IT5005 Artificial Intelligence

1. Uninformed and Informed Search

Sirigina Rajendra Prasad
AY2025/2026: Semester 1

Uninformed/Informed Search: Applications

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

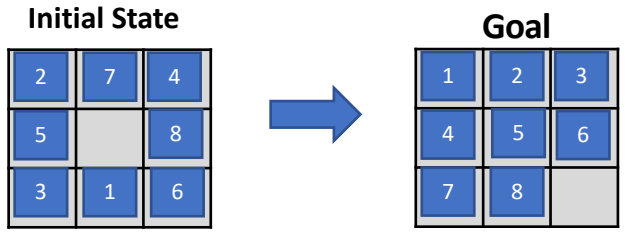
- Games:
  - Pacman, etc.
  - Maze Navigation

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

Examples

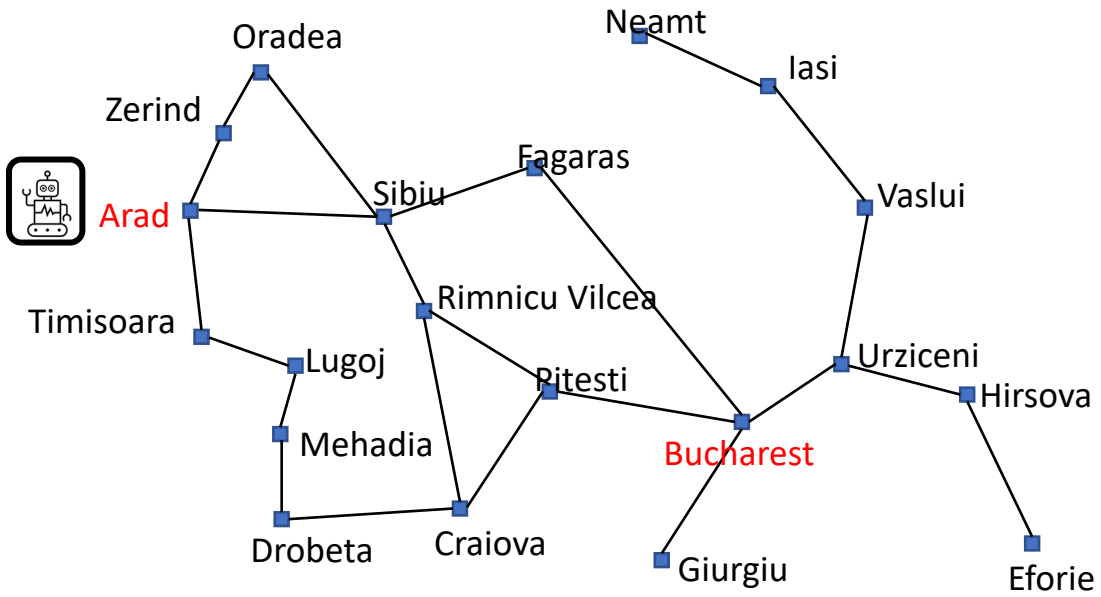
- Two-Room Vacuum World:

- 8-Puzzle



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

Examples



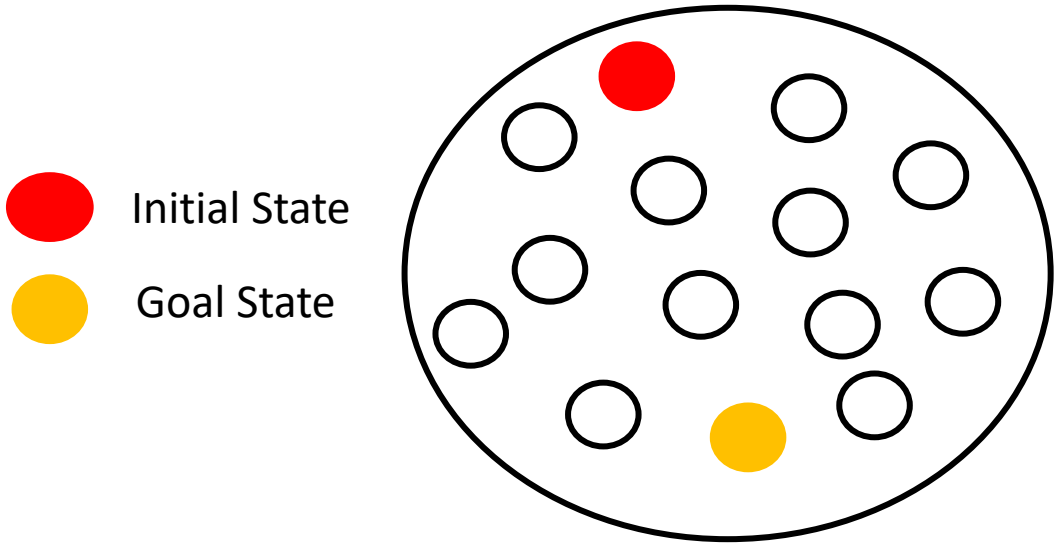
“Map of Romania”

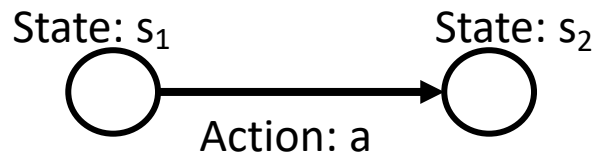
- **Route Planning**

  Goal: Find a path from Arad to Bucharest

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

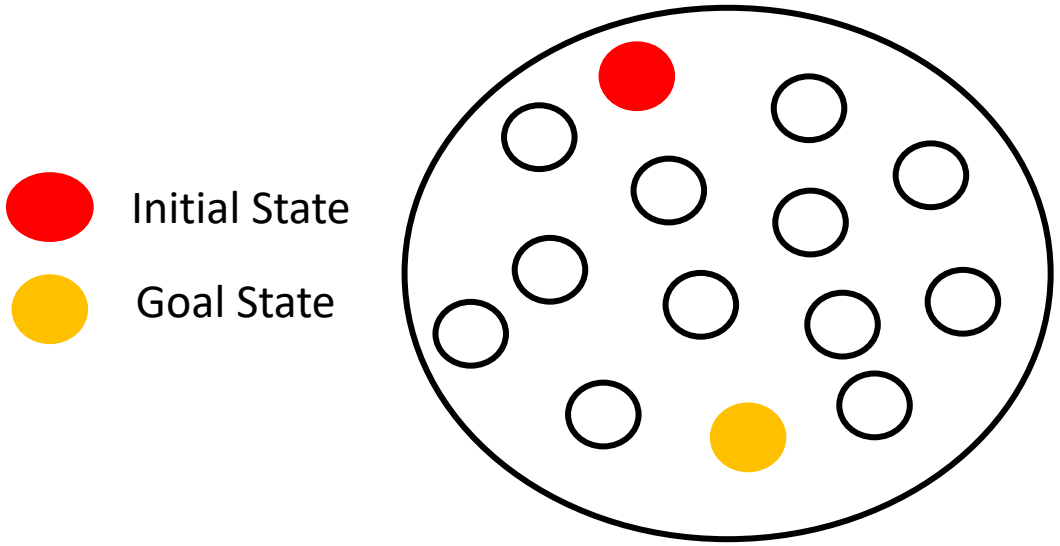


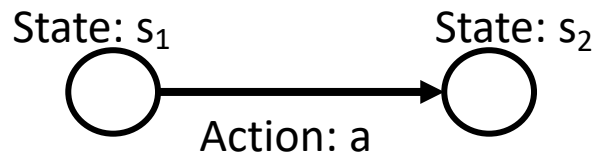


Action Cost: $c(s_1,a,s_2)$

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: $c(s_1,a,s_2)$

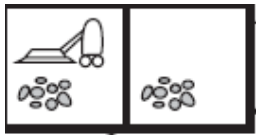
Modeling or Problem Formulation

```
# Modelling
class Problem(object):
    """The abstract class for a formal problem. A new domain subclasses this,
    overriding `actions` and `results`, and perhaps other methods.
    The default heuristic is 0 and the default action cost is 1 for all states.
    When you create an instance of a subclass, specify `initial`, and `goal` states
    (or give an `is_goal` method) and perhaps other keyword args for the subclass."""

    def __init__(self, initial=None, goal=None, **kwds):
        self.__dict__.update(initial=initial, goal=goal, **kwds)

    def actions(self, state):         raise NotImplementedError
    def result(self, state, action): raise NotImplementedError
    def is_goal(self, state):         return state == self.goal
    def action_cost(self, s, a, s1): return 1
    def h(self, node):                return 0
```

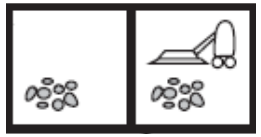
Two-Room Vacuum World: Modeling

1. States

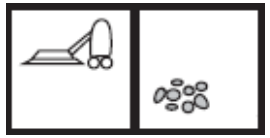
State: 1



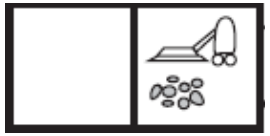
State: 2



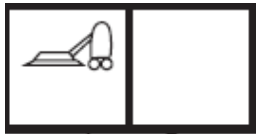
State: 3



State: 4



State: 5



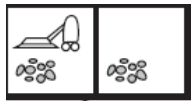
State: 6

State: 7

State: 8

Two-Room Vacuum World: Modeling

2. Initial State



3. Actions:

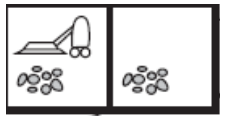
Move Left (L)
Move Right (R)
Suck dirt (S)
No-op (N)

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

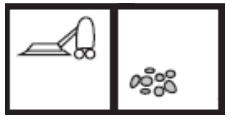
Example:

state: s_1

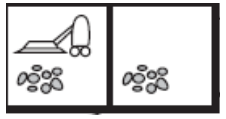


Suck Dirt

state: s_2 = Result(s_1, S)

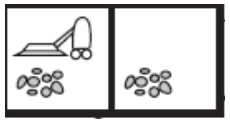


state: s_1



Move Left

state: s_2 = Result(s_1, L)



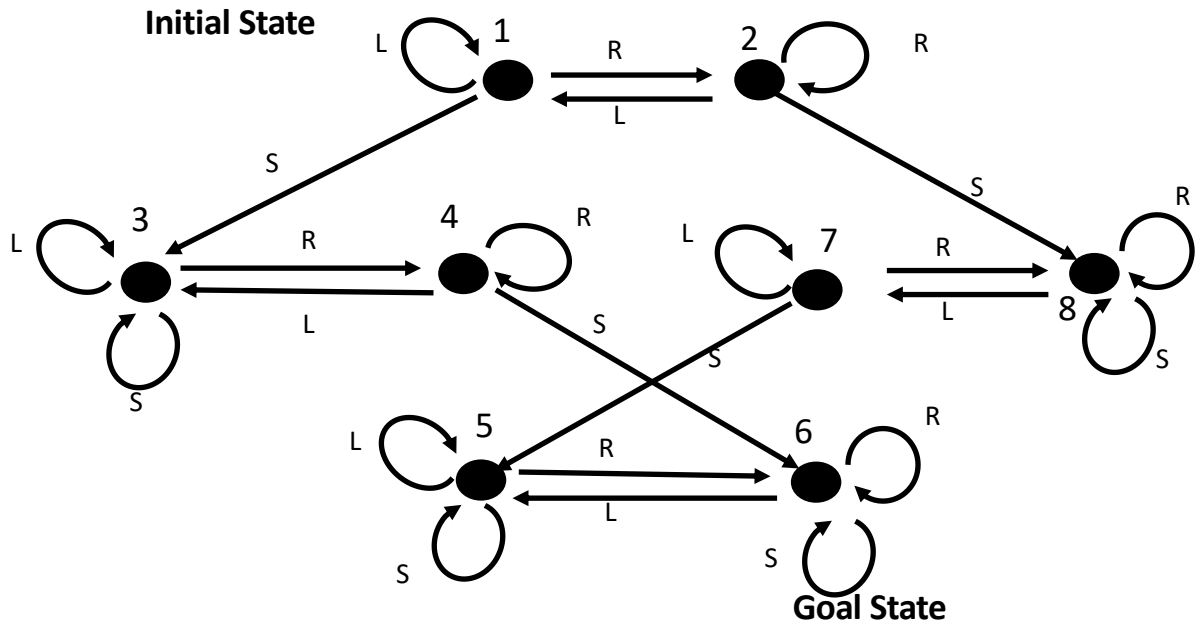
5. Goal State

Check for this state

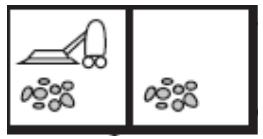
6. Action Cost

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

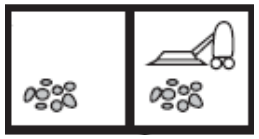
Two-Room Vacuum World: Modeling



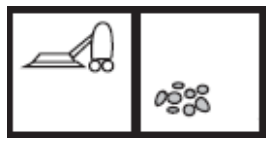
State: 1



State: 2

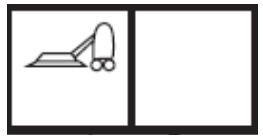


State: 3



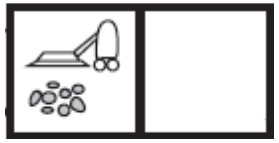
State: 4

State: 5

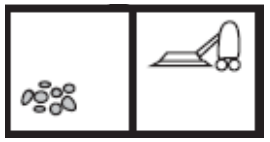


State: 6

State: 7



State: 8



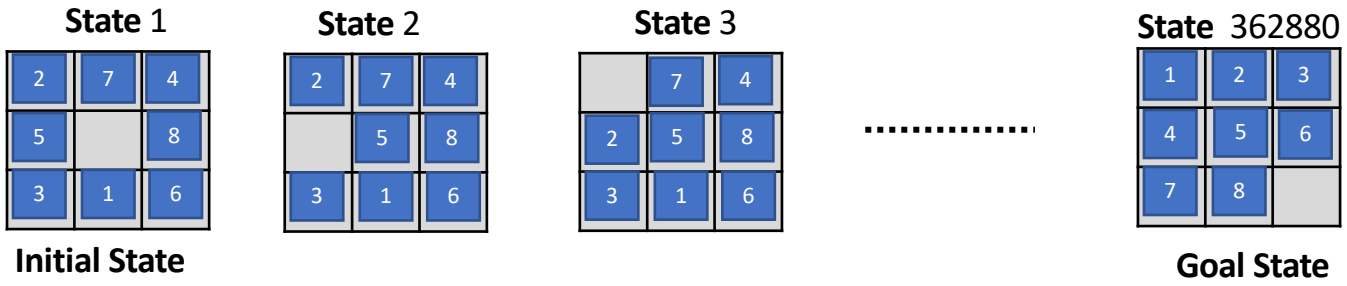
Action Cost:

Suck Dirt (S): 1
Move Left (L): 2
Move Right (R): 2
No-Op: 0

8-Puzzle: Modeling

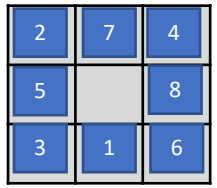
1. State Representation

Number of states = 9! = 362,880



8-Puzzle: Modeling

2. Initial State



3. Actions:

Movement of blank space

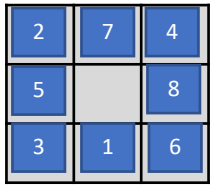
- Slide Left (L)
- Slide Right (R)
- Slide Below (B)
- Slide Above (A)

4. Transition Model:

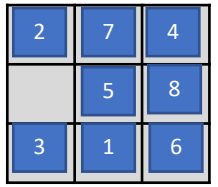
Given a state and action return the resultant state

Eg:

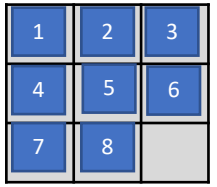
State: s_1



State: s_2 = Result(s_1,L)



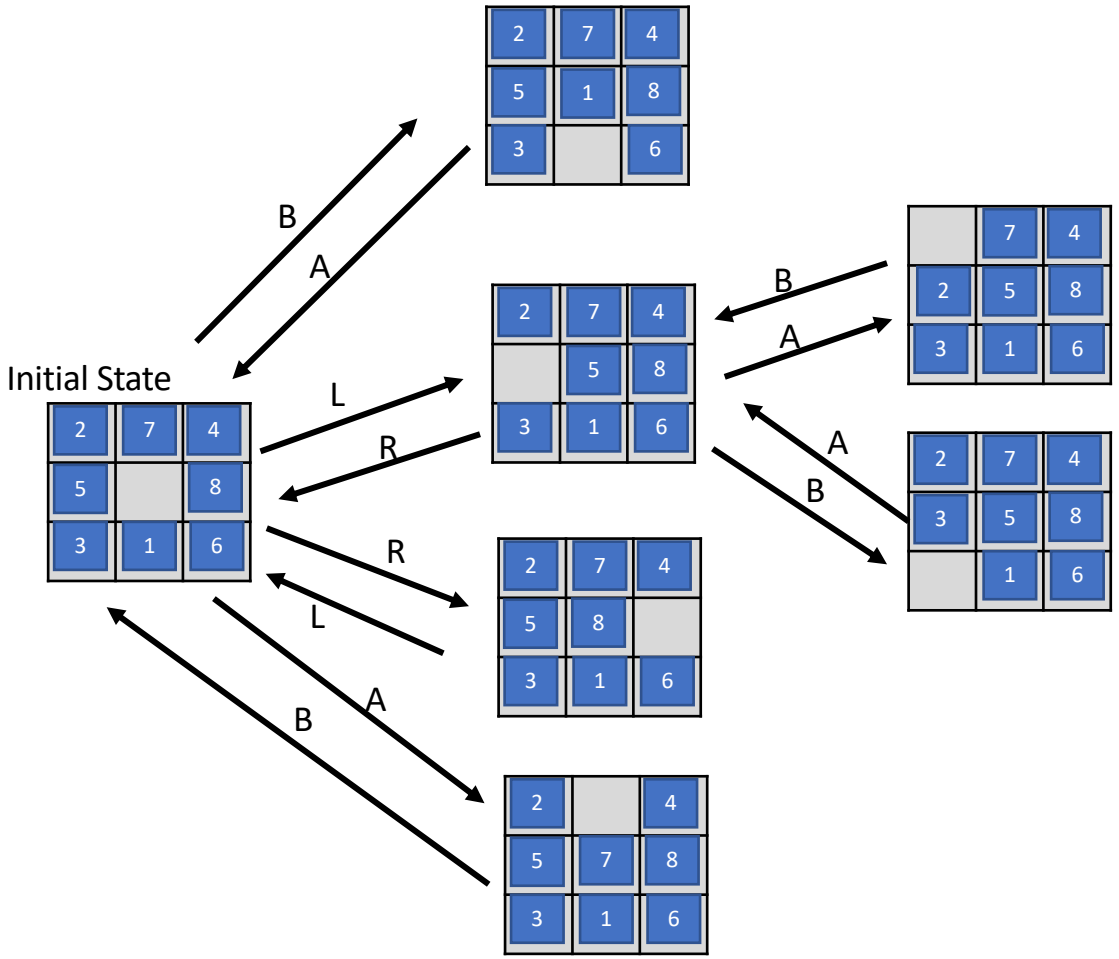
5. Goal State



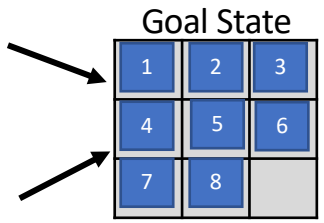
6. Action Cost

Each action costs 1

8-Puzzle: State Space Model



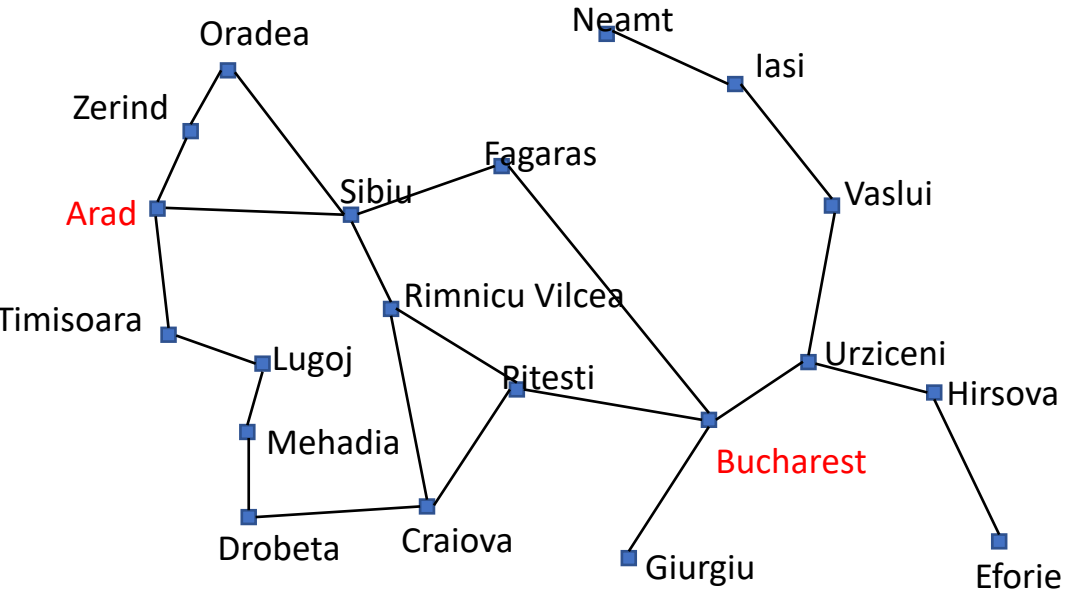
State Space Graph for 8-Puzzle
(impossible to show complete state space graph)



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

Romania Map (Route Planning): Modeling

Problem Formulation



1. States:

$Arad, Oradea,$ etc.

