IT5005 Artificial Intelligence

Sirigina Rajendra Prasad
AY2022/2023: Semester 1

2. Uninformed Search

Learning Objectives of the Session

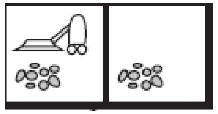
- Understand workflow of goal-based agents with atomic representation
- Modeling a problem into an implicit state space graph
- Learn how to search for a solution on implicit state space graphs
  - Uninformed search

Recap

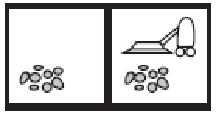
Atomic Representation

Two-Room Vacuum World: Eight States

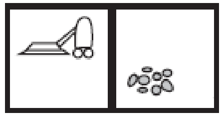
State: 1



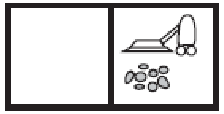
State: 2



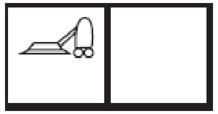
State: 3



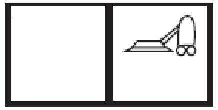
State: 4



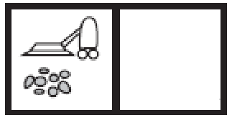
State: 5



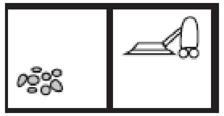
State: 6



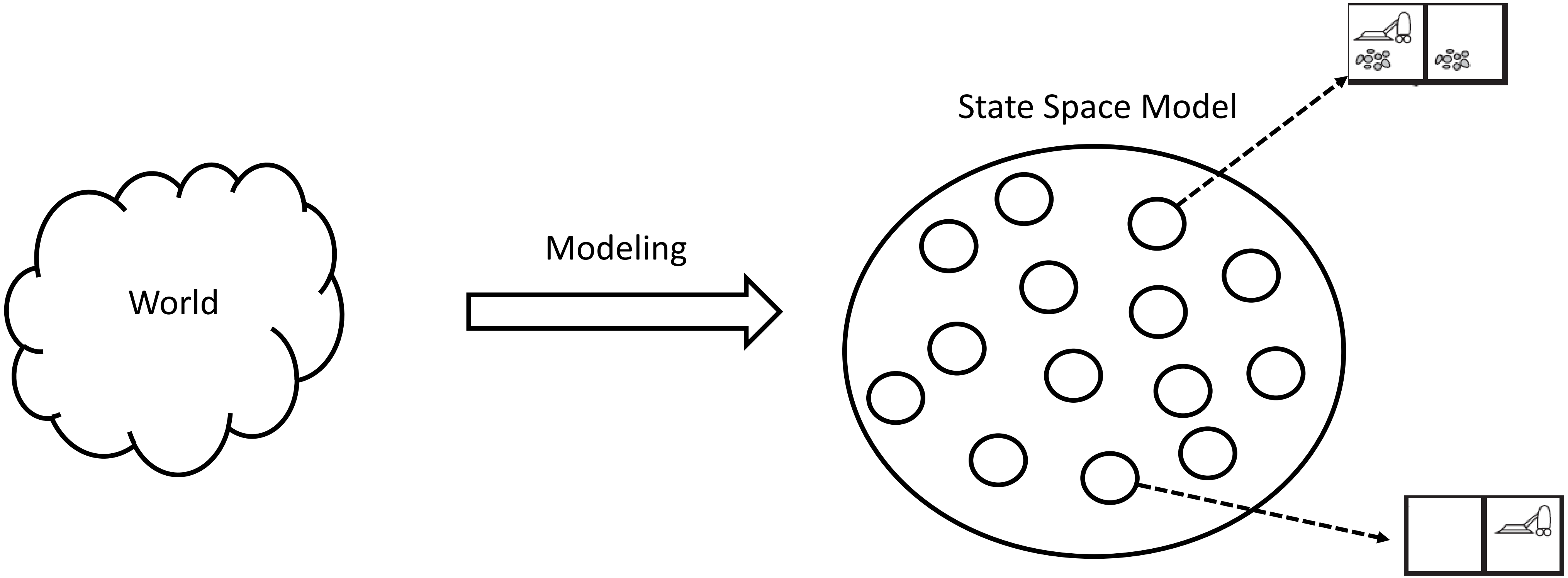
State: 7



State: 8

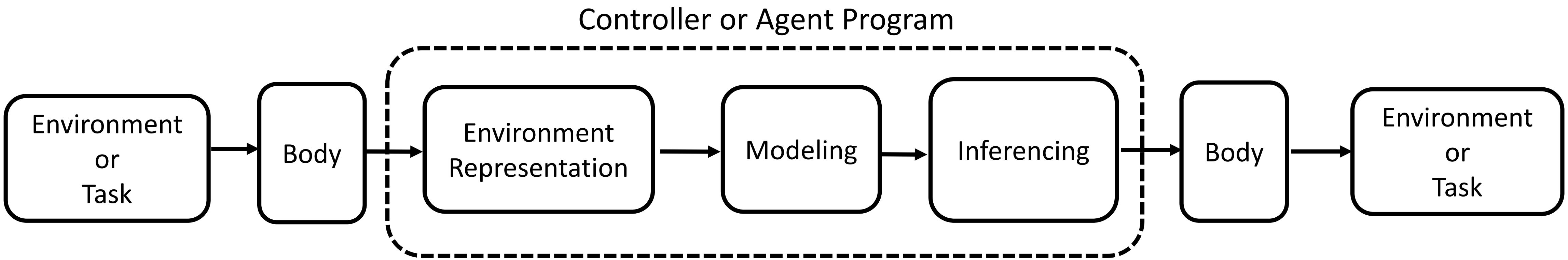


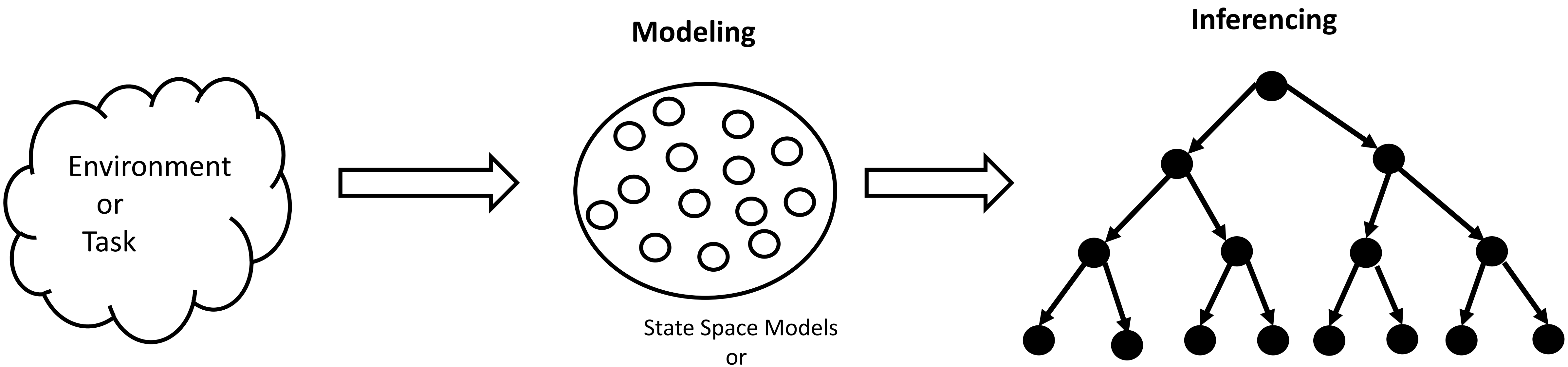
Modeling with Atomic Representation



- Each node (vertex) is related to a state of the world
- Goal should also be represented by state (node/vertex)

Problem Solving by Search: Workflow

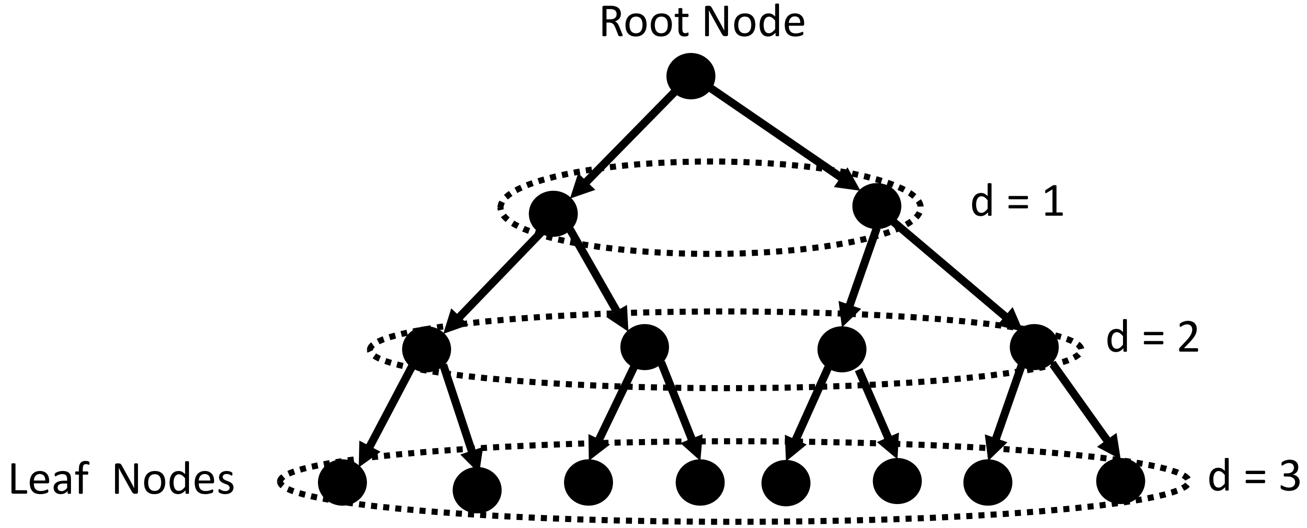




State Space Models
or
State Space Graphs

Search for solution on State Space Graph

Preliminaries: Trees



$$\begin{aligned}
\text{Branching Factor} &= b \\
\text{Maximum Depth} &= m \\
\text{Number of Nodes at depth } d &= b^d \\
\text{Number of Leaf Nodes} &= b^m \\
\text{Number of Nodes} &= 1 + b + b^2 + \ldots b^d + .. + b^m
\end{aligned}$$

m: maximum depth

Preliminaries: Performance Measures

- Completeness
  - Complete if algorithm can find a solution (achieve the goal)
- Optimality (*aka* rationality)
  - Optimal if algorithm finds an optimal (lowest cost) path to solution
- Time Complexity
  - Time taken to find the solution
  - Measured in terms of number of nodes generated
- Space Complexity
  - Memory needed to find the solution
  - Measured in terms of number of nodes stored while building the graph

Uninformed Search

Applications

- Puzzles
  - Slide Puzzles
  - Missionaries and Cannibals Problem, etc.

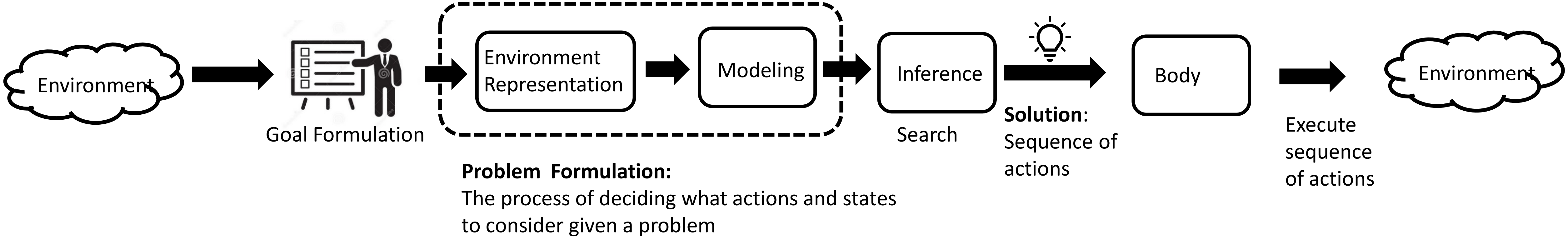
- Games:
  - Pacman, etc.
  - Maze Navigation

- Real-World Applications
  - Route Planning
  - Robot Motion Planning
  - VLSI Layout Planning, etc.

Agenda

- Workflow
- Goal Formulation
- Problem Formulation or Modeling
- Building Blocks of Search Tree
- Inference

Workflow



Environment
Goal Formulation
Environment Representation
Modeling
Inference
Search
Solution:
Sequence of actions
Body
Execute sequence of actions
Environment
Problem Formulation:
The process of deciding what actions and states to consider given a problem

Agenda

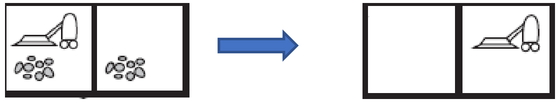
- Workflow
- Goal Formulation
- Problem Formulation or Modeling
- Building Blocks of Search Tree
- Inference

Goal Formulation: Examples

- Two-Room Vacuum World:

Vacuum cleaner world
with two locations

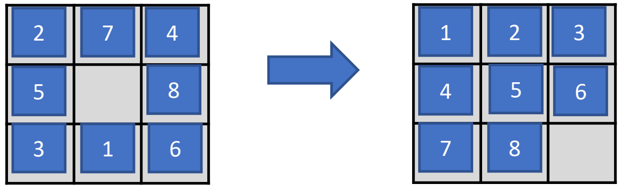
Goal



- 8-Puzzle

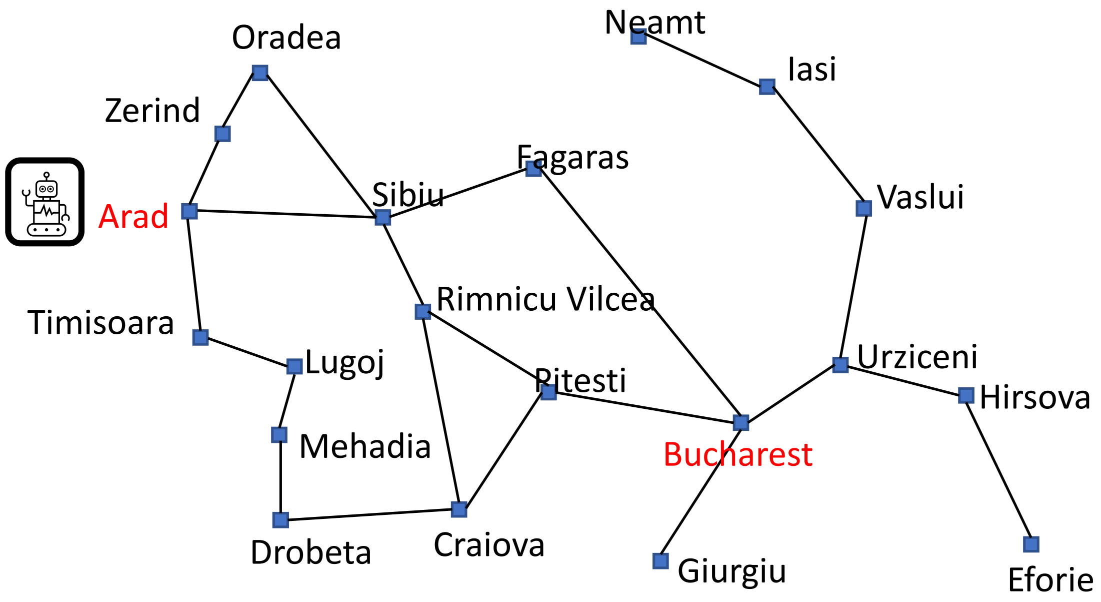
Initial State

Goal



https://mathworld.wolfram.com/15Puzzle.html

Goal Formulation: Examples



"Map of Romania"

