Introduction to Artificial Intelligence

DA 221

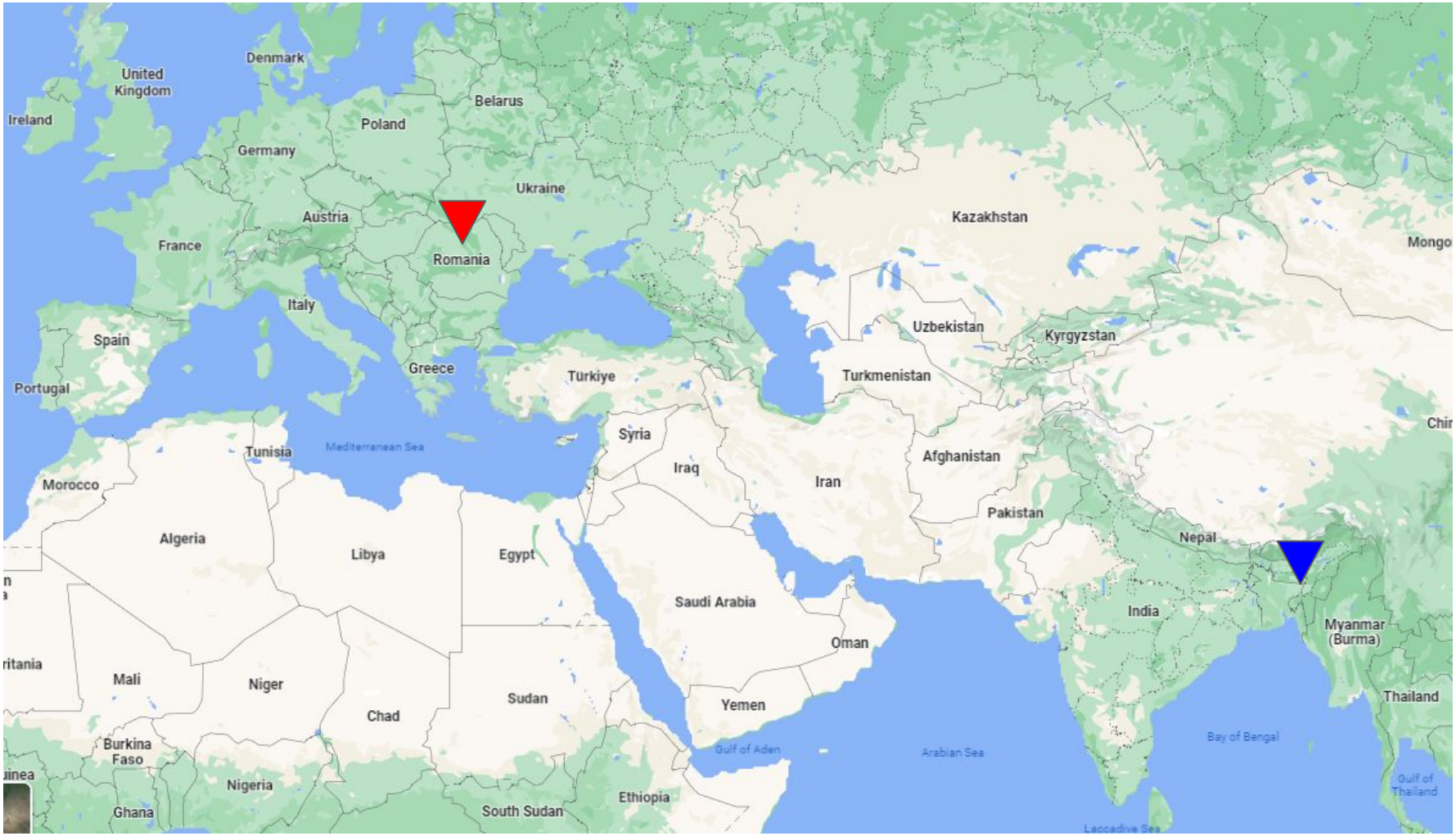
Jan - May 2023

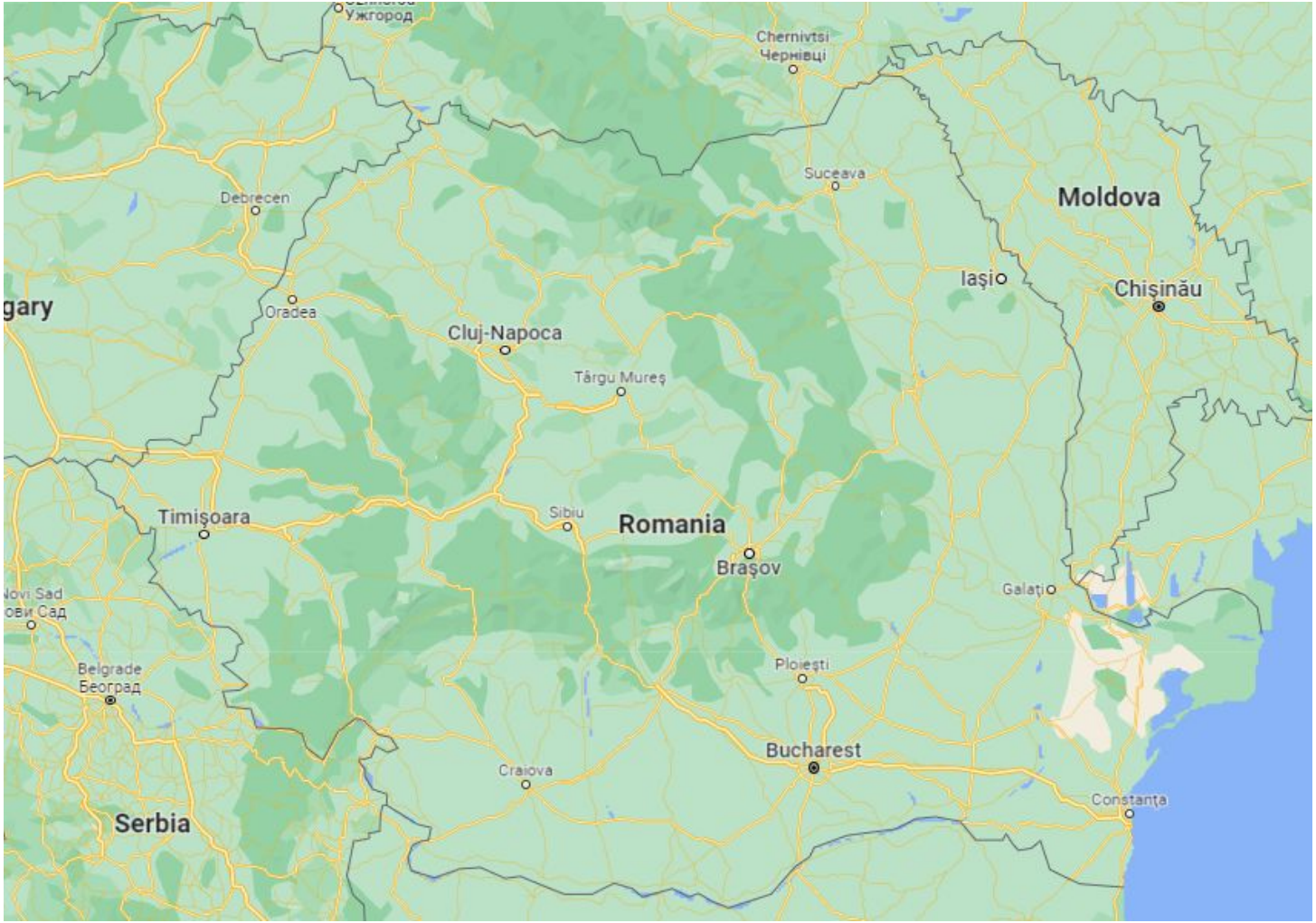
IIT Guwahati

Instructors: Neeraj Sharma (& Arghyadip Roy)