2. Initial State:

$Arad$

3. Actions:

Eg: for $Arad$:

$\{ToZerind, ToSibiu, ToTimisoara\}$

4. Transition Model:

$RESULT(Arad, ToZerind) = Zerind$

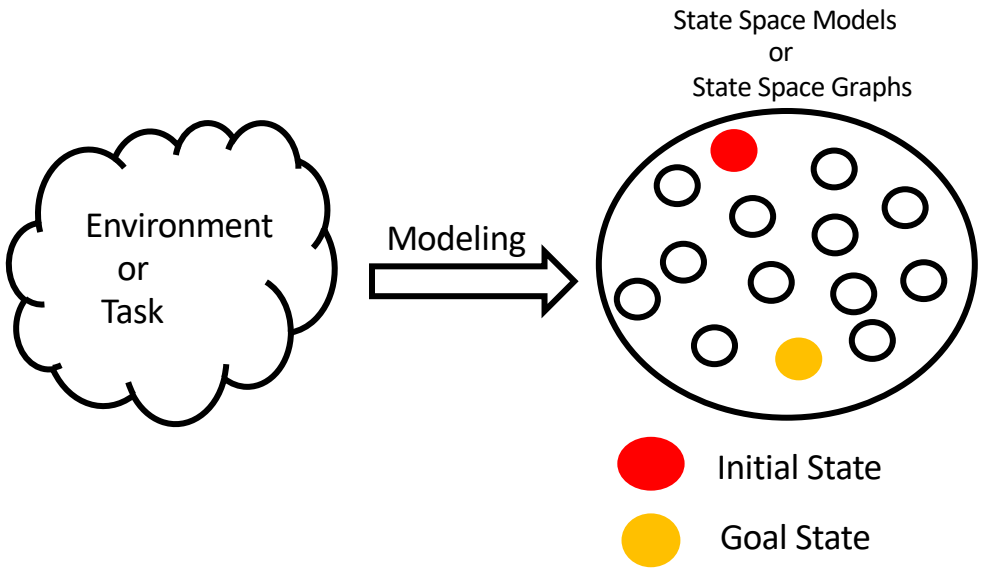
5. Goal-Test:

$IS - GOAL(Bucharest)$

6. Action-Cost:

Distance between cities

So Far: Modeling

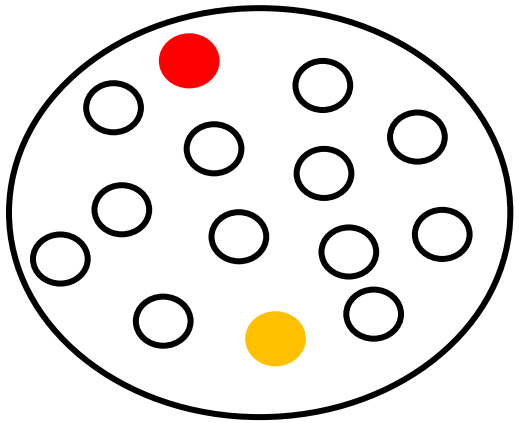


Note:

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

Next: Inference (Search for the solution)

Implicit State Space Graph

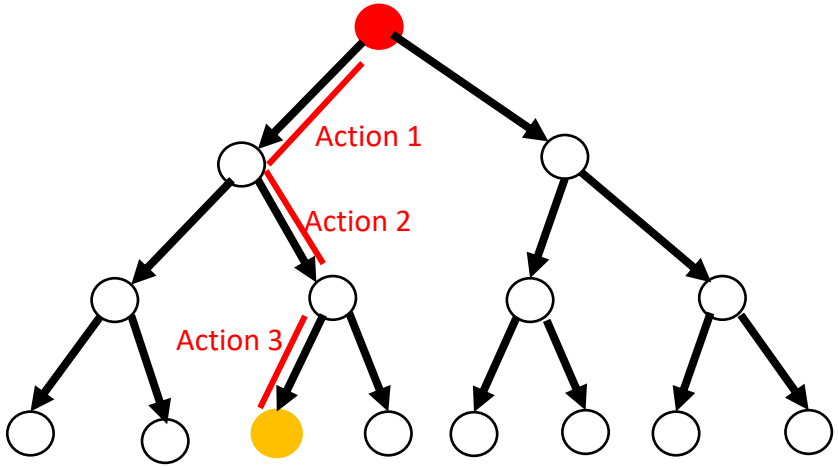


Initial State

Goal State

Inferencing

(Search for the solution)



Search Tree

Inferencing:

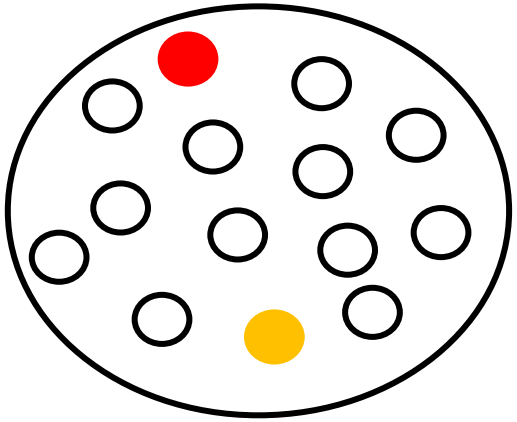
Finding the path from Initial State to Goal State

Building Blocks of Search Tree

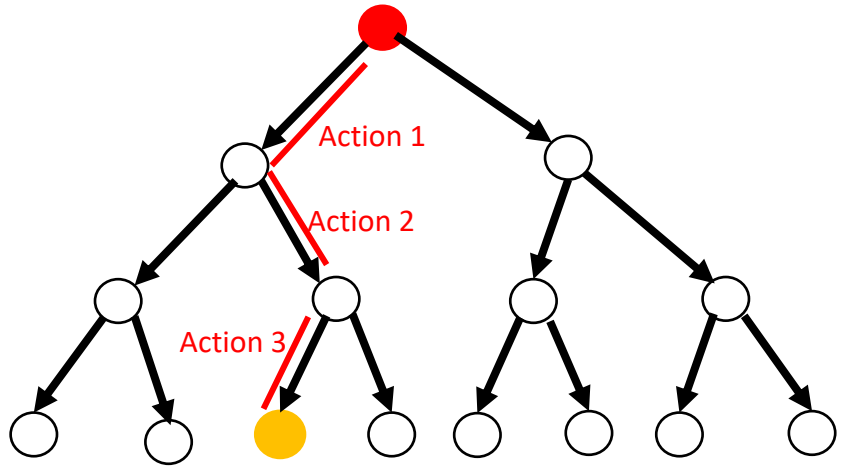
- Need to create the search tree dynamically

- Data structures
  - to generate nodes in search tree
  - for navigating around the search tree

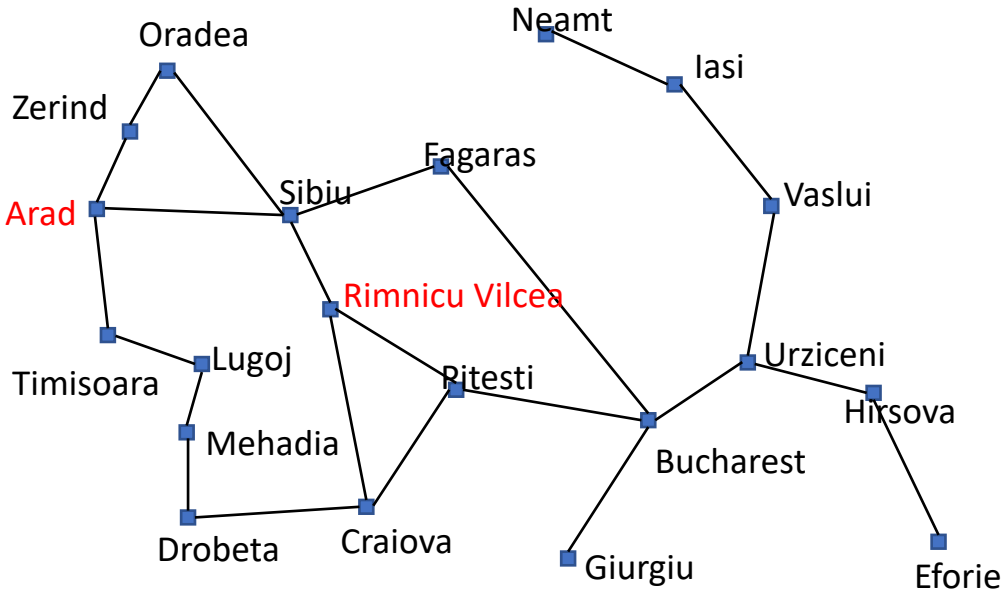
Implicit State Space Graph

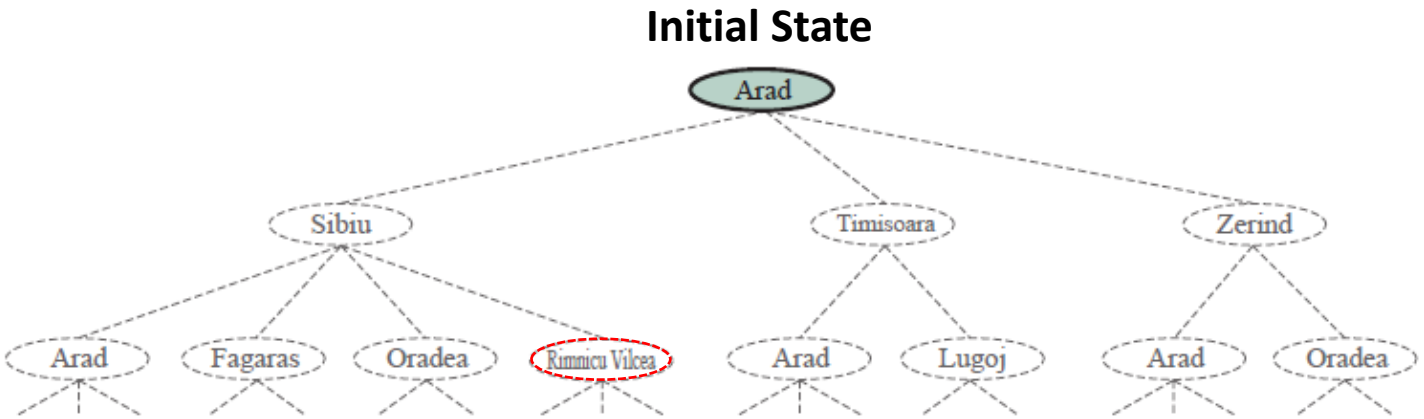


Search Tree



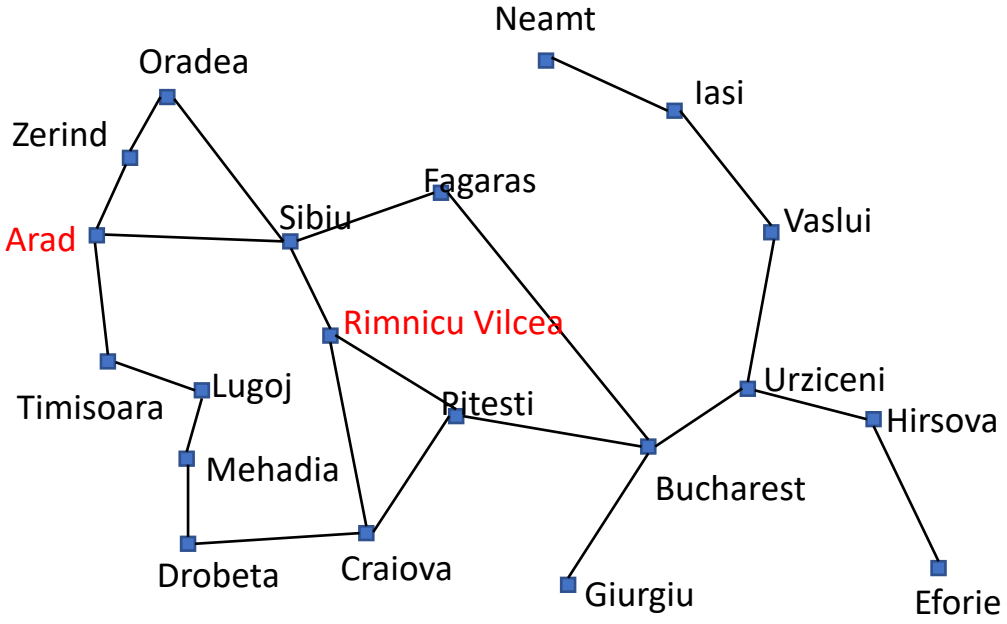
Search Tree: Arad to Rimnicu Vilcea

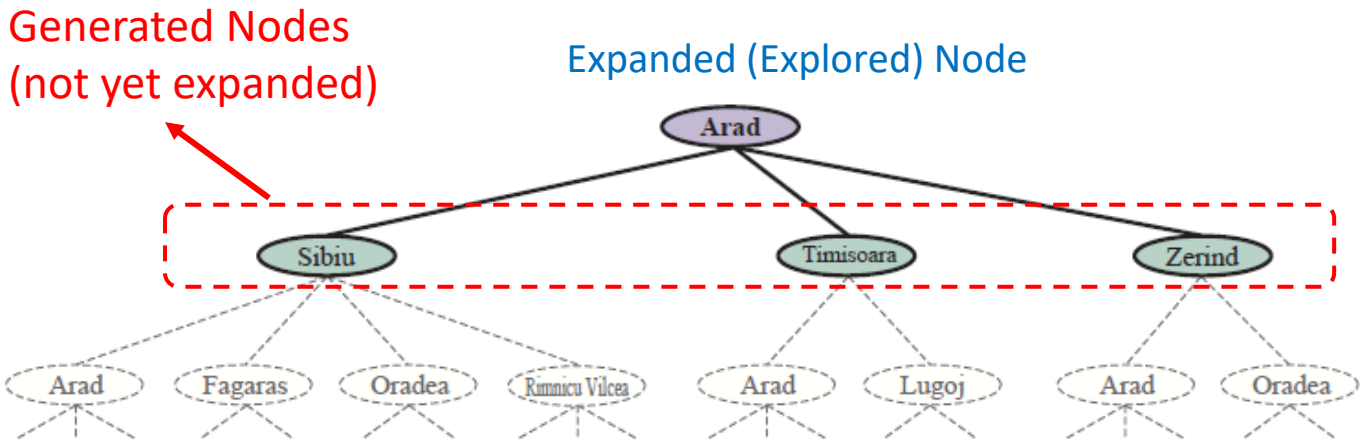




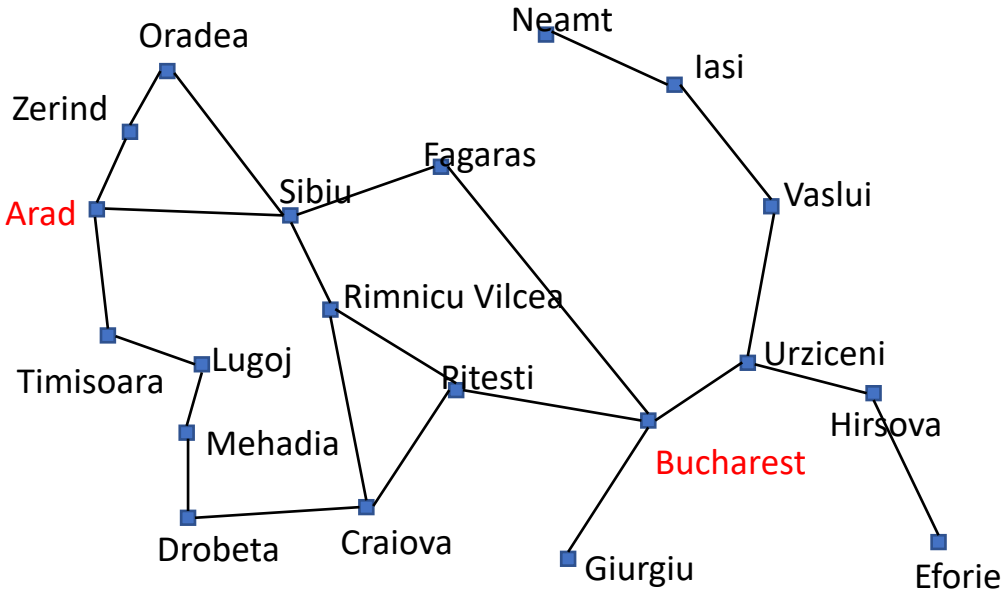
Actions at Initial State: $\{ToZerind, ToSibiu, ToTimisoara\}$

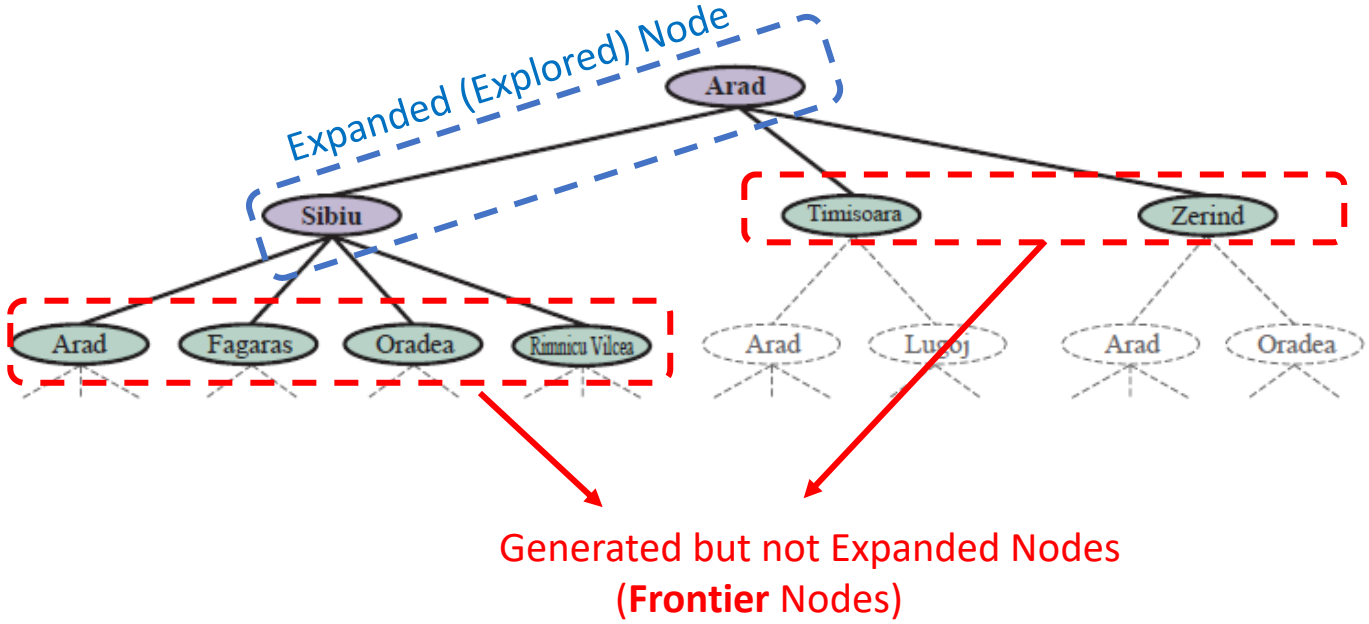
Search Tree: Arad to Rimnicu Vilcea





Search Tree: Arad to Rimnicu Vilcea



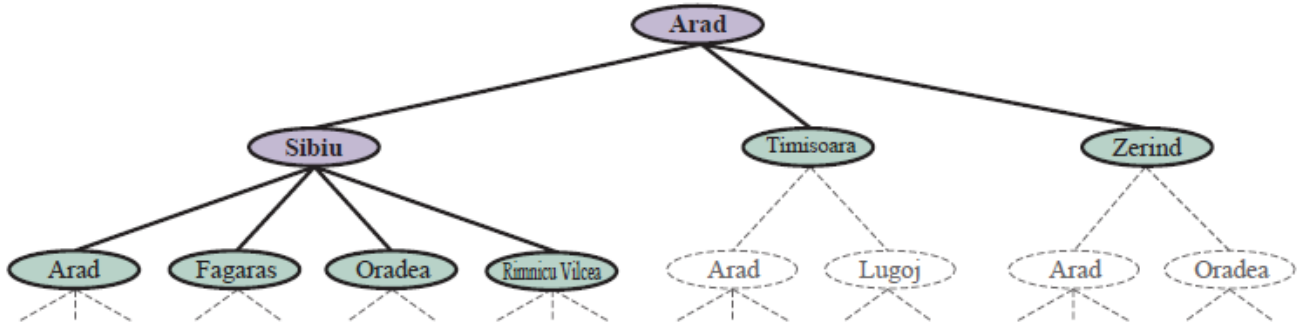


Reached Nodes:

