T~~	~~118~~
~~Z~~	~~75~~
~~T~~	~~146~~
~~L~~	~~229~~
F	239
~~R~~	~~220~~
C	336
P	317
M	299

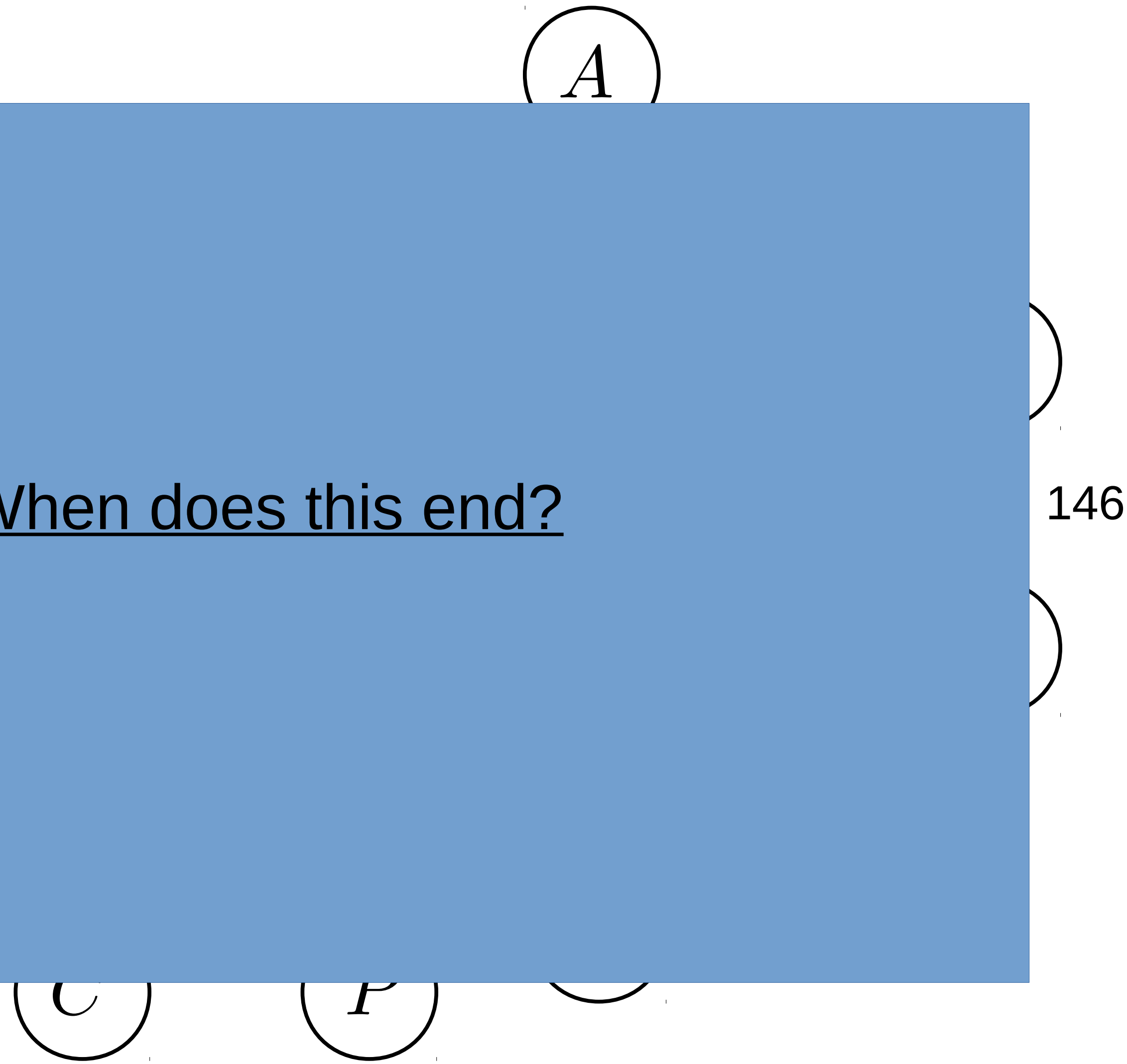


Explored set: A, Z, T, S, R, L

UCS

Fringe

~~A~~
~~S~~
~~T~~
~~Z~~
~~T~~
~~L~~
F
~~R~~
C
P
M



When does this end?

