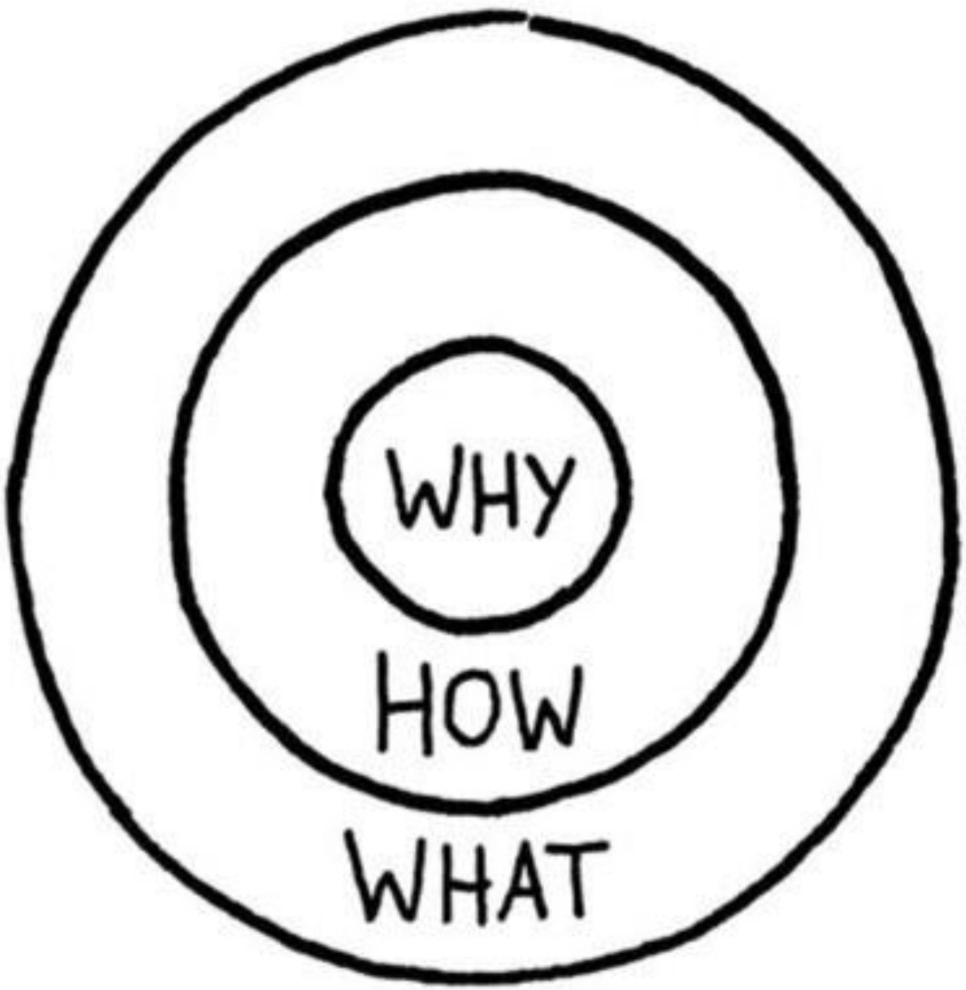
MACHINE LEARNING

WEEK 03

اَللّٰهُمَّ اُرْزُقْنِی عِلْمًا نَافِعًا وَاسِعًا عَمِیْقًا

اَللّٰهُمَّ اُرْزُقْنِى رِزْقًا وَاسِعًا حَلَالًا طَيِّبًا
مُبَارَكًا مِنْ عِنْدِكَ

GOLDEN CIRCLE



Why = The Purpose
What is your cause? What do you believe?

Apple: We believe in challienging the status quo and doing this differently

How = The Process
Specific actions taken to realize the Why.

Apple: Our products are beautifully designed and easy to use

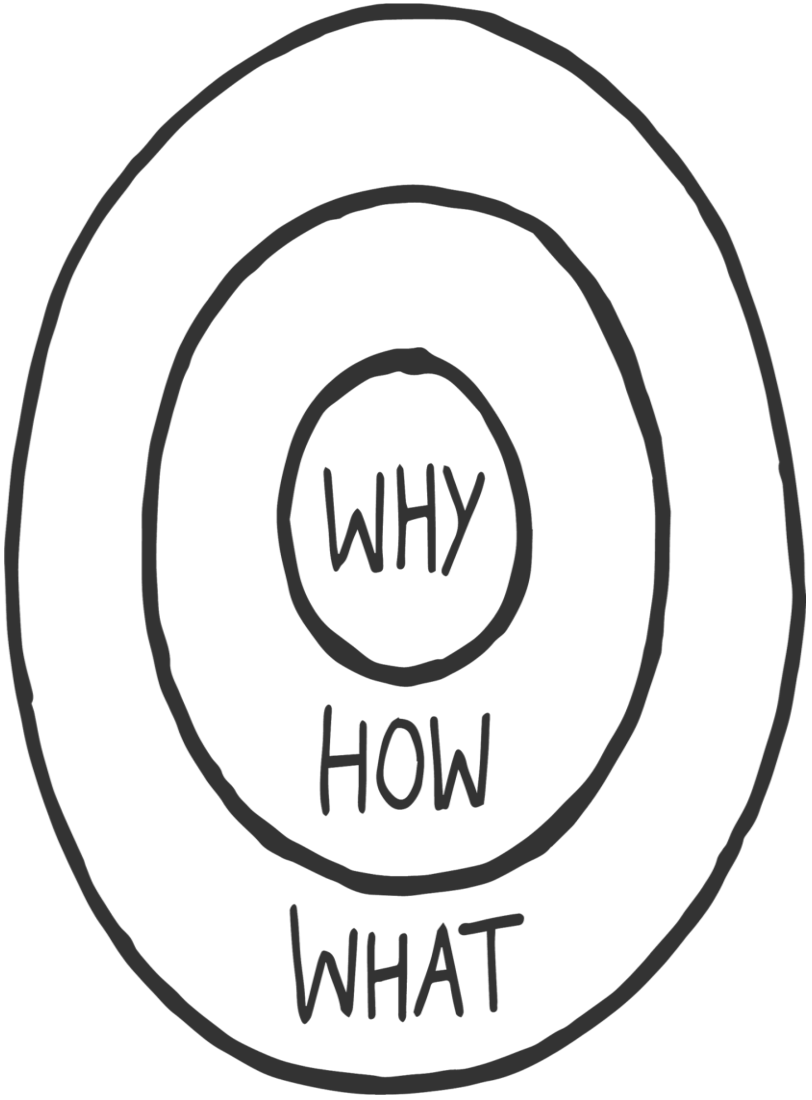
What = The Result
What do you do? The result of Why. Proof.