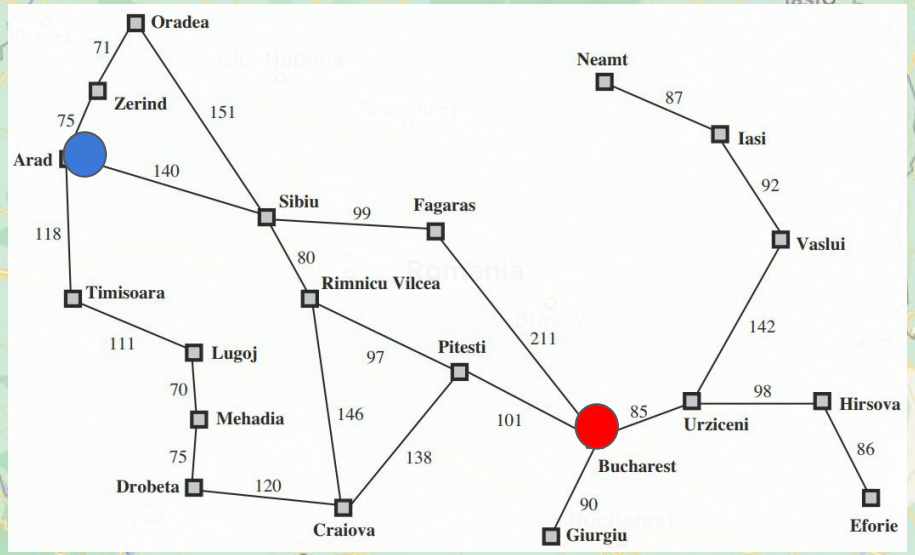
Lecture 07: Neeraj Sharma

Uninformed Search Strategies … bringing more context

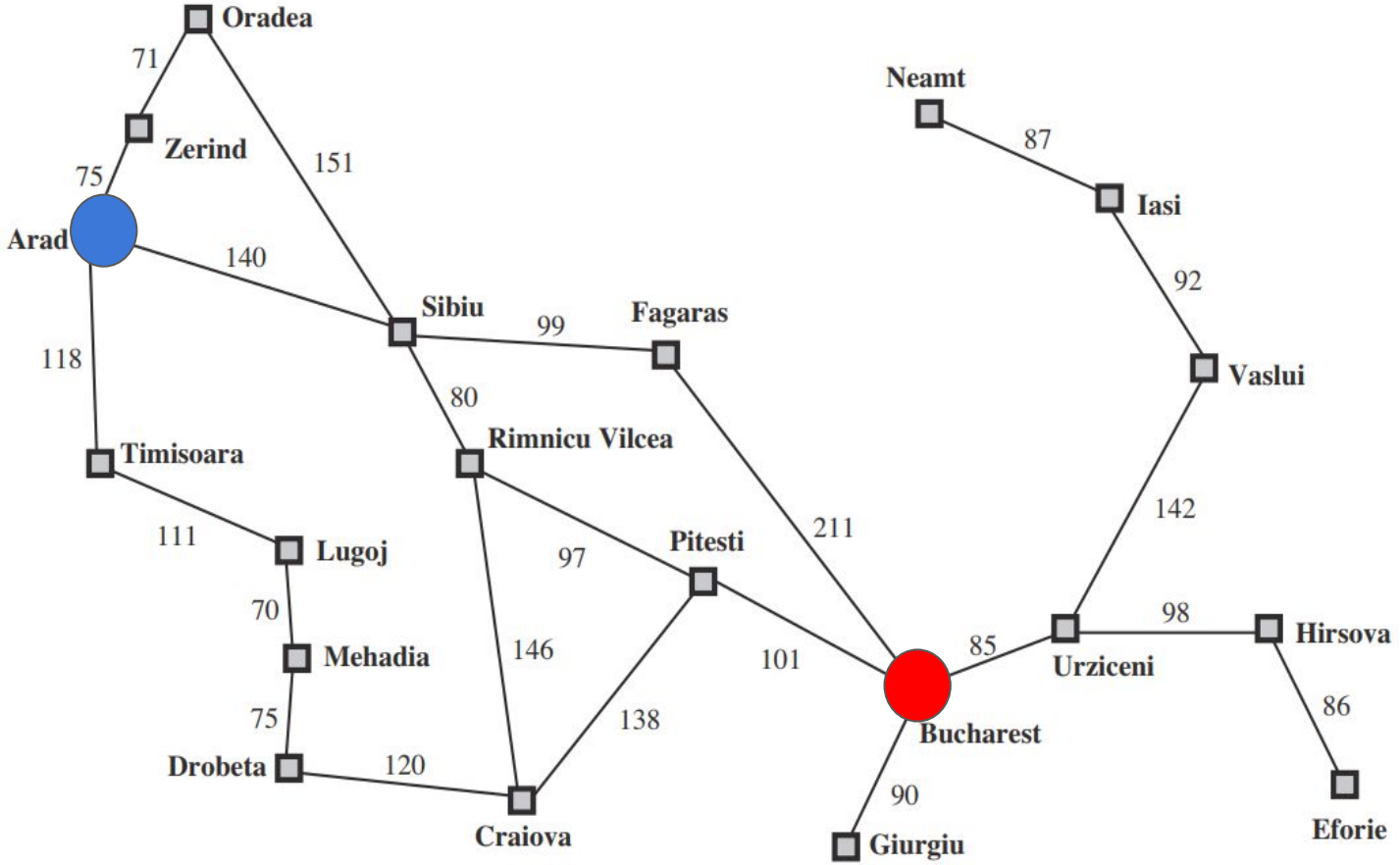

United Kingdom
Ireland
Denmark
Belarus
Poland
Germany
Ukraine
Austria
France
Romania
Italy
Spain
Portugal
Greece
Türkiye
Mediterranean Sea
Tunisia
Morocco
Algeria
Libya
Egypt
Syria
Iraq
Iran
Saudi Arabia
Oman
Yemen
Sudan
Chad
Niger
Mali
Burkina Faso
Nigeria
Ghana
South Sudan
Ethiopia
Gulf of Aden
Arabian Sea
Kazakhstan
Uzbekistan
Kyrgyzstan
Turkmenistan
Afghanistan
Pakistan
Nepal
India
Myanmar (Burma)
Thailand
Bay of Bengal
Gulf of Thailand