Expanded Nodes and Frontier Nodes

Data Structures for Search Tree

- Generation of search tree

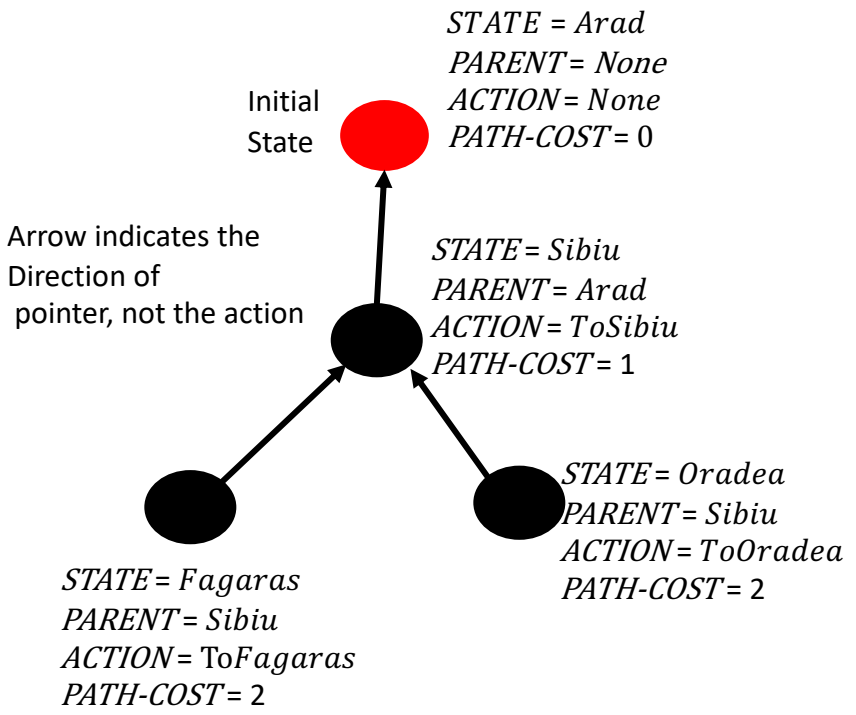


- Representation of
  - Reached nodes
  - Frontier nodes

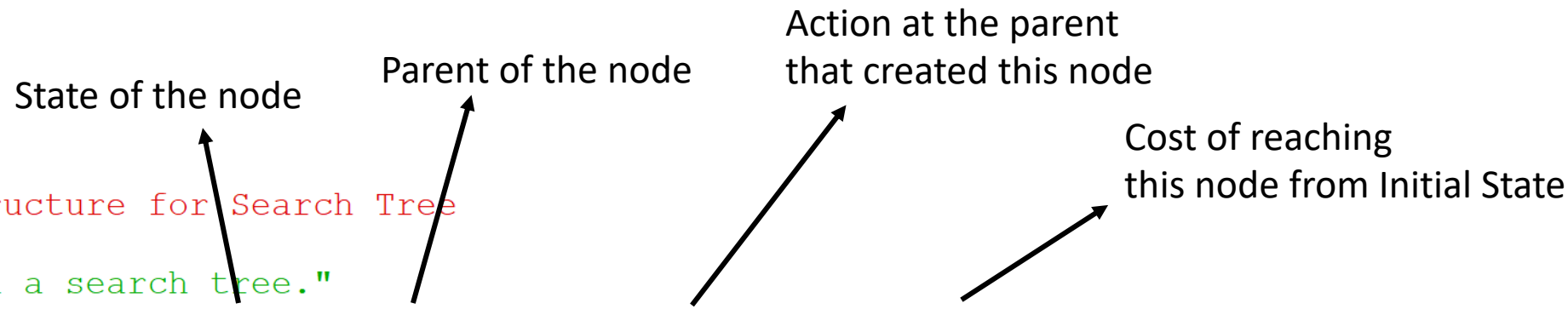
Generation of Search Tree

- 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



Node




```
#Node Data Structure for Search Tree
class Node:
    "A Node in a search tree."
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.__dict__.update(state=state, parent=parent, action=action, path_cost=path_cost)

    def __repr__(self): return '<{}>'.format(self.state)
    def __len__(self): return 0 if self.parent is None else (1 + len(self.parent))
    def __lt__(self, other): return self.path_cost < other.path_cost
```

Expanding a node

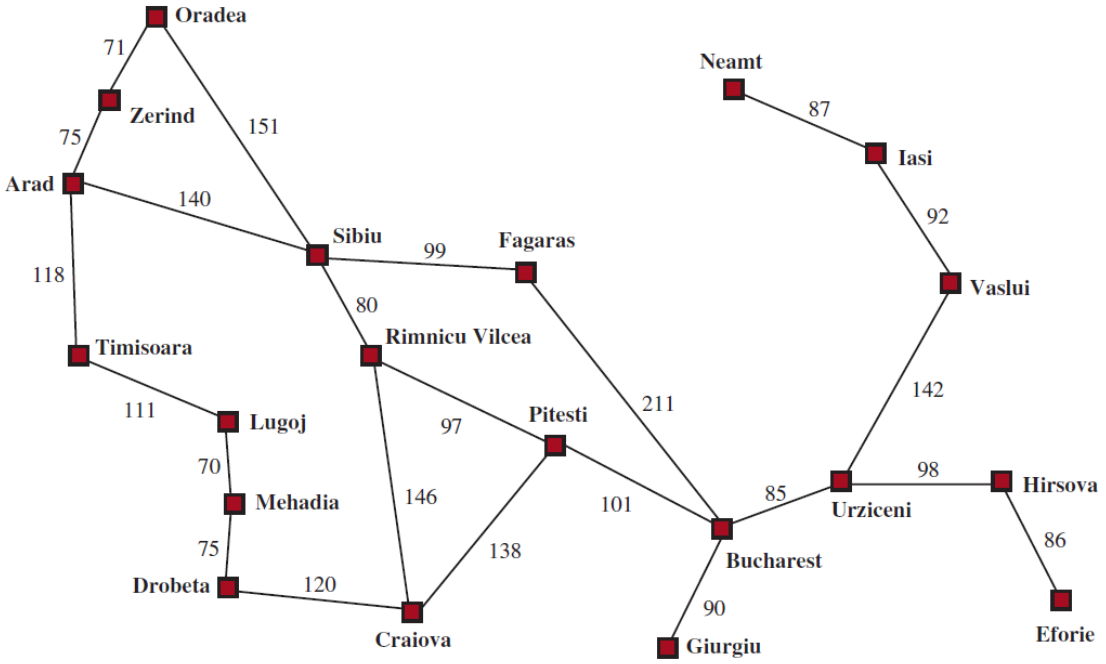
Node generator function

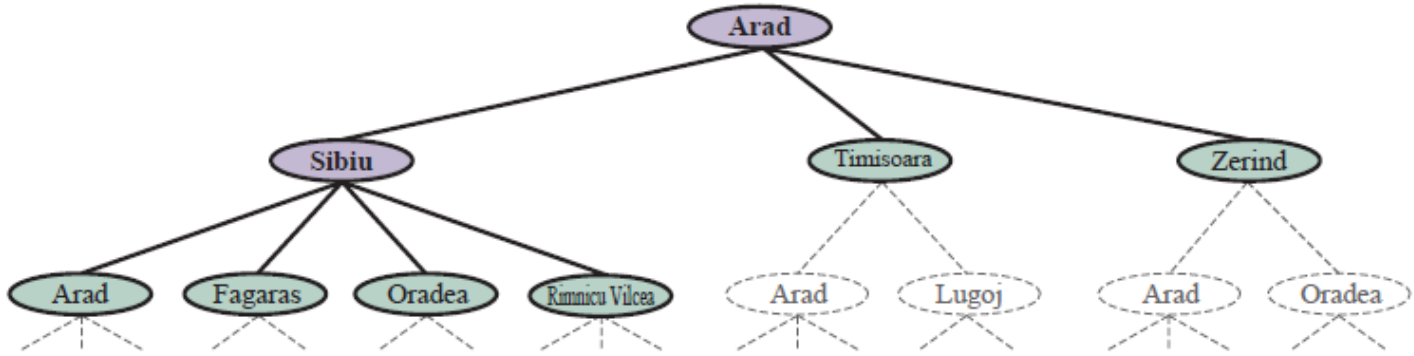
```
function EXPAND(problem, node) yields nodes
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s' ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
        yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

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

Expanding a node: Example

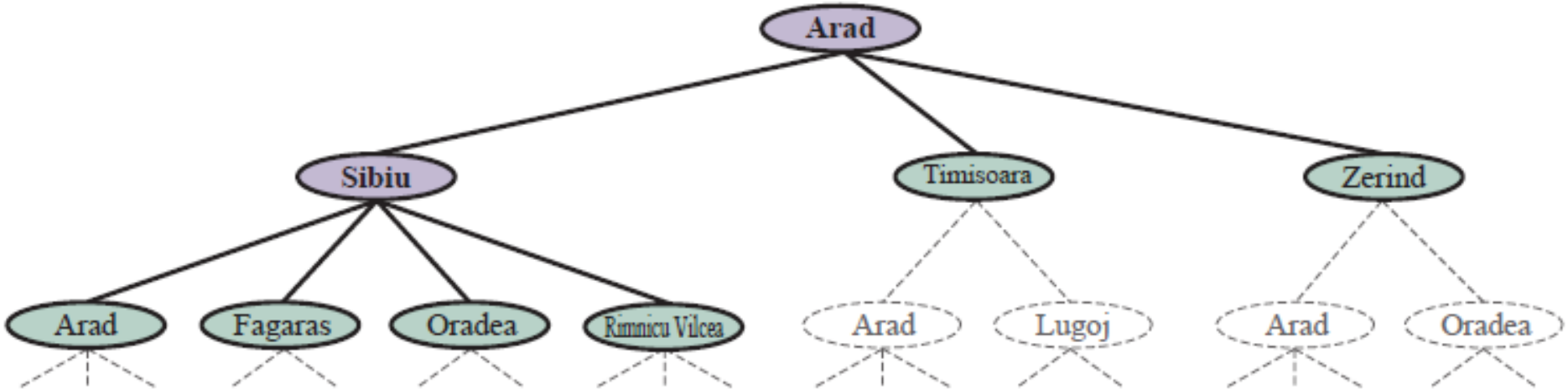
```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```





Node Vs State

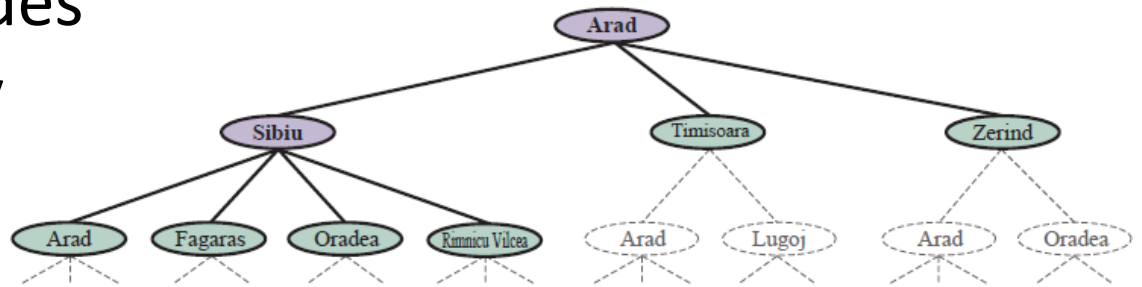
- Multiple nodes can have same state



Ex: Root node and multiple leaf nodes have same state (Arad)

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

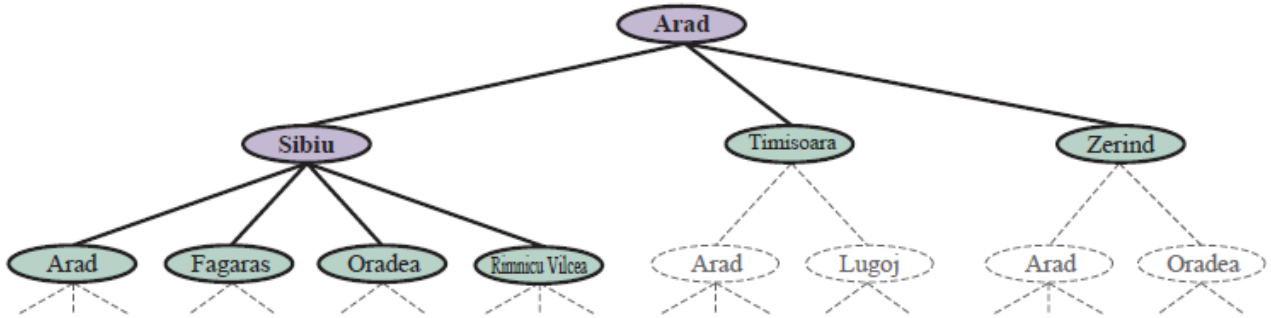
- Can be implemented using Hash Table
  - Key-value pairs
    - Key is state
    - Value is node
  - Dictionary in Python

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

Building Blocks of Search Tree

Problem:

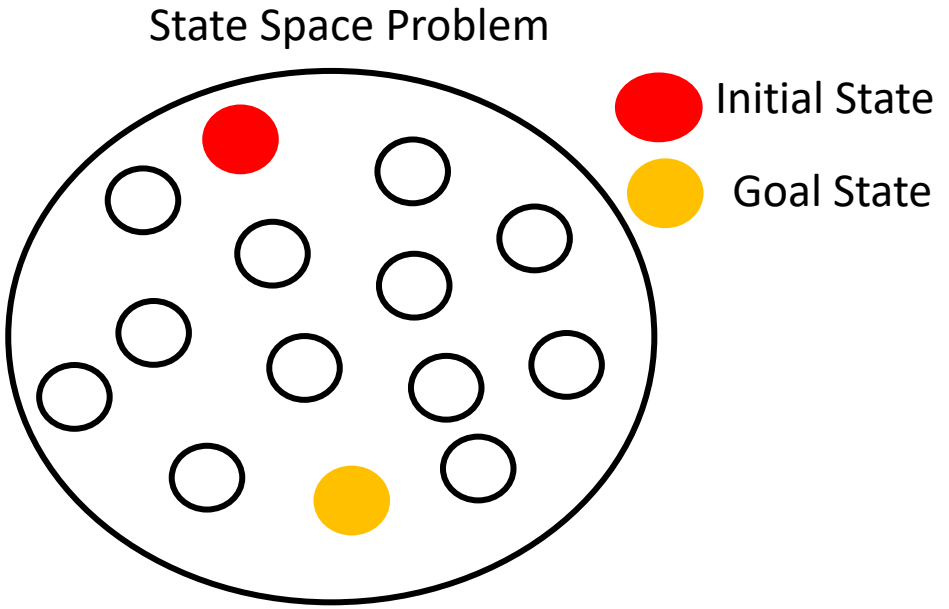
Initial State: $problem.INITIAL - STATE$

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

Actions: $problem.ACTIONS(s)$

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

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



Node:

State: $node.STATE$

Parent: $node.PARENT$

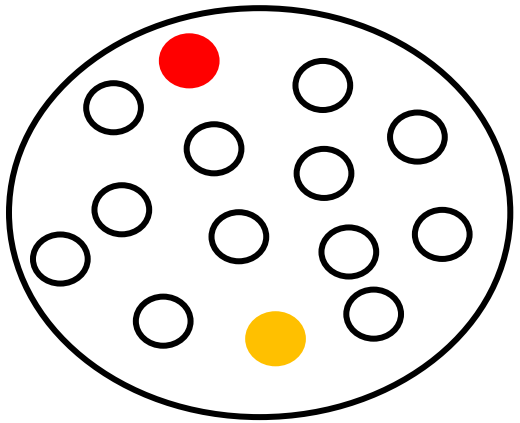
Path-Cost: $node.PATH - COST$

Action: $node.ACTION$

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

Inference (Search for the solution)

Implicit State Space Graph

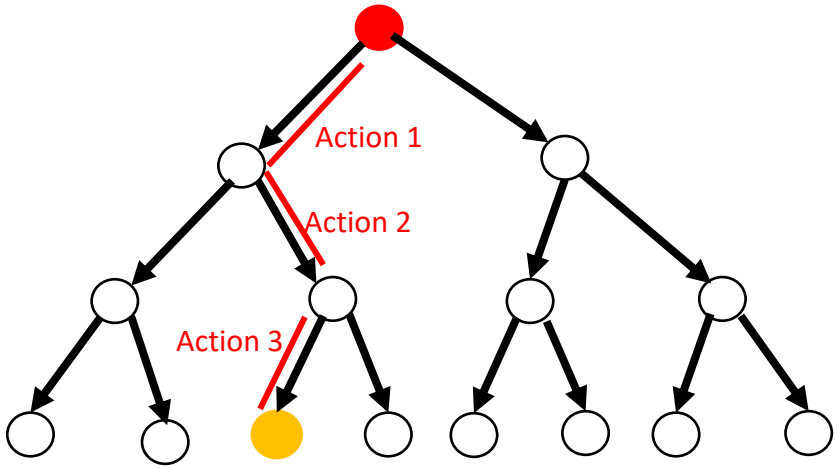


Initial State

Goal State

Inferencing

(Search for the solution)



Search Tree

Inferencing:

Finding the path from Initial State to Goal State

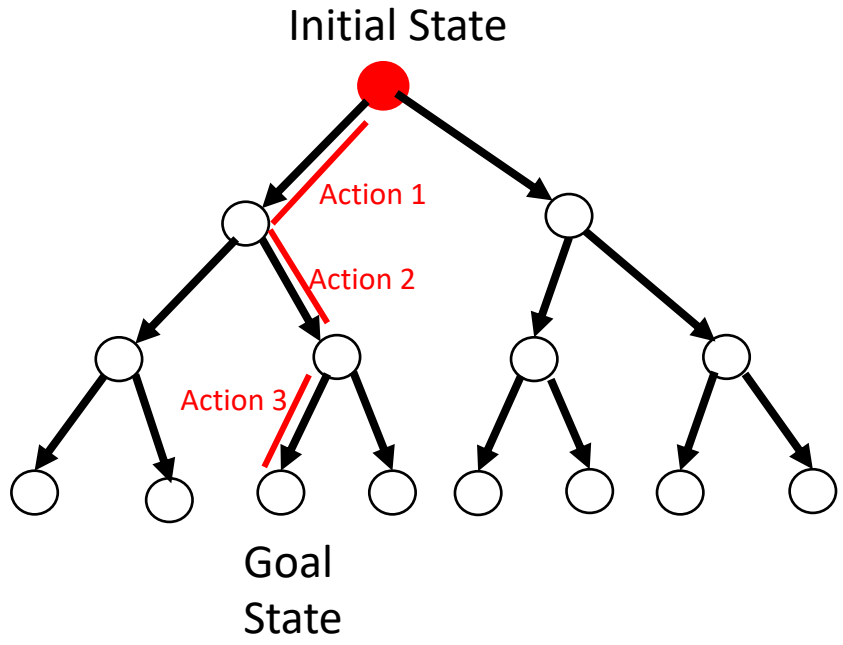
Solution: Sequence of Actions

[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

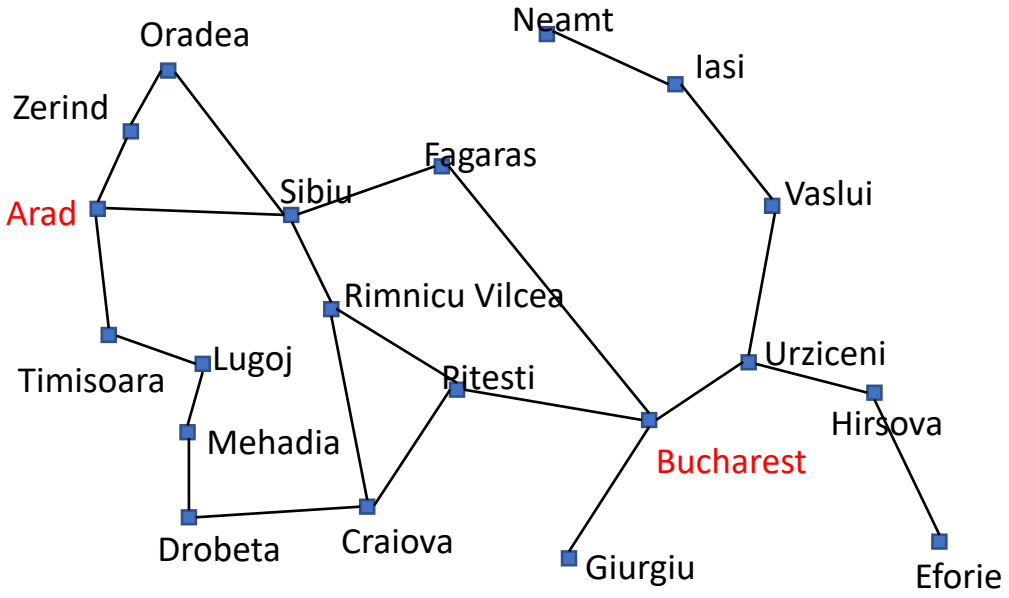
- Graph Search
  - Stores reached nodes
- Tree Search
  - Does not store reached nodes

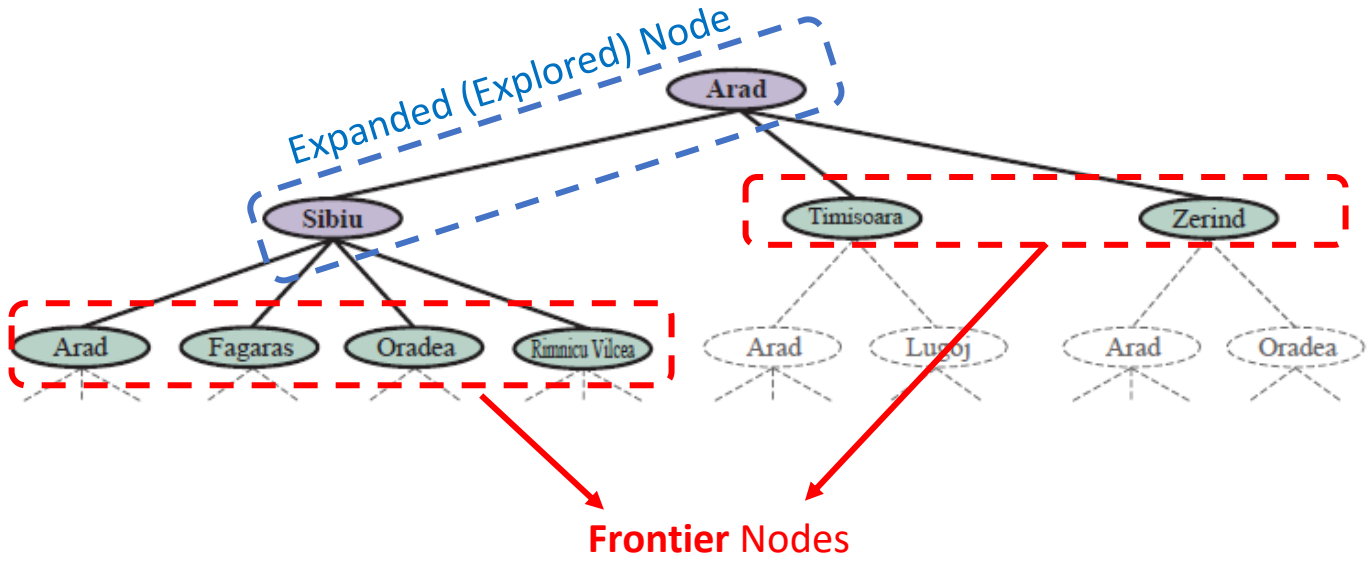
Storing reached nodes?