Apple: We make computers

GOLDEN CIRCLE
MACHINE LEARNING

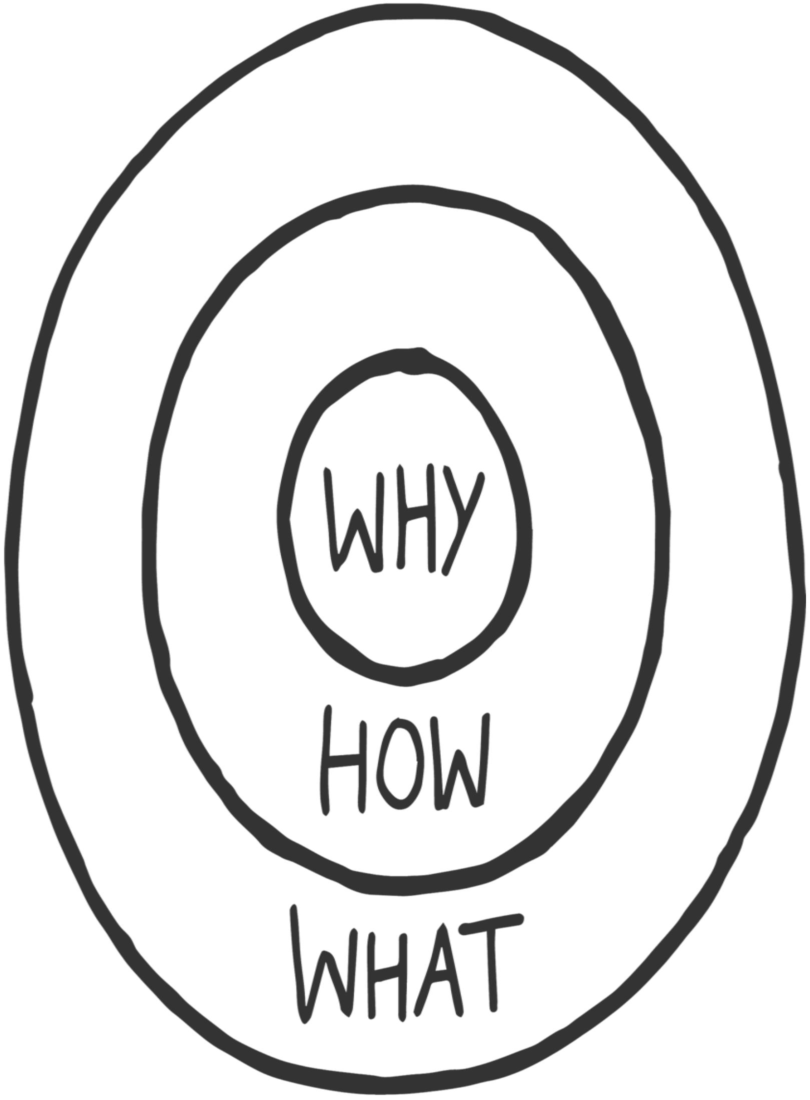
WHY

Making Machine to Learn from Past Experience



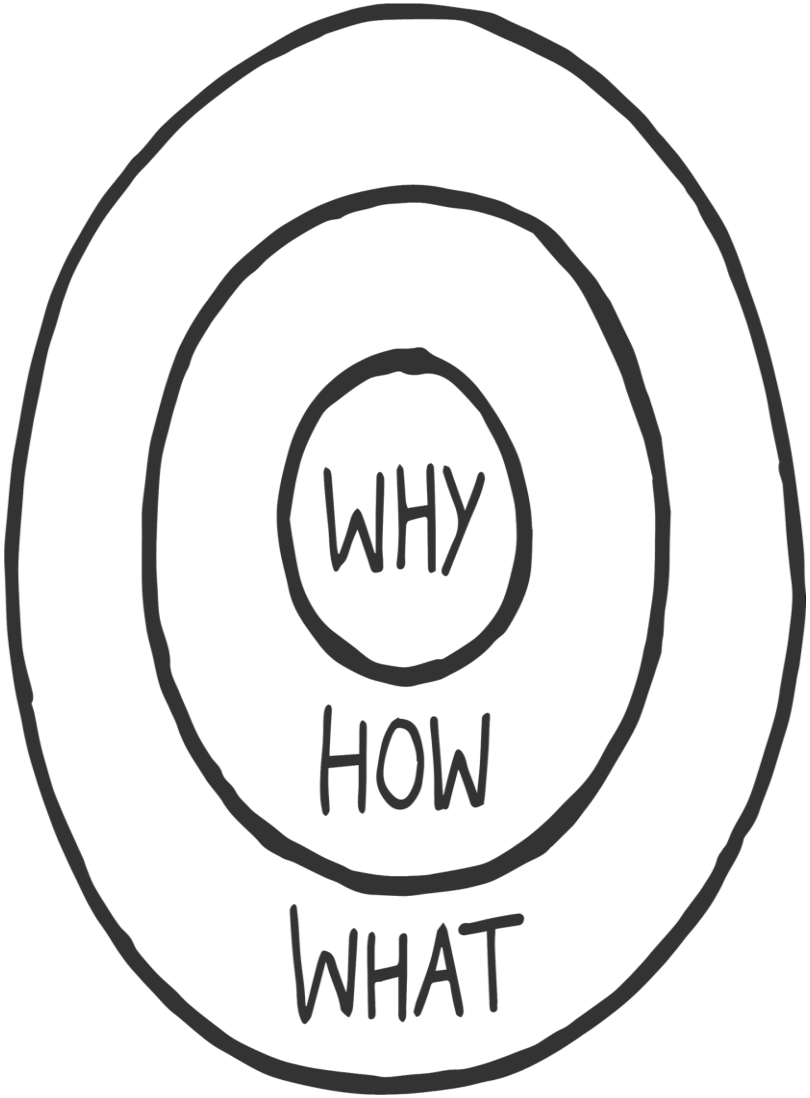
HOW

Symbolic
Connectionis
Evolutionary
Stochistic or
Probablistic



WHAT

Search Algorithm
Classification
Regression
Clustering
Strategy to Achieving
Goals (RL)
NLP
Vision



WEEK 03: SEARCH BASED LEARNING

- Real World Problem to Learning Problem Modeling
- Search Trees and Goal Finding.
- Heuristic Search.

ASSIGNMENT 02

Review Python

Understand **PACMAN** Project

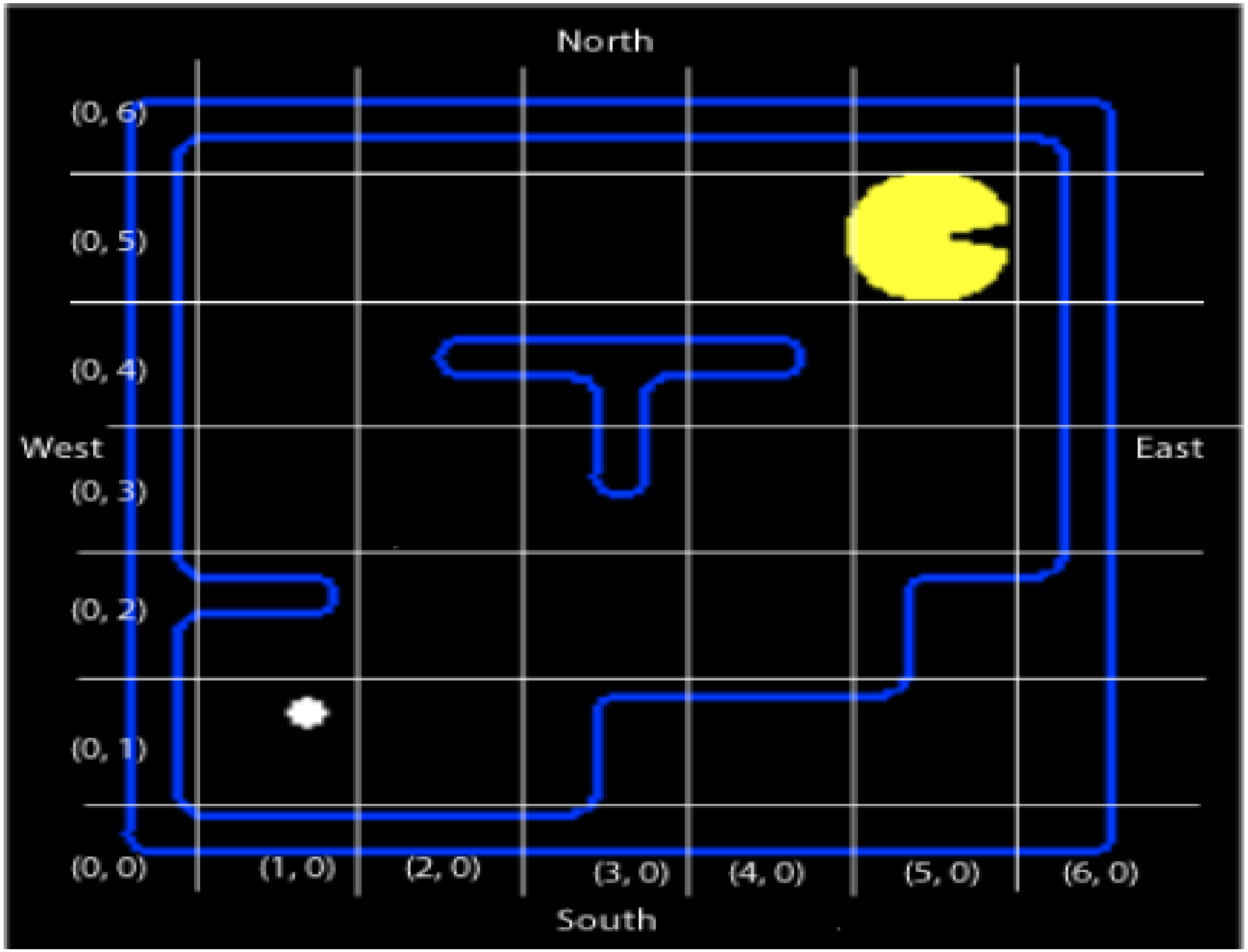
Move the PACMNA around See the Detail in the Assignment folder.

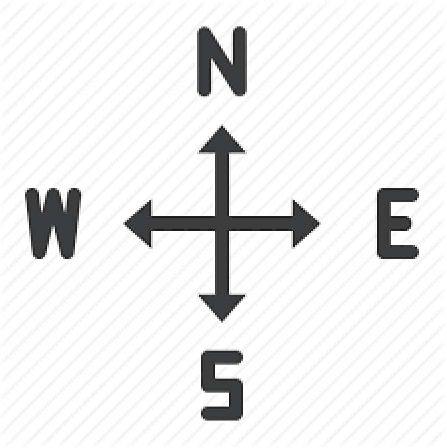
REAL WORLD PACMAN

COMPUTATIONAL MODEL

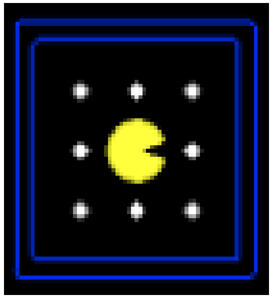
?

COMPUTATION MODEL



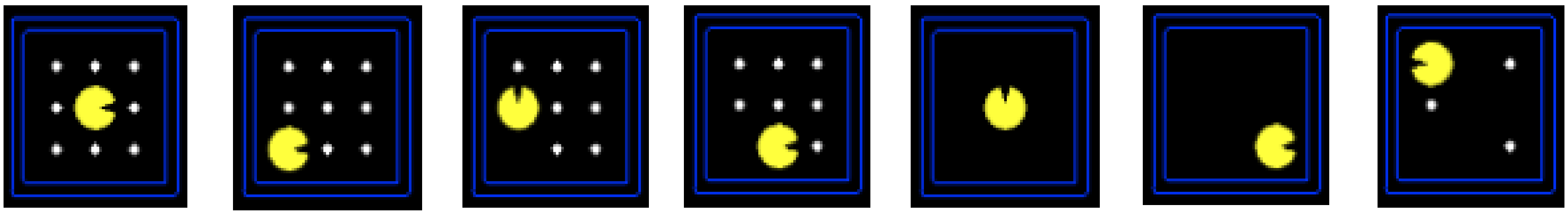


- **Start state**

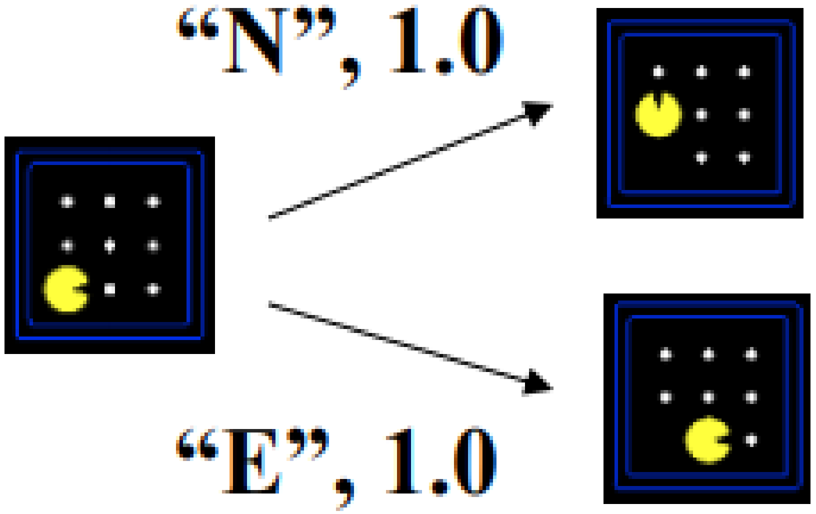


https://courses.cs.washington.edu/courses/cse473/13au/slides/03-search.pdf

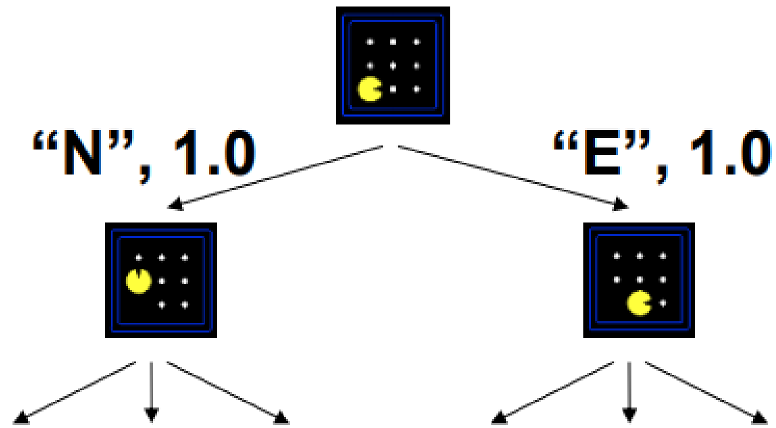
STATE SPACE



SUCCESSOR FUNCTION



SEARCH TREE



A search tree:

- Root contains Start state
- Children = successor states
- Edges = actions and step-costs
- Path from Root to a node is a "plan" to get to that state
- For most problems, we can never actually build the whole tree (why?)