Ужгород
Chernivtsi
Чернівці
Suceava
Moldova
Debrecen
Iaşi
Chişinău
gary
Oradea
Cluj-Napoca
Târgu Mureş
Timişoara
Sibiu
Romania
Braşov
Galaţi
Novi Sad
Нови Сад
Ploieşti
Belgrade
Београд
Bucharest
Craiova
Constanţa
Serbia

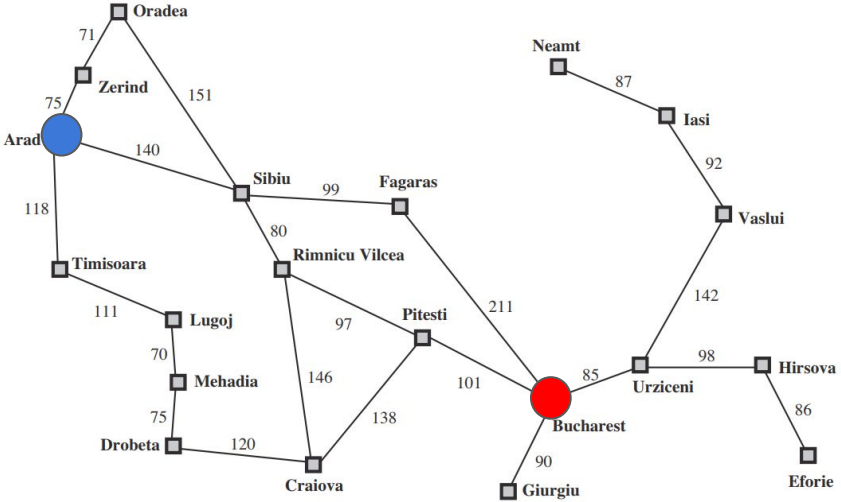
Go from Arad to Bucharest

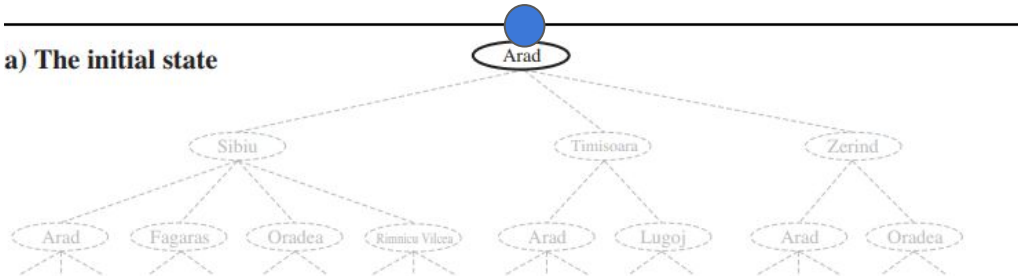


The agent's goal in Romania is the singleton set {In(Bucharest)}