If model contains

Cycle or Loopy Paths
Redundant Paths

} Storing reached nodes helps in pruning search tree





Reached Nodes:

Expanded Nodes and Frontier Nodes

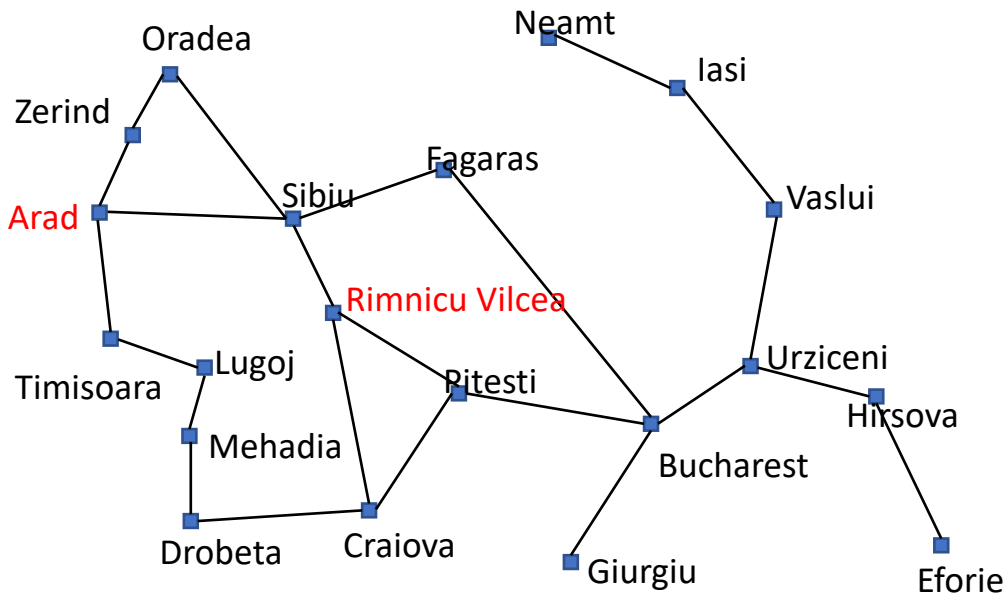
Breadth-First Search (BFS)

```
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
```

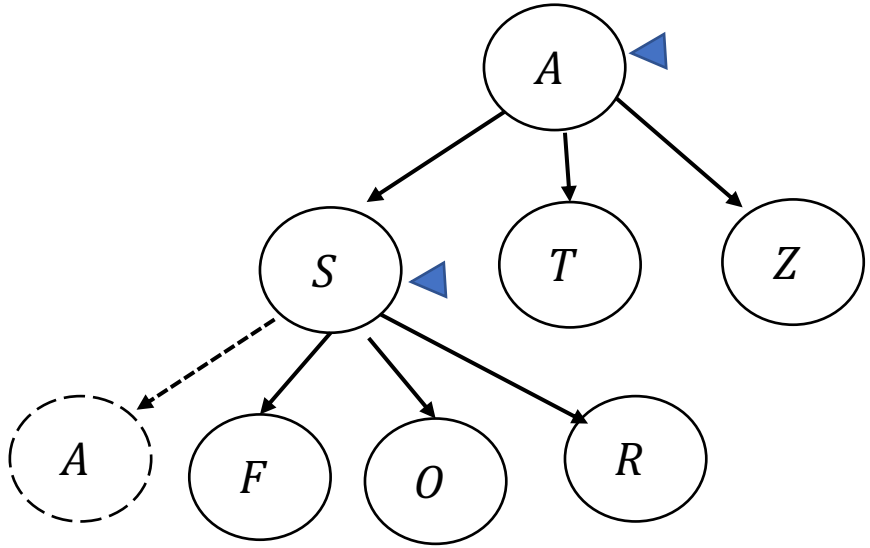
graph search or tree search?

Trace of BFS

Node	s	Is-Goal(s)	s not in Reached	Frontier	Reached
A		No		$[A]$	$\{A\}$
A	S	No	Yes	$[S]$	$\{A,S\}$
	T	No	Yes	$[S,T]$	$\{A,S,T\}$
	Z	No	Yes	$[S,T,Z]$	$\{A,S,T,Z\}$
S	A	No	No	$[T,Z]$	$\{A,S,T,Z\}$
	F	No	Yes	$[T,Z,F]$	$\{A,S,T,Z,F\}$
	O	No	Yes	$[T,Z,F,O]$	$\{A,S,T,Z,F,O\}$
	R	Yes			



Search Tree



Depth-First Search (DFS)

function DEPTH-FIRST-SEARCH($problem$):

$node \leftarrow$ NODE($problem.INITIAL - STATE$)

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

$frontier \leftarrow$a LIFO queue (stack) with *node* as element

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

$node \leftarrow POP(frontier)$

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

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

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

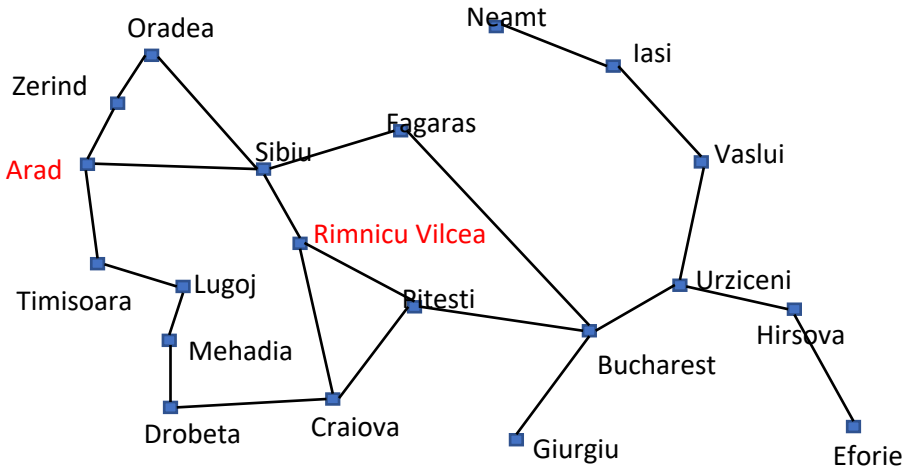
add $child$ to $frontier$

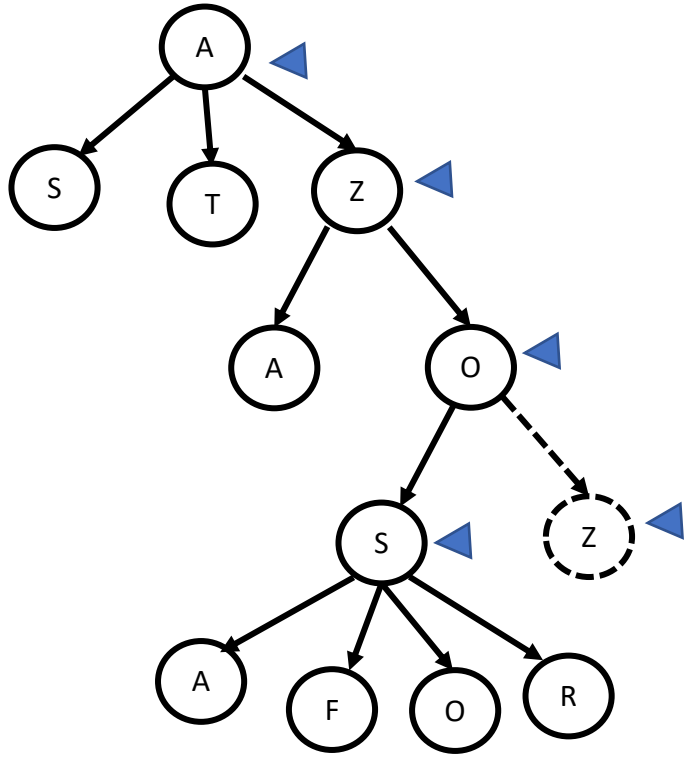
return $failure$

How to check cycles?

Trace of DFS

Node	Is-Goal(Node)	Is-Cycle(Node)	Frontier
A	No		$[A]$
A	No	No	$[S, T, Z]$
Z	No	No	$[S, T, A, O]$
O	No	No	$[S, T, A, S, Z]$
Z	No	Yes	$[S, T, A, S]$
S	No	No	$[S, T, A, A, F, O, R]$
R	Yes		

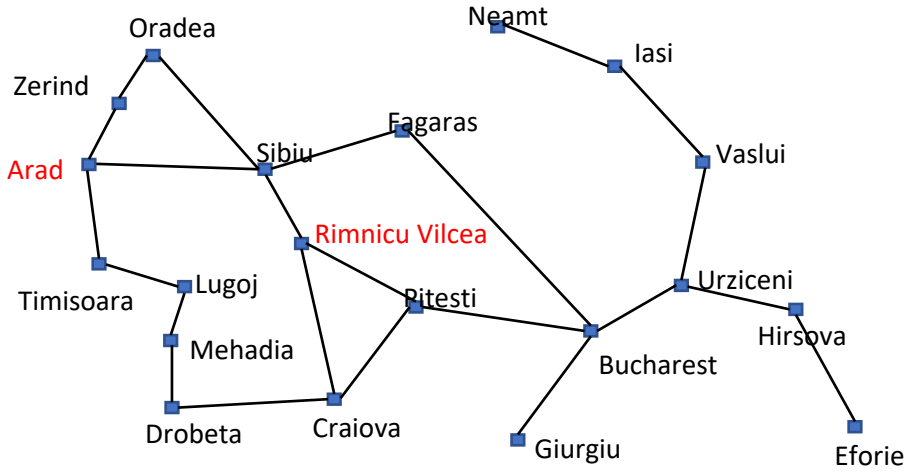




Depth-limited Search (DLS)