STATES.

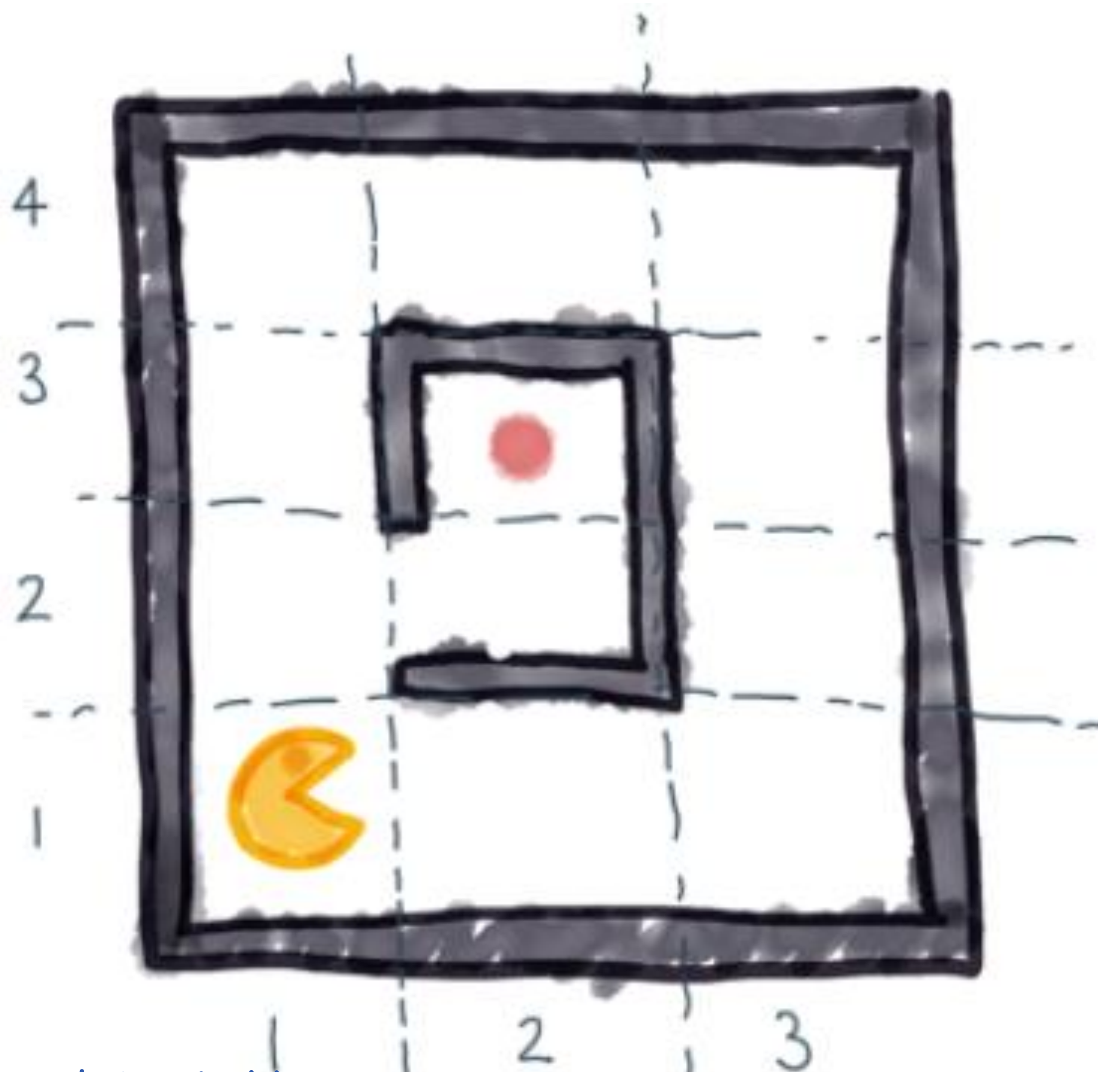
startState=problem.getStartState()

(4,3)

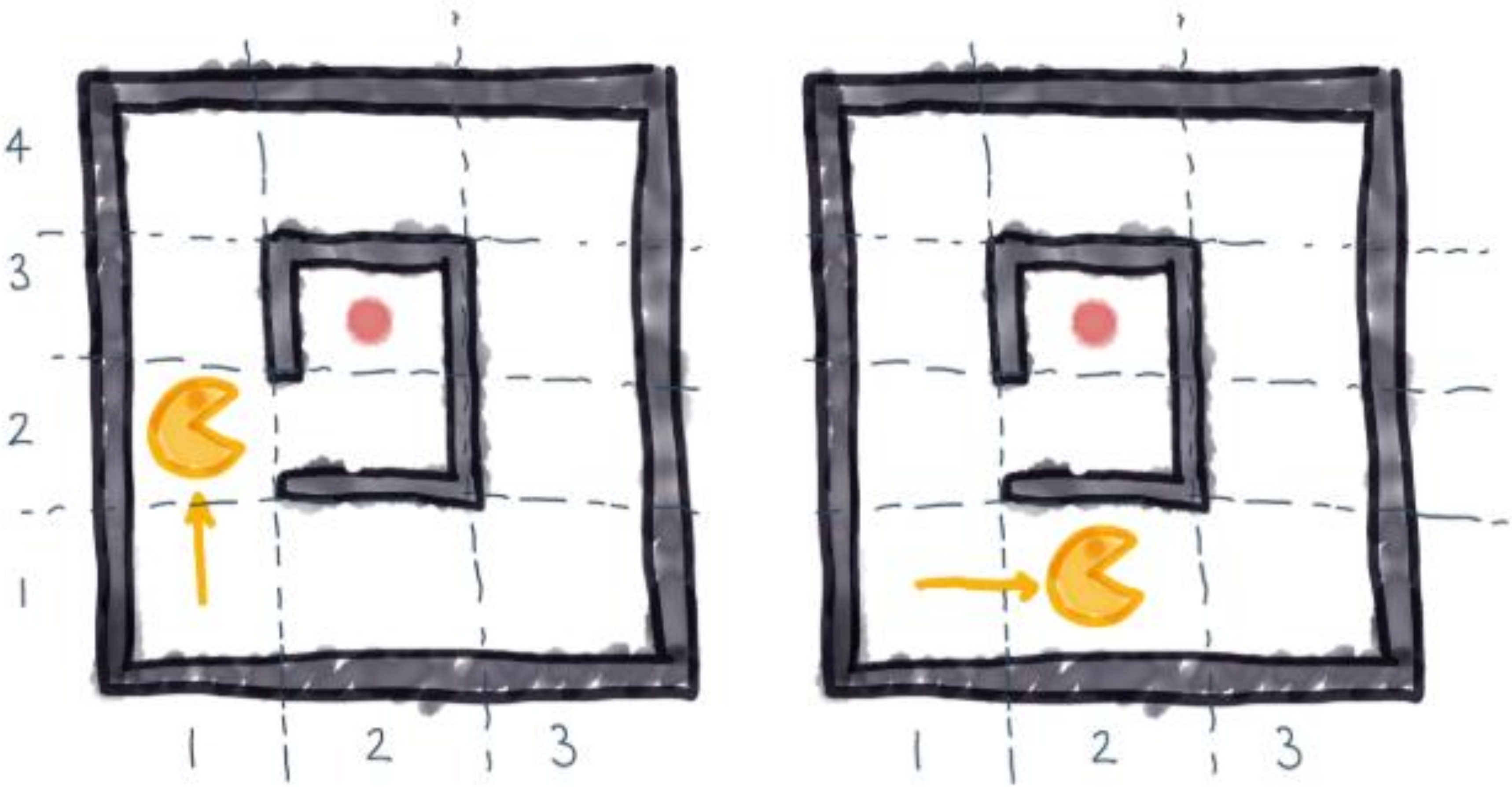
Problem.getSuccessor(startState)

[((3,3),w,cost),((4,2),s,1)]

HOW TO MOVE PACMAN

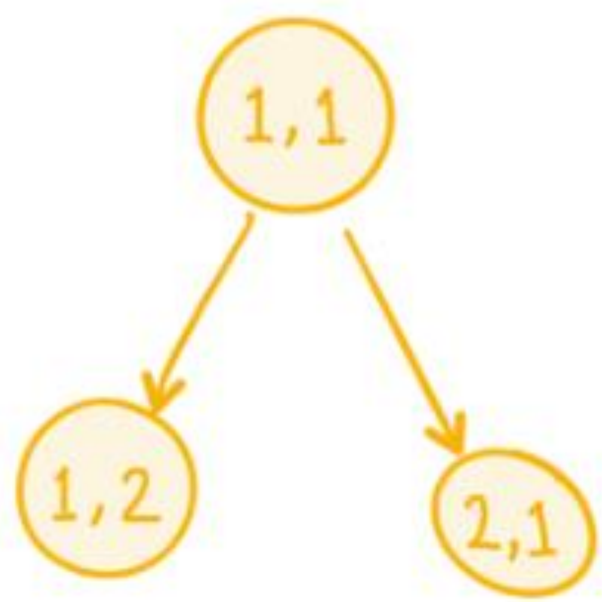


STATES



TREE SPACE

Representing this in a tree looks like this



SEARCHING STRATEGIES

RECALL THE GRAPH ALGORITHM

```
function GRAPH-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    initialize frontier with initialState
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.remove()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.add(neighbor)

    return FAILURE
```

GRAPH ALGORITHM WITH PATH

```
function GRAPH-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    initialize frontier with initialState
    explored = Set.new()
      Paths = {} // initialize a dictionary
    while not frontier.isEmpty():
        state = frontier.remove()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)        Return Paths = {state}

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.add(neighbor)
                  Paths.add ( paths[state]+ child(action))

    return FAILURE
```