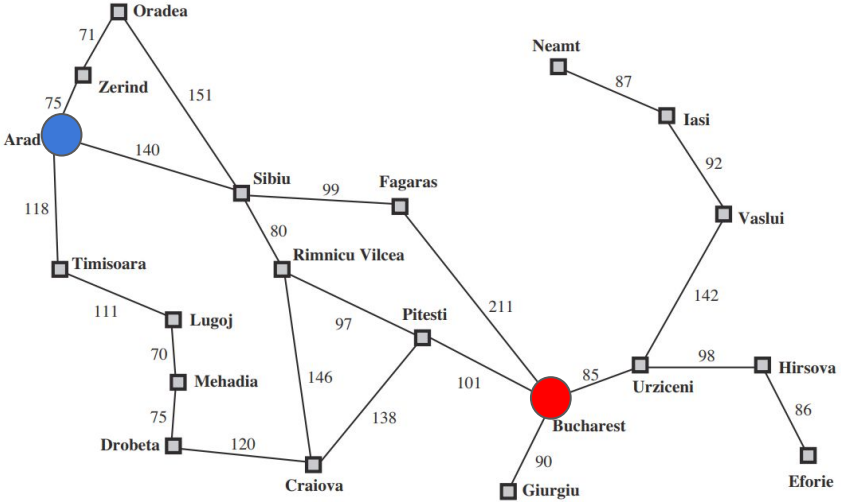


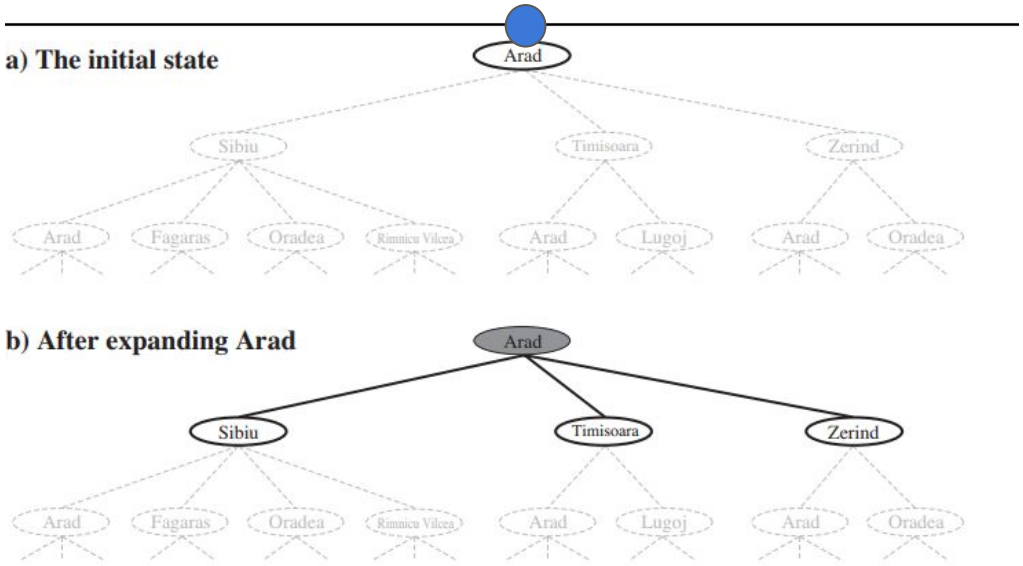
Creating a Search Tree



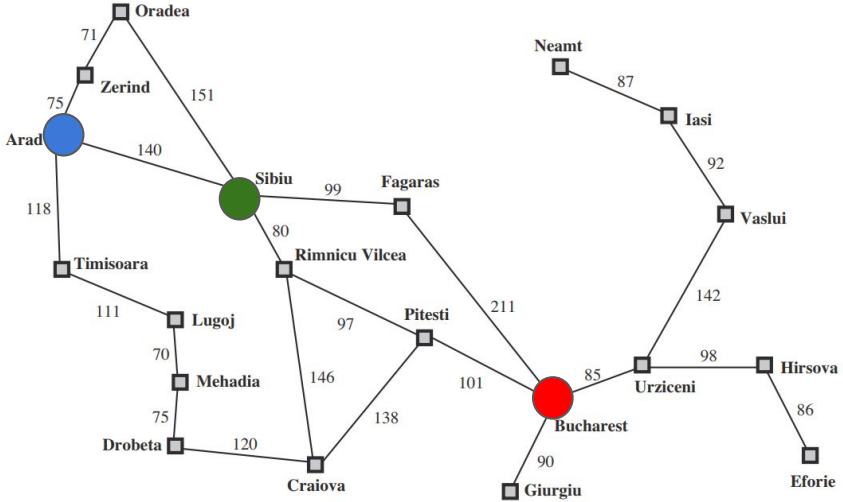


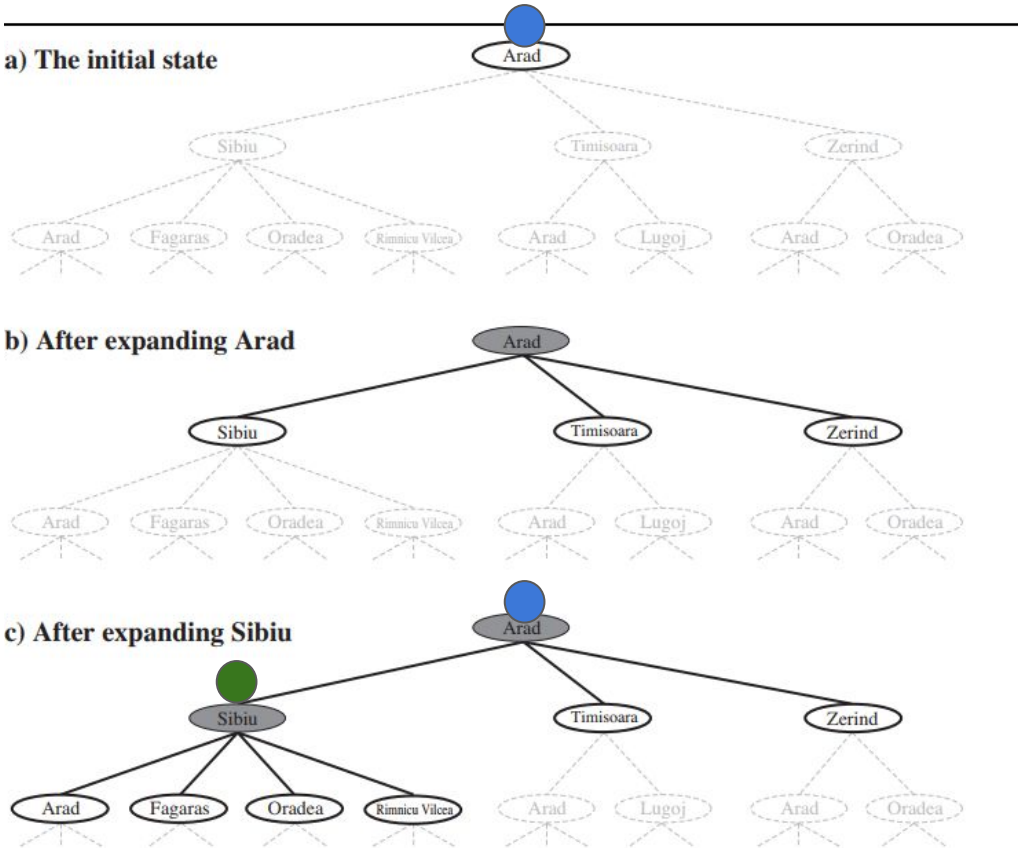
Creating a Search Tree



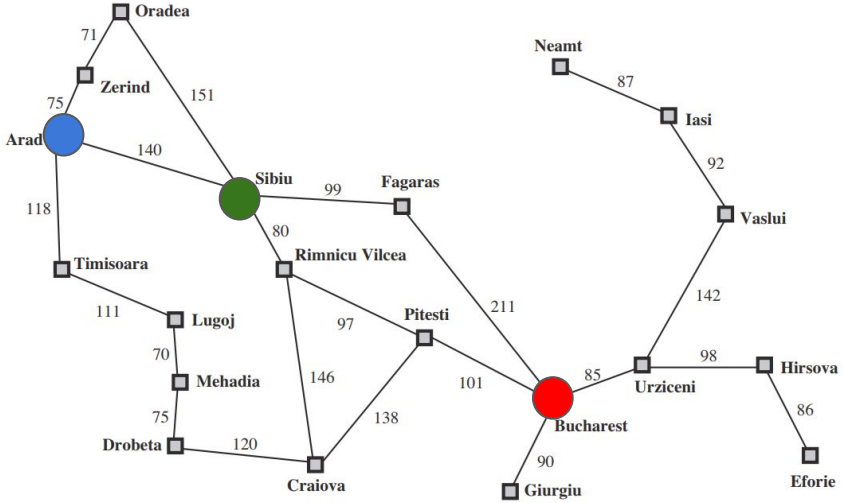


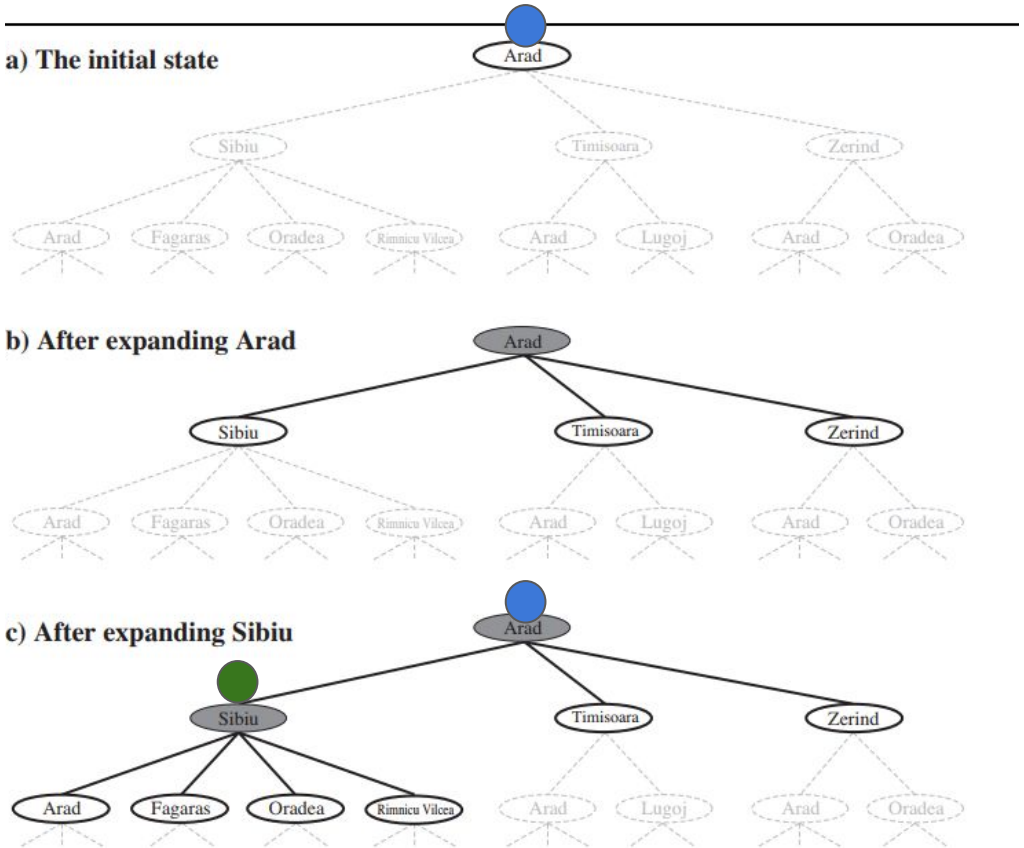
Creating a Search Tree





Creating a Search Tree





What is happening?