- **Route Planning**

  Goal: Find a path from Arad to Bucharest

Agenda

- Workflow
- Goal Formulation
- Problem Formulation or Modeling
- Building Blocks of Search Tree
- Inference

Modeling or Problem Formulation

1. States

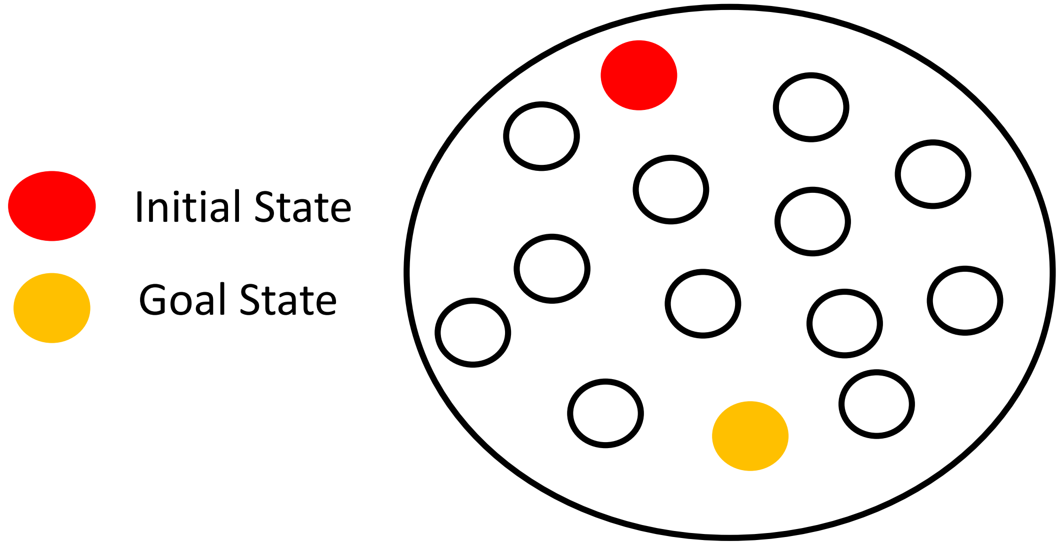
2. Initial State

3. Actions: $Actions(s)$
   - Legal (applicable) actions for an agent at a state s

4. Transition Model: $RESULT(s, a)$
   - Defines the result state for an action at a given state
   - Ex: $s_2 = RESULT(s_1, action)$

5. Goal State:
   - Test goal state using $IS - GOAL(s)$

6. Action Cost Function: $ACTION - COST(s, a, s')$
   - Cost of an action at a state

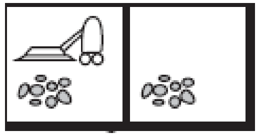


Action Cost: $c(s_1,a,s_2)$

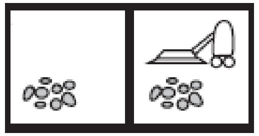
Two-Room Vacuum World: Modeling

1. States

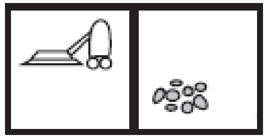
State: 1



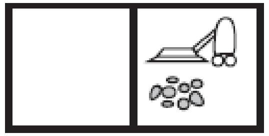
State: 2



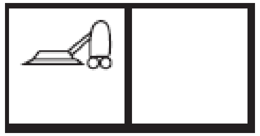
State: 3



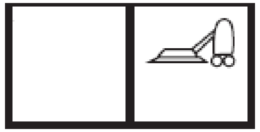
State: 4



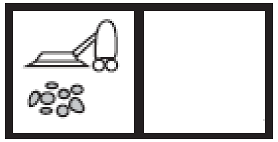
State: 5



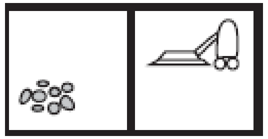
State: 6



State: 7

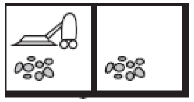


State: 8



Two-Room Vacuum World: Modeling

2. Initial State



3. Actions:

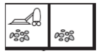
Move Left (L)
Move Right (R)
Suck dirt (S)
No-op (N)

4. Transition Model:

If room is dirty, *Suck Dirt* make it clean
If room is clean, *Suck Dirt* does nothing
If agent is in left room, move Right takes it to right room
If agent is in left room, move left does nothing, etc

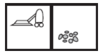
Example:

state: s_1

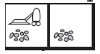


Suck Dirt

state: s_2 = Result(s_1, S)

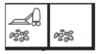


state: s_1

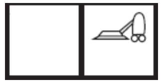


Move Left

state: s_2 = Result(s_1, L)



5. Goal State

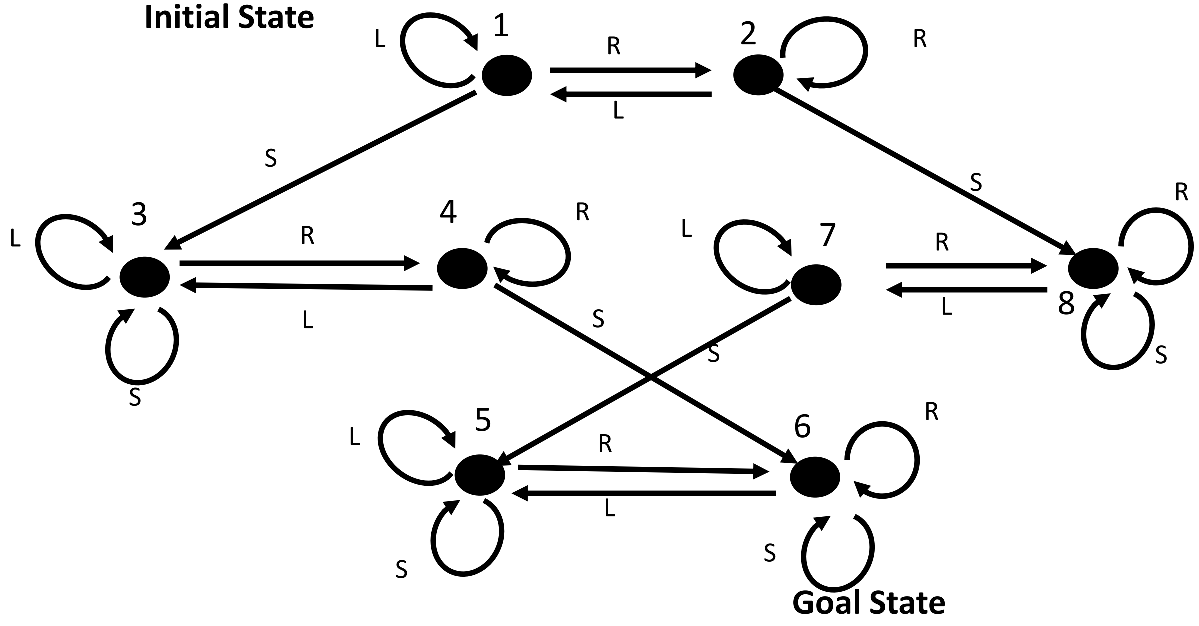


Check
for this state

6. Action Cost

Suck Dirt: 1
Move Left: 2
Move Right: 2
No-Op: 0

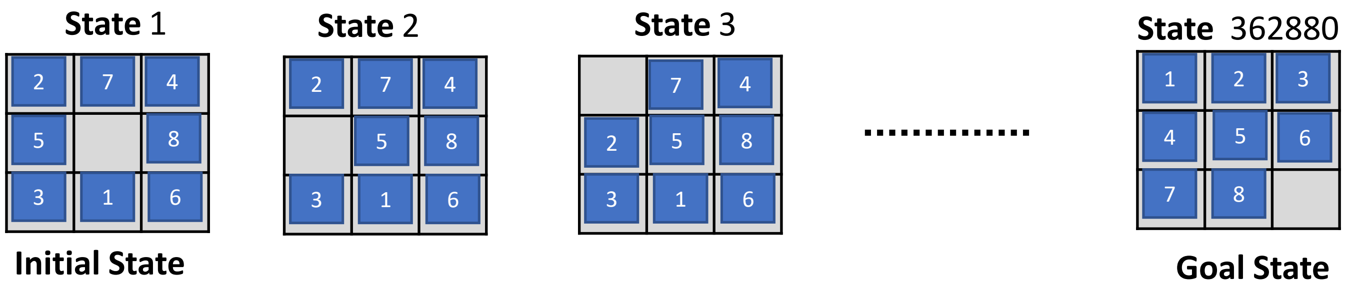
Two-Room Vacuum World: Modeling



8-Puzzle: Modeling

1. State Representation

Number of states = 9! = 362,880



8-Puzzle: Modeling

2. Initial State

2	7	4
5		8
3	1	6

3. Actions:

Movement of blank space

- Slide Left (L)
- Slide Right (R)
- Slide Below (B)
- Slide Above (A)

4. Transition Model:

Given a state and action
return the resultant state

Eg:

State: s_1

2	7	4
5		8
3	1	6

State: $s_2 = \text{Result}(s_1, L)$

2	7	4
	5	8
3	1	6

5. Goal State

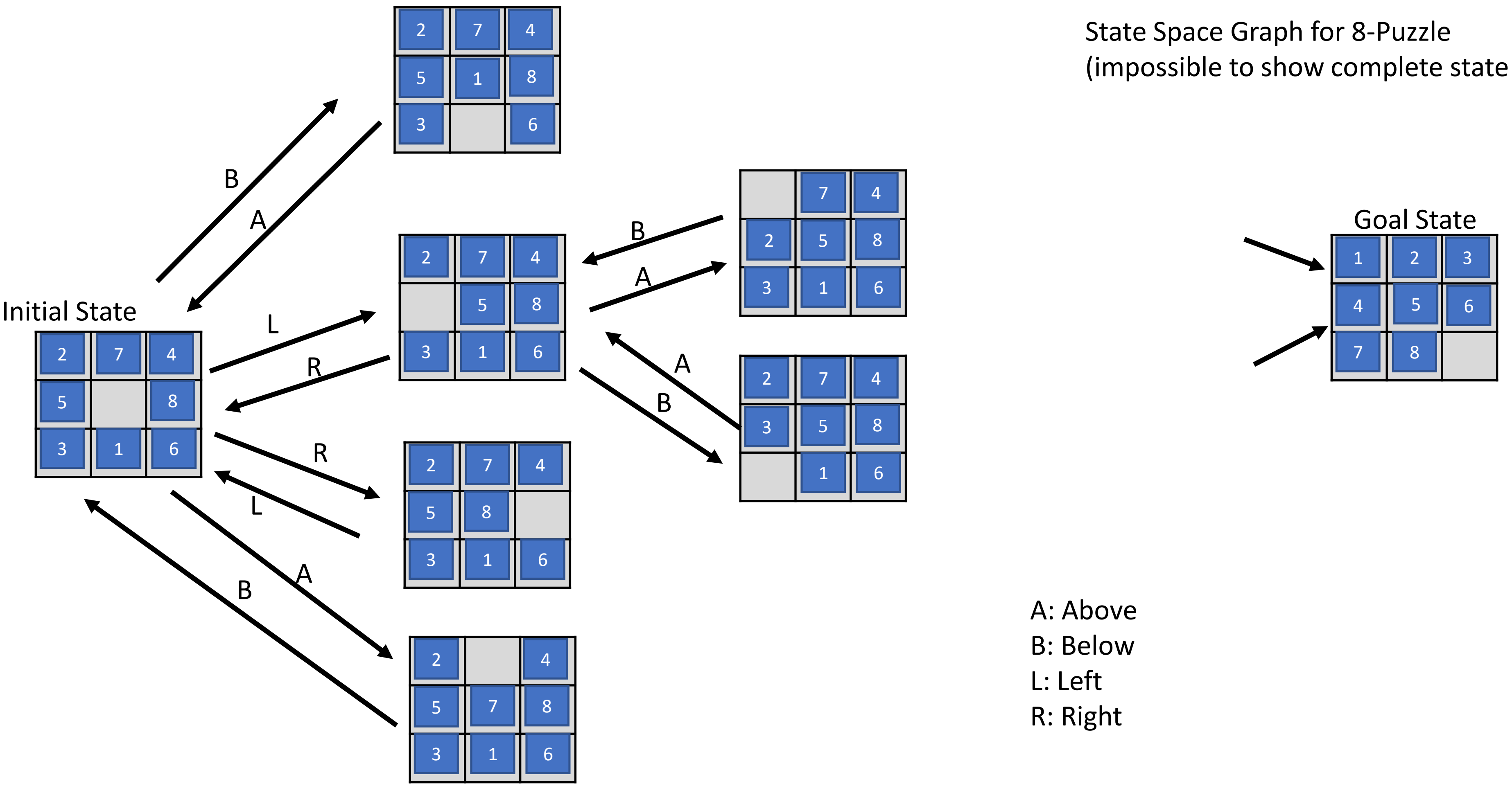
1	2	3
4	5	6
7	8	

6. Action Cost

Each action costs 1

8-Puzzle: State Space Model

State Space Graph for 8-Puzzle
(impossible to show complete state space graph)