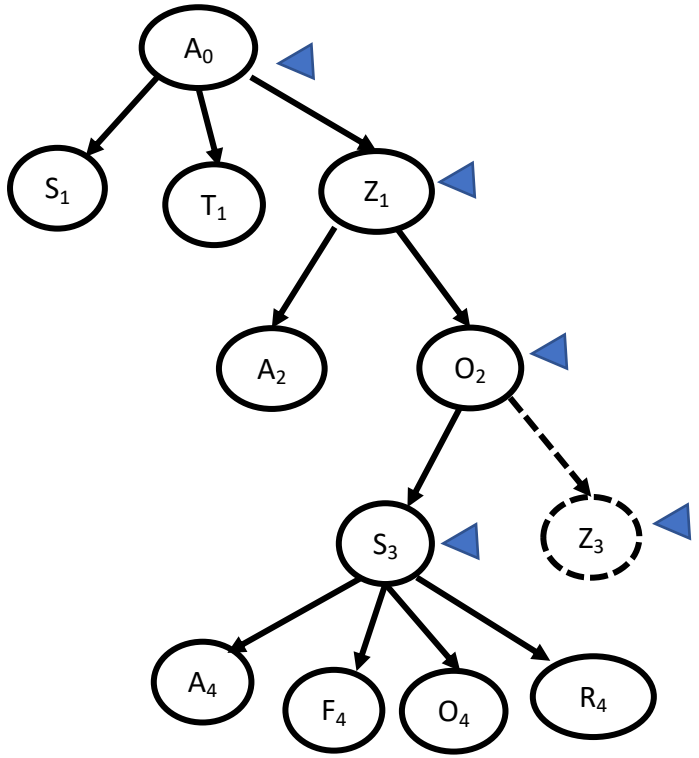
```
function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node)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

Trace of DLS



$l = 3$

Node	Is-Goal(Node)	Depth(Node)>l	Is-Cycle(Node)	Frontier	Result
A_0	No	No		$[A_0]$	Failure
A_0	No	No	No	$[S_1, T_1, Z_1]$	Failure
Z_1	No	No	No	$[S_1, T_1, A_2, O_2]$	Failure
O_2	No	No	No	$[S_1,T_1,A_2,S_3,Z_3]$	Failure
Z_3	No	No	Yes	$[S_1,T_1,A_2,S_3]$	Failure
S_3	No	No	No	$[S_1,T_1,A_2,A_4,F_4,O_4,R_4]$	Failure
R_4	Yes				



Subscript indicates depth of the node

Performance Measures

- Completeness
  - Complete: if algorithm can reach goal
  - Incomplete: if algorithm cannot reach goal
- Sound
  - If algorithm provides correct solution
- Optimality (*aka* rationality)
  - Optimal: if algorithm finds an optimal (lowest cost) path to goal
- Time Complexity
  - Time taken to find the solution
  - Measured in terms of number of nodes generated and visited while searching for the solution
- Space Complexity
  - Memory needed to find the solution
  - Measured in terms of number of nodes stored while searching for the solution

Performance of Uninformed Search Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

Iterative Deepening Search (IDS)

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

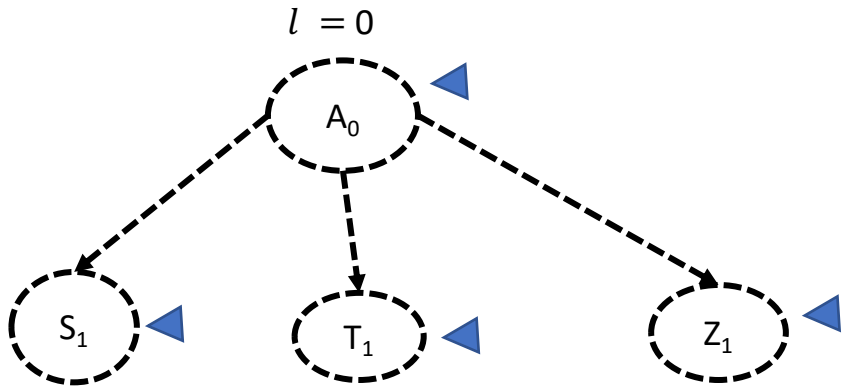
Trace of IDS

$l = 0$

Node	Is-Goal(Node)	Depth(Node)>l	Is-Cycle(Node)	Frontier	Result
A_0	No			$[A_0]$	Failure
A_0	No	No	No	$[S_1, T_1, Z_1]$	Failure
Z_1	No	Yes		$[S_1, T_1]$	Cutoff
T_1	No	Yes		$[S_1]$	Cutoff
S_1	No	Yes		[ ]	Cutoff

Subscript indicates depth of the node

Frontier is empty
Returns Result (Cutoff)

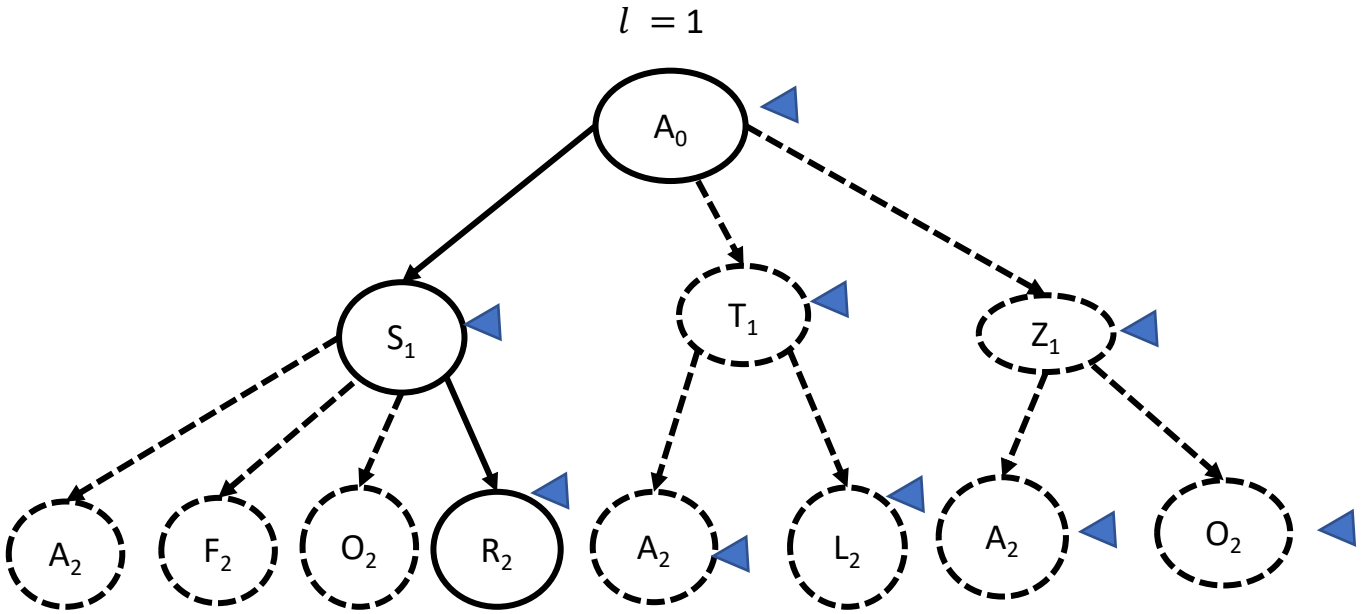


Trace of IDS

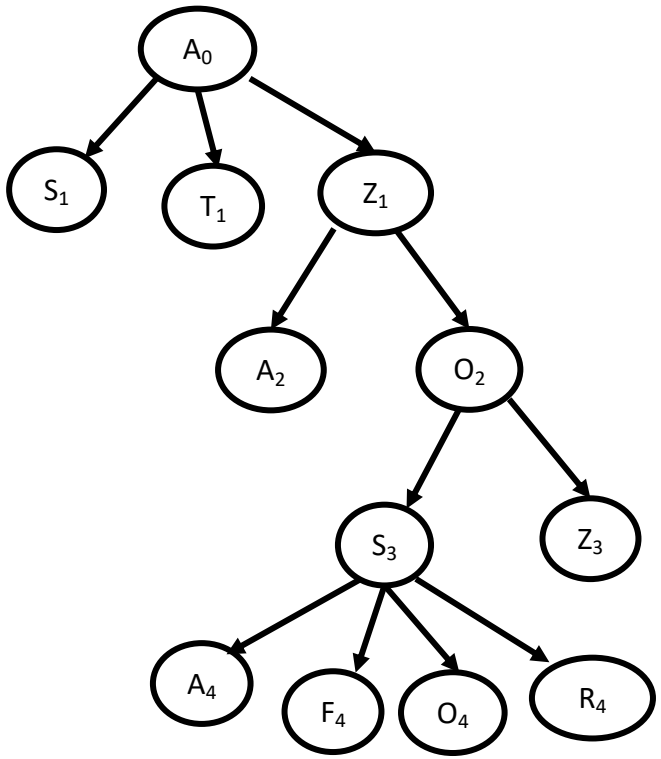
$l = 1$

Node	Is-Goal(Node)	Depth(Node)>l	Is-Cycle(Node)	Frontier	Result
A_0	No			$[A_0]$	Failure
A_0	No	No	No	$[S_1, T_1, Z_1]$	Failure
Z_1	No	No	No	$[S_1, T_1, A_2, O_2]$	Failure
O_2	No	Yes		$[S_1, T_1, A_2]$	Cutoff
A_2	No	Yes		$[S_1, T_1]$	Cutoff
T_1	No	No	No	$[S_1, A_2, L_2]$	Cutoff
L_2	No	Yes		$[S_1, A_2]$	Cutoff
A_2	No	Yes		$[S_1]$	Cutoff
S_1	No	No	No	$[A_2, F_2, O_2, R_2]$	Cutoff
R_2	Yes				

Search Tree of IDS

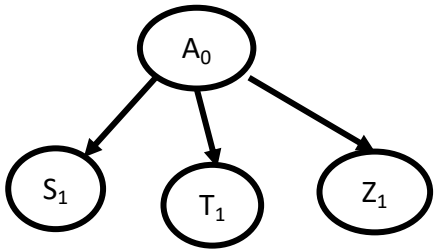


DLS

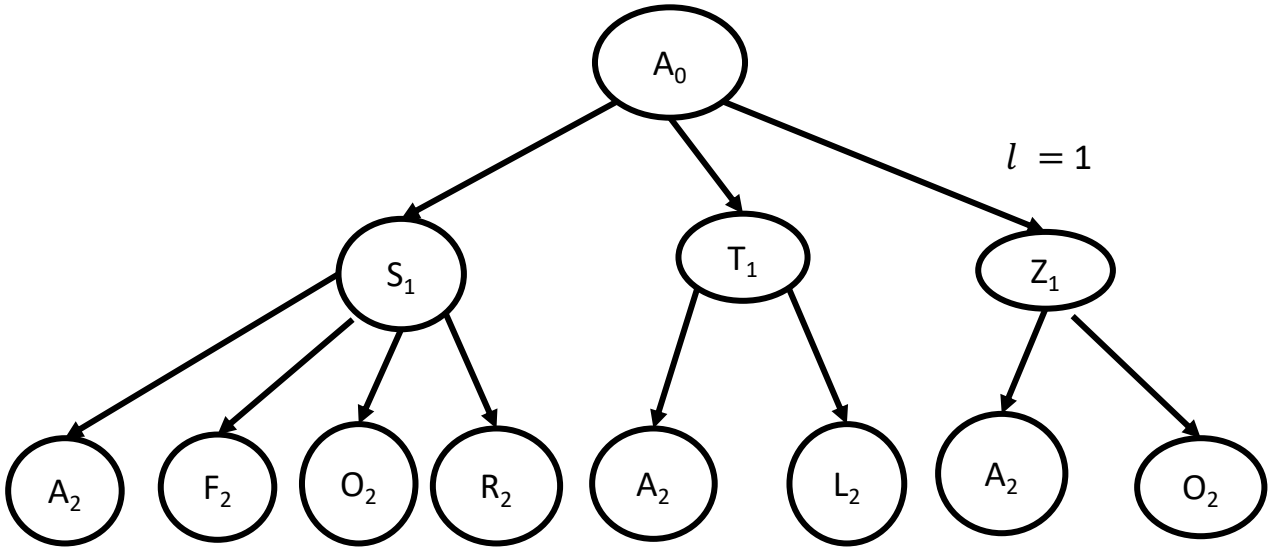

A_0
S_1
T_1
Z_1
A_2
O_2
S_3
Z_3
A_4
F_4
O_4
R_4

IDS

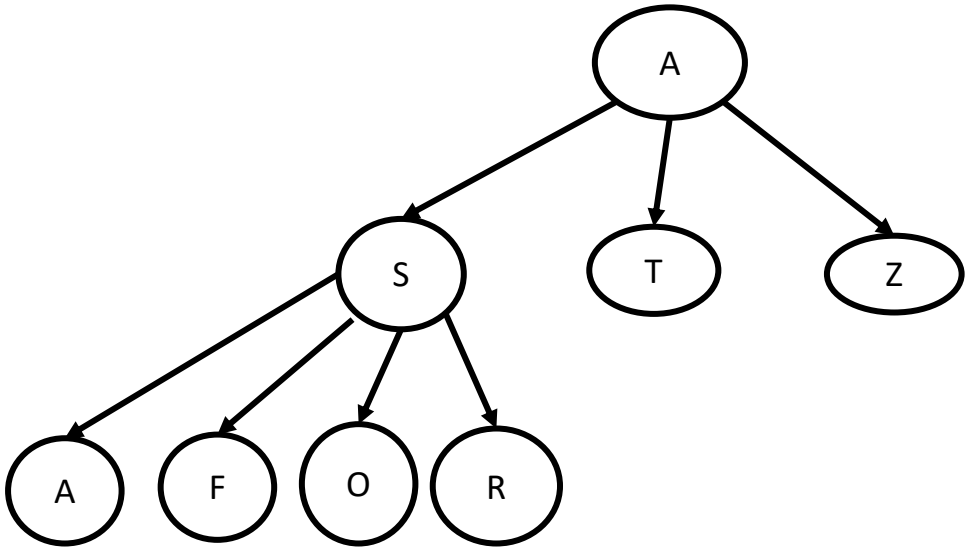
$l = 0$


A_0
S_1
T_1
Z_1

$l = 1$

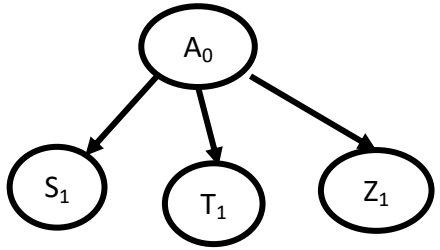

A_0
S_1
T_1
Z_1
A_2
F_2
O_2
R_2
A_2
L_2
A_2
O_2

BFS

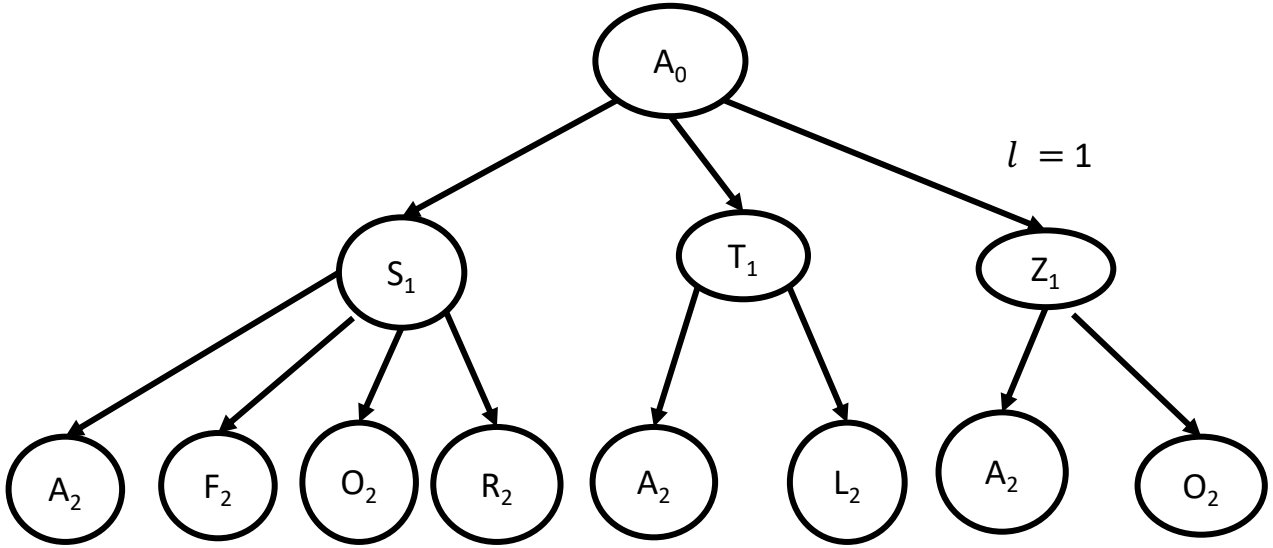


IDS

$l = 0$



$l = 1$



Overhead in IDS

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

Depth Limit (l)	Number of Generated Nodes
0	b
1	$b + b^2$
2	$b + b^2 + b^3$
…	…
$d-1$	$b + b^2 + b^3 + \cdots + b^d$

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

Uniform-Cost Search

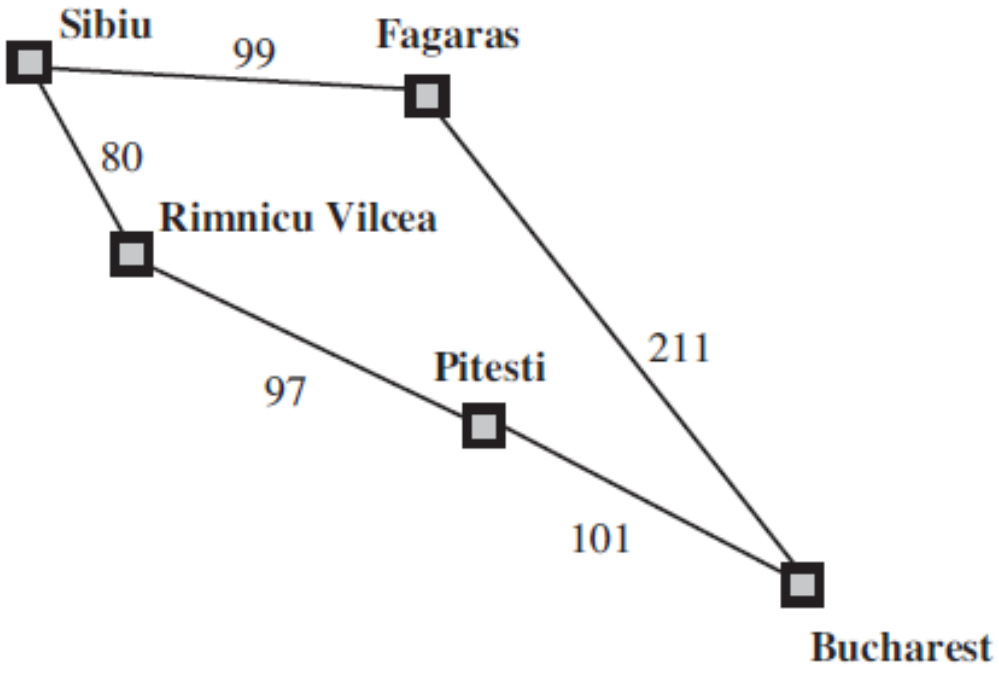
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

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*

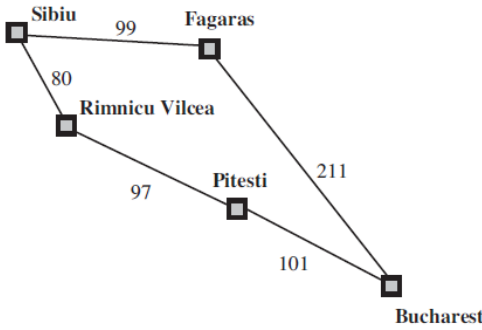
$reached = \{state: node\}$

If a node with same state is already in $reached$ with higher path-cost than $child$, replace that node with the $child$ node

Uniform-Cost Search: Romania Map



Trace of UCS



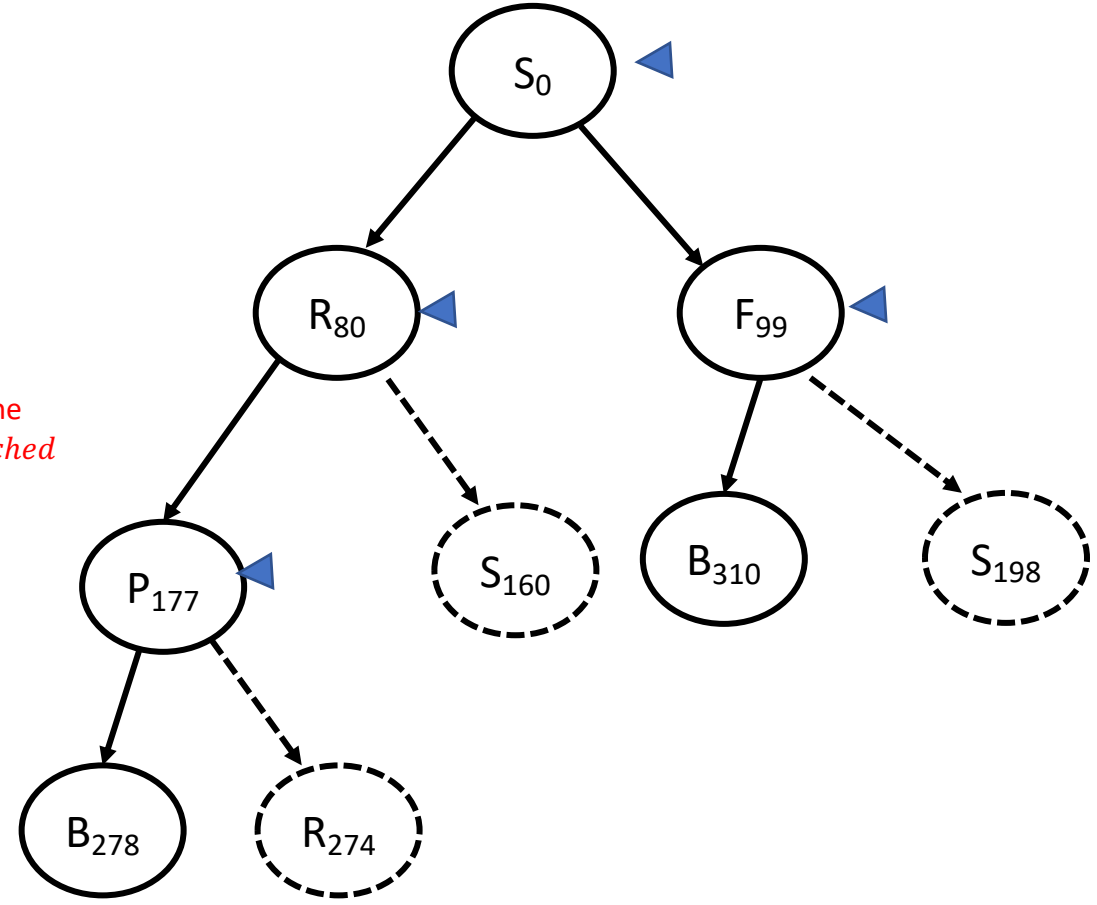
Priority Queue
(ordered by path-cost)

Lookup Table (Dictionary)
Key: State
Value :Node

Node	Is-Goal(Node)	s	Frontier	Reached	
S_0			$[S_0]$	$\{S_0\}$	
S_0	No	R_{80}	$[R_{80}]$	$\{S_0,R_{80}\}$	
S_0	No	F_{99}	$[R_{80}, F_{99}]$	$\{S_0,R_{80},F_{99}\}$	
R_{80}	No	P_{177}	$[F_{99}, P_{177}]$	$\{S_0,R_{80},F_{99},P_{177}\}$	
R_{80}	No	S_{160}	$[F_{99}, P_{177}]$	$\{S_0,R_{80},F_{99},P_{177}\}$	s is not added to the $frontier$ and $reached$
F_{99}	No	B_{310}	$[P_{177}, B_{310}]$	$\{S_0,R_{80},F_{99},P_{177}, B_{310}\}$	
F_{99}	No	S_{198}	$[P_{177}, B_{310}]$	$\{S_0,R_{80},F_{99},P_{177}, B_{310}\}$	
P_{177}	No	B_{278}	$[B_{310}, B_{278}]$	$\{S_0,R_{80},F_{99},P_{177}, B_{278}\}$	B_{278} replaced B_{310}
P_{177}	No	R_{274}	$[B_{310}, B_{278}]$	$\{S_0,R_{80},F_{99},P_{177}, B_{278}\}$	
B_{278}	Yes				

Subscript indicates path-cost

Search Tree



Performance Measures

- Completeness
  - Complete: if algorithm can reach goal
  - Incomplete: if algorithm cannot reach goal