Explored set: A, Z, T, S, R, L

UCS

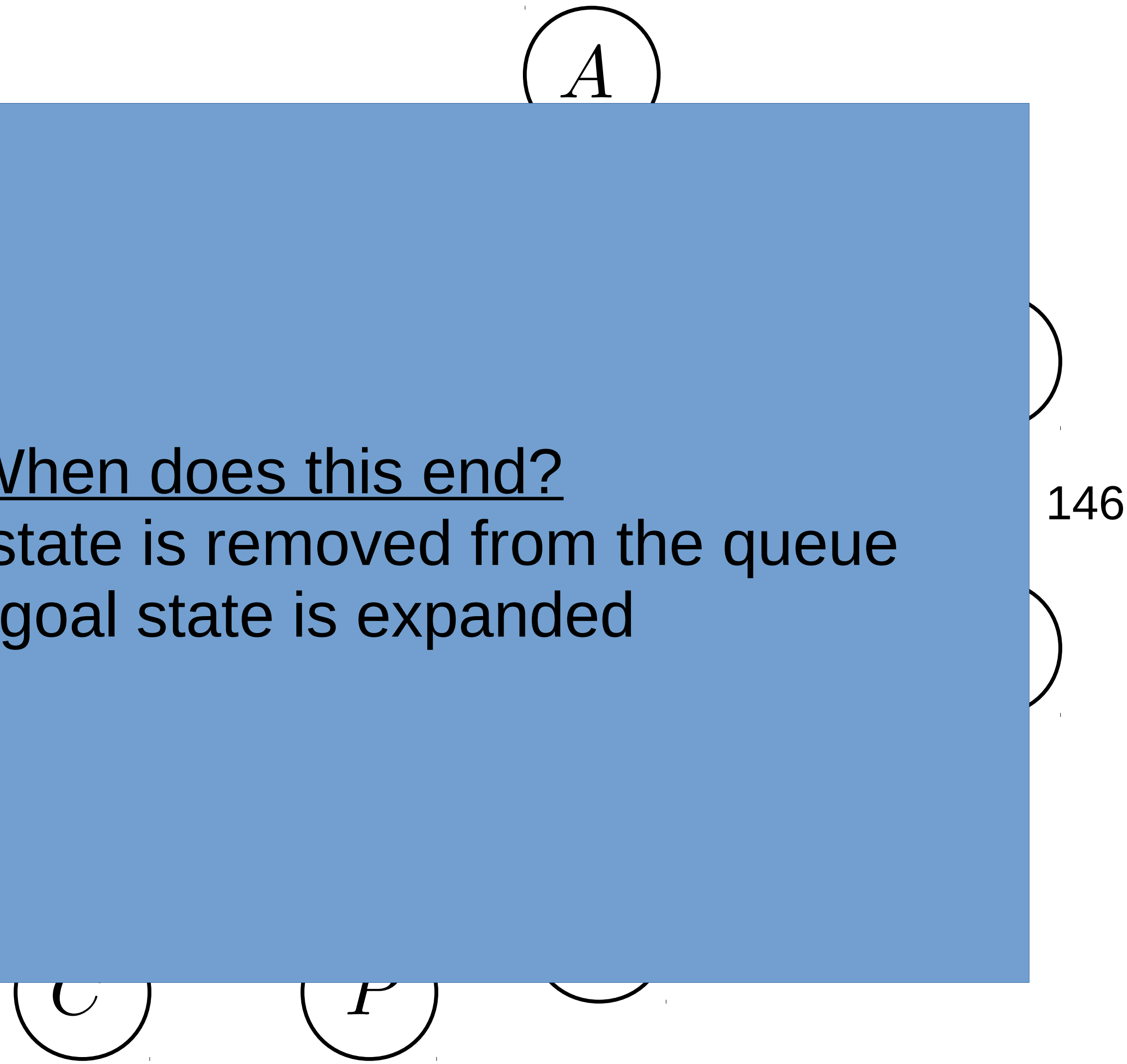
<u>Fringe</u>
~~A~~
~~S~~
~~T~~
~~Z~~
~~T~~
~~L~~
F
~~R~~
C
P
M