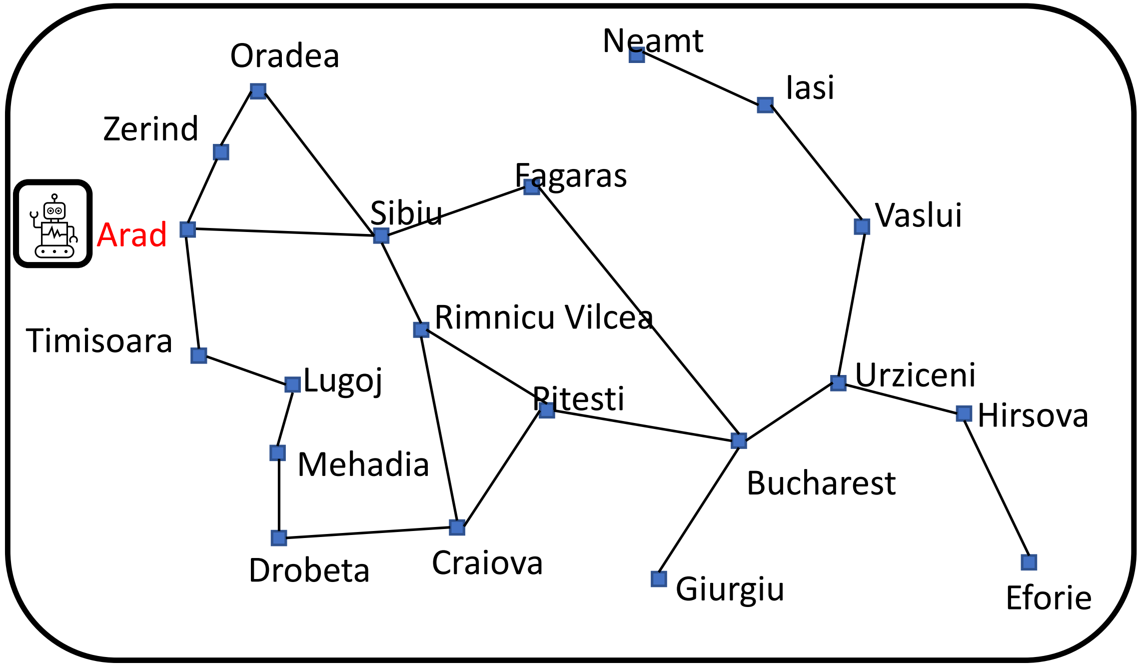
A: Above
B: Below
L: Left
R: Right

Romania Map (Route Planning): Modeling

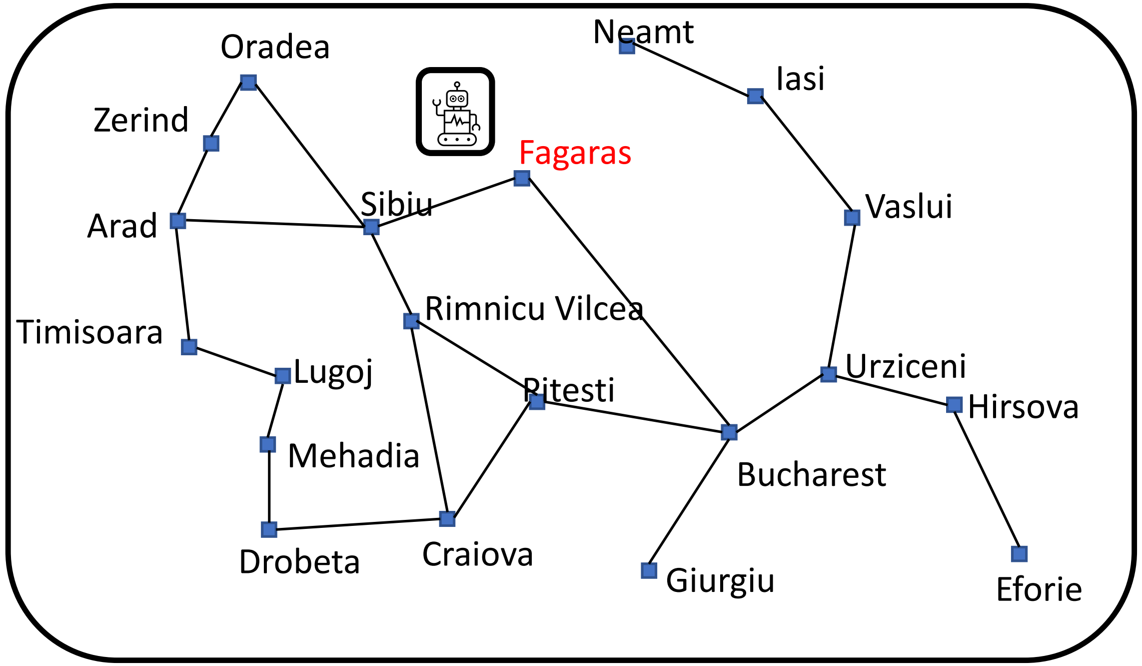
1. States

- State: location of agent

State: $Arad$



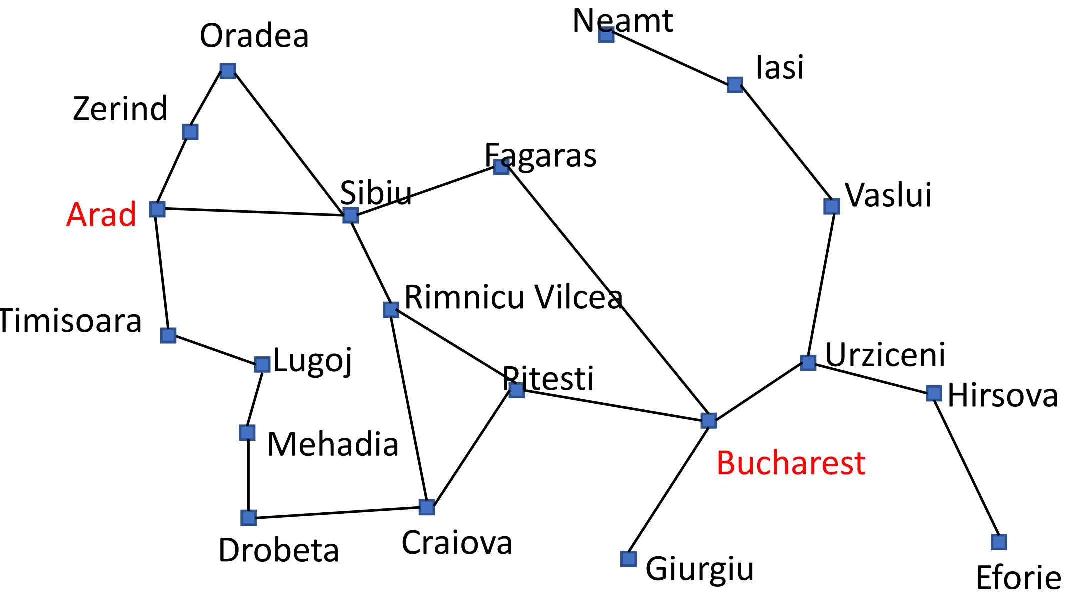
State: $Fagaras$



"Map of Romania" Number of States: 20

Romania Map (Route Planning): Modeling

Problem Formulation



1. States:

$Arad$, $Oradea$, etc.

2. Initial State:

$Arad$

3. Actions:

Eg: for $Arad$:

$\{ToZerind, ToSibiu, ToTimisoara\}$

4. Transition Model:

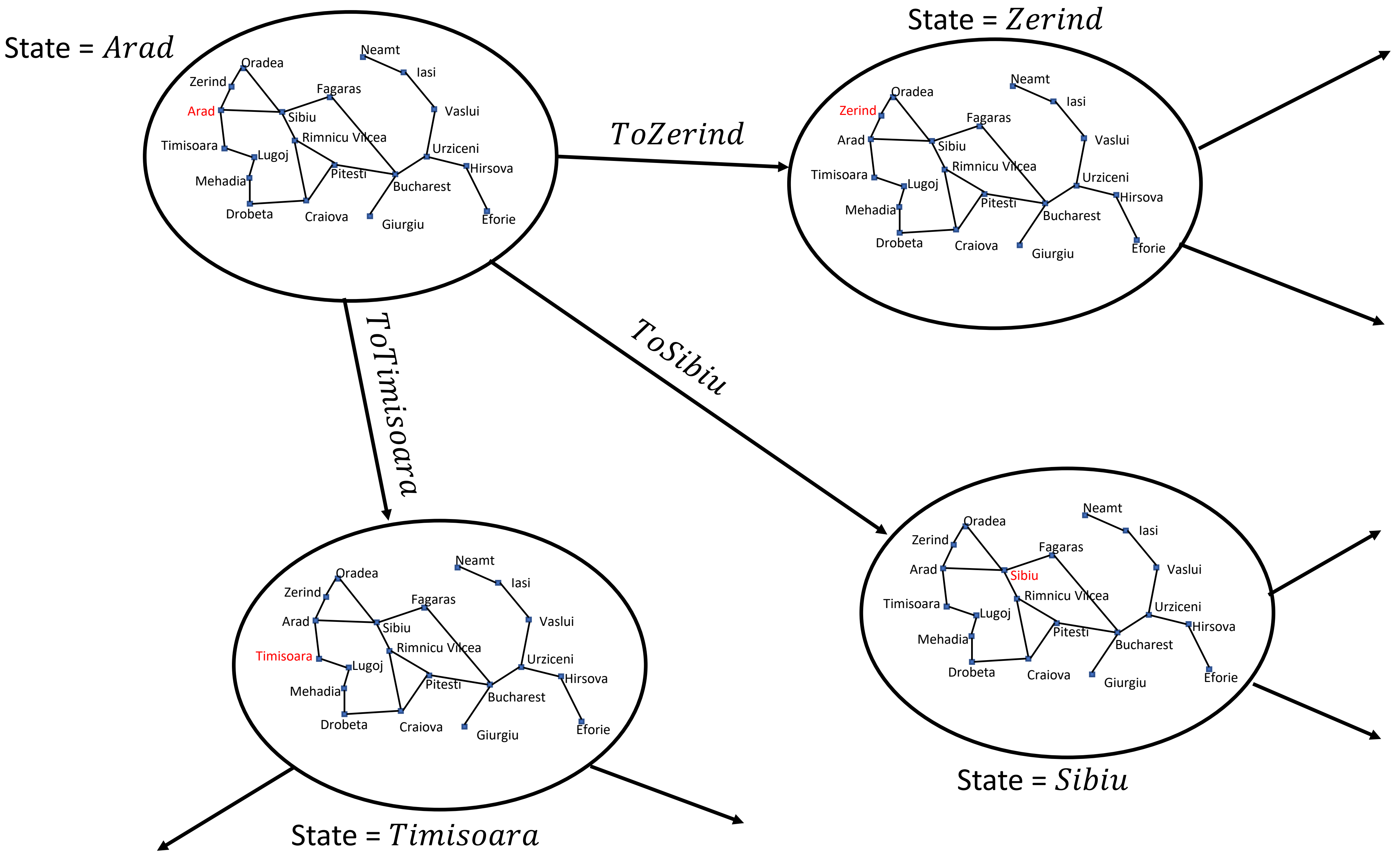
$RESULT(Arad, ToZerind) = Zerind$

5. Goal-Test:

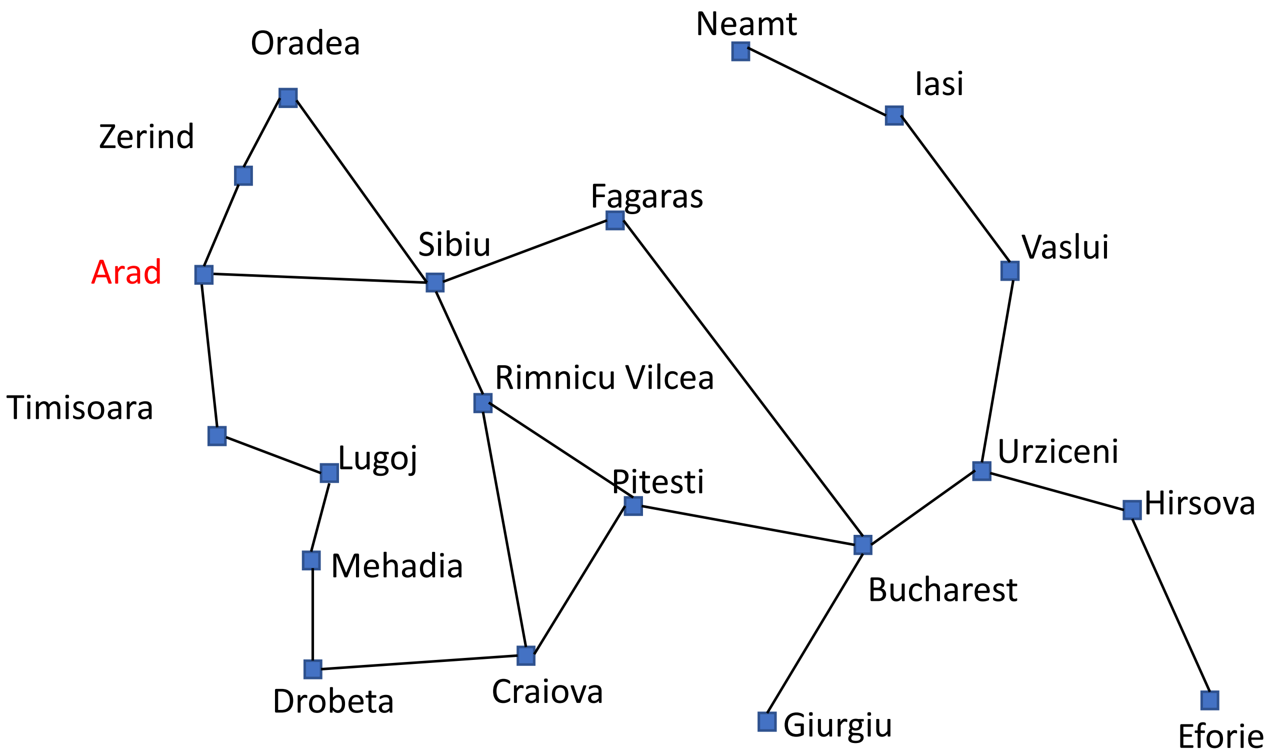
$IS - GOAL(Bucharest)$

6. Action-Cost:

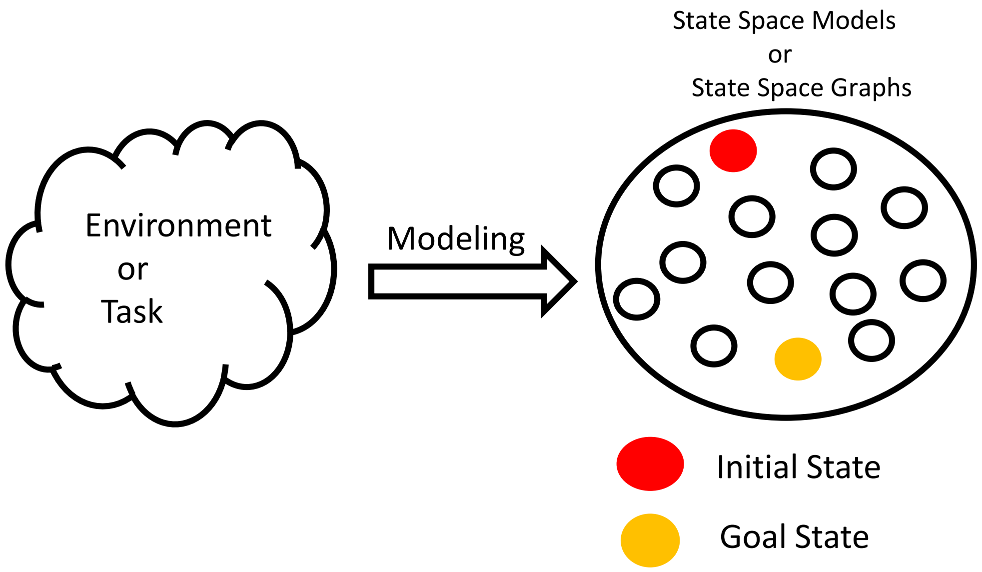
Distance between cities


State = $Arad$
State = $Zerind$
State = $Sibiu$
State = $Timisoara$
$ToZerind$
$ToSibiu$
$ToTimisoara$
Oradea
Zerind
Arad
Timisoara
Lugoj
Mehadia
Drobeta
Craiova
Sibiu
Rimnicu Vilcea
Fagaras
Pitesti
Bucharest
Giurgiu
Neamt
Iasi
Vaslui
Urziceni
Hirsova
Eforie