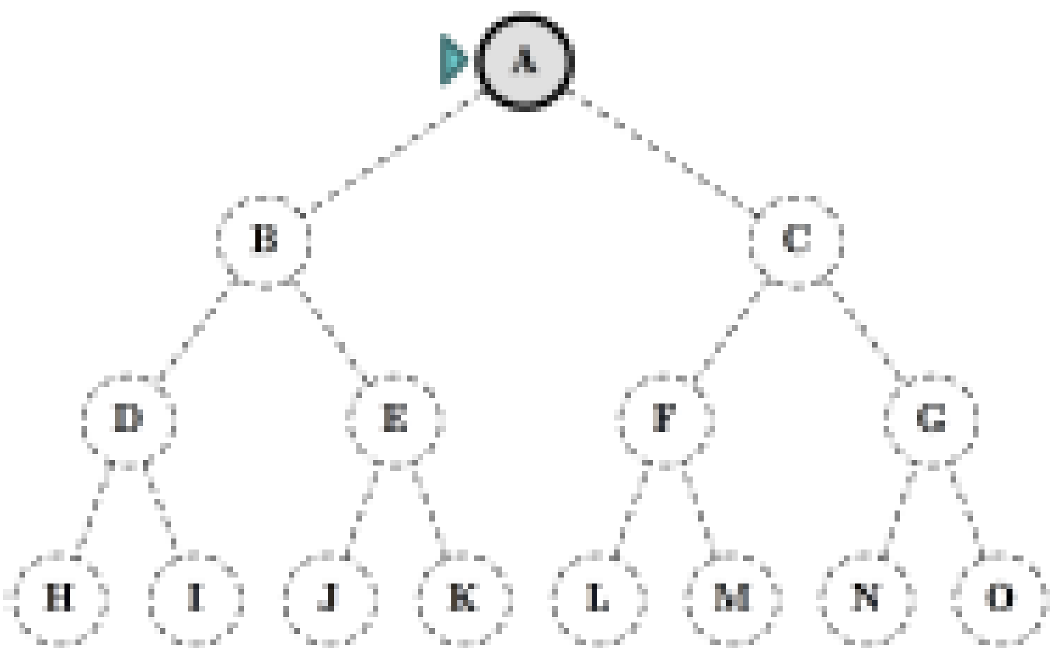
Dr. Muhammad Awais Hassan
Department of Computer Science UET, Lahore

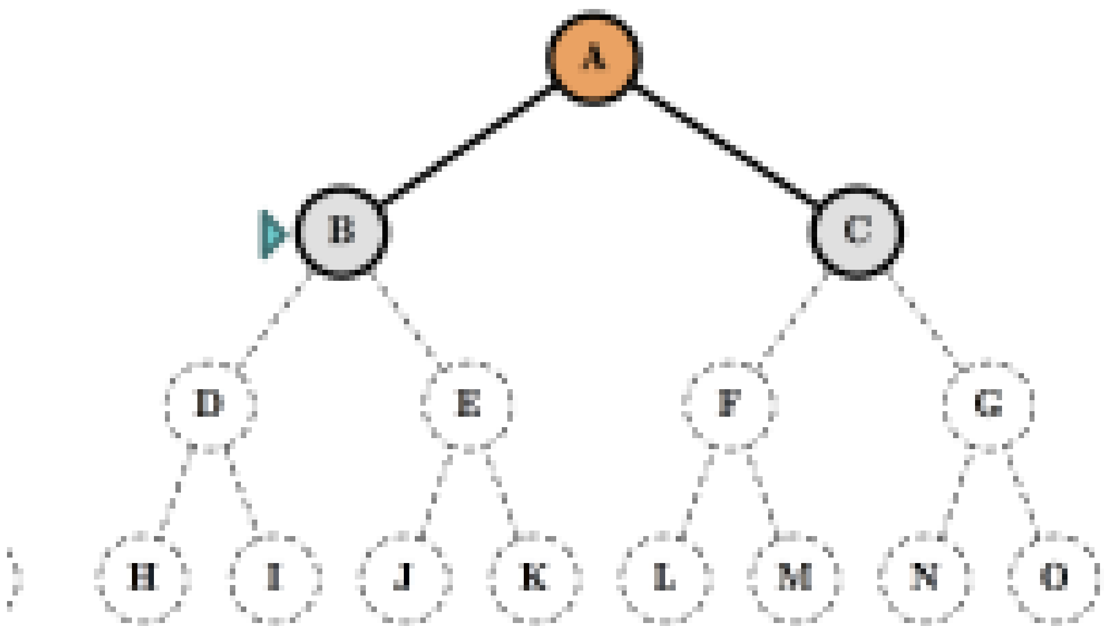
WHAT IF WE CHANGE THE FRONTIER WITH THE QUEUE

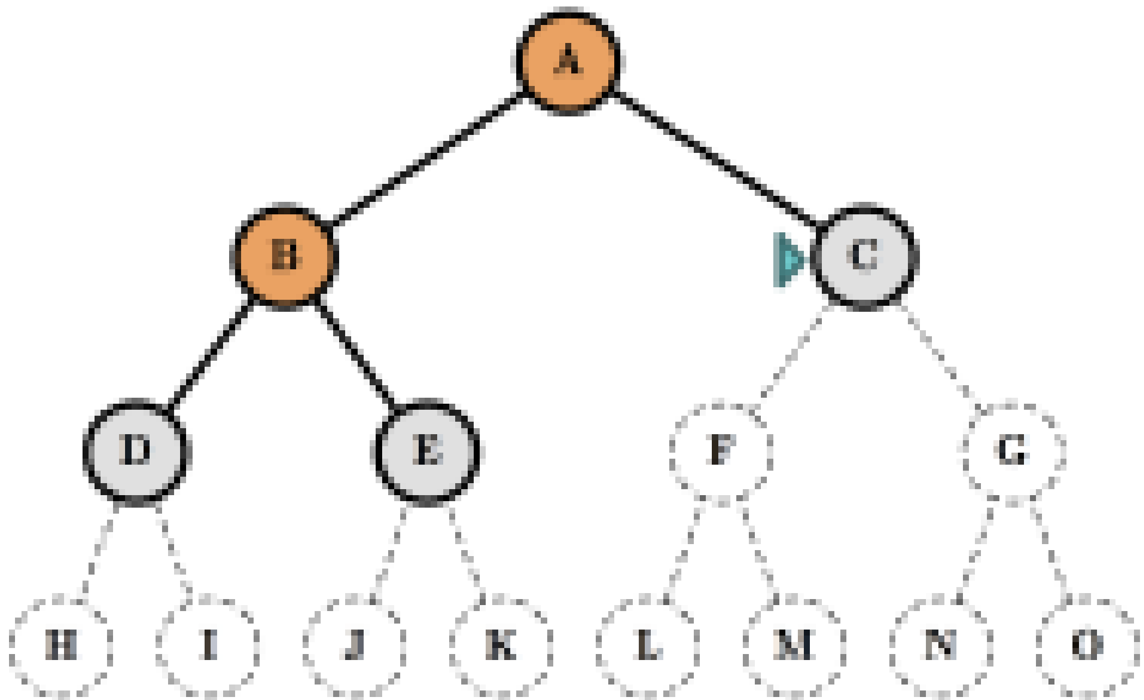
HOW IT SHALL WORK ?