<u>When does this end?</u>
– when the goal state is removed from the queue

146

Explored set: A, Z, T, S, R, L

UCS



Fringe
~~A~~
~~S~~
~~T~~
~~Z~~
~~T~~
~~L~~
F
~~R~~
C
P
M

When does this end?
– when the goal state is removed from the queue
– NOT when the goal state is expanded

Explored set: A, Z, T, S, R, L

UCS

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

```
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
            replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

UCS Properties

Is UCS <u>complete</u>?
– is it guaranteed to find a solution if one exists?

What is the <u>time complexity</u> of UCS?
– how many states are expanded before finding a sol'n?
  – b: branching factor
  – C*: cost of optimal sol'n
  – e: min one-step cost
  – complexity = $O(b^{C^*/e})$

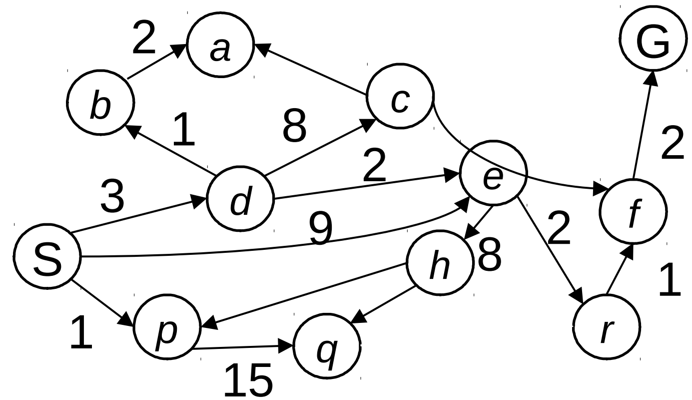
What is the <u>space complexity</u> of BFS?
– how much memory is required?
  – complexity = $O(b^{C^*/e})$

Is BFS optimal?
– is it guaranteed to find the best solution (shortest path)?

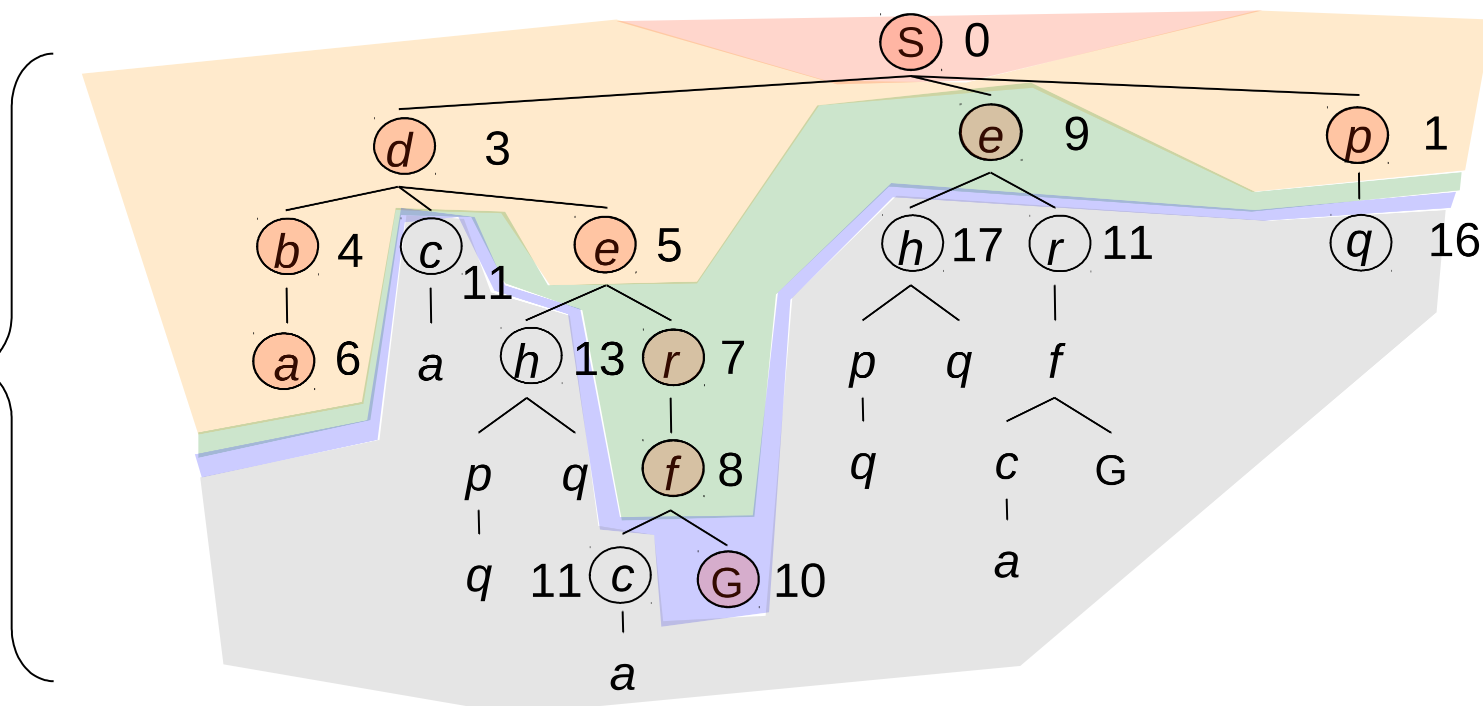
UCS vs BFS

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)



Cost contours

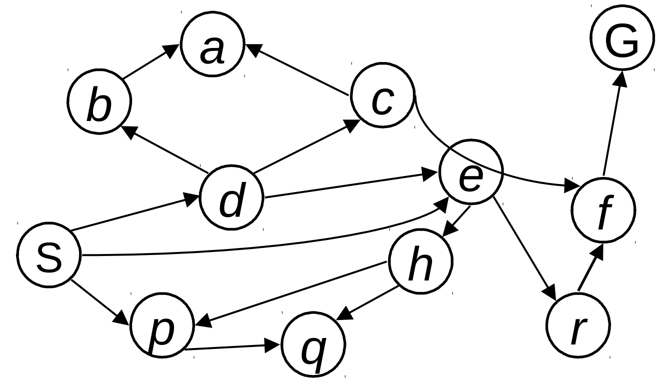