- Generation of nodes
- Expansion of nodes

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

function TREE-SEARCH(*problem*) **returns** a solution, or failure
    initialize the frontier using the initial state of *problem*

function TREE-SEARCH(*problem*) **returns** a solution, or failure
    initialize the frontier using the initial state of *problem*
    loop do

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
```

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

"Algorithms that forget their history are doomed to repeat it": create an explored set to help keep track of node visits.

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

initialize the frontier using the initial state of *problem*

initialize the explored set to be empty

loop do

if the frontier is empty **then return** failure

choose a leaf node and remove it from the frontier

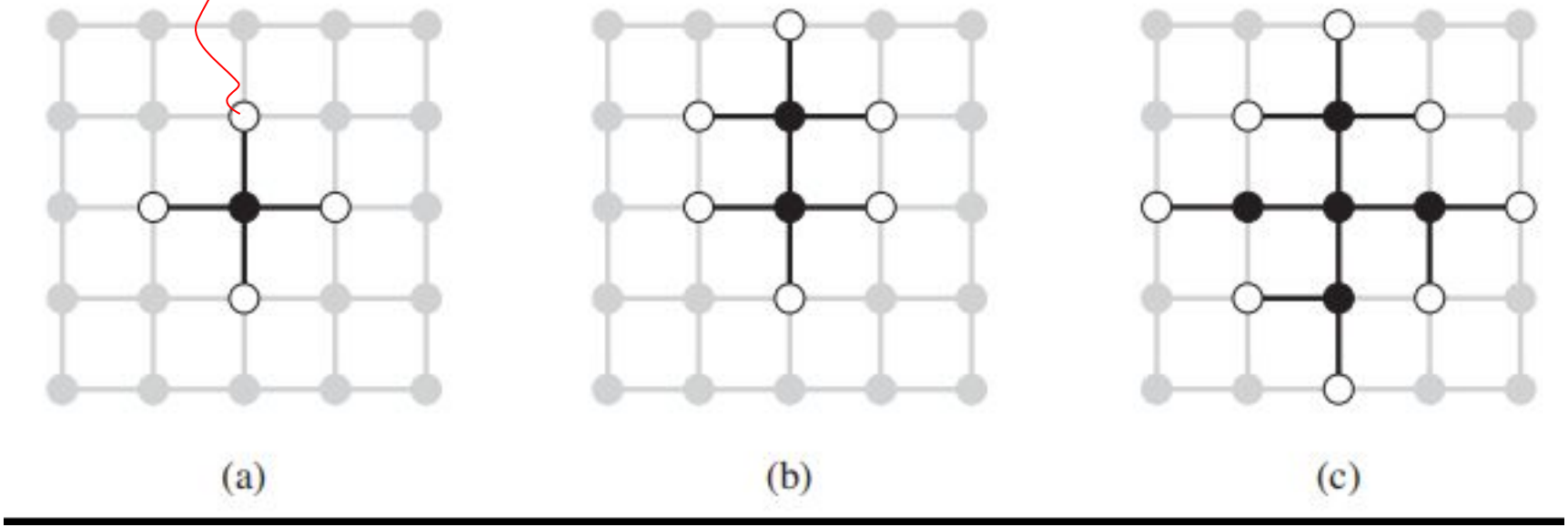
if the node contains a goal state **then return** the corresponding solution

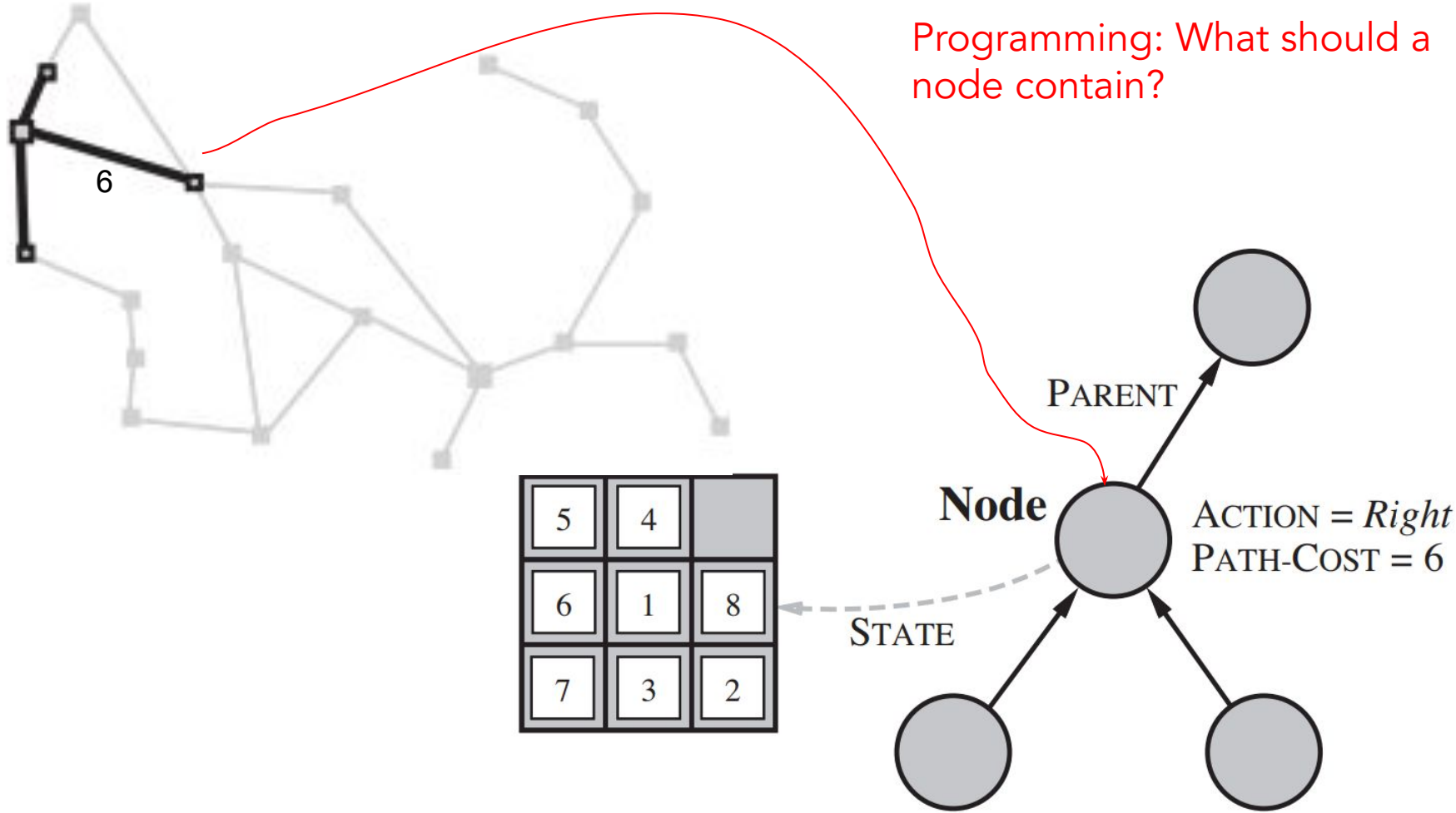
add the node to the explored set

expand the chosen node, adding the resulting nodes to the frontier

only if not in the frontier or explored set

Frontier: separates unexplored from explored




Programming: What should a node contain?
6
5 4
6 1 8
7 3 2
Node
PARENT
STATE