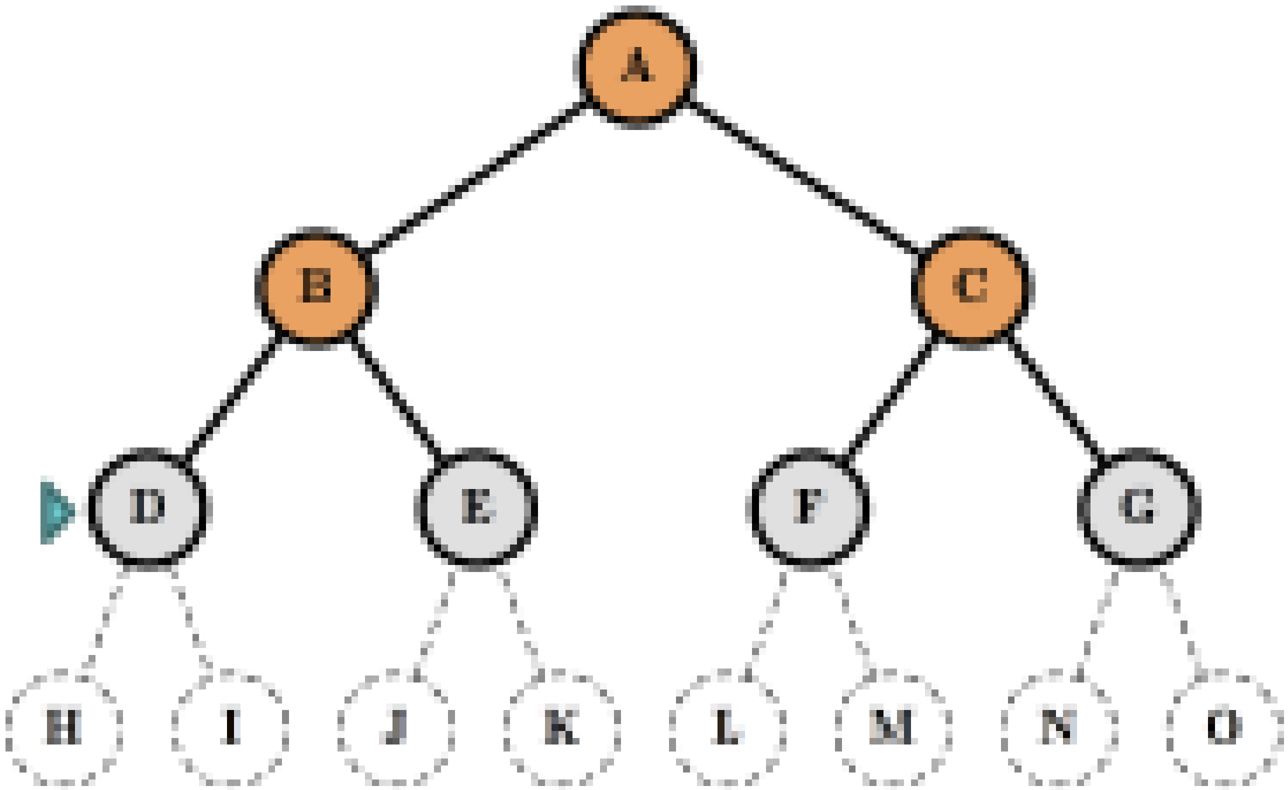
DRY RUN

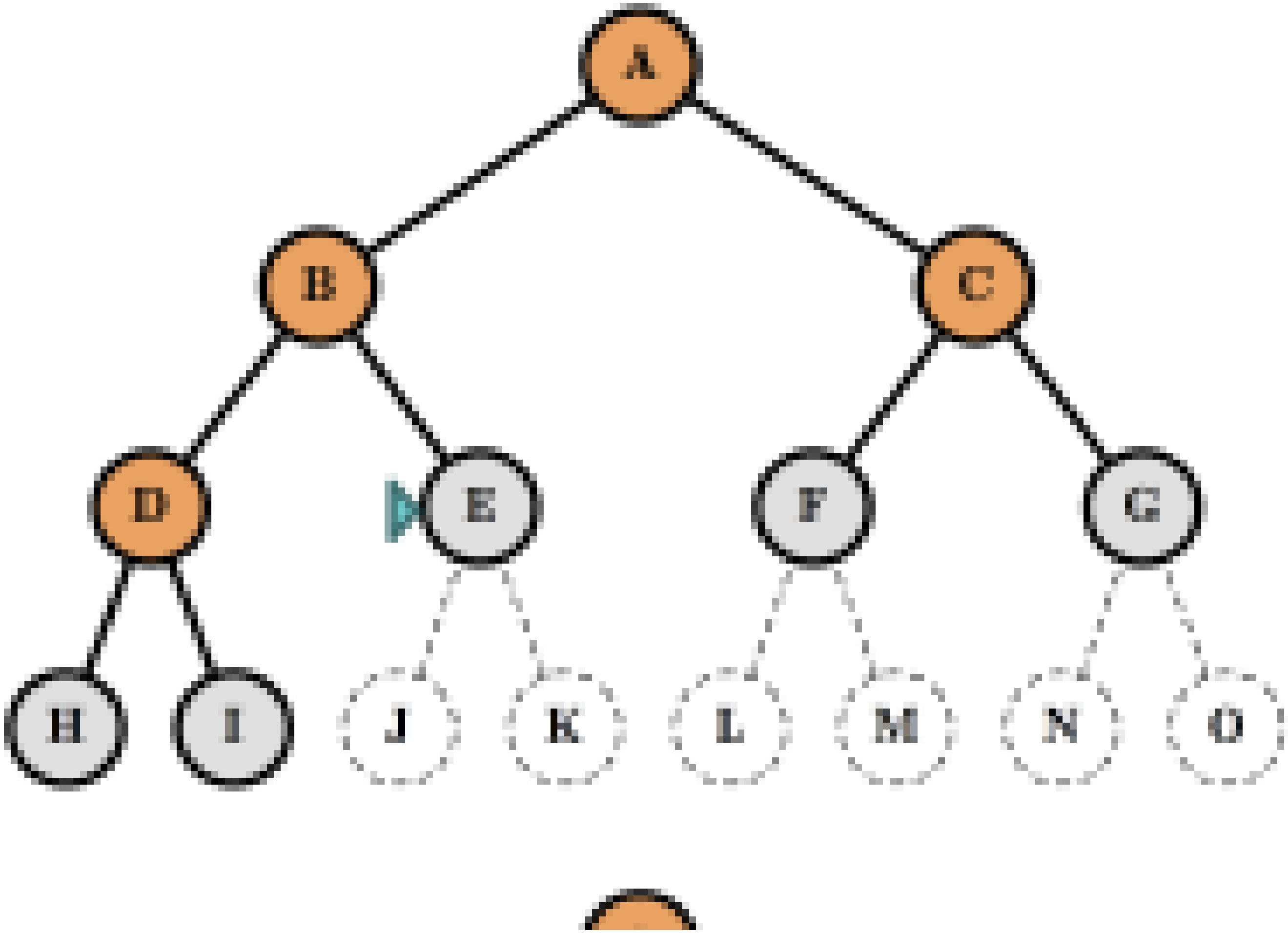
- Give the dry run on the code at following tree when fringe is Queue