- Sound
  - If algorithm provides correct solution
- Optimality (*aka* rationality)
  - Optimal: if algorithm finds an optimal (lowest cost) path to goal
- Time Complexity
  - Time taken to find the solution
  - Measured in terms of number of nodes generated and visited while searching for the solution
- Space Complexity
  - Memory needed to find the solution
  - Measured in terms of number of nodes stored while searching for the solution

Performance of Uninformed Search Algorithms

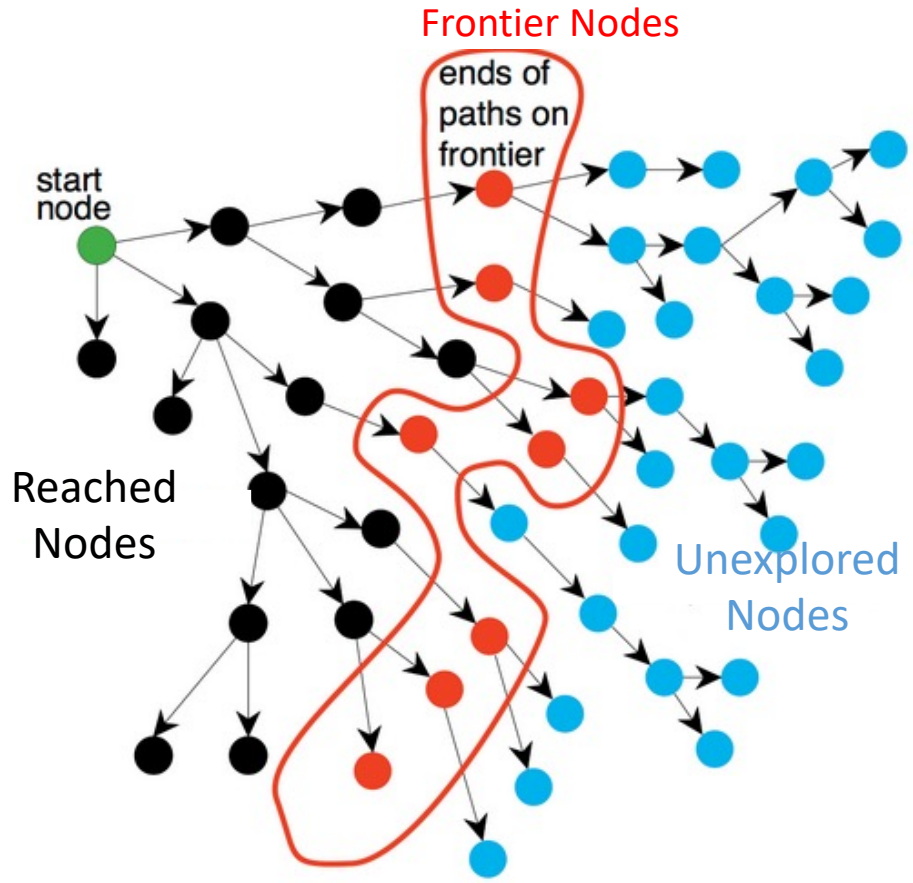
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

Summary of Uninformed Search

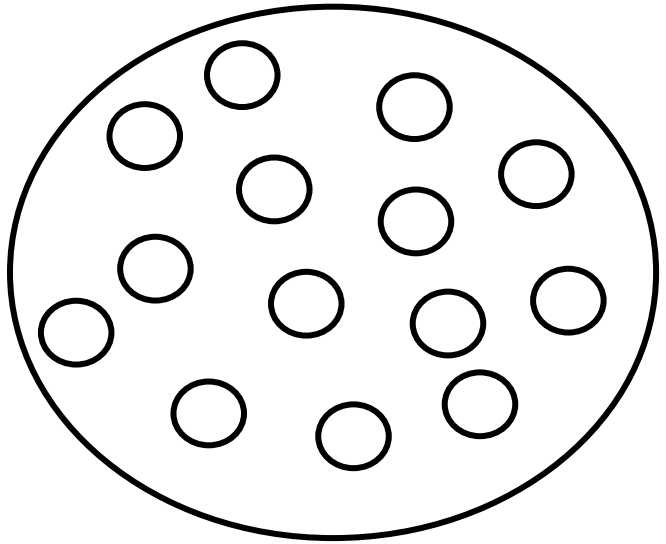
- Classification based on traversal methods
  - Breadth-First Search: Frontier is a Queue (FIFO)
  - Depth-First Search: Frontier is a stack (LIFO)
  - Uniform-Cost Search: Frontier is a priority queue

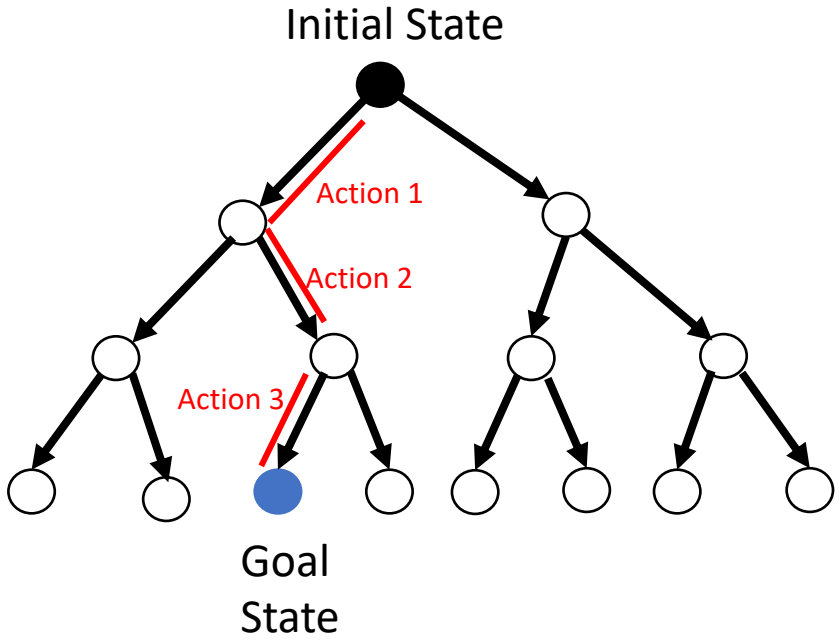
- Tree-Search Vs Graph-Search
  - Tree-Search
    - Doesn't store reached nodes
    - Leads to cycles and redundant paths
    - Cycles can be avoided with Cycle-Check
  - Graph-Search
    - Maintains a set of reached states
    - Avoids cycles and redundant paths



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

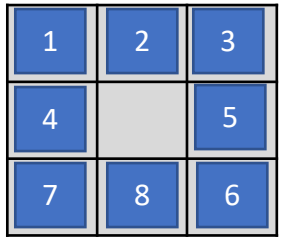
Need well-defined goal
Interaction is offline
No limits on resources (memory and time)

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

8-Puzzle

Sliding Puzzle: 8-puzzle

1. Initial State



2. Actions:

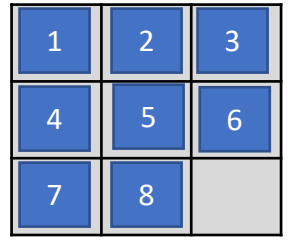
Movement of blank space

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

3. Transition Model:

Given a state and action return the resultant state

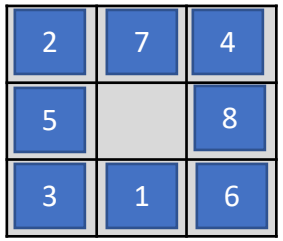
4. Goal Test



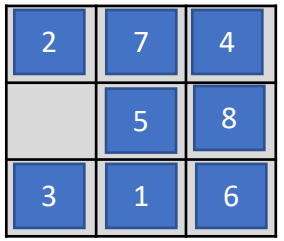
Check for this state

Eg:

State: s_1



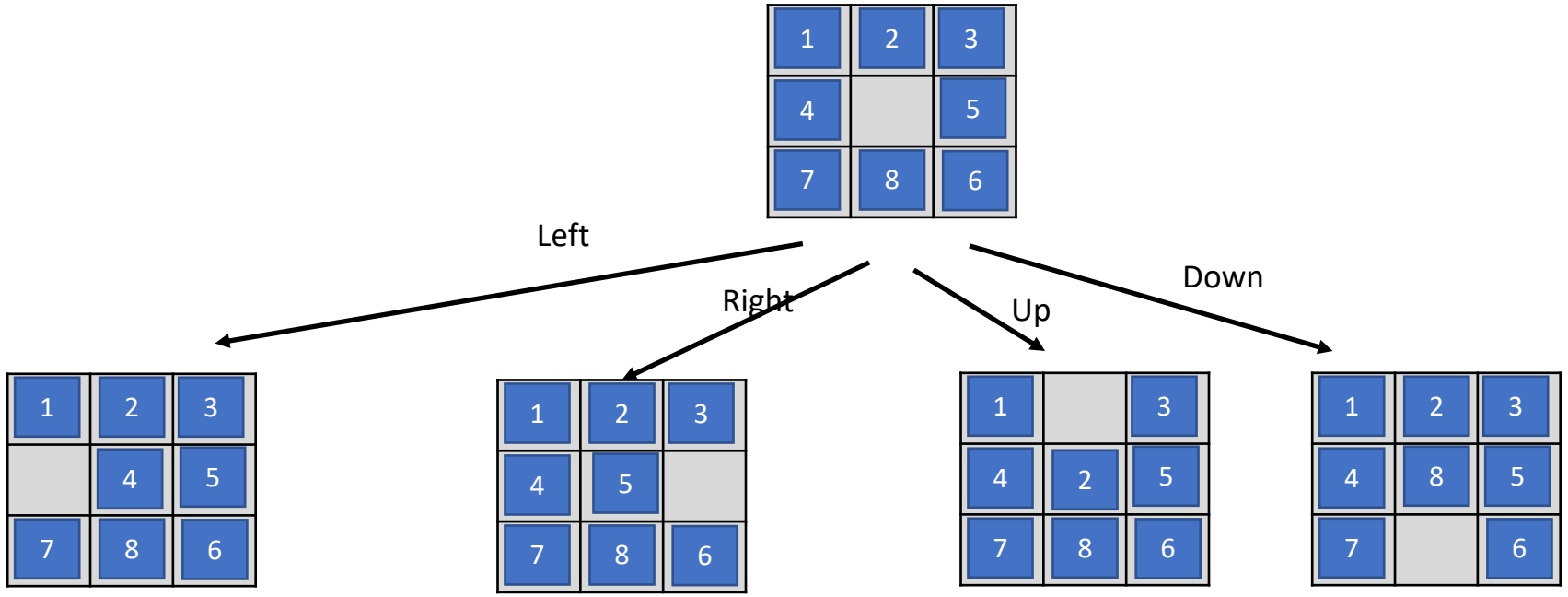
State: s_2 = Result(s_1,L)



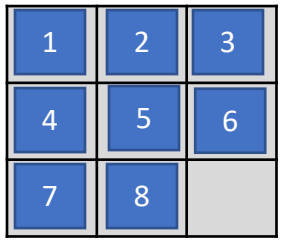
5. Path Cost

Cost of reaching a state from Initial State

8-Puzzle

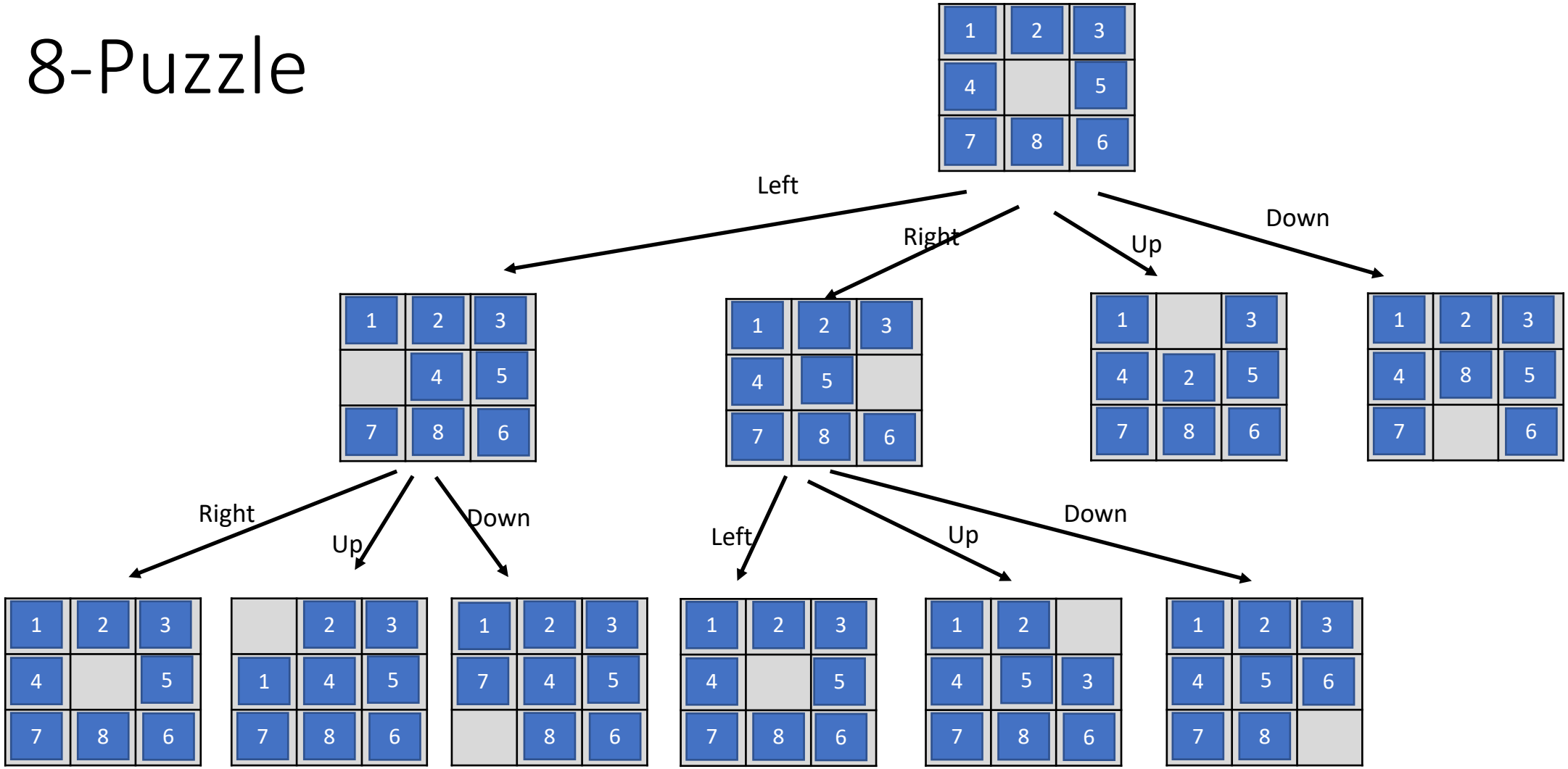


Uninformed Search Algorithms:
Select nodes from frontier based on order or path-cost



Goal State

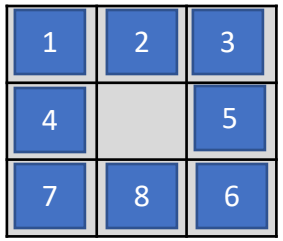
8-Puzzle



Uninformed Search

8-Puzzle

- Average branching factor: $\frac{4+4*2+4*3}{8} = 3$

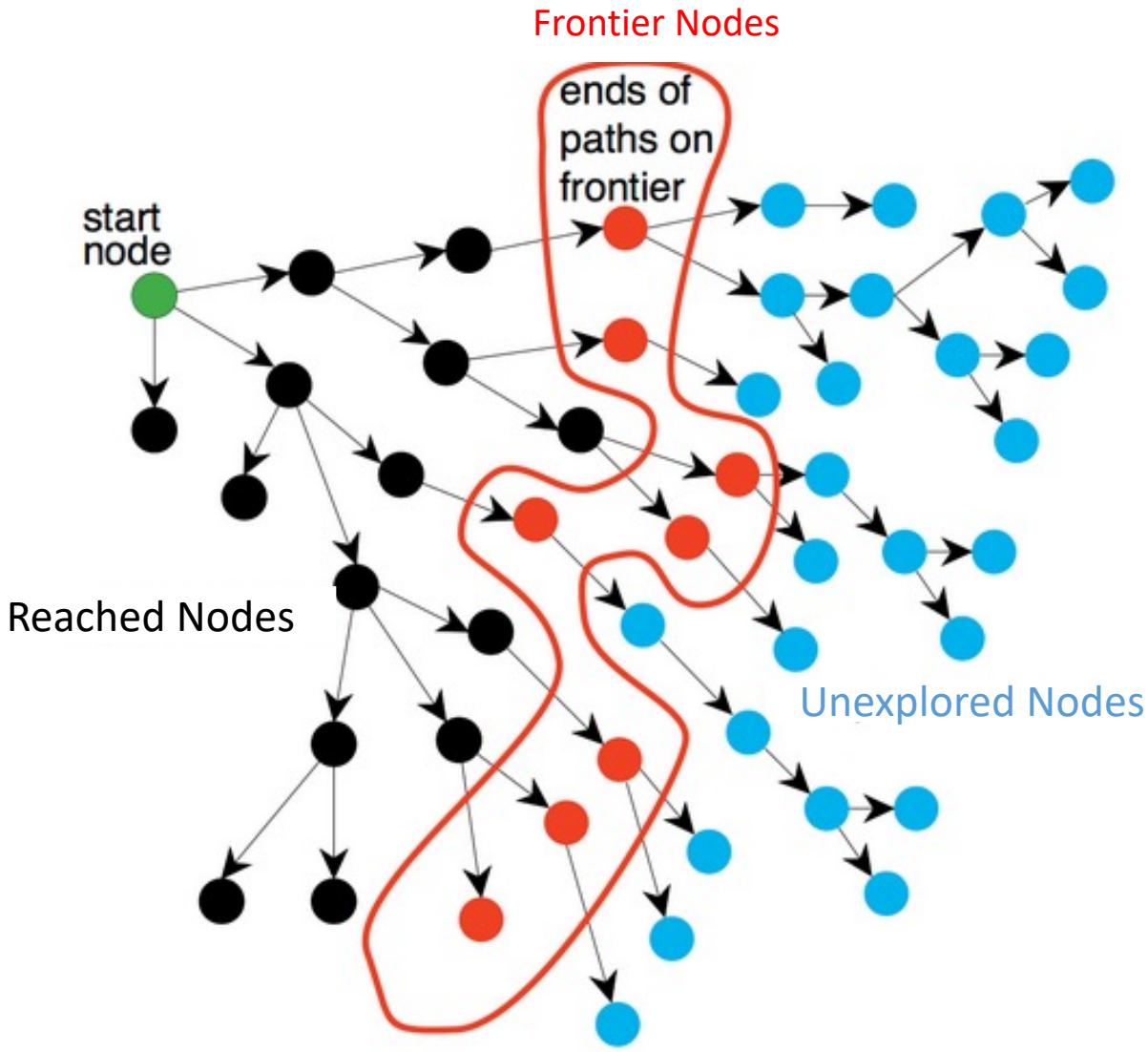


- Average solution depth: 22 steps (from empirical studies)
- Uninformed search algorithms generate 3^{22} (b^d) nodes
- How to reduce this number?

Informed Search

- Uninformed (Uniform Cost) Search:
  - Uses only the information provided in problem formulation
  - Only track path-cost

- Informed Search:
  - Uses additional domain-specific information called “heuristic”



What is heuristic?

Heuristic

From Wikipedia, the free encyclopedia

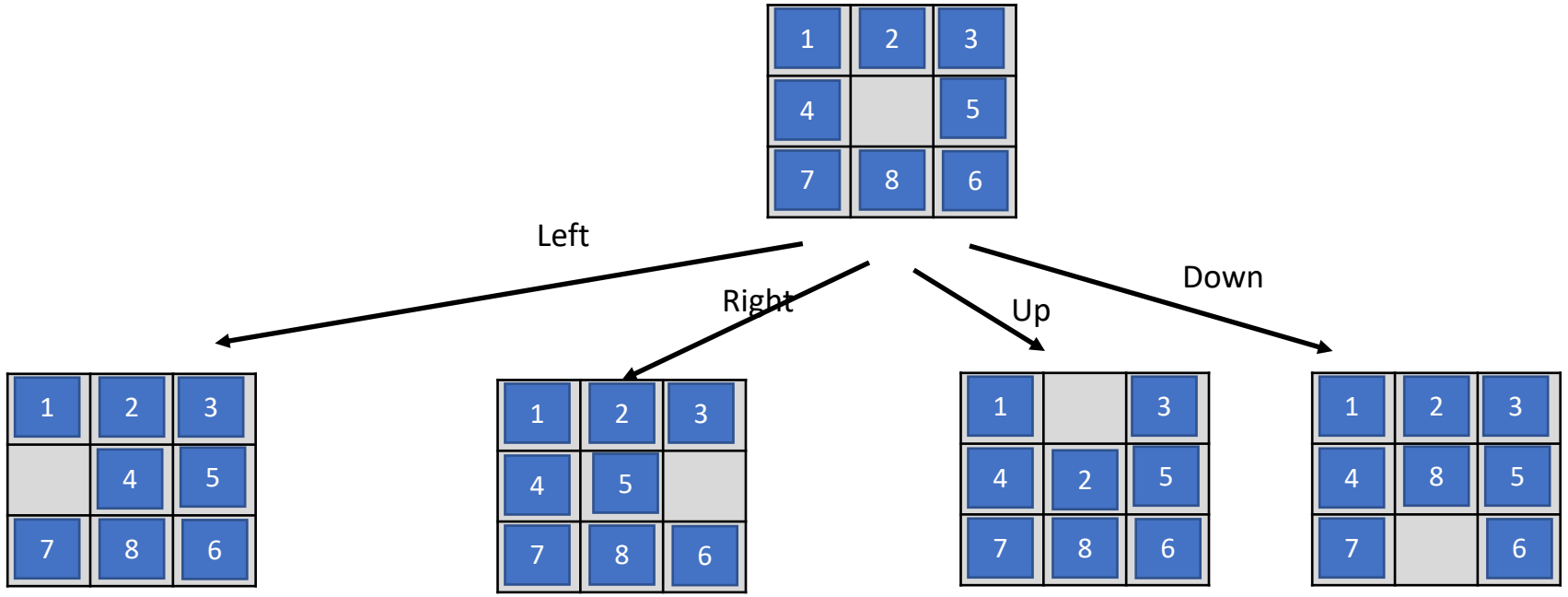
For other uses, see Heuristic (disambiguation).

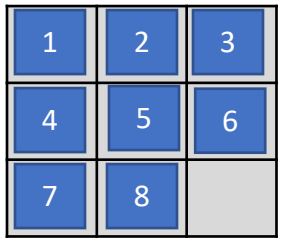
A **heuristic** or **heuristic technique** (/hjʊəˈrɪstɪk/; Ancient Greek: εὑρίσκω, *heurískō*, 'I find, discover'), is any approach to problem solving or self-discovery that employs a practical method that is not guaranteed to be optimal, perfect, or rational, but is nevertheless sufficient for reaching an immediate, short-term goal or approximation. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution. Heuristics can be mental shortcuts that ease the cognitive load of making a decision.[1][2]

‘Rule-of-Thumb’ or an ‘Educated-Guess’

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

8-Puzzle





Goal State

Which node do we prefer for expansion?

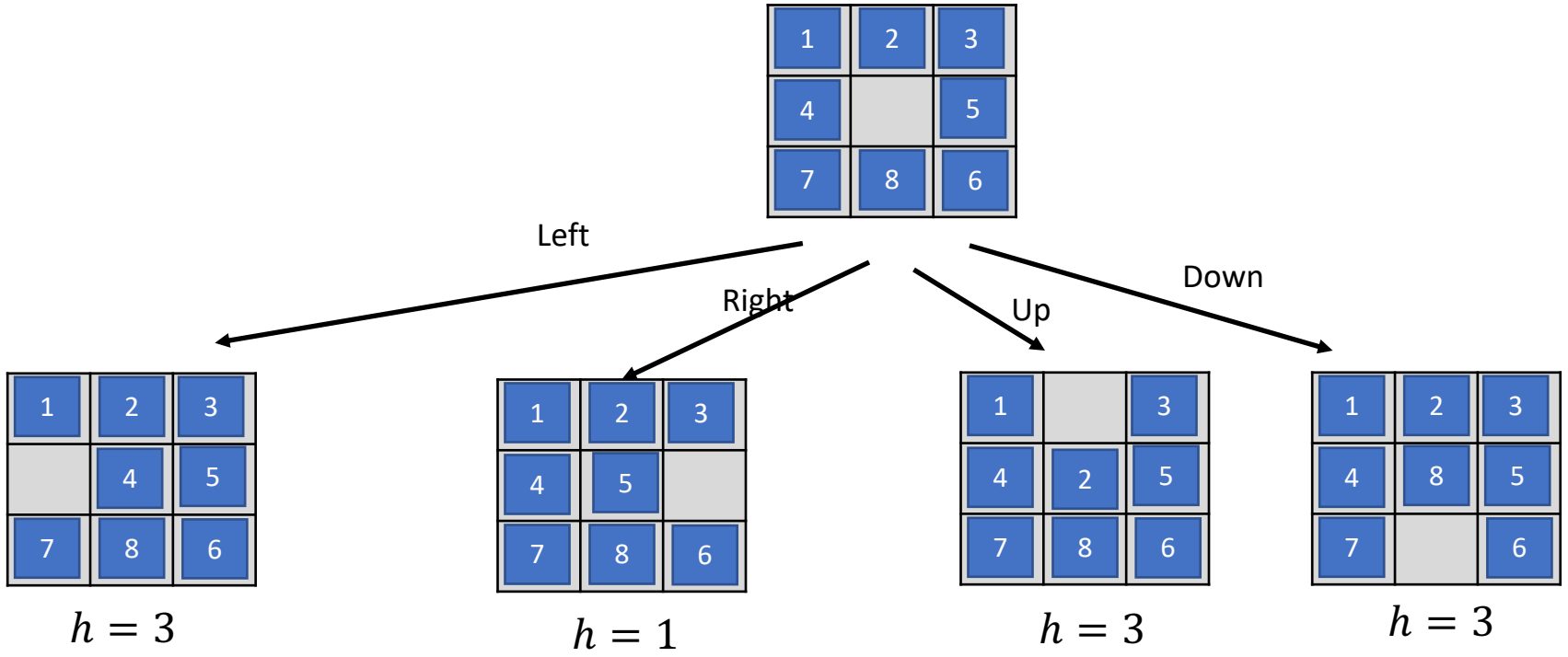
or

Which node in frontier is close to the goal state?

or

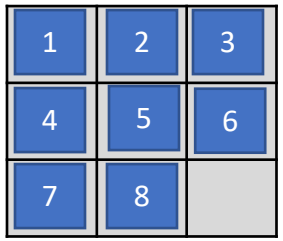
Which node have least path-cost towards to the goal state?

8-Puzzle



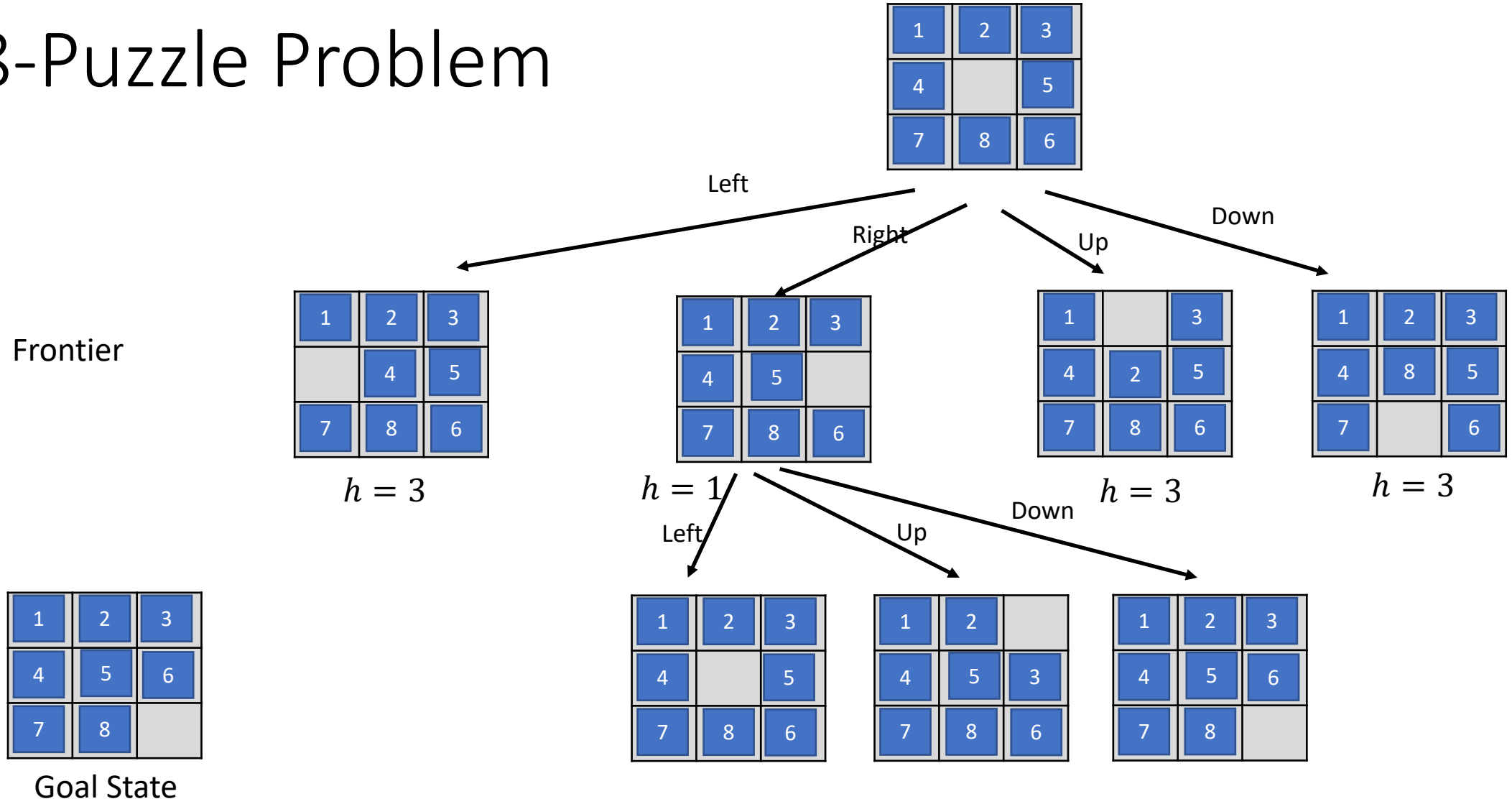
Estimated path-cost to Goal State:

$h(n)$ = Number of misplaced tiles

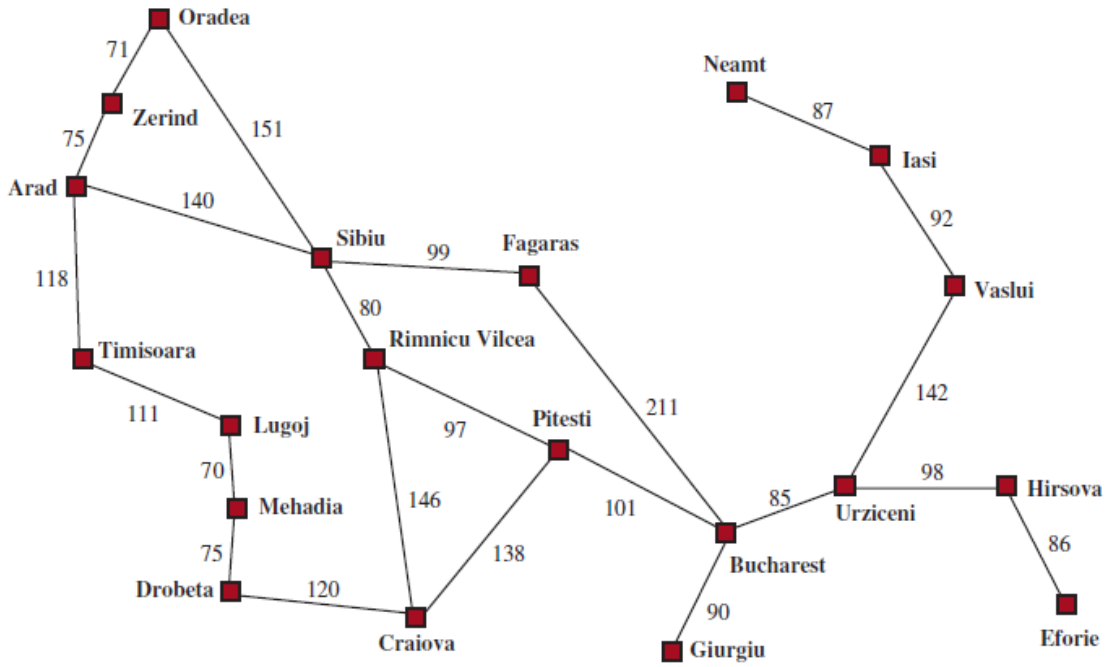


Goal State

8-Puzzle Problem



Romania Map



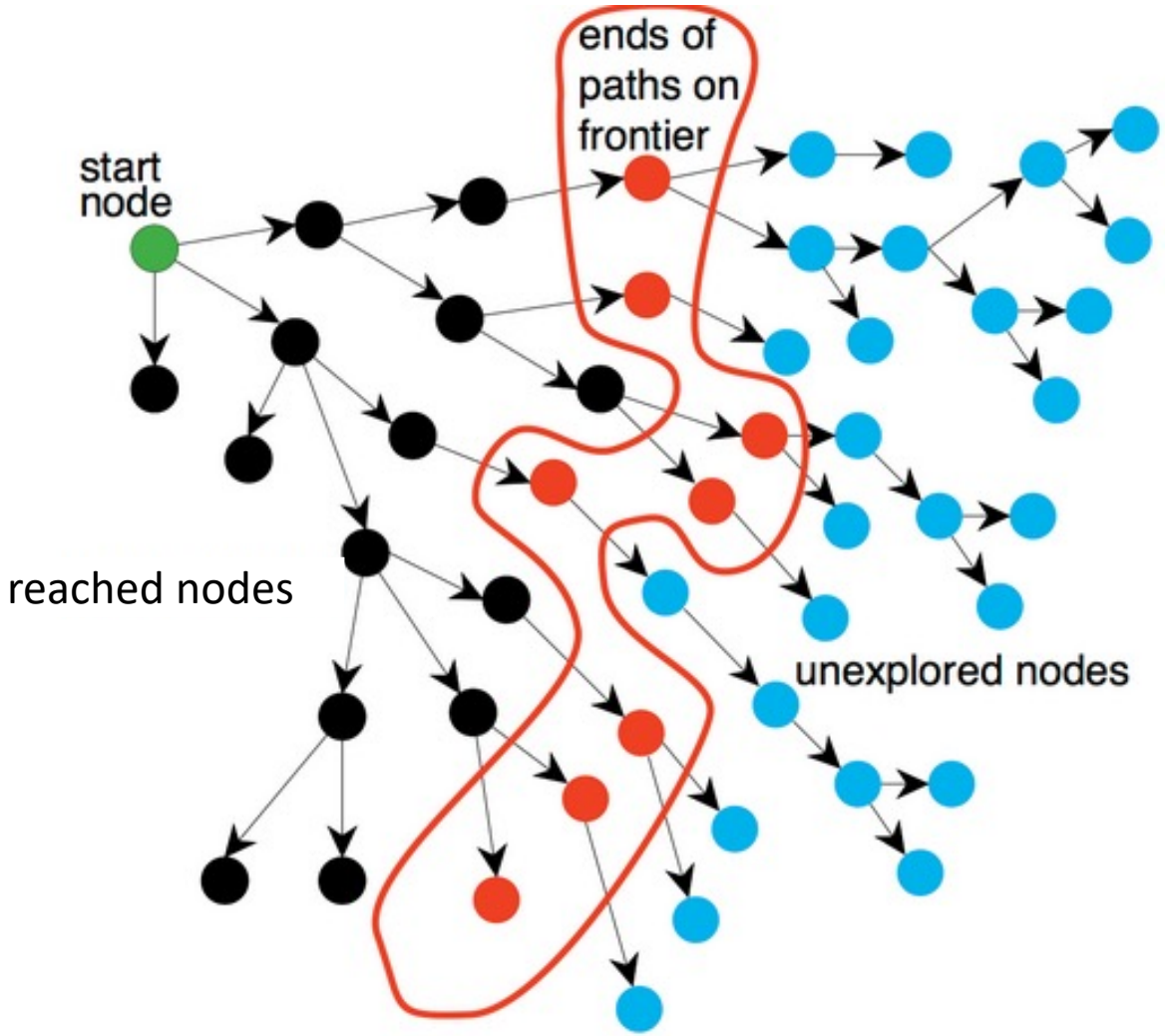
Heuristic: Straight Line Distance (SLD)

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Values of h_{SLD}—straight-line distances to Bucharest.

Goal: Bucharest

Heuristic Function: $h(n)$



$h(n_i)$: heuristic function