ACTION = Right
PATH-COST = 6

Given the components for a parent node,

CHILD-NODE takes {parent node, action} and returns the resulting child node:

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Frontier:

Store such that the search algorithm can easily choose the next node to expand according to its preferred strategy.

The appropriate data structure for this is a queue. The operations on a queue are as follows:

- EMPTY?(queue): returns TRUE only if there are no more elements in the queue.
- POP(queue): removes the first element of the queue and returns it.
- INSERT(element, queue): inserts an element and returns the resulting queue.

Frontier:

Store such that the search algorithm can easily choose the next node to expand according to its preferred strategy.

Queue types based on how elements can be popped out

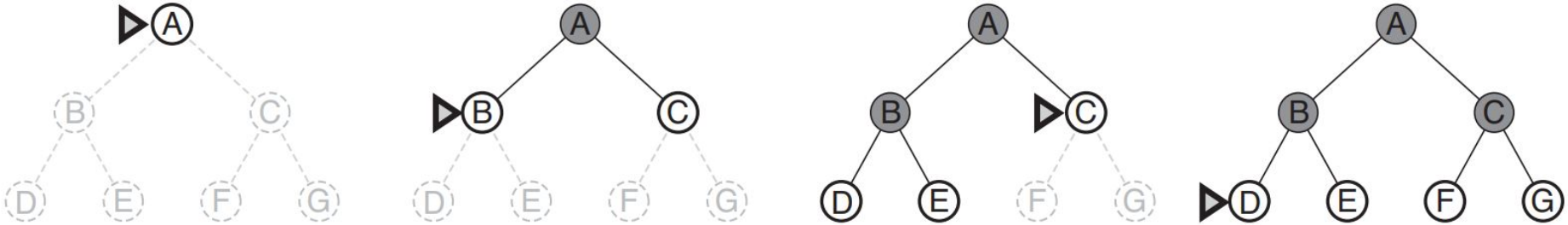
- FIFO Queue
- LIFO Queue
- Priority Queue

Expansion set:

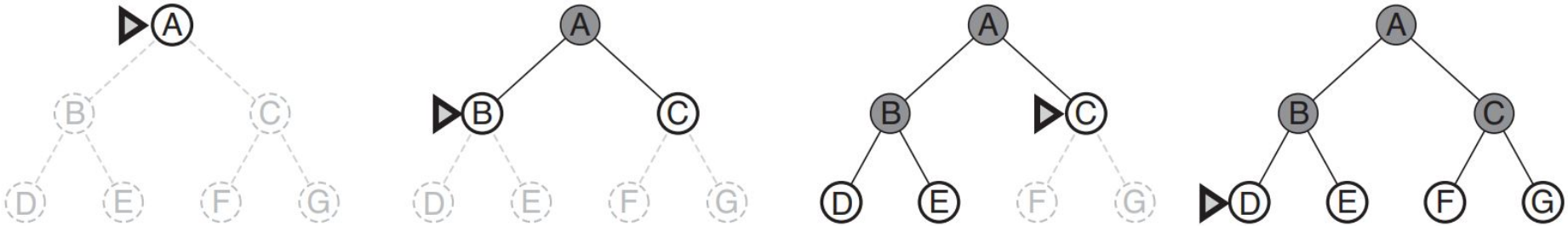
Needs to be regularly queried for checking if a state has been explored.

- Efficient storage, lookup and expansion
- Hash tables can be used
- Canonical form:
  - Bit vector representation
  - sorted /ordered list

Breadth First search



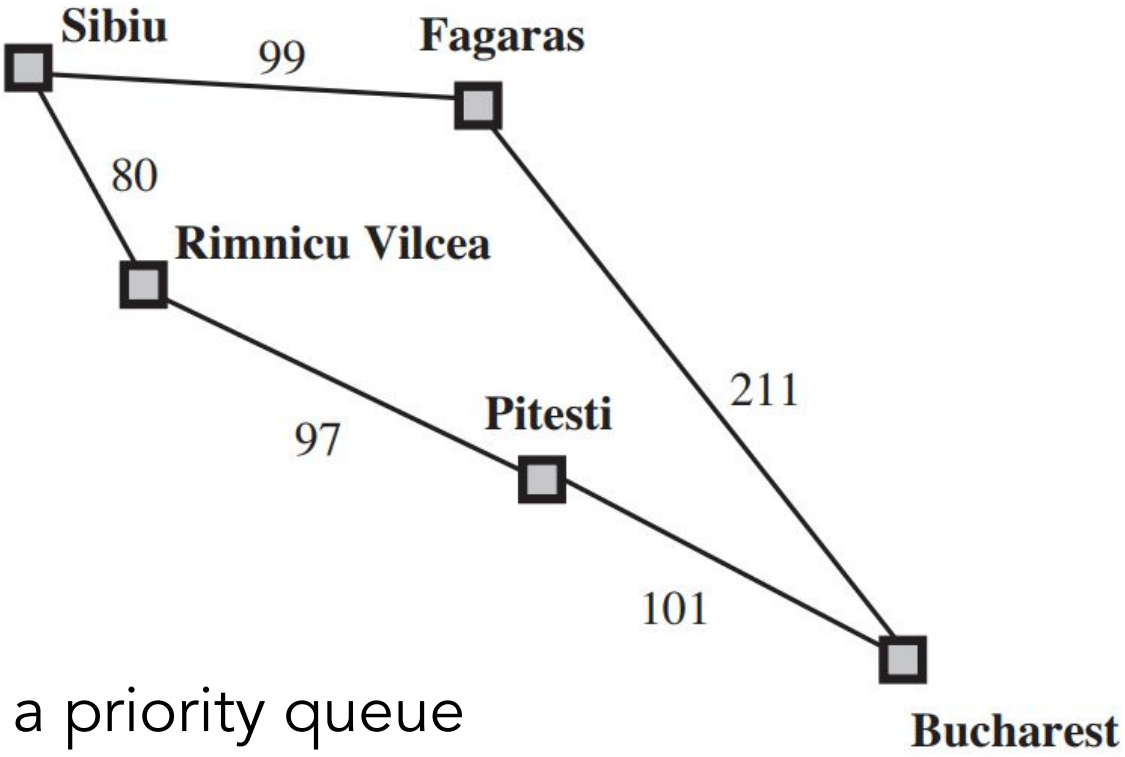
Breadth First search



Frontier operates as a FIFO queue