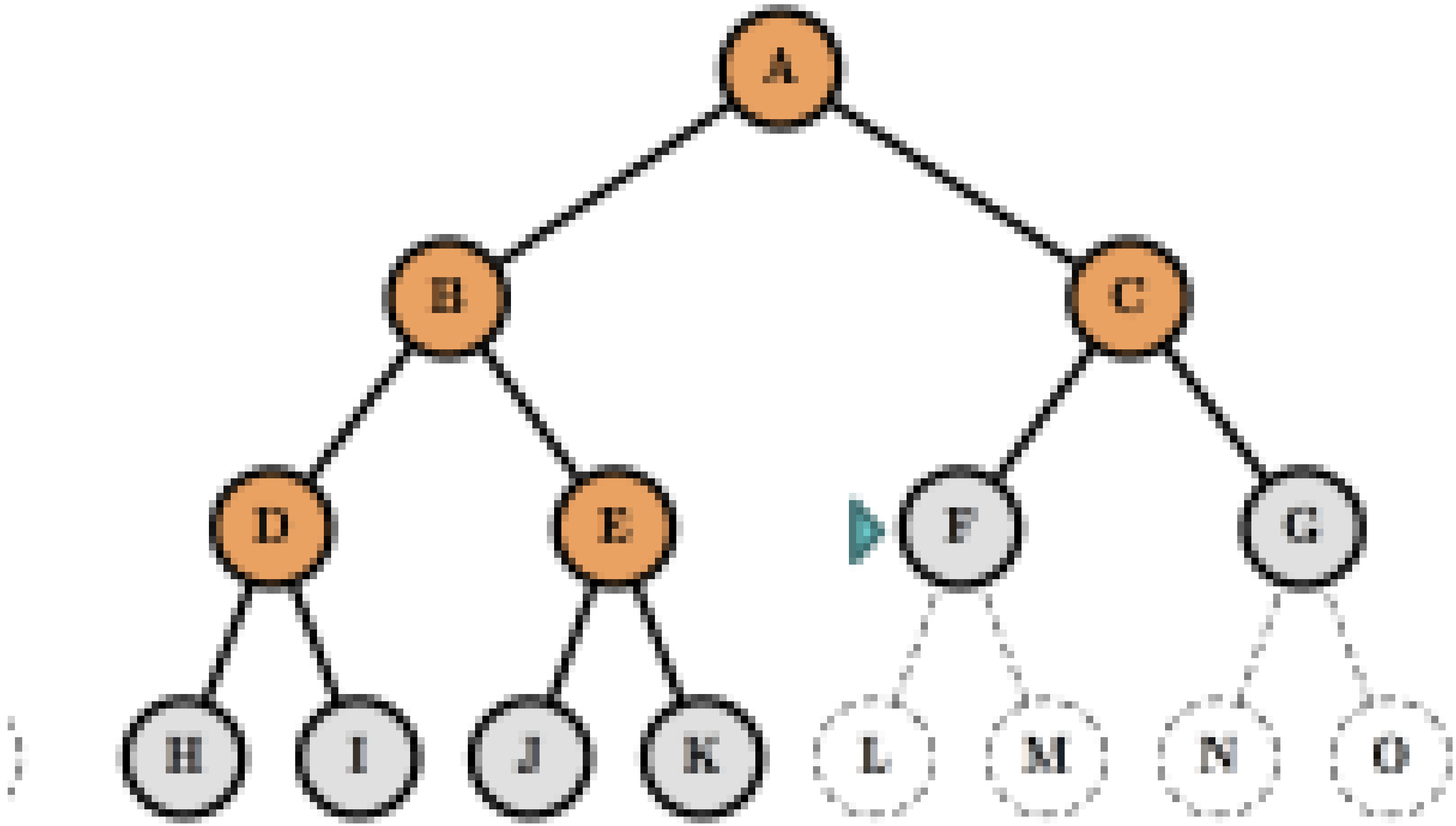



A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

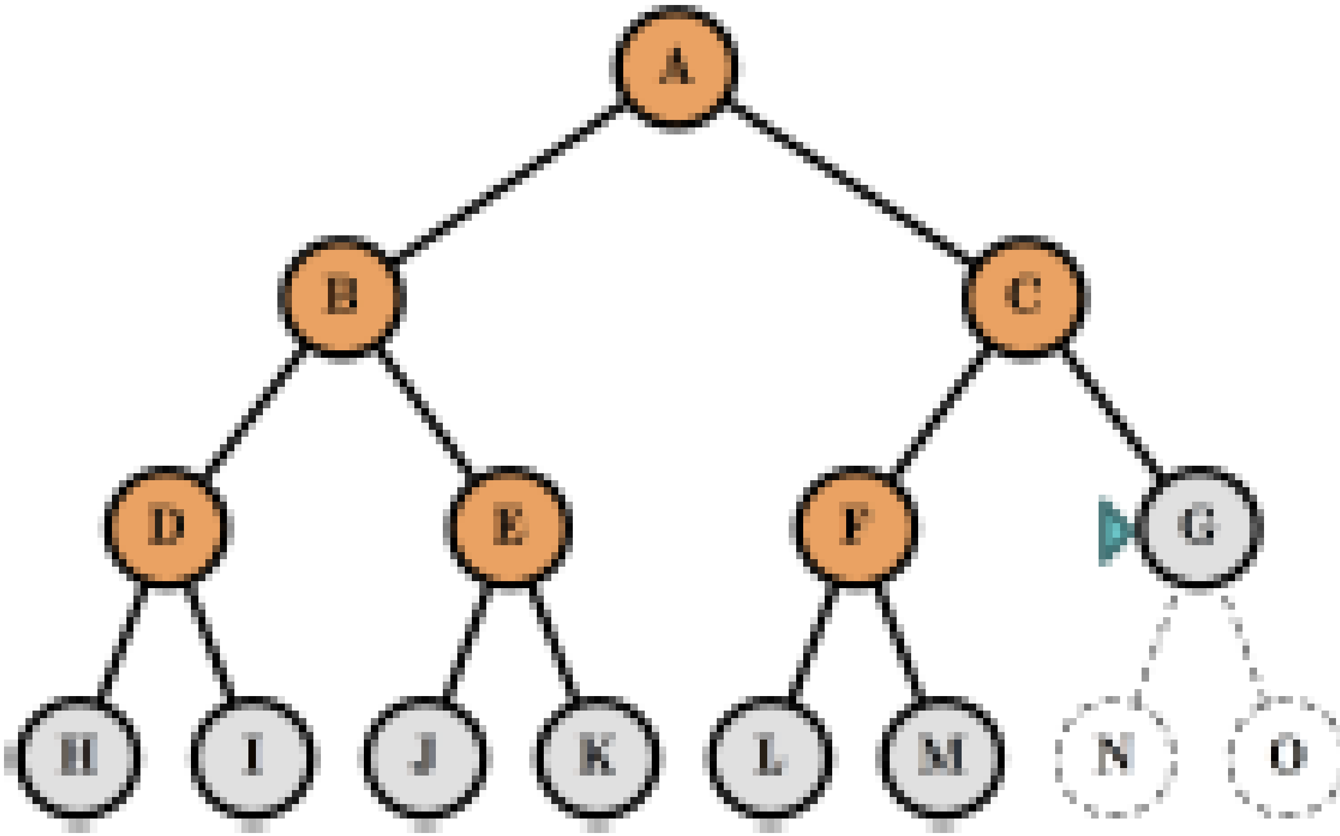

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

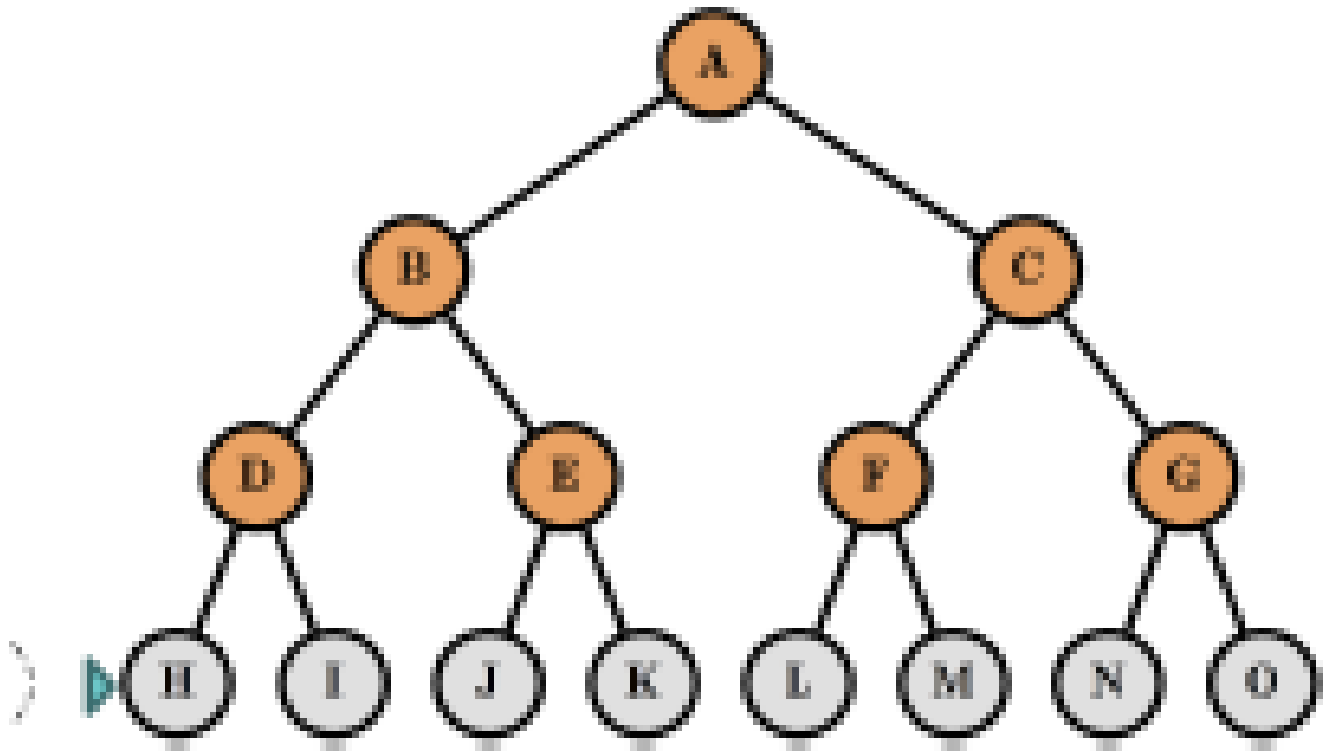

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

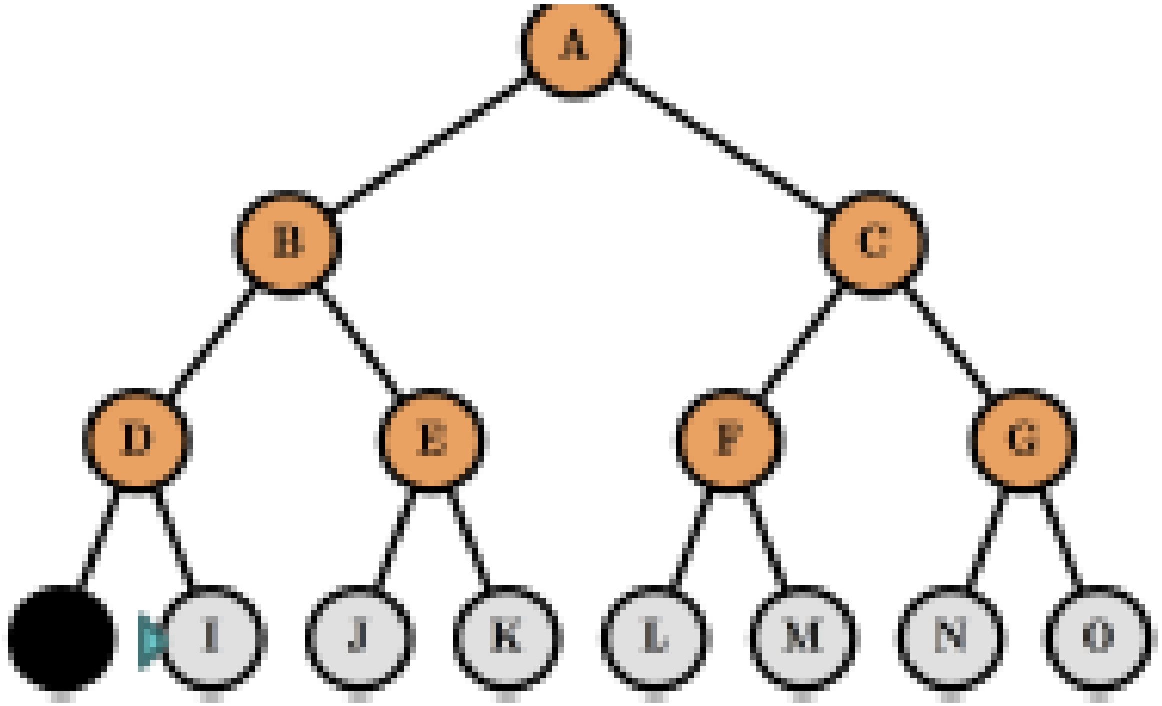

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O



A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
I
J
K
L
M
N
O

- The strategy is called breadth First Algorithm

BFS

- **Complete** Yes (if b is finite)
- **Time** $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$
- **Space** $O(b^d)$
  Note: If the *goal test* is applied at expansion rather than generation then $O(b^{d+1})$
- **Optimal** Yes (if cost $= 1$ per step).
- **implementation**: fringe: FIFO (Queue)

Question: If time and space complexities are exponential, why use BFS?

BFS PERFORMANCE ANALYSIS

- Assume that branching factor b is 10
- Computer can process One million node per second. Means 110 nodes in .11 milliseconds
- And to store a node it requires 1000 bytes. Means 107 kilobytes for 110 nodes.
- Construct a four column (depth, Nodes, time required and space requirement) up to depth of 16

TIME AND SPACE REQUIREMENT OF BFS

Depth	Nodes	Time	Space
2	110	.11 milliseconds	107 KB
4			
6			
8			
10			
12			
14			
16			

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

There are **10^{23}** in 8x8 chess board

Simple Game Tic Tac Toe Search Tree require 1 gigabyte space.

GO

19x19 Boad has complexity of $\mathbf{10^{361}}$

RECALL THE GRAPH ALGORITHM

```
function GRAPH-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    initialize frontier with initialState
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.remove()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.add(neighbor)

    return FAILURE
```

GRAPH ALGORITHM WITH PATH

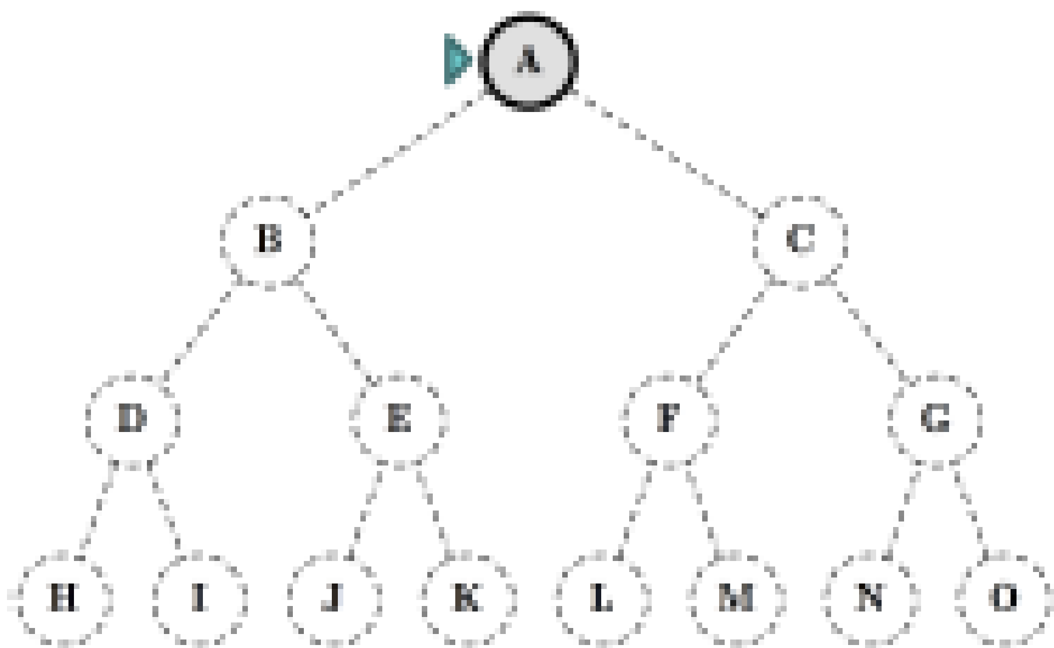
```python
from util import Queue
def breadthFirstSearch(problem):
    fringe=Queue()
    explored=set()

    startStateBlock = problem.getStartState()
    fringe.push((startStateBlock,[]));

    while(not fringe.isEmpty()):
        state = fringe.pop()
        stateBlock = state[0]
        statePath  = state[1].copy()
        explored.add(stateBlock)
        if(problem.isGoalState(stateBlock)):
            return statePath
        children=problem.getSuccessors(stateBlock)
```