(Provides an estimated cost of cheapest path from node n_i to goal node)

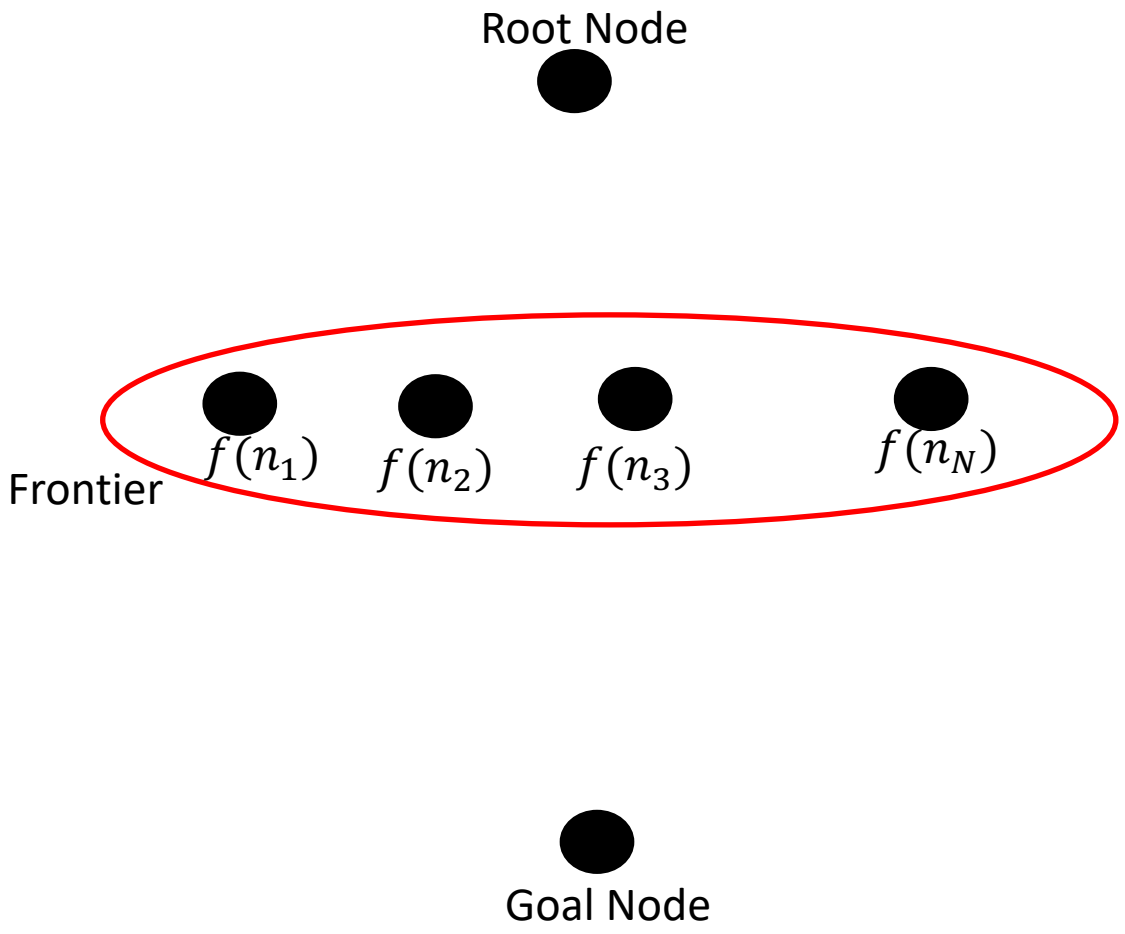
Heuristic Cost Guided Search

- Two Strategies
  - Greedy Best-First Search
  - A* Search

Greedy Best-First Search

- Also known as Greedy Search

- Node selection at frontier
  - Always select node with least estimated cost or least heuristic cost
  - $f(n_i) = h(n_i)$
  - $f(n_i)$ is called Evaluation Function



Greedy Best-First Search

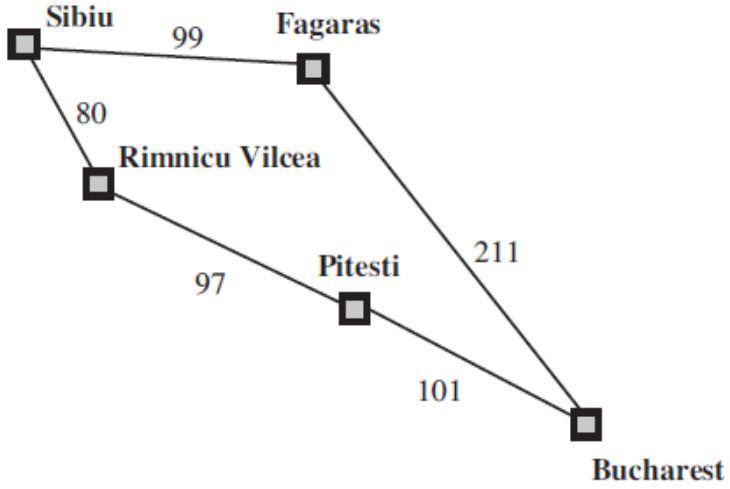
```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
    node ← NODE(STATE=problem.INITIAL)
    frontier ← a priority queue ordered by f, with node as an element
    reached ← a lookup table, with one entry with key problem.INITIAL and value node
    while not IS-EMPTY(frontier) 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
```

Greedy Best-First Search

Subscript indicates heuristic cost from this state to goal state

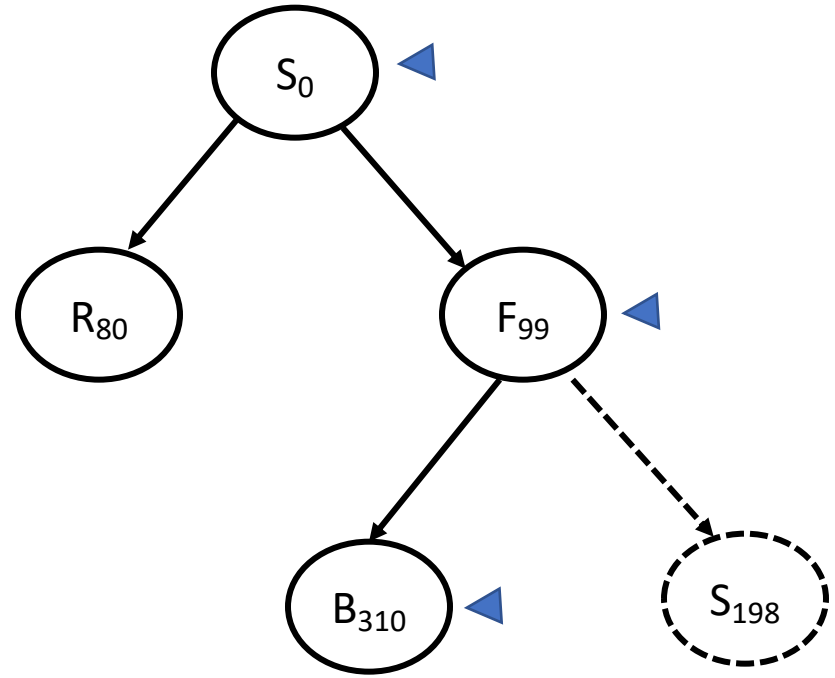
Node	Is-Goal(Node)	s	Frontier	Reached
S	No		$[S_{253}]$	$\{S_0\}$
S	No	R_{80}	$[R_{193}]$	$\{S_0,R_{80}\}$
S	No	F_{99}	$[R_{193}, F_{176}]$	$\{S_0,R_{80}, F_{99}\}$
F	No	B_{310}	$[R_{193}, B_0]$	$\{S_0,R_{80},F_{99}, B_{310}\}$
F	No	S_{198}	$[R_{193}, B_0]$	$\{S_0,R_{80},F_{99}, B_{310}\}$
B	Yes			

s is not added to the $frontier$ and $reached$



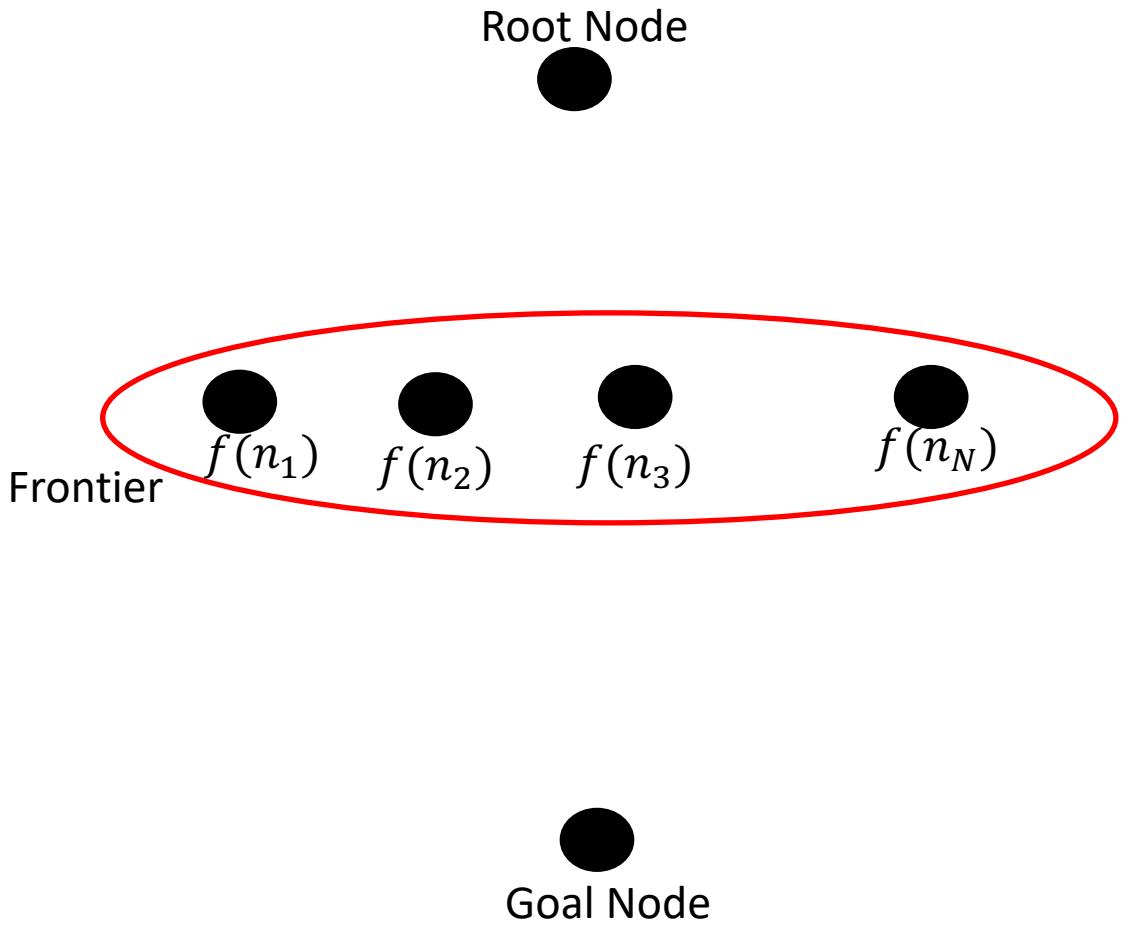
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Subscript indicates path-cost from root node to this state



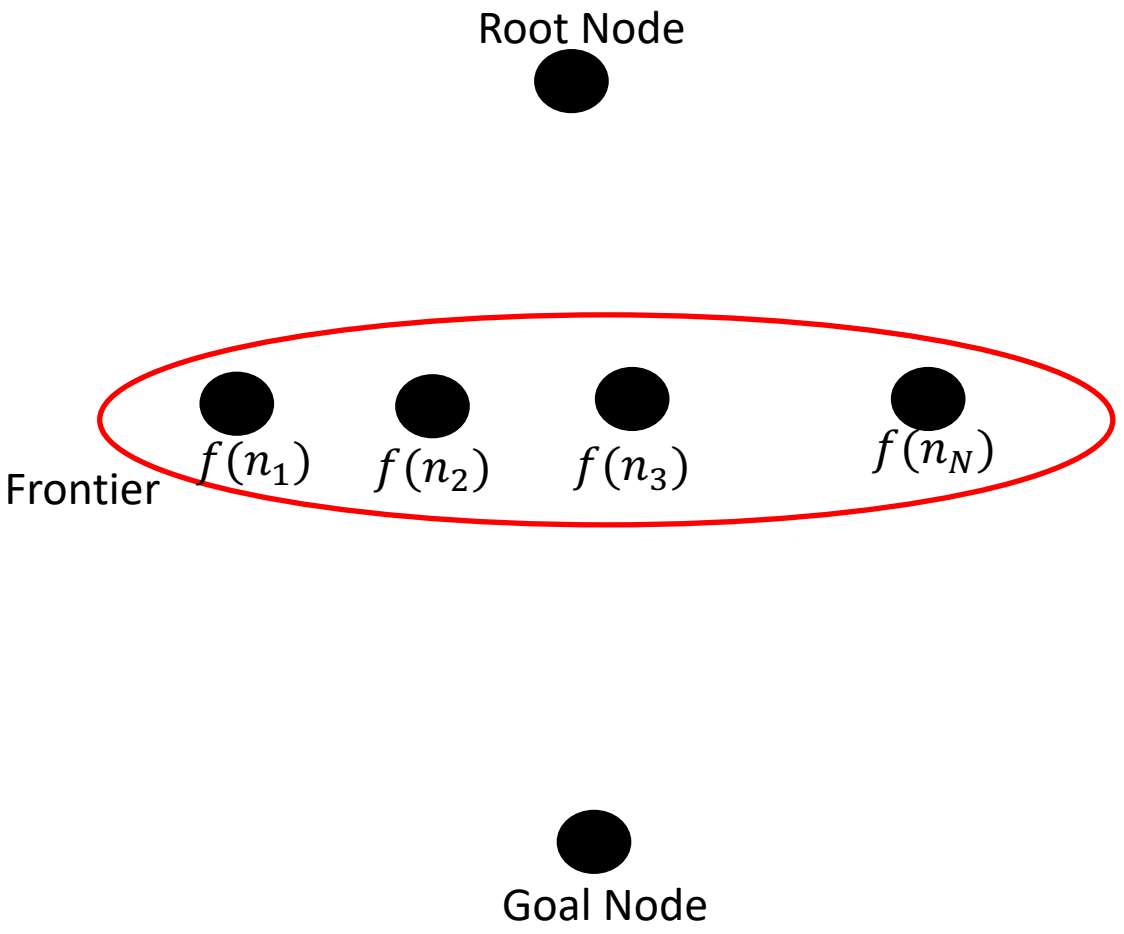
Greedy Best-First Search

- What information does it ignore?



A* Search

- Considers both past (i.e., path-cost) and estimated future cost (i.e., heuristic cost)
- Uses estimated cost from initial state to goal state
  - $f(n_i) = g(n_i) + h(n_i)$
    - $f(n_i)$: estimated cost of solution through node n_i
    - $g(n_i)$: actual path-cost from root node to node n_i
    - $h(n_i)$: estimated cost from node n_i to goal node
- Select node with least $f(n)$



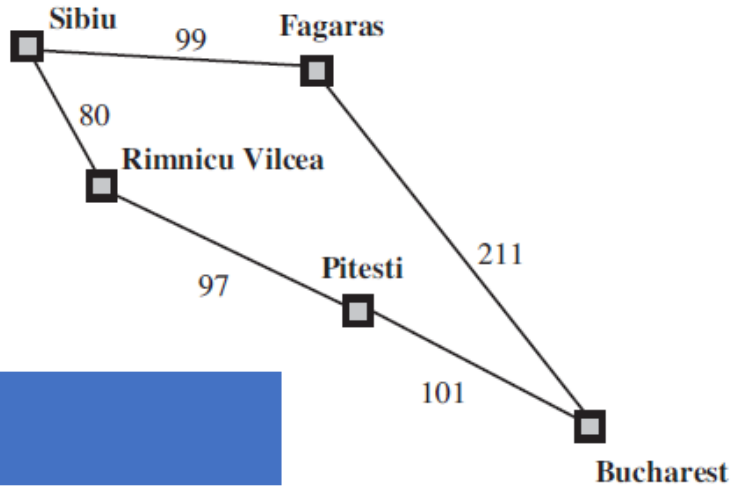
https://cacm.acm.org/magazines/2020/1/241713-a-search/fulltext

A* Search

Subscript indicates path-cost from root node to this state

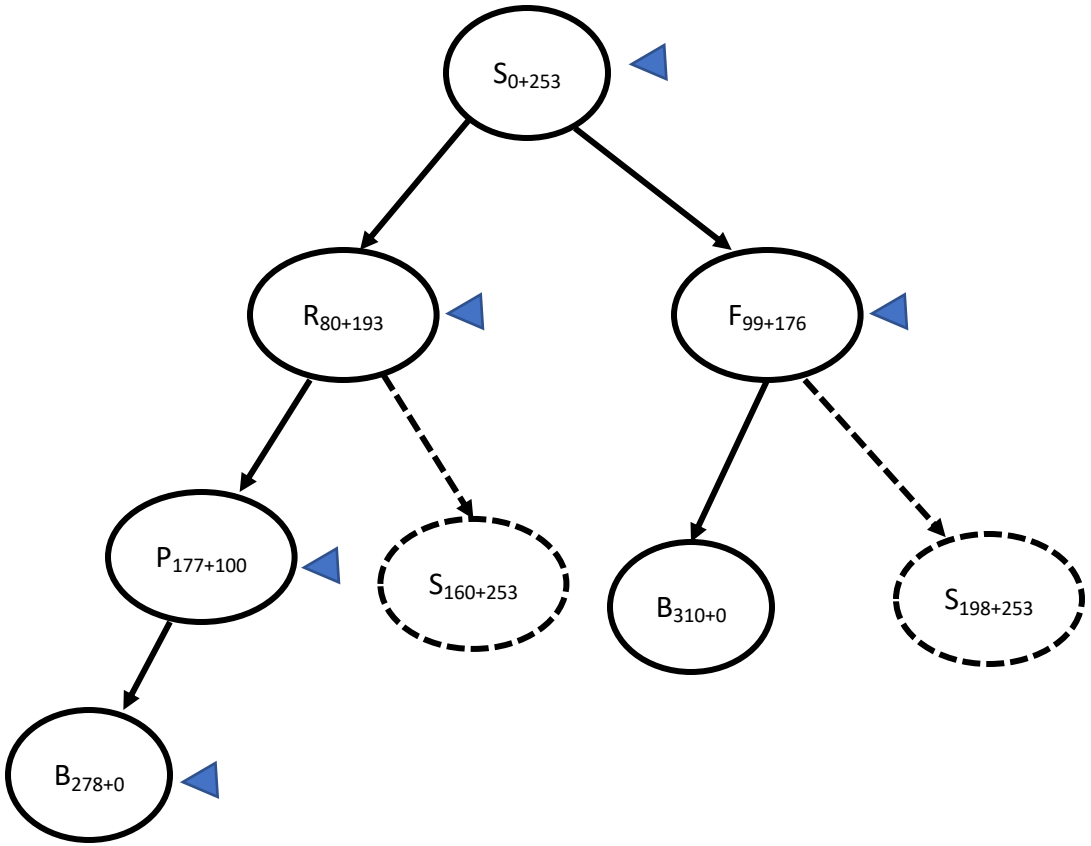
Node	Is-Goal(Node)	s	Frontier	Reached
S_0	No		$[S_{0+253}]$	$\{S_0\}$
S_0	No	R_{80}	$[R_{80+193}]$	$\{S_0,R_{80}\}$
S_0	No	F_{99}	$[R_{80+193}, F_{99+176}]$	$\{S_0,R_{80},F_{99}\}$
R_{80}	No	P_{177}	$[F_{99+176}, P_{177+100}]$	$\{S_0,R_{80},F_{99},P_{177}\}$
R_{80}	No	S_{160}	$[F_{99+176}, P_{177+100}]$	$\{S_0,R_{80},F_{99},P_{177}\}$
F_{99}	No	B_{310}	$[P_{177+100}, B_{310+0}]$	$\{S_0,R_{80},F_{99},P_{177}, B_{310}\}$
F_{99}	No	S_{198}	$[P_{177+100}, B_{310+0}]$	$\{S_0,R_{80},F_{99},P_{177}, B_{310}\}$
P_{177}	No	B_{278}	$[B_{310+0}, B_{278+0}]$	$\{S_0,R_{80},F_{99},P_{177}, B_{278}\}$
P_{177}	No	R_{274}	$[B_{310+0}, B_{278+0}]$	$\{S_0,R_{80},F_{99},P_{177}, B_{278}\}$
B_{278}	Yes			

Two terms in the subscript:
first term indicates path-cost from root node
second term indicates heuristic cost to reach goal node



Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search: Search Tree



Two terms in the subscript:

first term indicates path-cost from root node

second term indicates heuristic cost to reach goal node

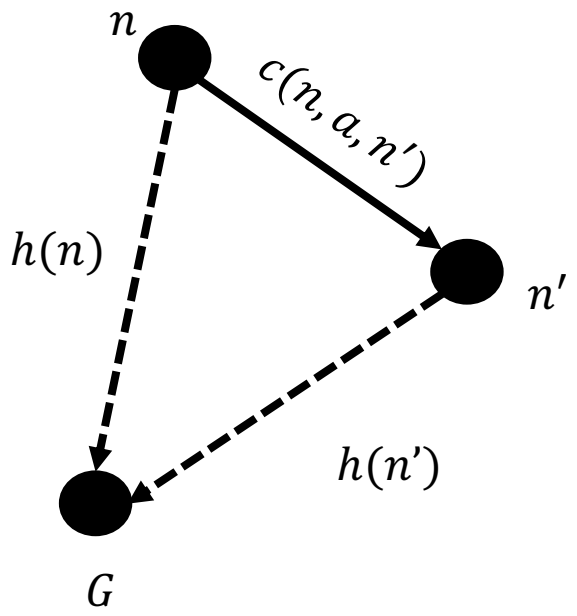
Note: Node only have path-cost attribute

Admissible Heuristics

- $h(n)$ is admissible, if $\forall n, h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach goal state from node n
- Never overestimate the cost to reach goal
- Optimistic when estimating the cost to goal
  - Ex: Straight Line Distance (SLD) or Line of Sight (LOS) distance is always less than actual distance
- A* search is optimal with admissible heuristics

Consistent Heuristic

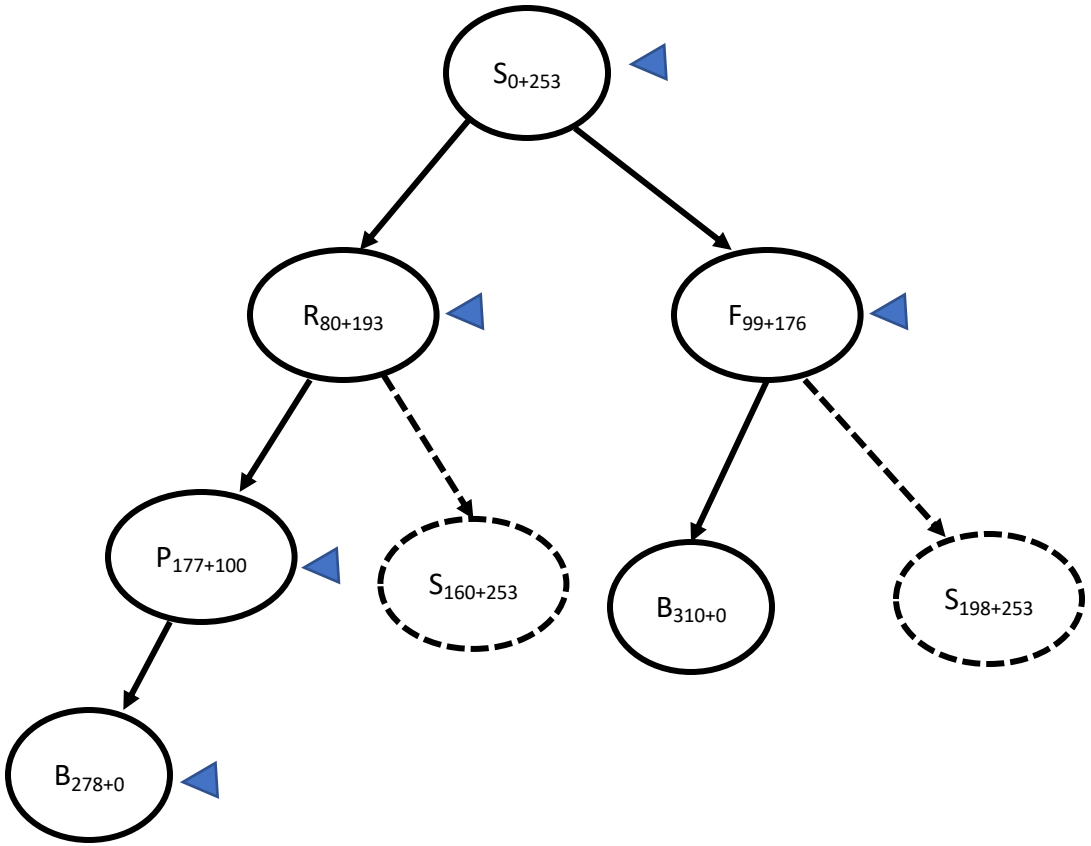
- A heuristic is consistent if, for every node n and every successor n' of n generated by an action a, $h(n) \leq c(n, a, n') + h(n')$



Consistency Vs Admissibility

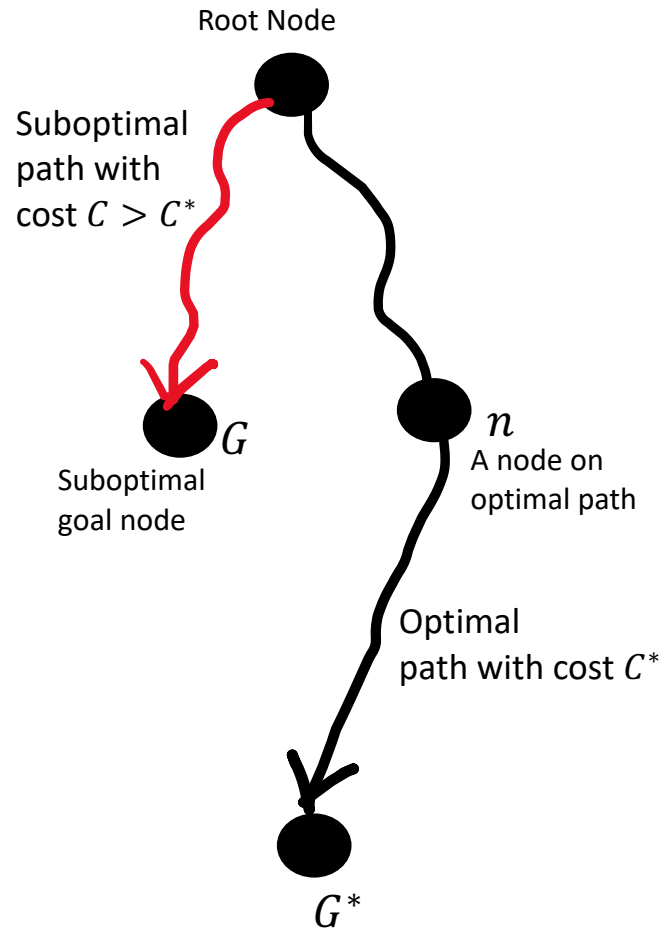
- Consistency is a stronger condition than admissibility
  - Consistency $\Longrightarrow$Admissibility
  - Admissibility $\not\Rightarrow$ Consistency

A* Search: Search Tree



Optimality of Admissible Heuristics

- Assume:
  - Optimal path has cost C^*