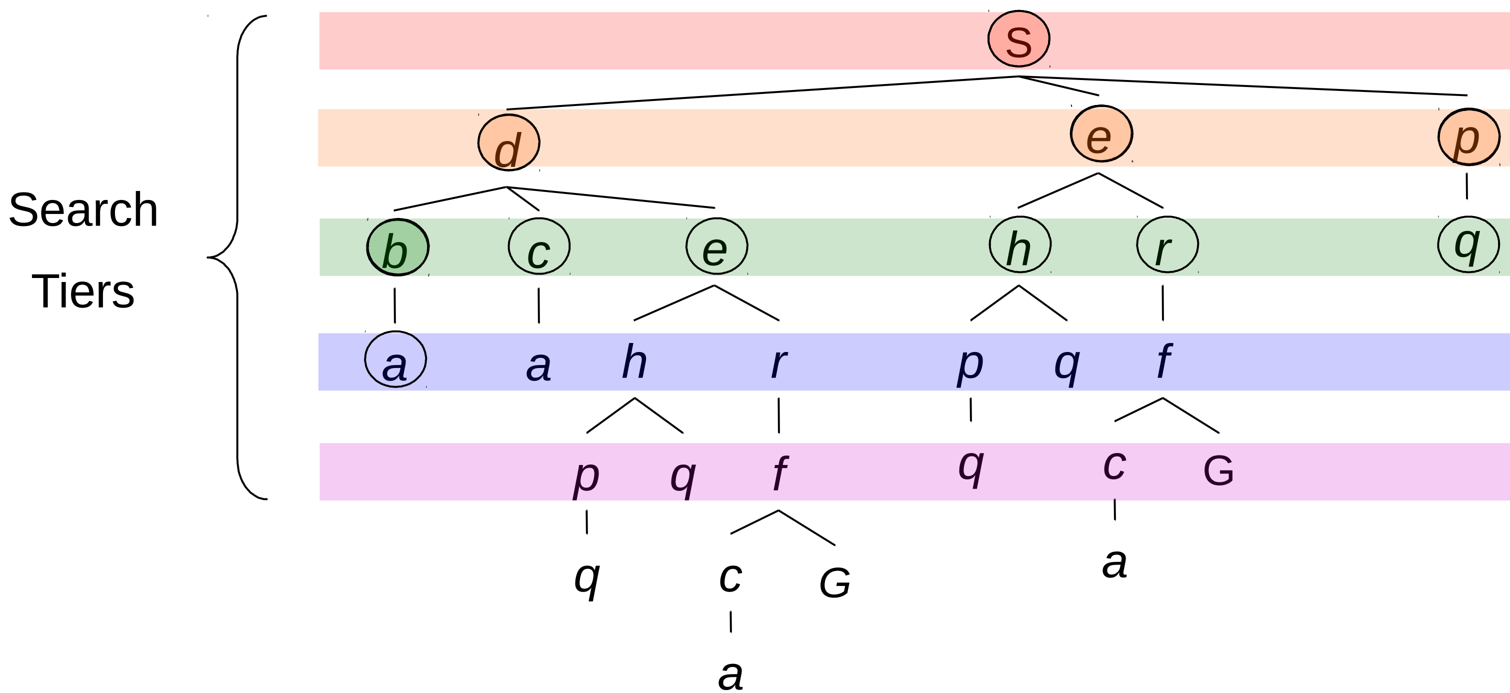
Slide: Adapted from Berkeley CS188 course notes (downloaded Summer 2015)

UCS vs BFS

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue

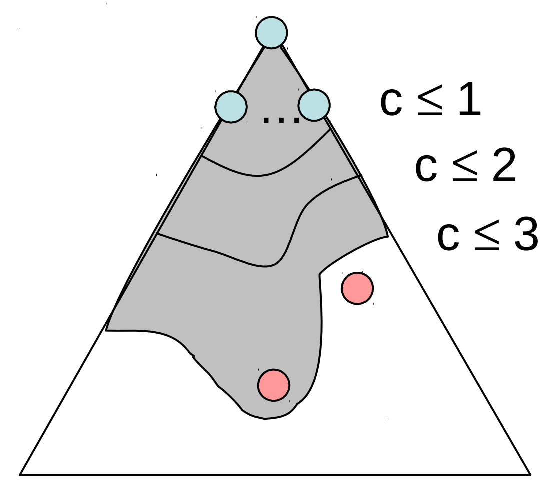


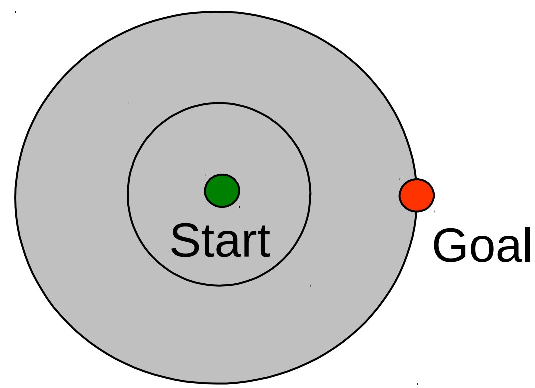


Slide: Adapted from Berkeley CS188 course notes (downloaded Summer 2015)

UCS vs BFS

- Remember: UCS explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every “direction”
  - No information about goal location
- We’ll fix that soon!





Slide: Adapted from Berkeley CS188 course notes (downloaded Summer 2015)