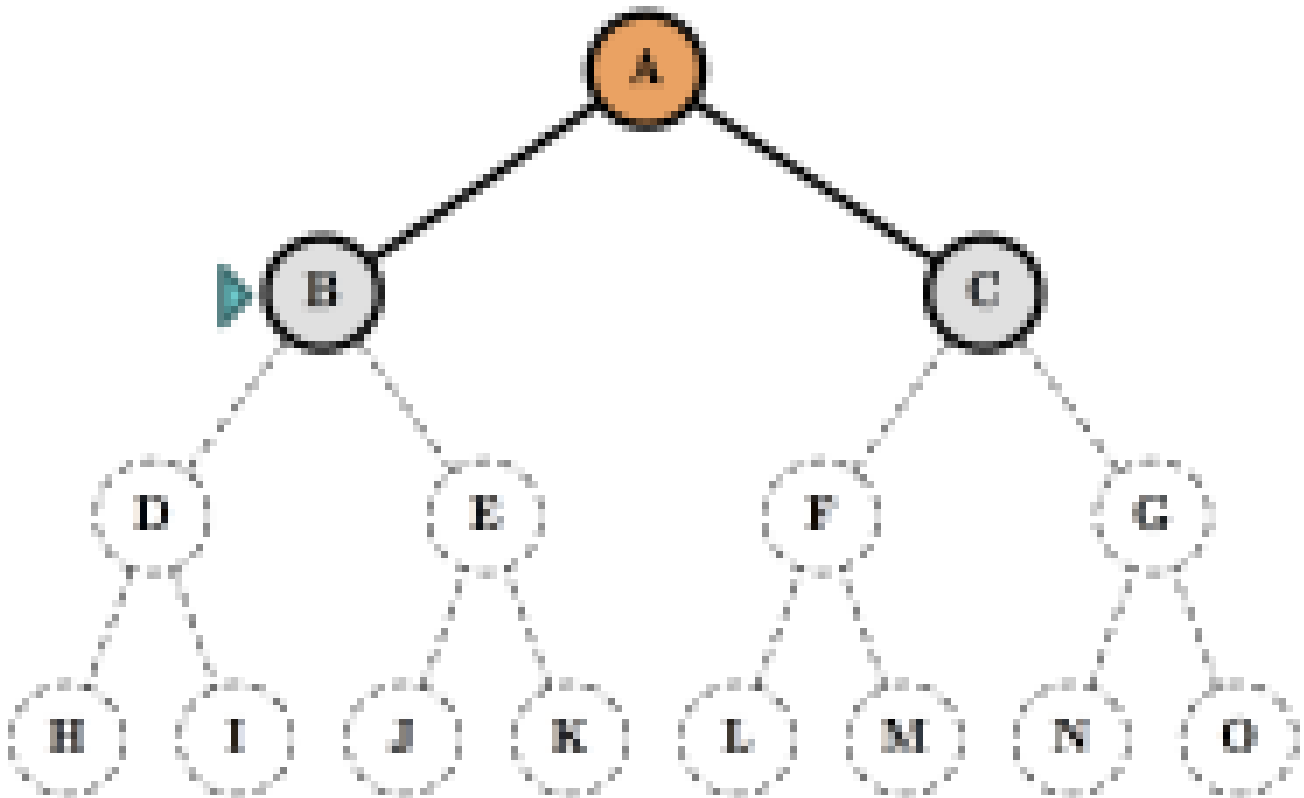
GRAPH ALGORITHM WITH PATH

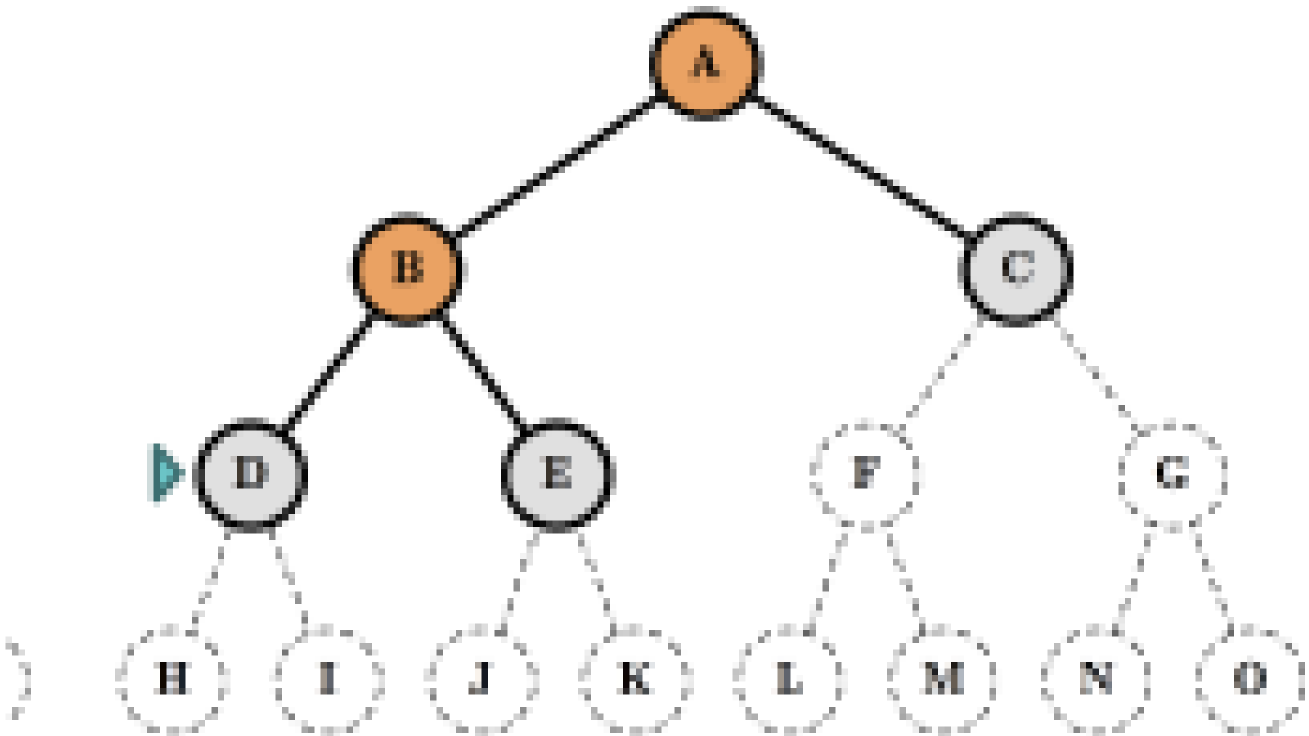
```
    for child in children:

        actionToReachChild=child[1]
        childPath = statePath.copy()
        childPath.append(actionToReachChild)

        openList= [x[0] for x in fringe.list if x[0]==child[0]]


        inProcess=child[0] in explored or child[0] in openList
        if(not inProcess):
            fringe.push((child[0],childPath)) #child[0] returning the state

return []
```