  - Algorithm returns another path with cost $C > C^*$
  - A node n is on shortest path and is unexpanded
- Proof by Contradiction:
  - $f(n) > C > C^*$ - Otherwise n would have been expanded
  - $f(n) = g(n) + h(n)$ - By definition
  - $f(n) = g^*(n) + h(n)$ – Becuase n is on optimal path
  - $f(n) \leq g^*(n) + h^*(n)$ – Because of admissibility $h(n) \leq h^*(n)$
  - $f(n) \leq C^*$ – By definition $C^* = g^*(n) + h^*(n)$



$g^*(n)$: optimal path-cost of node n from root node
$h^*(n)$: optimal path-cost from node n to goal node

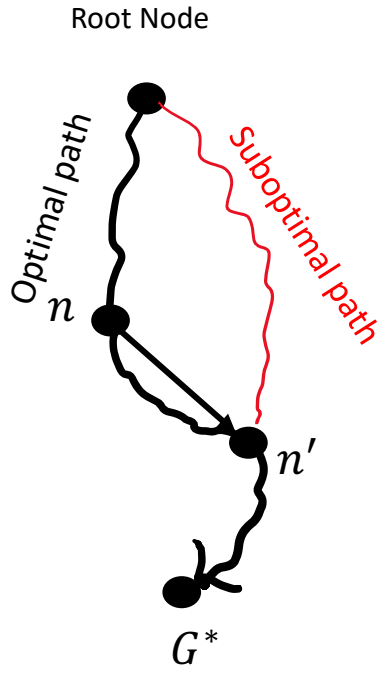
Monotonicity of $f(n)$

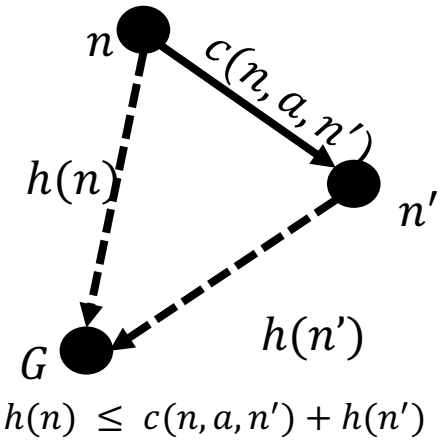
- Let n' be the successor of n
  $\Rightarrow g(n') = g(n) + c(n, a, n')$
  $\Rightarrow f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n')$
  $\Rightarrow f(n') \geq g(n) + h(n) = f(n)$ (because $h(n)$ is consistent)

- So, always n will be selected ahead of n'

- $f(n)$ is nondecreasing along any path with consistent heuristic

- With consistent heuristic, once a node is selected from frontier, the optimal path till that node is found





$h(n) \leq c(n, a, n') + h(n')$

Search Contours

Check Fig. 3.18

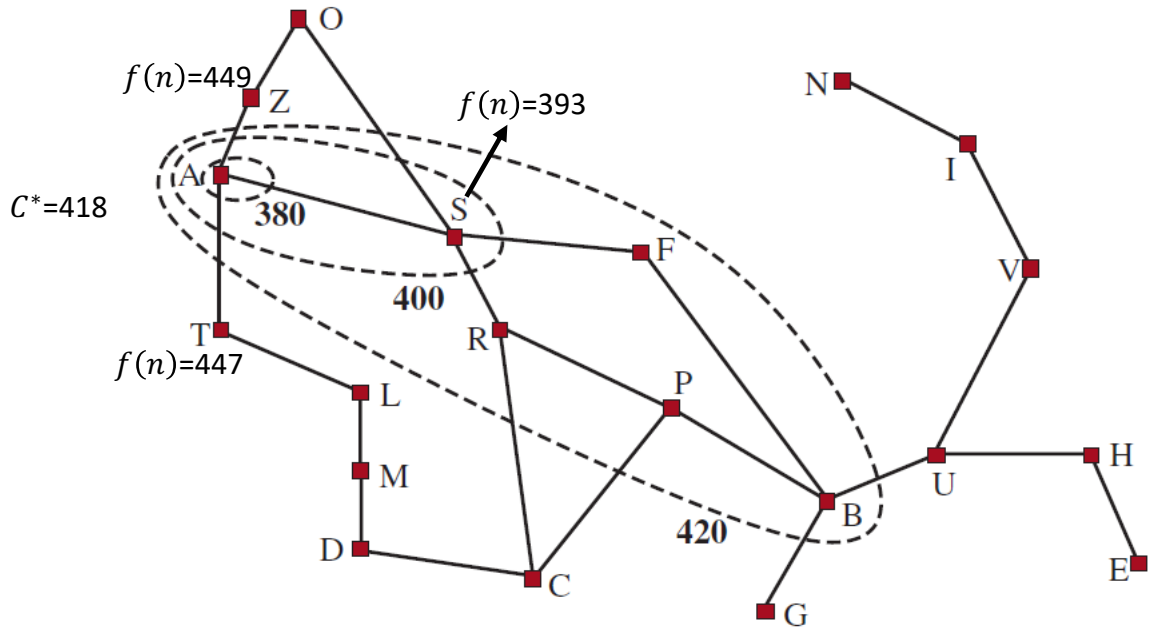


Figure 3.20 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f = g + h$ costs less than or equal to the contour value.

A* expands all nodes with $f(n) < C^*$

A* might expand some nodes on goal contour,
i.e., nodes with $f(n) = C^*$

A* never expands nodes with $f(n) > C^*$

What about UCS?

Effect of heuristic cost on number of nodes explored?

- Extreme cases:
  - $h(n) = 0$
    - Explores all nodes with path-cost less than optimal cost C^*
    - Same as uniform-cost search
  - $h(n) = h^*(n)$
    - $h^*(n)$ is true cost
    - Explores only nodes on optimal path

- Number of nodes explored are in general between the above two extremes

How to derive admissible heuristics?

- From relaxed problems
  - Reduce the restrictions on actions
  - Cost of optimal solution to a relaxed problem is admissible heuristic for original problem

- From subproblems
  - Pattern databases

How to derive admissible heuristics?

- From Landmarks
  - Select a few landmark points (pivots or anchors)
  - Compute C^* (distance between each state and landmark)
  - Store C^* and use it to calculate heuristic

$$h_L(n) = \min_{L \in Landmarks} C^*(n, L) + C^*(L, goal)$$

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n, L) - C^*(goal, L)|$$

Dominant Heuristics

- Let $h^*(n)$ be the optimal cost
- Let $h_i(n)$ and $h_j(n)$ are two admissible heuristic functions
- If $\forall n, h_i(n) > h_j(n)$, then $h_i(n)$ dominates $h_j(n)$
- We can get dominant heuristic ($h\ (n)$) from nondominant heuristics:
  - $h\ (n)\ =\ \max(h_1(n), \ldots., h_N(n))$

Metrics for efficiency

Only for your reading

- Effective branching factor (b^*)
  - A lean search tree implies less number of generated nodes
  - Characterized by effective branching factor

- Effective search depth

$$N = b^* + (b^*)^2 + (b^*)^3 + \cdots + (b^*)^d$$

A* Search: Saving Memory

Only for your reading

- Reference counts
  - Count number of times a state is reached
  - Delete the state from $reached\ table$ once target count is met

- Beam Search
  - Keep only top k nodes in frontier

- Iterative Deepening A* Search

- Recursive Best-First Search

A* Search: Improving Time Complexity

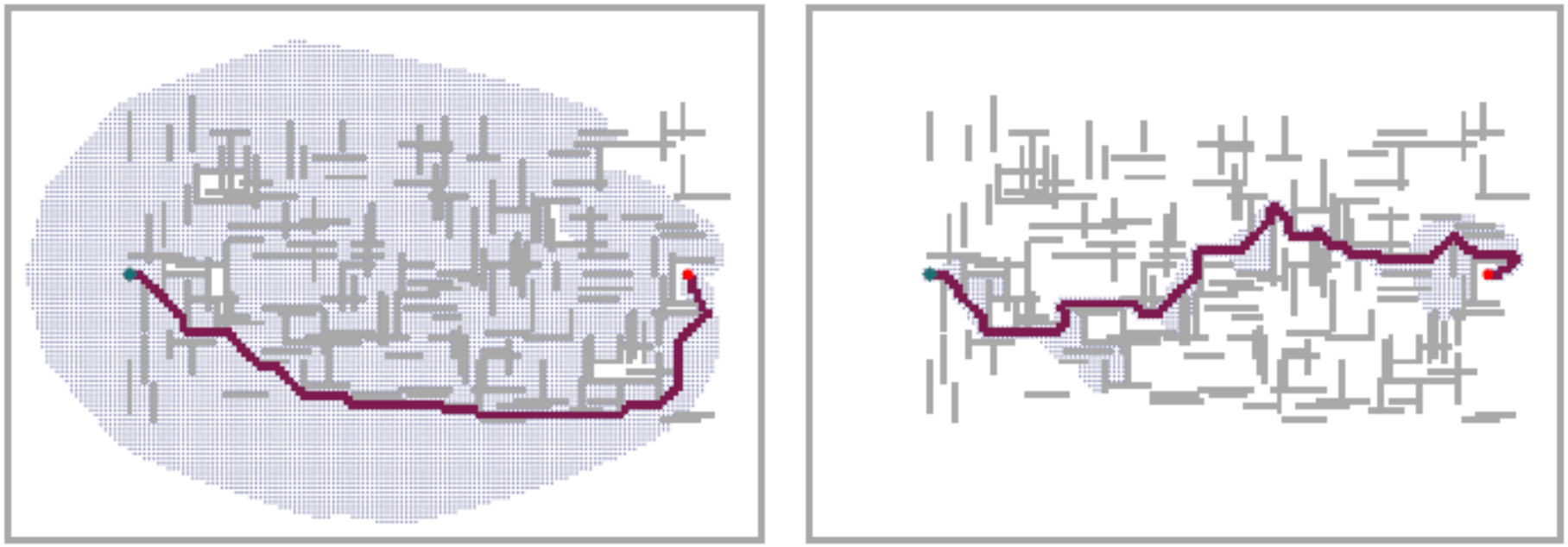
Only for your reading

- Weighted A* Search
  - A* Search with:

    $f(n) = g(n) + W\ h(n), \quad 1 < W < \infty$
  - Route Planning:
    - W is called detour index
    - Leads to violation of admissibility
  - Bounded suboptimality
    - Solution cost lies between C^* and WC^*
  - Explores less number of nodes at the expense of path-cost

Weighted A* Search

Only for your reading



(a) (b)

Figure 3.21 Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

Review

- Uninformed Search Algorithms
  - Breadth-First Search
  - Depth-limited Search
  - Iterative Deepening Search
  - Uniform-Cost Search
- Informed Search Algorithms
  - Heuristic function
  - Greedy Best-First Search
  - A* Search

Summary

- Uninformed search algorithms are inefficient
- Efficiency can be improved through heuristics
- Heuristics aid in pruning search tree