```
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)
```

Uniform cost search

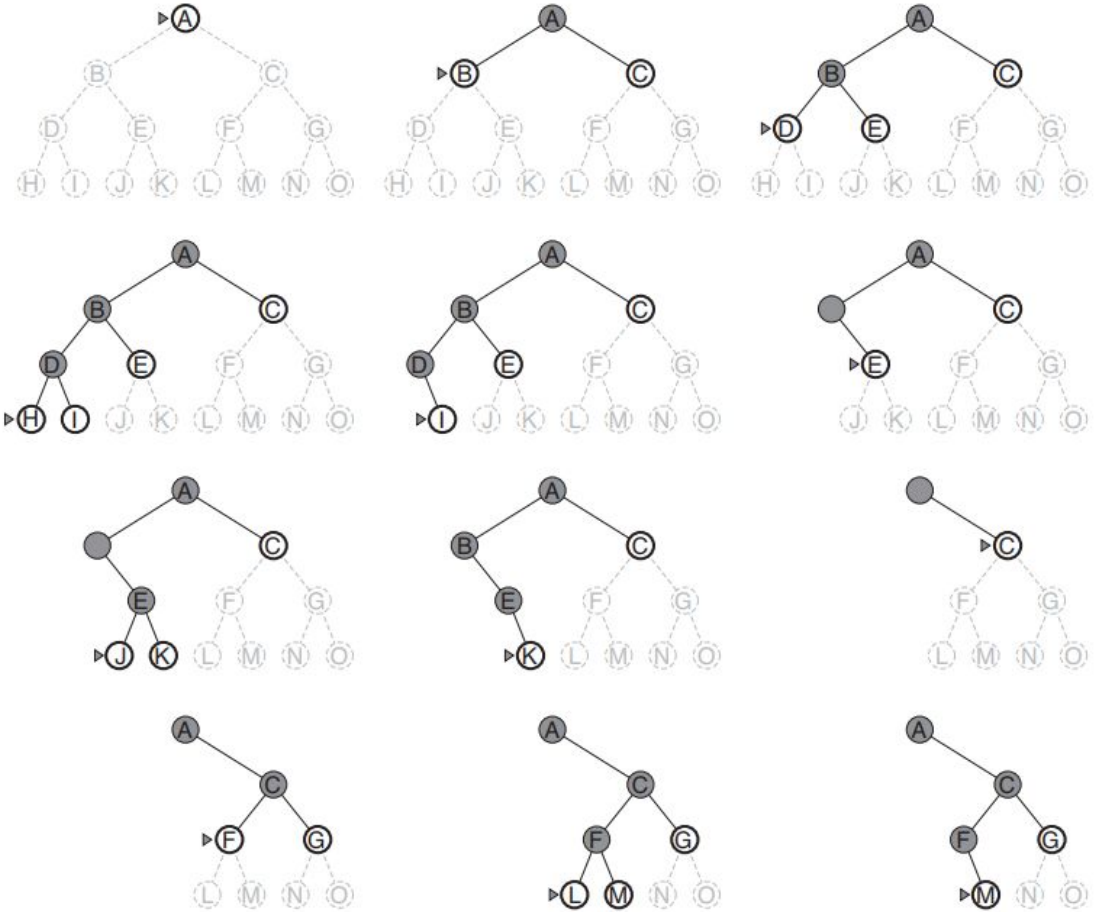


Introduces the notion of cost.

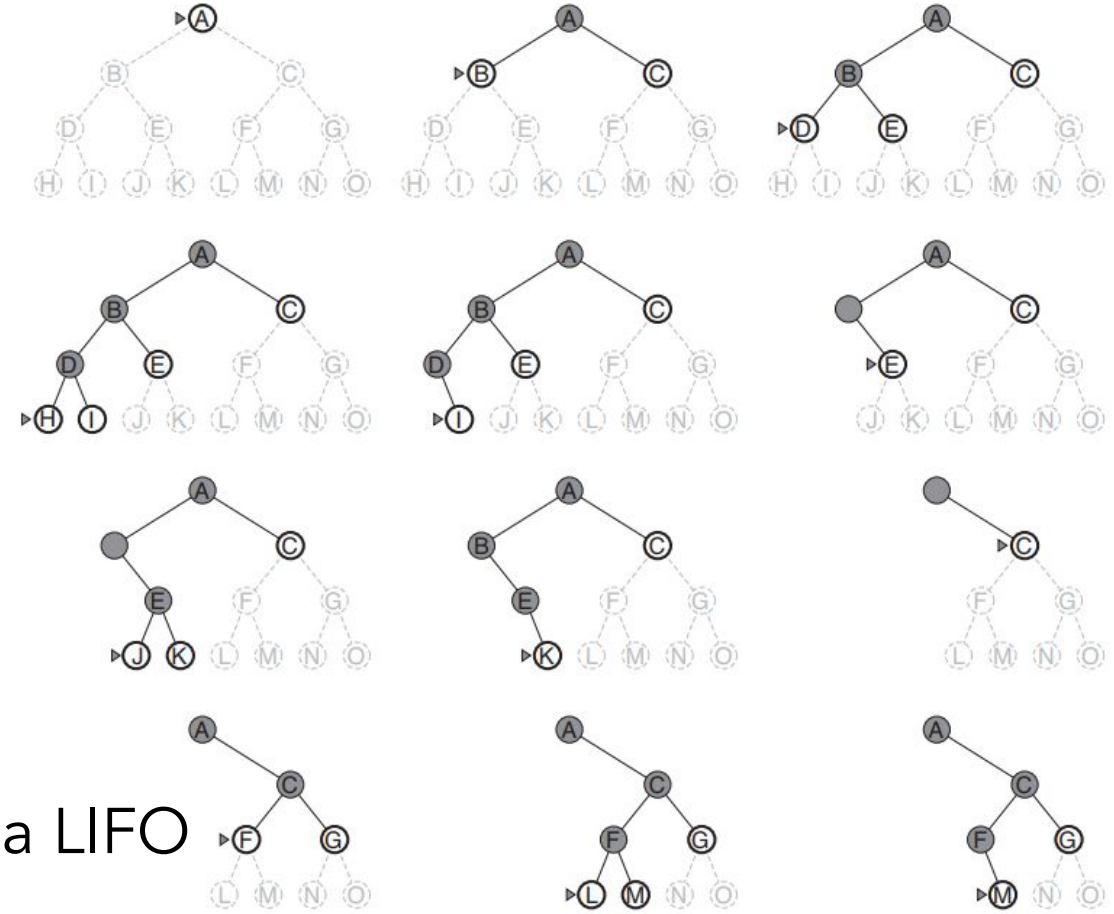
Frontier operates as a priority queue