Romania Map (Route Planning): Modeling



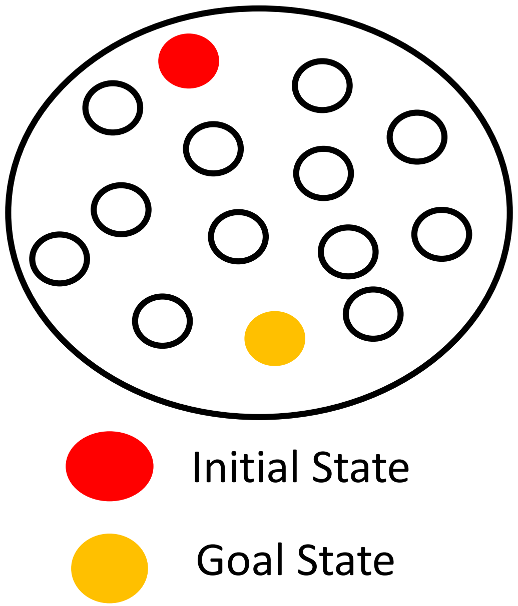
Until Now: Modeling



Note:
- Implicit Graph.
- Only provided a framework (model/problem formulation) to generate this graph

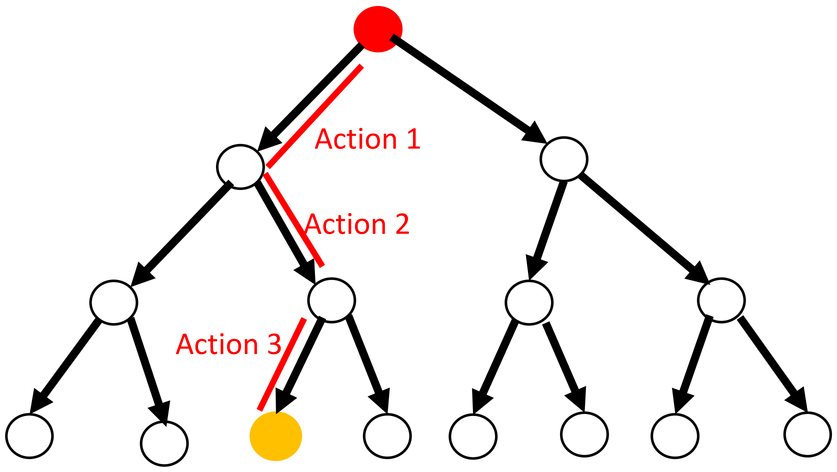
Next: Inference (Search for the solution)

Implicit State Space Graph



Inferencing

(Search for the solution)



Search Tree

Inferencing:

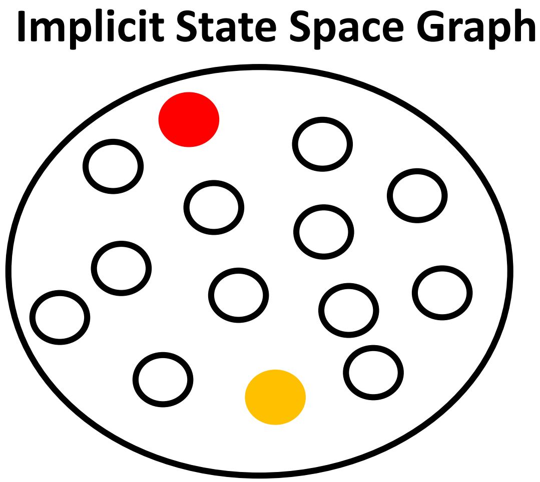
Finding the path from Initial State to Goal State

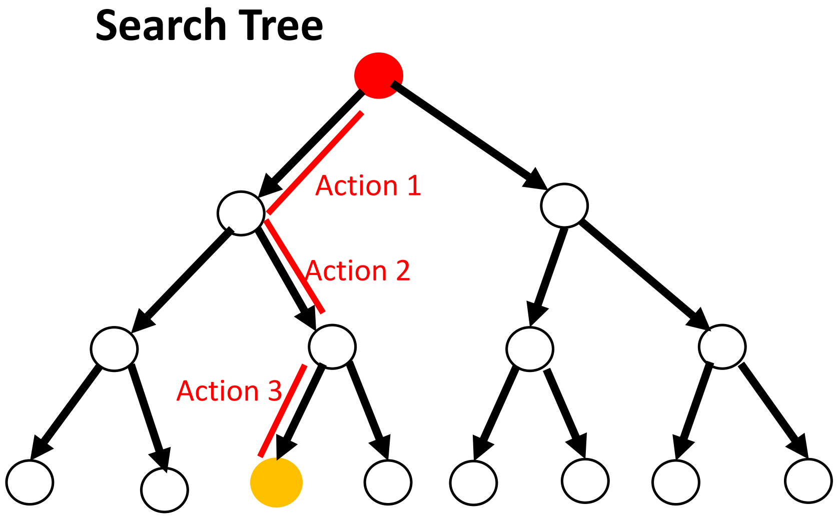
Agenda

- Workflow
- Goal Formulation
- Problem Formulation or Modeling
- Building Blocks of Search Tree
- Inference

Building Blocks of Search Tree

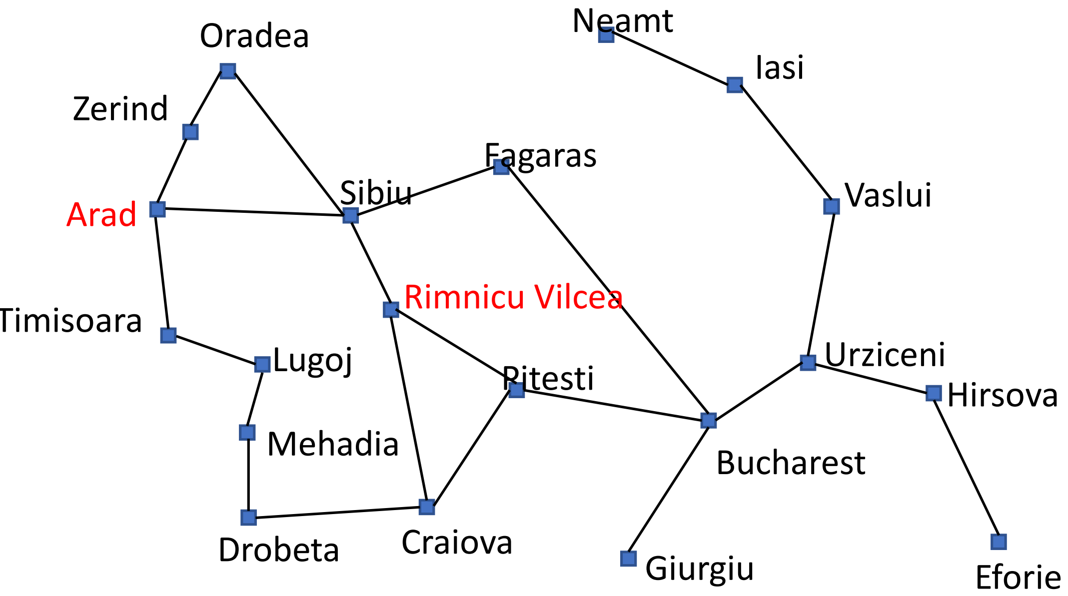
- Need to create the search tree dynamically
- Require data structures
  - to generate nodes in search tree
  - for navigating around the search tree
- Let's start with node
- Illustration with Romania map





Romania Map: Route Planning

Problem Formulation



1. States:

$Arad$, $Oradea$, etc.

2. Initial State:

$Arad$

3. Actions:

Eg: for Arad:

$\{ToZerind, ToSibiu, ToTimisoara\}$

4. Transition Model:

$RESULT(Arad, ToZerind) = Zerind$

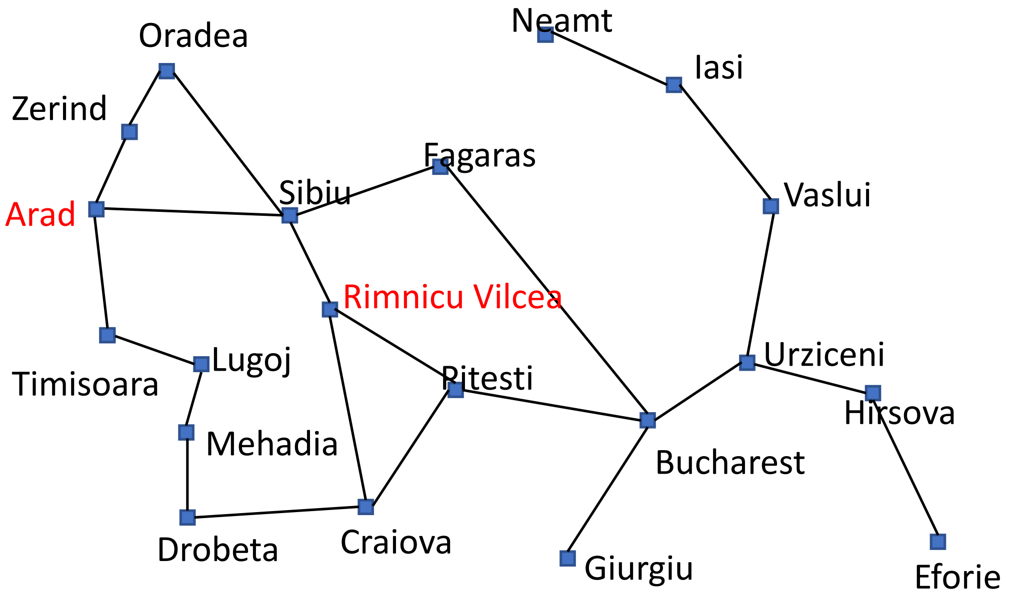
5. Goal-State: Rimnicu Vilcea

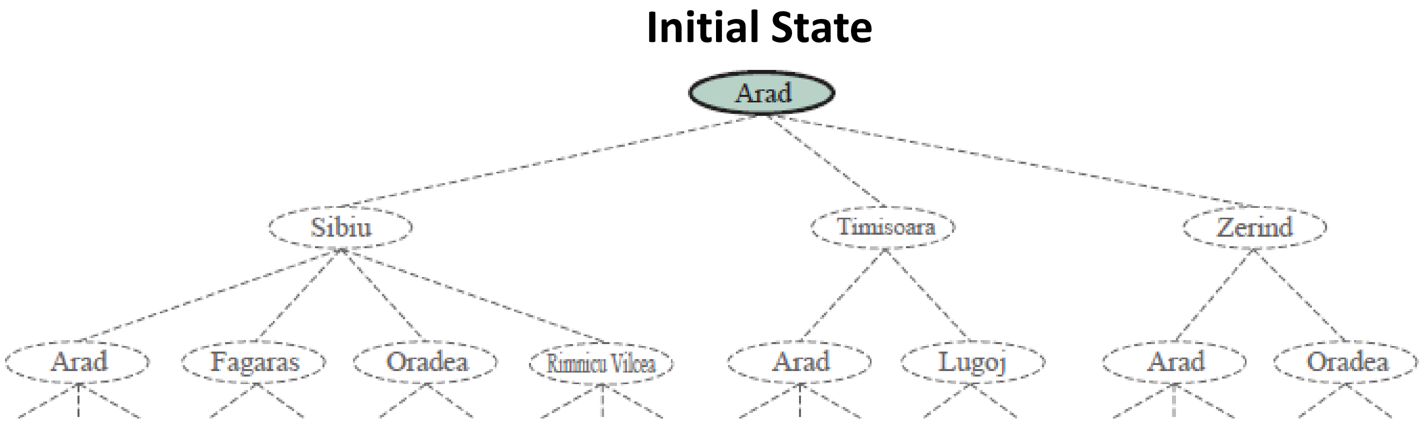
$IS - GOAL(Rimnicu\ Vilcea)$

6. Action-Cost:

Distance between cities

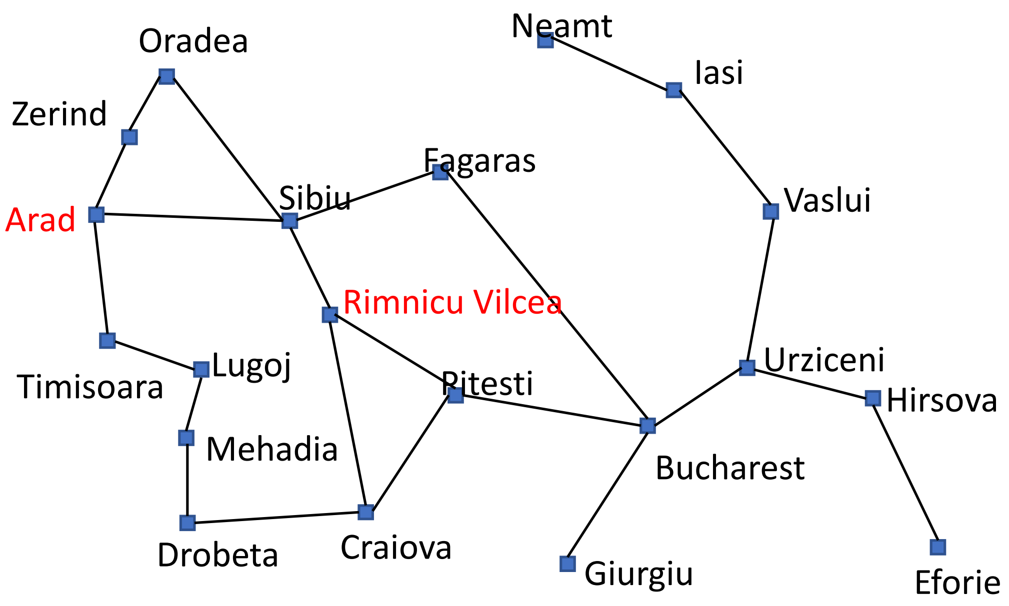
Search Tree: Arad to Rimnicu Vilcea

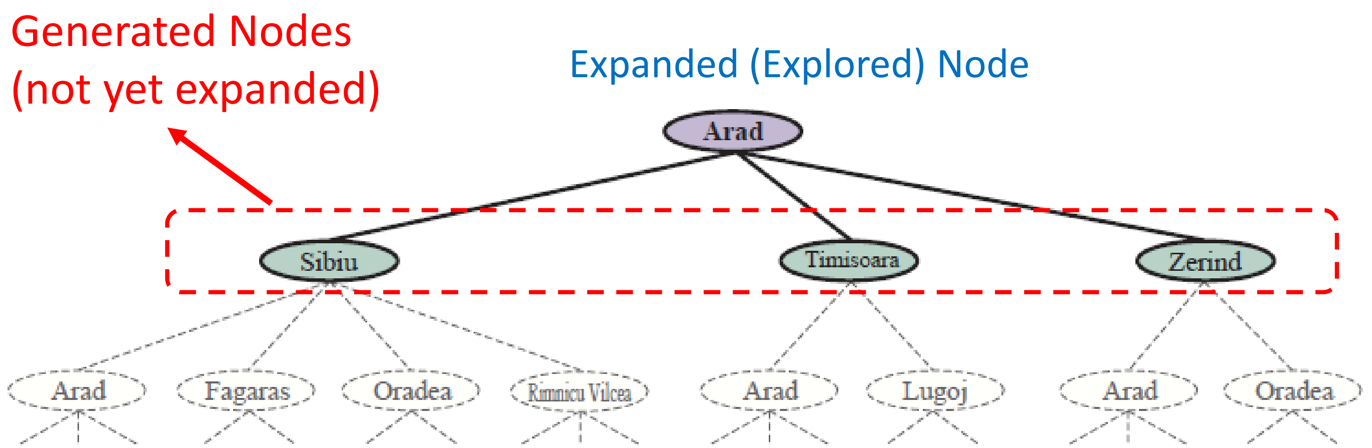




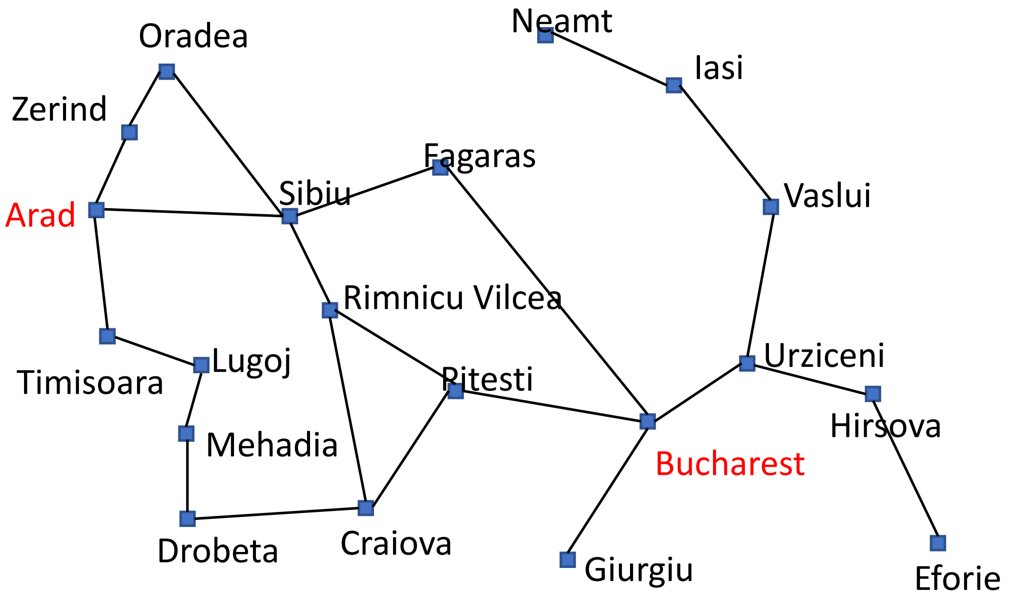
Actions at InitialState: $\{ToZerind, ToSibiu, ToTimisoara\}$

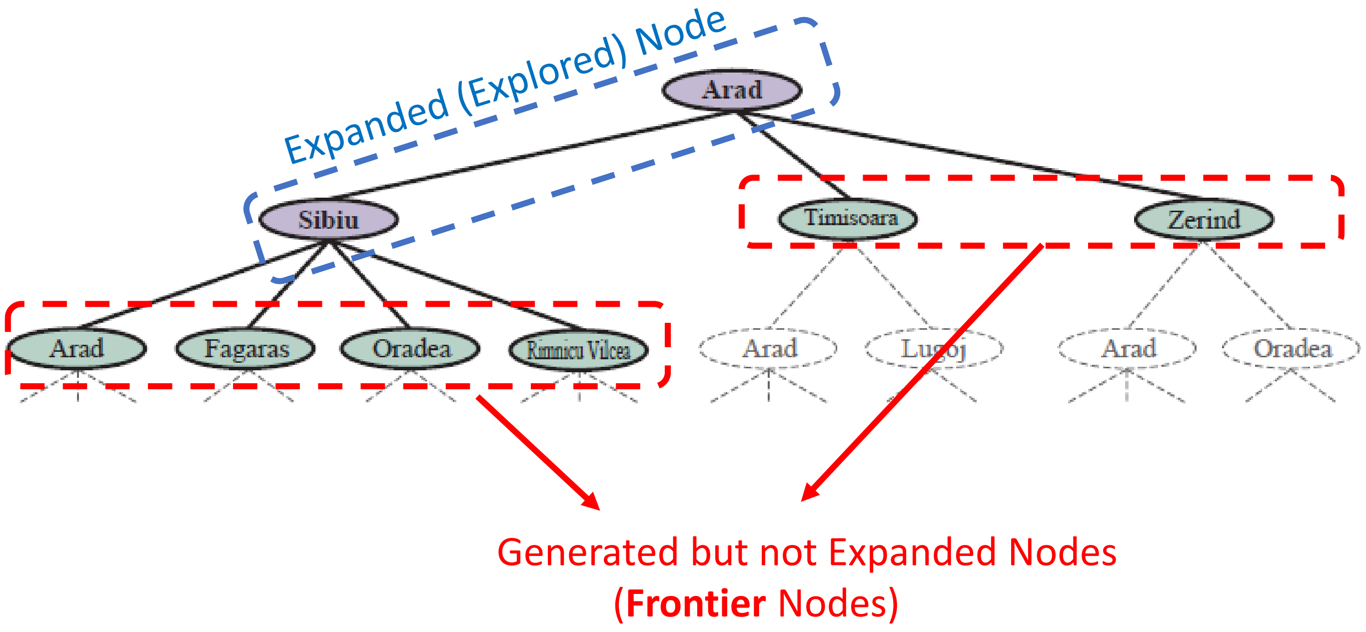
Search Tree: Arad to Rimnicu Vilcea





Search Tree: Arad to Rimnicu Vilcea



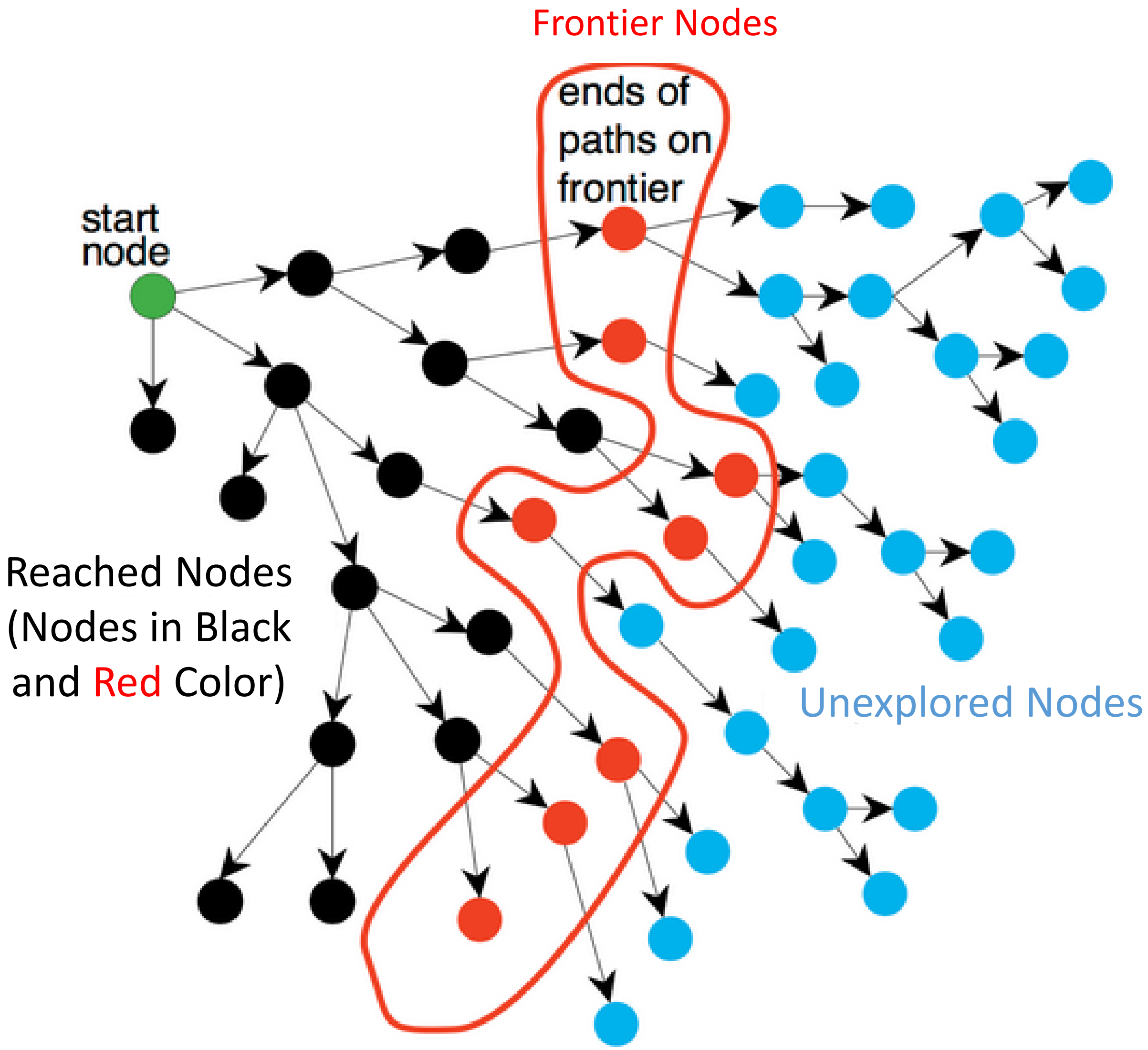


Reached Nodes:

Expanded Nodes and Frontier Nodes

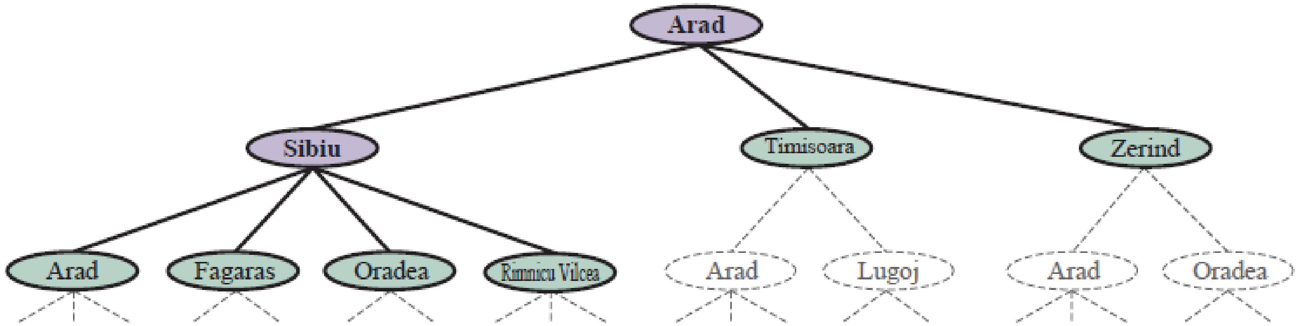
Search Tree Terminology

- Reached Nodes
  - Nodes in black and red color
  - Includes both expanded and not expanded nodes
- Frontier Nodes
  - Nodes in red color
  - Generated but not yet expanded
- Unexplored Nodes
  - Nodes in blue color
  - Not yet generated



Data Structures for Search Tree

- Generation of search tree

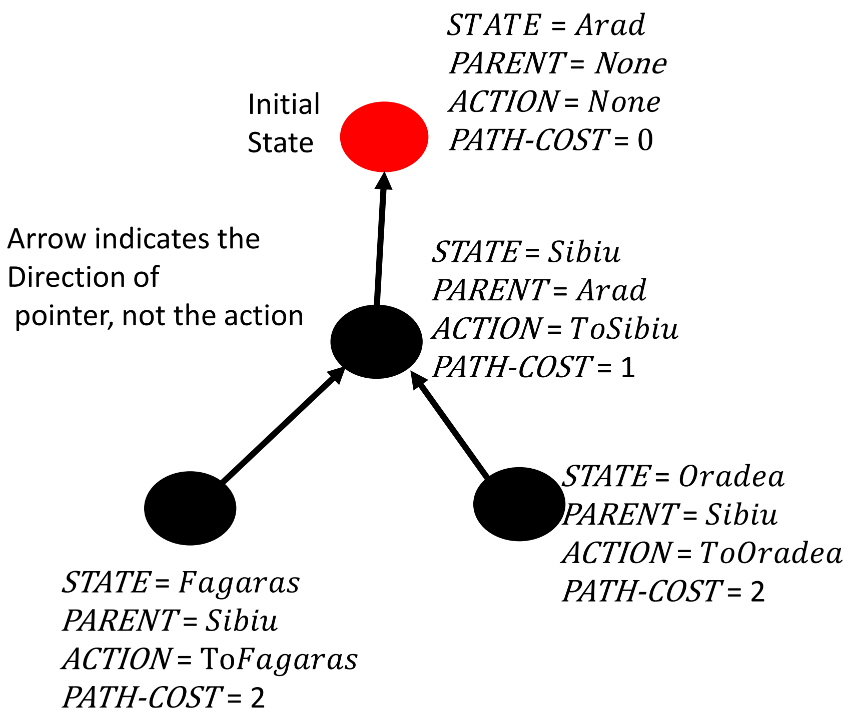


- Representation of
  - Reached nodes
  - Frontier nodes

Generation of Search Tree

- Node (node) contains four components:
  - State Information ($node.STATE$)
    - Eg: $Sibiu$
  - Parent Information ($node.PARENT$)
    - Pointer from child to parent node
    - Need this for backtracking
  - Action ($node.ACTION$)
    - Action at parent node that leads to this node
    - Need this for final solution
  - Path-Cost ($node.PATH-COST$)
    - Cost of reaching this node from initial state
    - For checking optimality of a path