```
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
```

Depth First Search



Depth First Search



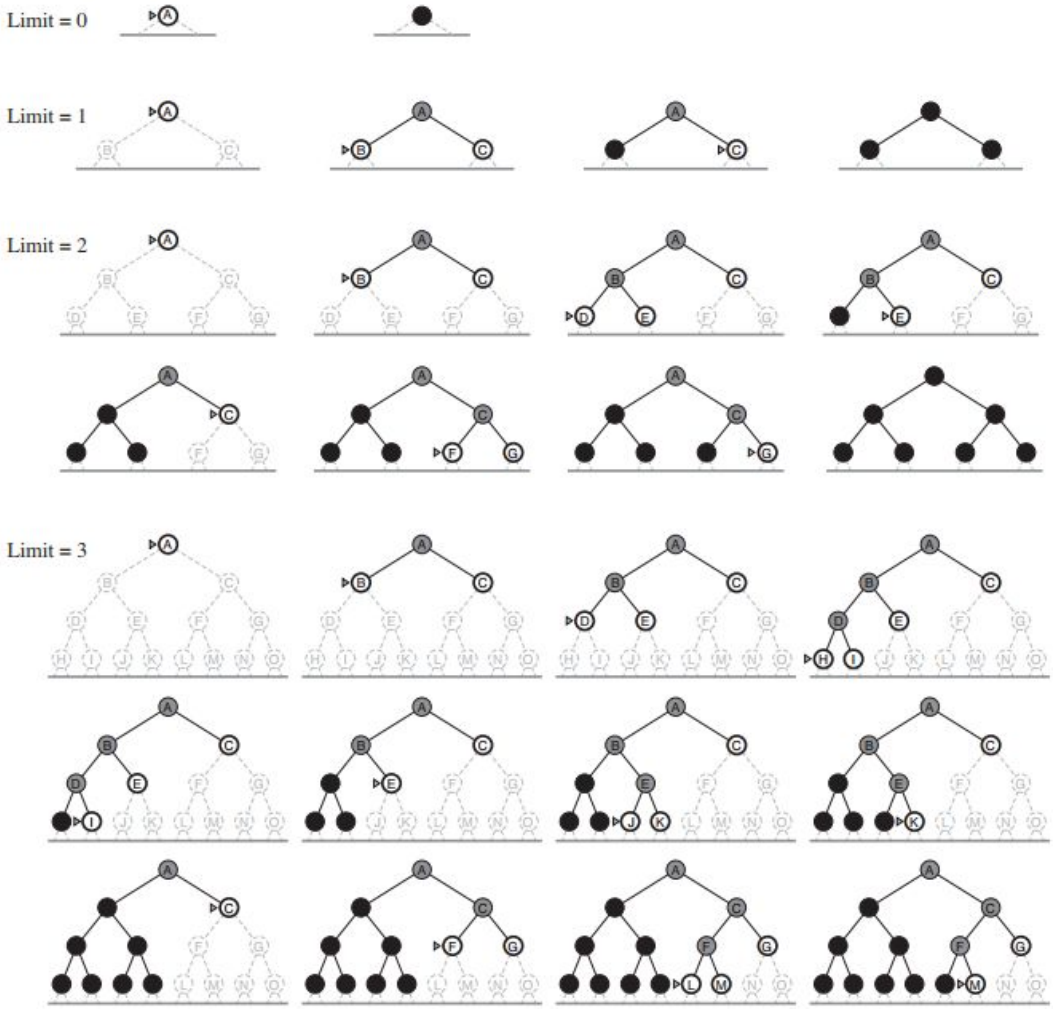
Frontier operates as a LIFO queue

Depth Limited Search

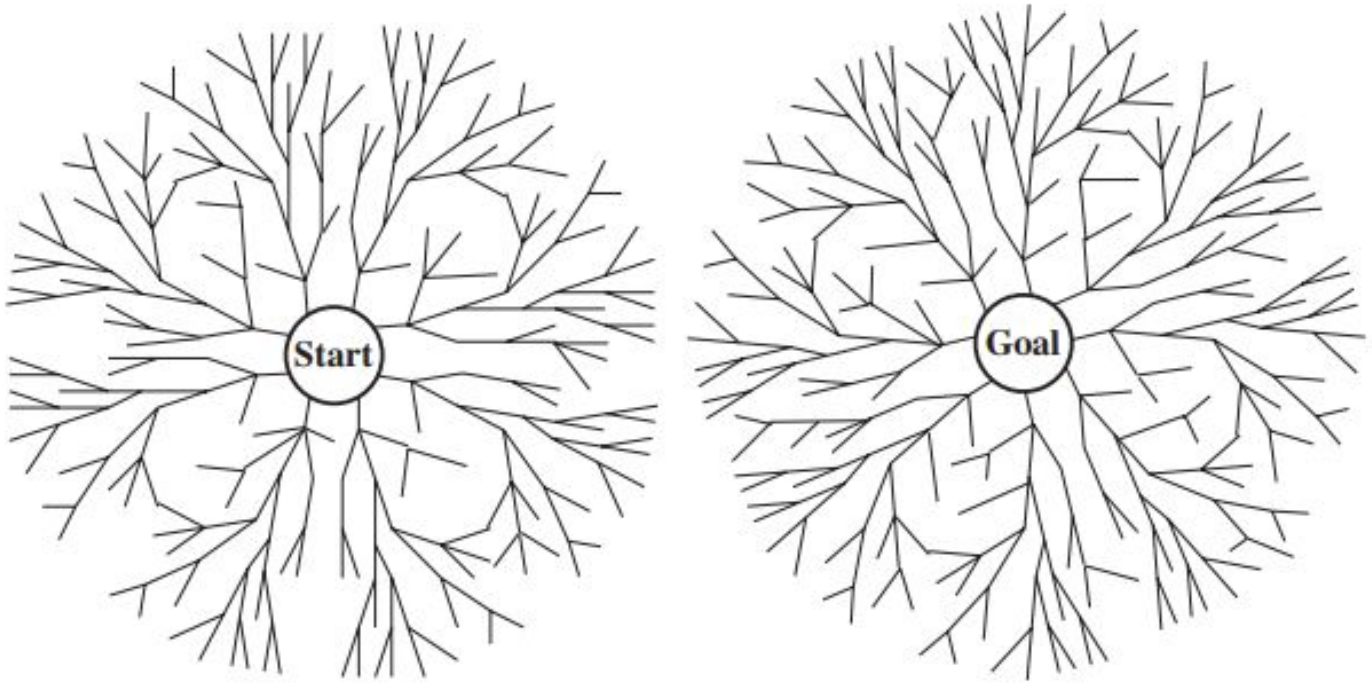
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Iterative Deepening Depth-First Search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```


Limit = 0
Limit = 1
Limit = 2
Limit = 3

Bi-directional Search



Comparison of Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution,
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

b:branching factor or maximum number of successors for any node
d: the depth of the shallowest goal
m: maximum length of any path in the state space