Node Data Structure



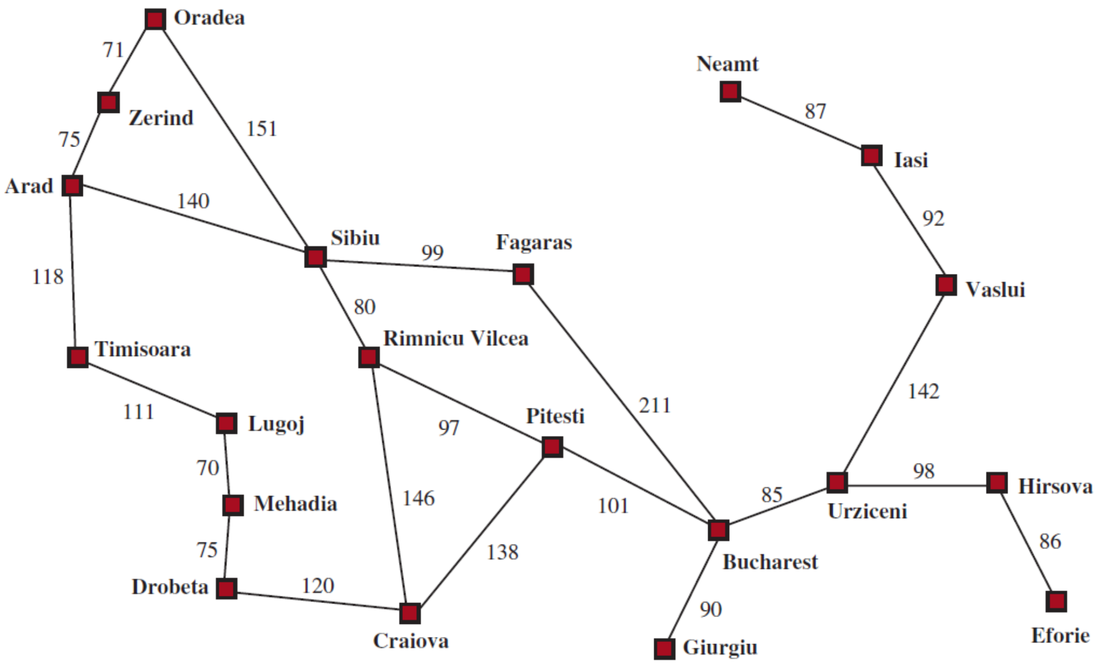
Expanding a node

Node generator function

function $\text{EXPAND}(\mathit{problem}, \mathit{node})$ **yields** nodes
$s \leftarrow \mathit{node}.\text{STATE}$
for each action **in** $\mathit{problem}.\text{ACTIONS}(s)$ **do**
$s' \leftarrow \mathit{problem}.\text{RESULT}(s, \mathit{action})$
$\mathit{cost} \leftarrow \mathit{node}.\text{PATH-COST} + \mathit{problem}.\text{ACTION-COST}(s, \mathit{action}, s')$
yield $\text{NODE}(\text{STATE}{=}s', \text{PARENT}{=}\mathit{node}, \text{ACTION}{=}\mathit{action}, \text{PATH-COST}{=}\mathit{cost})$

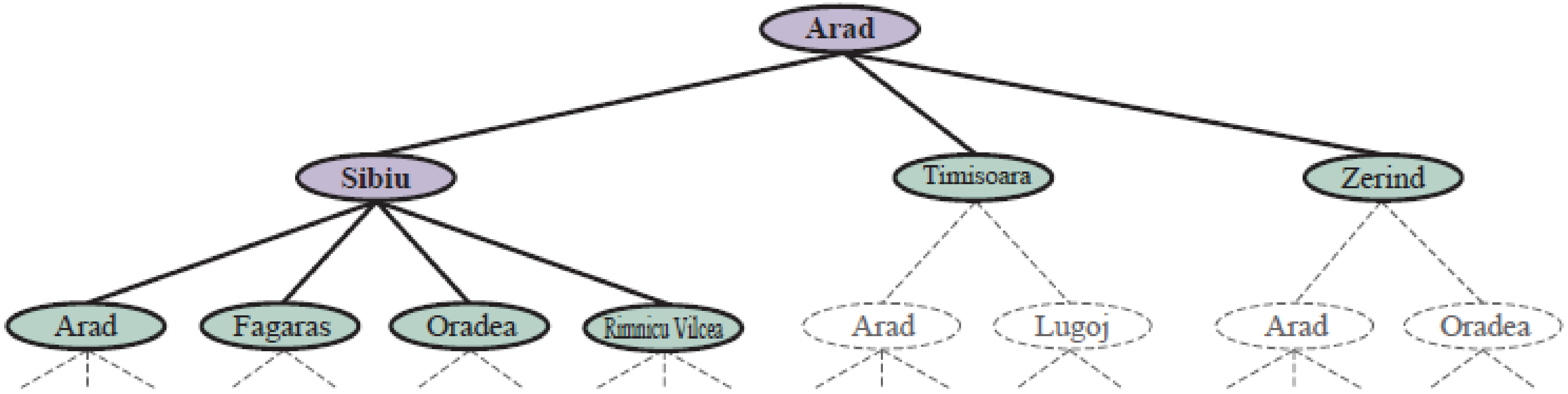
Yields a child node of the given node with state $node.STATE$ for each call

Expanding a node: Example

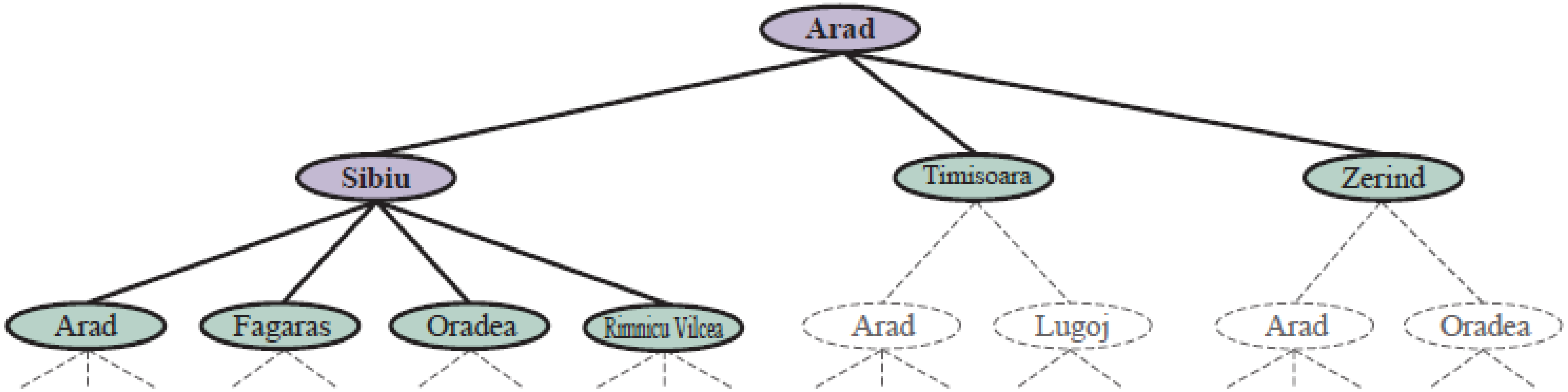


function EXPAND($problem$, $node$) **yields** nodes
  $s \leftarrow node.$STATE
  for each $action$ **in** $problem.$ACTIONS(s) **do**
    $s' \leftarrow problem.$RESULT(s, $action$)
    $cost \leftarrow node.$PATH-COST + $problem.$ACTION-COST(s, $action$, s')
    yield NODE(STATE=s', PARENT=$node$, ACTION=$action$, PATH-COST=$cost$)

Expanding a Node

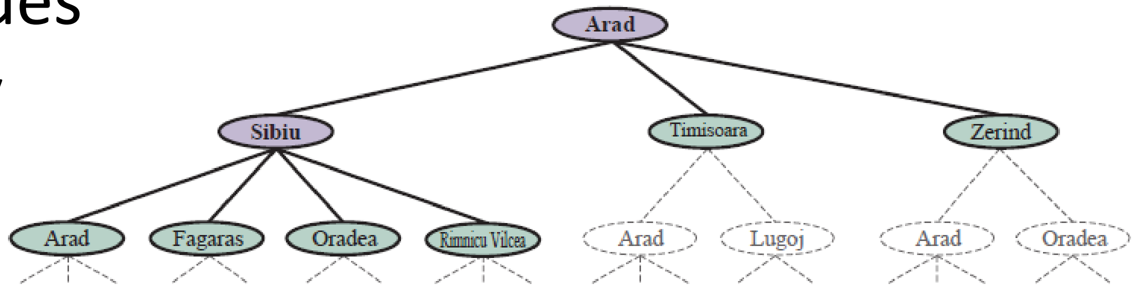


Node Vs State



Representation of $reached$ Nodes

- Need a data structure to store $reached$ nodes
  - To check whether a node with a state is already generated
    - Helps in pruning the tree



- Should support quick insert and quick access

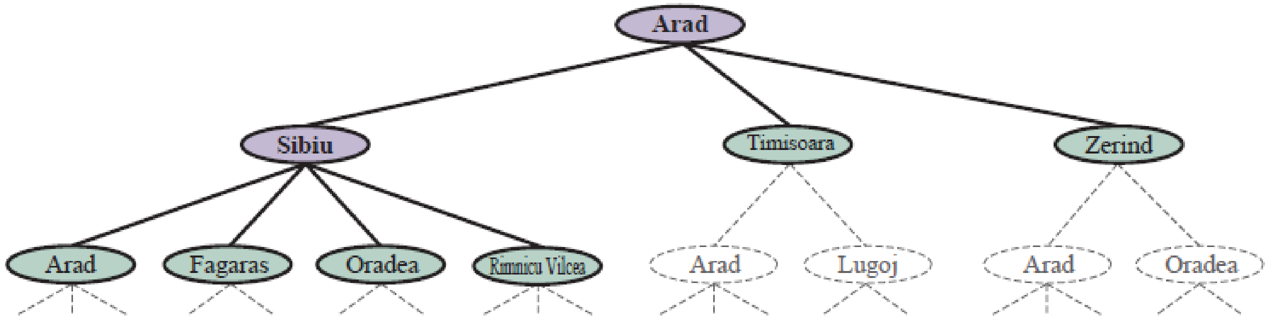
- Can be implemented using Hash Table
  - Key-value pairs
    - Key is state
    - Value is node
  - Dictionary in Python

https://en.wikipedia.org/wiki/Hash_table

Representation of $frontier$ Nodes

- Need a data structure to store $frontier$ nodes

- Should support following operations
  - $Is - Empty(frontier)$
  - $POP(frontier)$
  - $TOP(frontier)$
  - $ADD(node, frontier)$

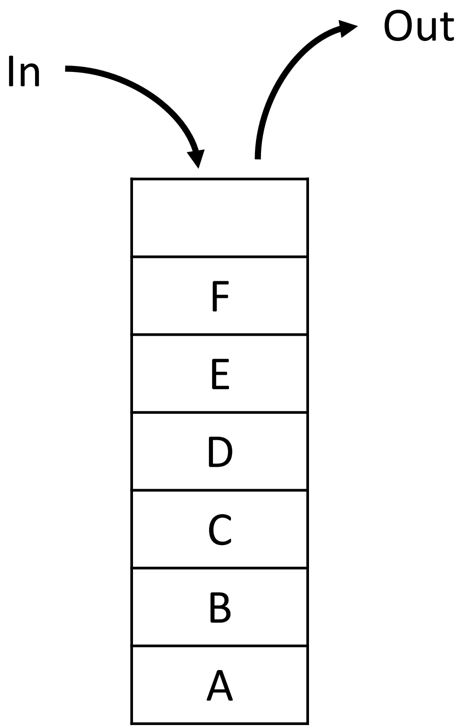


- Implementation depends on search strategy
  - Last-in-First-Out Queue (stack)
  - First-in-First-Out Queue (queue)
  - Priority Queue

Representation of $frontier$ Nodes

- Last-In-First-Out Queue (Stack)

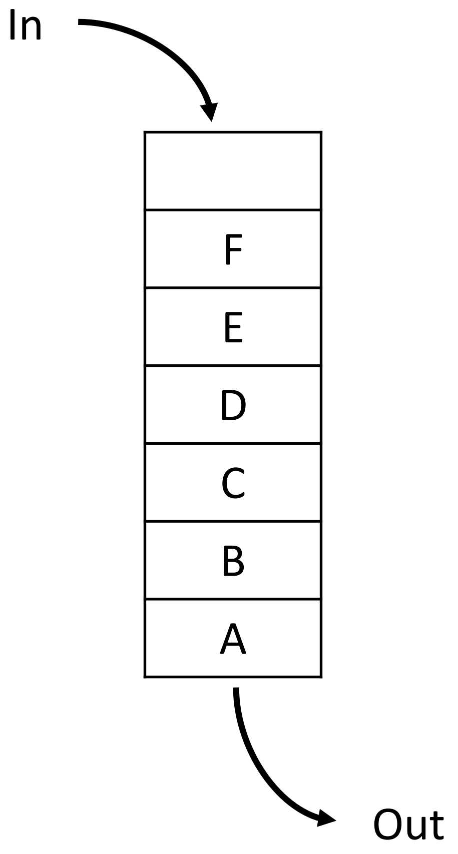
Similar to a stack of plates, adding or removing is only possible at the top.



https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Representation of $frontier$ Nodes

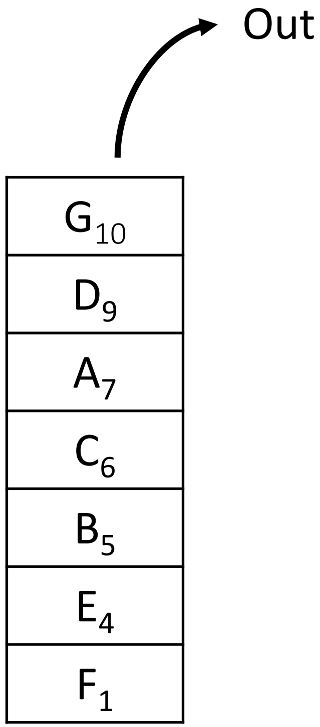
- First-In-First-Out Queue



Representation of $frontier$ Nodes

- Priority Queue
  - Nodes are ordered based on evaluation of nodes

Insertion depends
on value of node



https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Building Blocks of Search Tree

Problem:

Initial State: $problem.INITIAL - STATE$

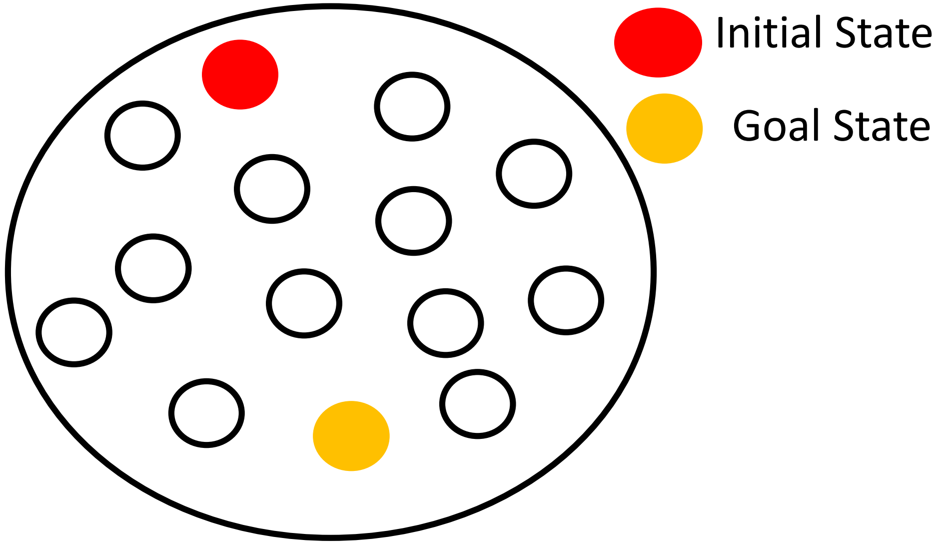
Goal: $problem.Is - GOAL(s)$

Actions: $problem.ACTIONS(s)$

Transition Model: $problem.RESULT(s, action)$

Action-Cost Function: $problem.ACTION - COST(s, action, s')$

State Space Problem



Node:

State: $node.STATE$

Parent: $node.PARENT$

Path-Cost: $node.PATH - COST$

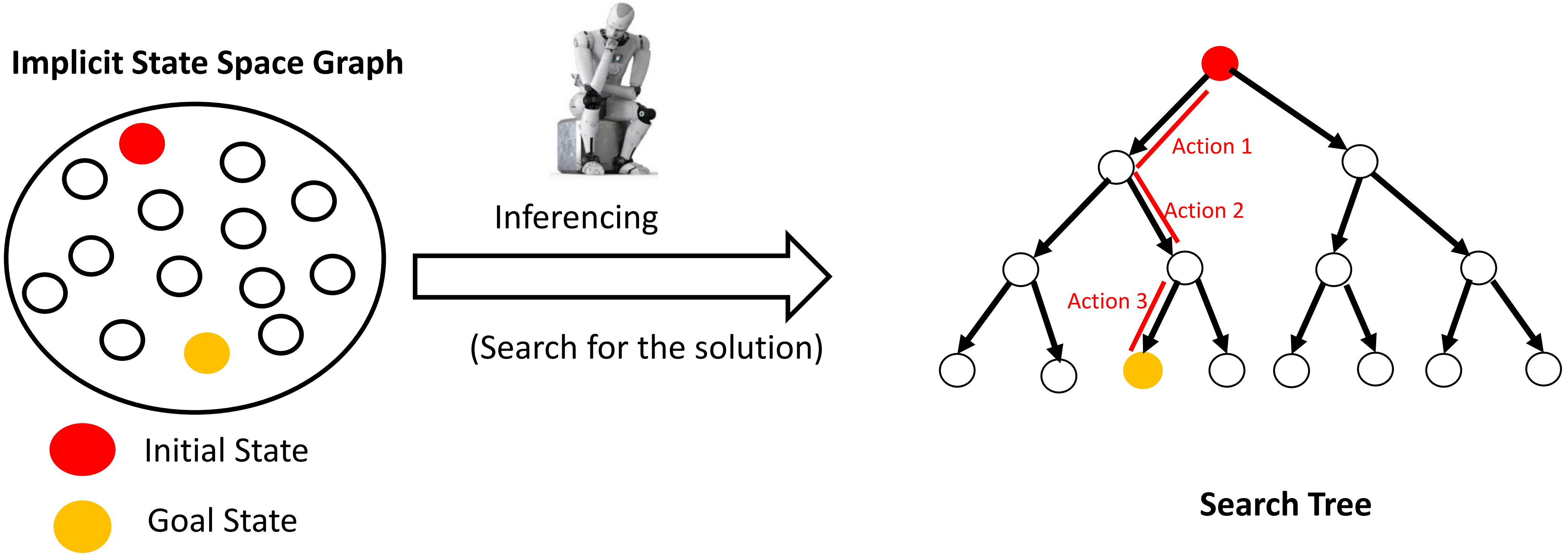
Action: $node.ACTION$

function EXPAND($problem$, $node$) **yields** nodes
  $s \leftarrow node.$STATE
  for each $action$ **in** $problem.$ACTIONS(s) **do**
    $s' \leftarrow problem.$RESULT(s, $action$)
    $cost \leftarrow node.$PATH-COST + $problem.$ACTION-COST(s, $action$, s')
    yield NODE(STATE=s', PARENT=$node$, ACTION=$action$, PATH-COST=$cost$)

Agenda

- Workflow
- Goal Formulation
- Problem Formulation or Modeling
- Building Blocks of Search Tree
- Inference

Inference (Search for the solution)



Inferencing:

Finding the path from Initial State to Goal State

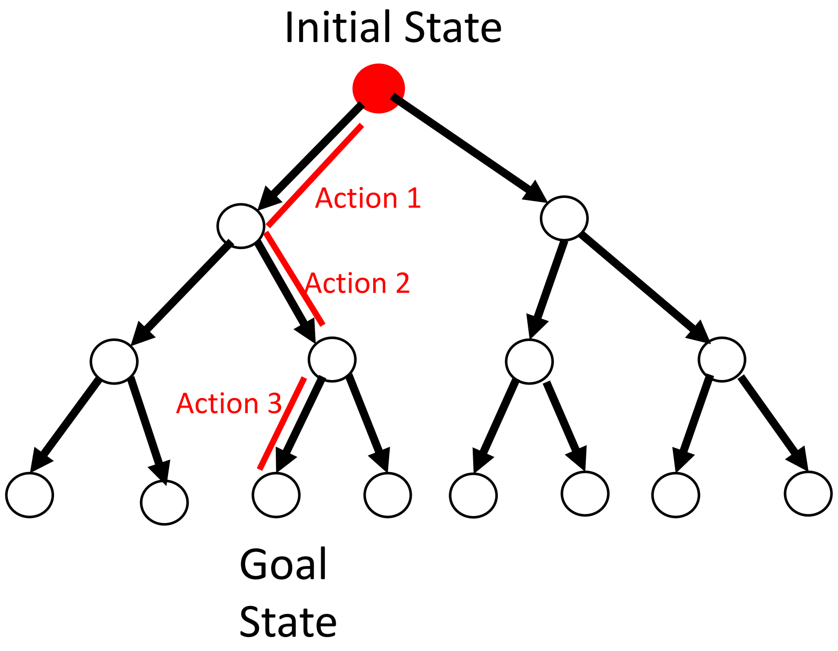
Solution: Sequence of Actions

[Action 1, Action 2, Action 3]

Uninformed Search Algorithms

- Based on storage of nodes
  - Graph Search
  - Tree Search

- Based on Traversal Strategy
  - Breadth-First Search
  - Depth-First Search
  - Depth-Limited Search
  - Iterative Deepening Search
  - Bidirectional Search
  - Uniform Cost Search



Why are they called uninformed search algorithms?

Graph Search vs Tree Search

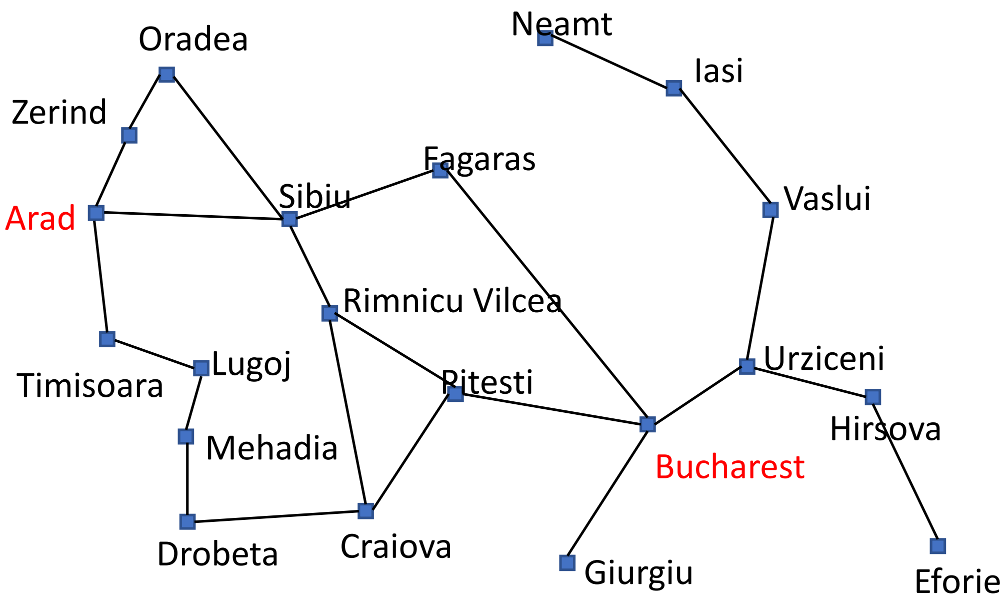
- Graph Search
  - Stores reached nodes
- Tree Search
  - Does not store reached nodes

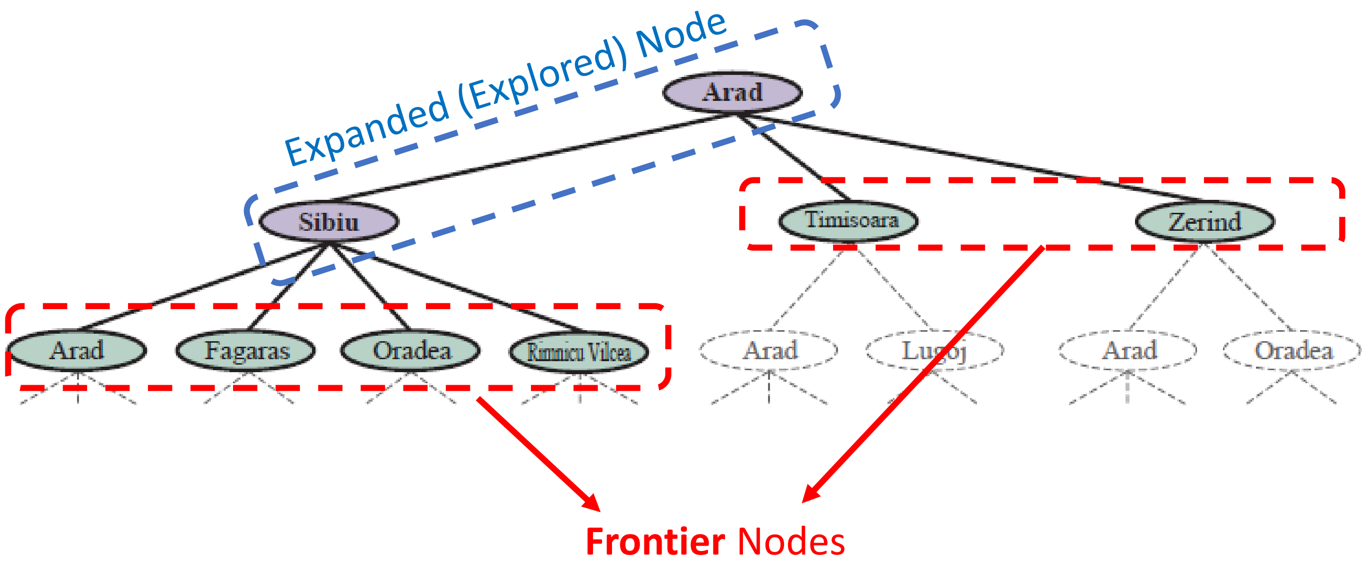
Storing reached nodes?

If model contains

Cycle or Loopy Paths
Redundant Paths

Storing reached nodes helps in pruning search tree





Reached Nodes:

Expanded Nodes and Frontier Nodes

Breadth-First Search (BFS)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← 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
```

graph search or tree search?

Trace of BFS

Node	s	Is-Goal(s)	Frontier	Reached

Search Tree of BFS

Performance of BFS

- Complete:
  - ?
- Optimal
  - ?
- Time Complexity
  - ?
- Space Complexity
  - ?

Depth-First Graph Search (DFGS)

```
function DEPTH-FIRST-SEARCH(problem):
        node ← NODE(problem.INITIAL-STATE)
        if problem.IS-GOAL(node.STATE) then return node
        frontier ←a LIFO queue (stack) with node as element
        reached ← {problem.INITIAL-STATE}
        while not IS-EMPTY(frontier) do
                node ← POP(frontier)
                for each child in EXPAND(problem, node) do
                        s ← 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
```

Trace of DFGS

Node	s	Is-Goal(s)	Frontier	Reached

Search Tree of DFGS

Performance of DFGS

- Complete:
  - ?
- Optimal
  - ?
- Time Complexity
  - ?
- Space Complexity
  - ?

Depth-First Tree Search (DFTS)

function DEPTH-FIRST-TREE-SEARCH(*problem):*

node ← NODE(*problem*.INITIAL-STATE)

if *problem*.IS-GOAL(*node*.STATE) **then** return *node*

frontier ←a LIFO queue (stack) with *node* as element

while not IS-EMPTY(*frontier*) **do**

node ← POP(frontier)

if *problem*.IS-GOAL(*node*.STATE) **then** return *node*

if not $IS - CYCLE$(node) **do**

for each $child$ in *EXPAND*(*problem*, *node*) **do**

add $child$ to $frontier$

return *failure*

How to check cycles?

Trace of DFTS

Node	s	Is-Goal(s)	Is-Cycle	Frontier

Search Tree of DFTS

Performance of DFTS

- Complete
  - ?
- Optimal
  - ?
- Time Complexity
  - ?
- Space Complexity
  - ?

Depth-limited Search (DLS)

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

tree search or graph search?

Trace of DLS

Node	Is-Goal(s)	Depth(Node) > l	Is-Cycle(Node)	Frontier	Result

Search Tree of DLS

Performance of DLS

- Complete
  - ?
- Optimal
  - ?
- Time Complexity
  - ?
- Space Complexity
  - ?

Iterative Deepening Search (IDS)

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

Trace of IDS

Trace of DLS

Node	Is-Goal(s)	Depth(Node) > l	Is-Cycle(Node)	Frontier	Result

Search Tree of IDS

Overhead in IDS

- Number of nodes generated with solution at depth-d:
  - Breadth-First Search
    - $N(BFS) = 1 + b^1 + b^2 + \cdots + b^d$
  - Iterative Depth-Limited Search
    - $N(IDLS) = (d)b + (d-1)\, b^2 + (d-2)b^3 + \cdots + (1)b^d$

Assumption: same number of branches for each node

Overhead in IDS

- Let $b = 10$ and $d = 5$
  - $N(BFS) = 111{,}110$
  - $N(IDLS) = 123{,}450$

- $Overhead = \frac{N(IDLS) - N(BFS)}{N(BFS)}$ = 11%

- IDLS: Huge savings in memory with little overhead in number of nodes generated compared to BFS

Performance of IDS

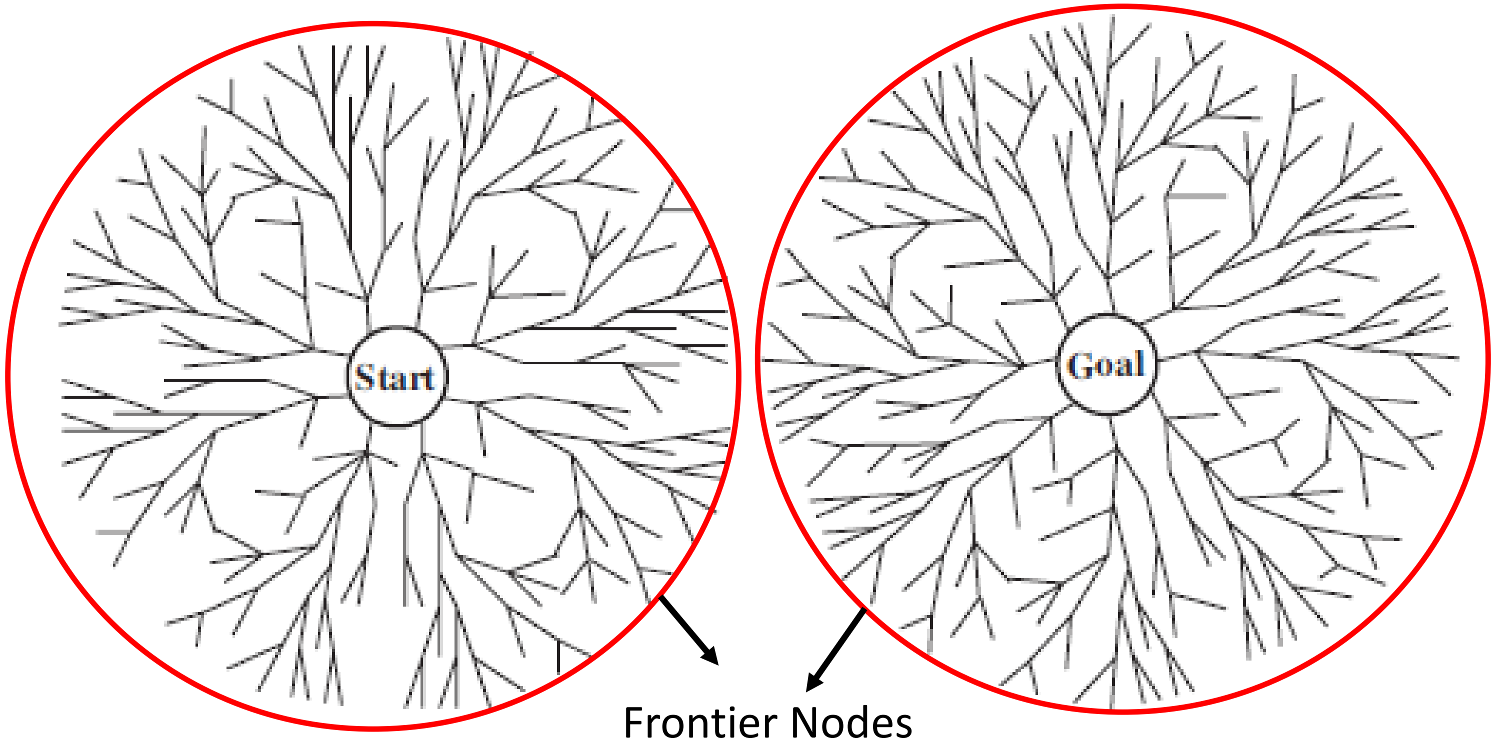
- Complete
  - ?
- Optimal
  - ?
- Time Complexity
  - ?
- Space Complexity
  - ?

Bidirectional Search

- Idea:
  - Search from initial and goal states
  - If frontiers of both meet, solution is found

Motivation:

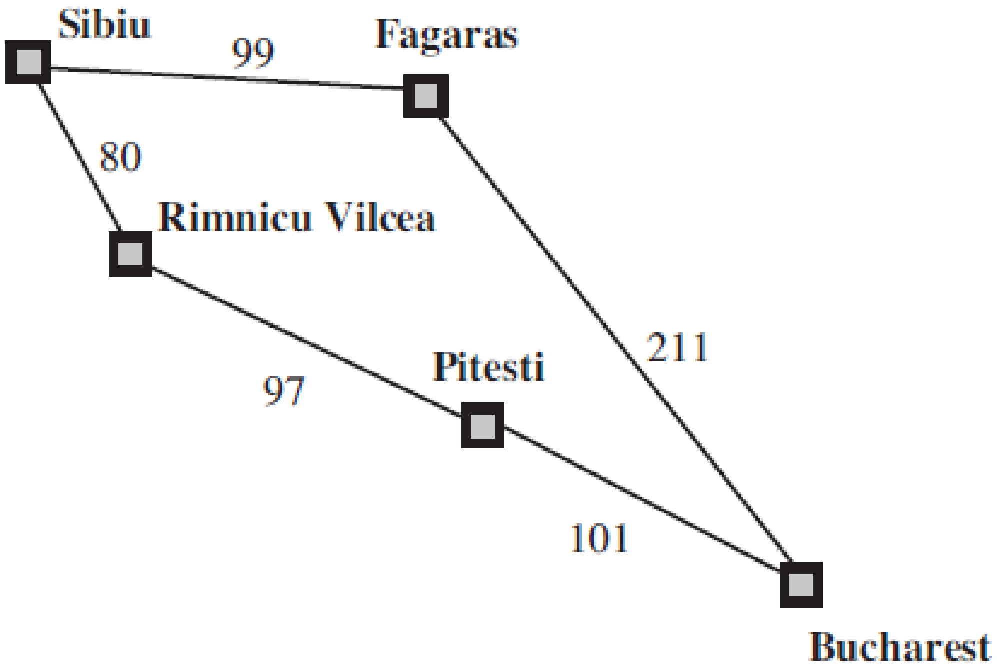
$$b^{\frac{d}{2}} + b^{\frac{d}{2}} < b^d$$



Performance of Bidirectional Search

- Complete
  - ?
- Optimal
  - ?
- Time Complexity
  - ?
- Space Complexity
  - ?

Uniform-Cost Search: Romania Map

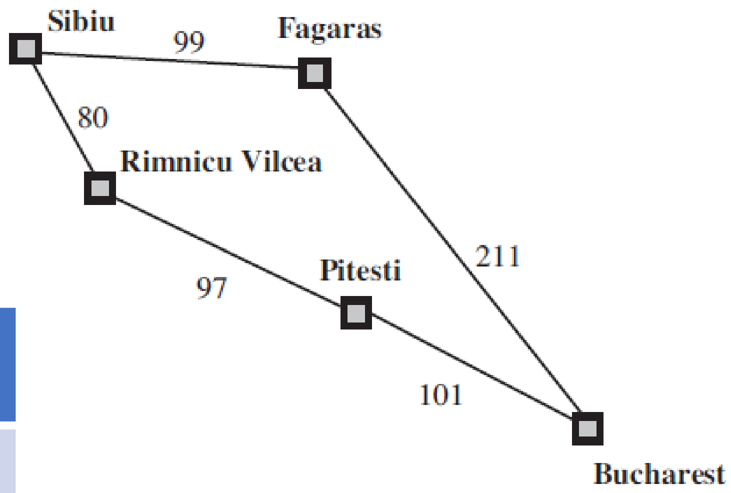


Uniform-Cost Search

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

function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

Trace of UCS

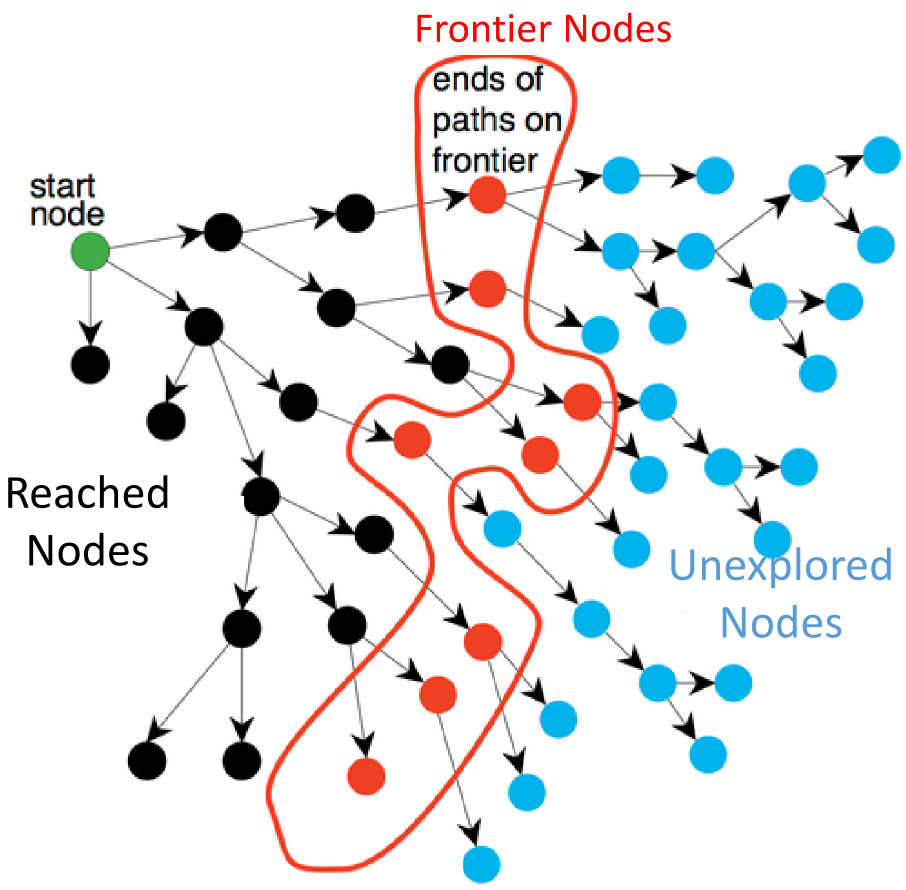


Node	Is-Goal(Node)	s	Frontier	Reached

UCS Search Tree

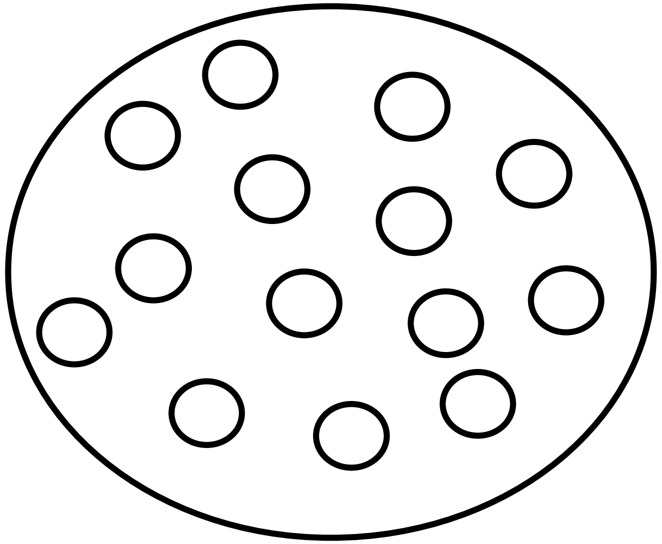
Summary of Uninformed Search

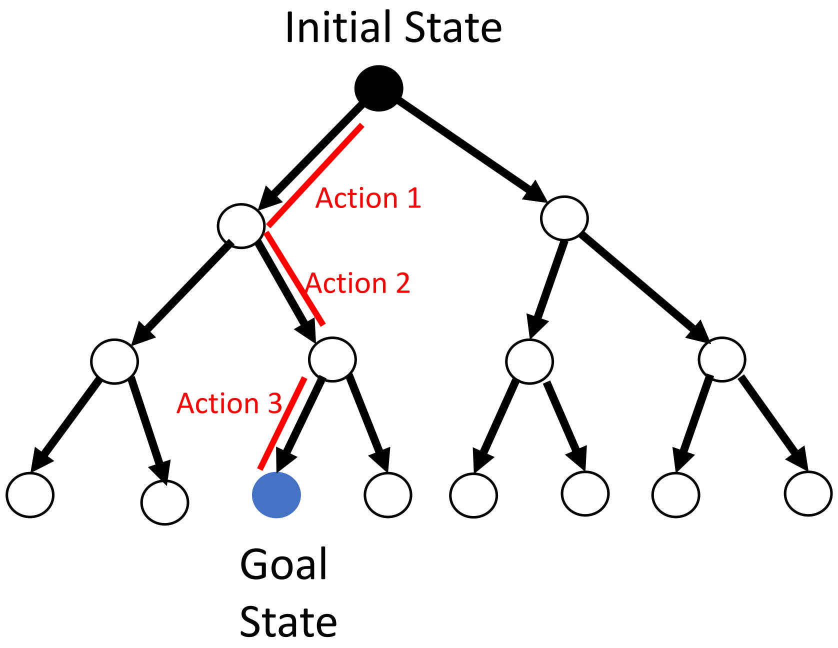
- Classification based on traversal methods
  - Breadth-First Search: Frontier is a Queue (FIFO)
  - Depth-First Search: Frontier is a stack (LIFO)
  - Uniform-Cost Search: Frontier is a priority queue
- Tree-Search Vs Graph-Search
  - Tree-Search
    - Doesn’t store reached nodes
    - Leads to cycles and redundant paths
    - Cycles can be avoided with Cycle-Check
  - Graph-Search
    - Maintains a set of reached states
    - Avoids cycles and redundant paths



Why is it called uninformed search?

Also called Blind Search or Brute Force Search





When can an agent use uninformed search algorithms?

Image: https://www.shutterstock.com/search/android+robot+thinking

Design Space

Dimension	Values
Environment	Static, Dynamic
Representation Scheme	States, Features, Relations
Observability	Fully Observable, Partially observable
Parameter Types	Discrete, Continuous
Uncertainty	Deterministic, Stochastic
Learning	Knowledge is given (known), knowledge is learned (unknown)
Number of Agents	Single Agent, Multiple Agent

Need well-defined goal
Interaction is offline
No limits on resources (memory and time)

AIMA3e (Section 2.3.2) and **AIFCA (Section 1.5)**

Conclusion

- Atomic Representation of World
- Implicit representation of state space graphs
- Inferencing using search algorithm
- Issues related to cycles in state space graphs