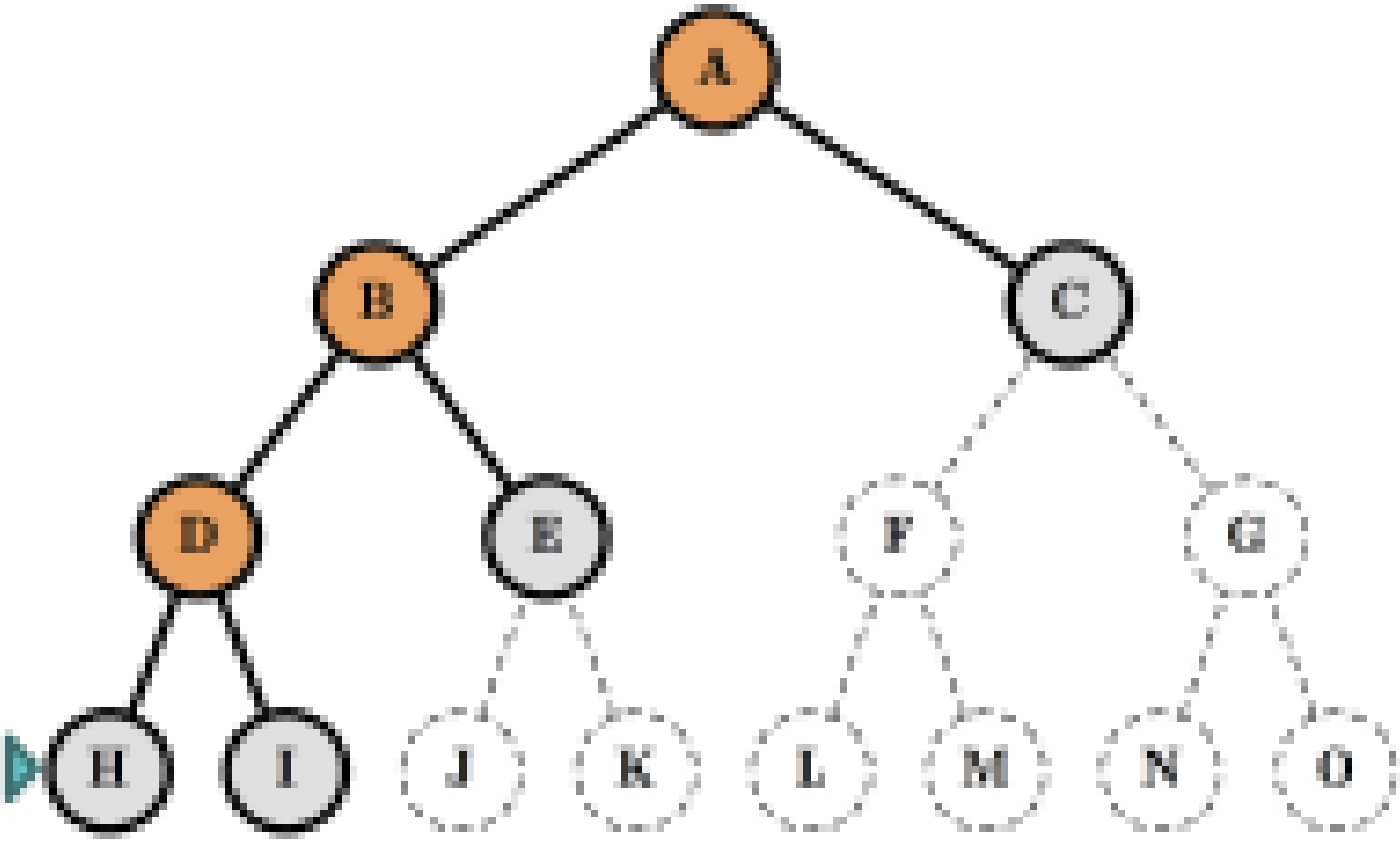
- What if we change it with the Stack

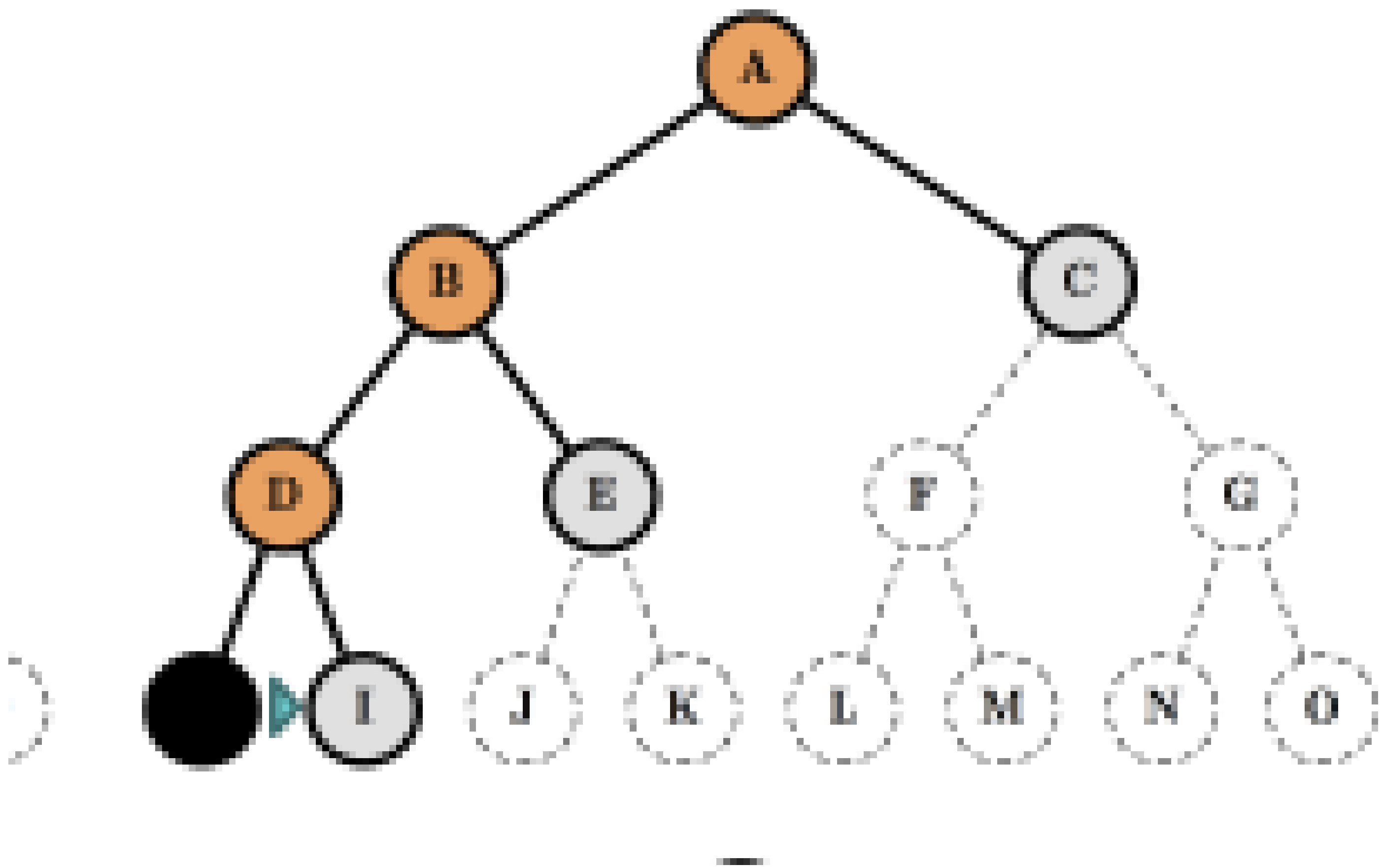
- Give the dry run on the code at following tree when fringe is **Stack**

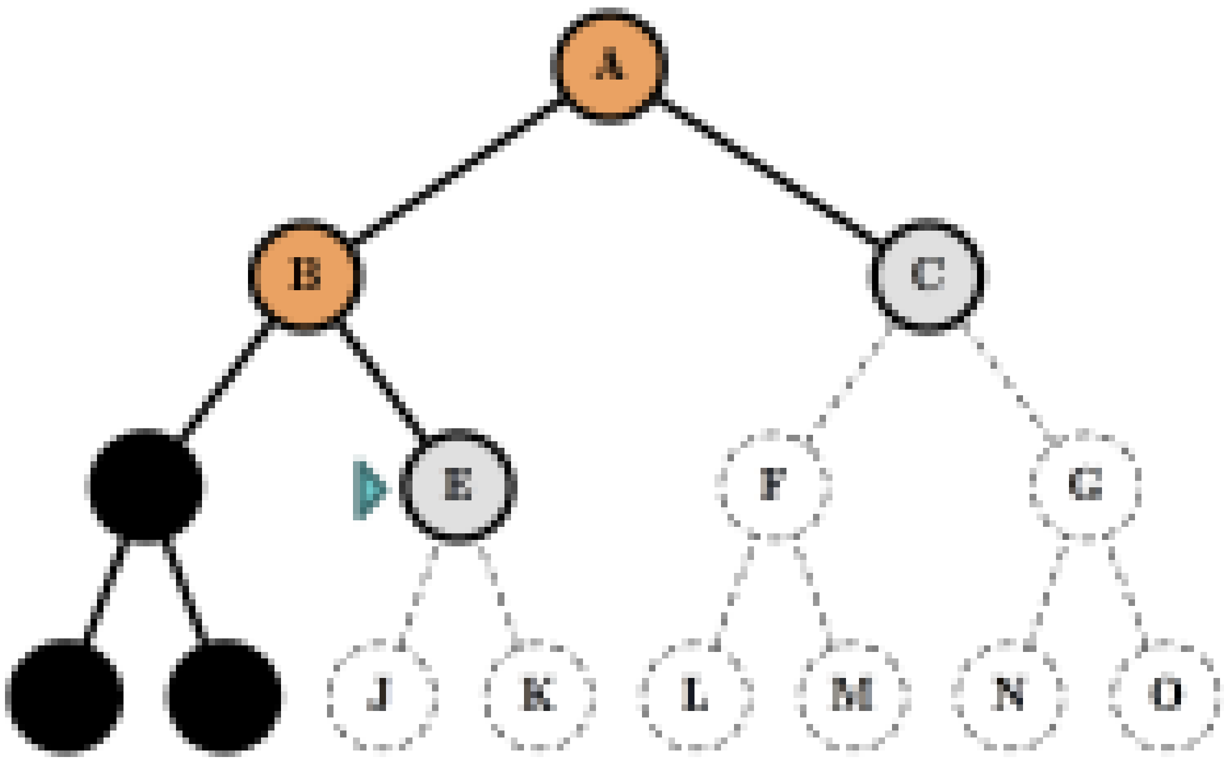


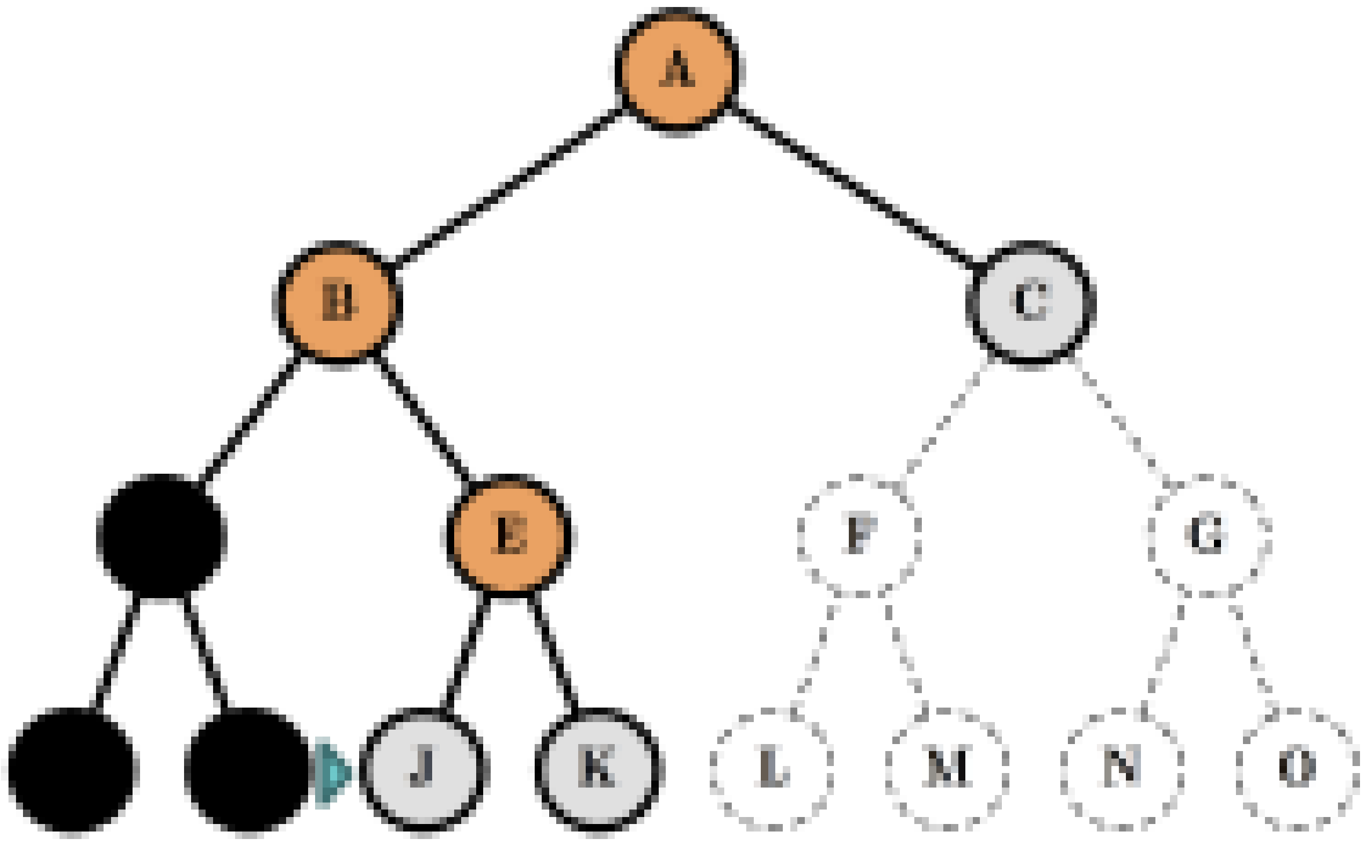

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

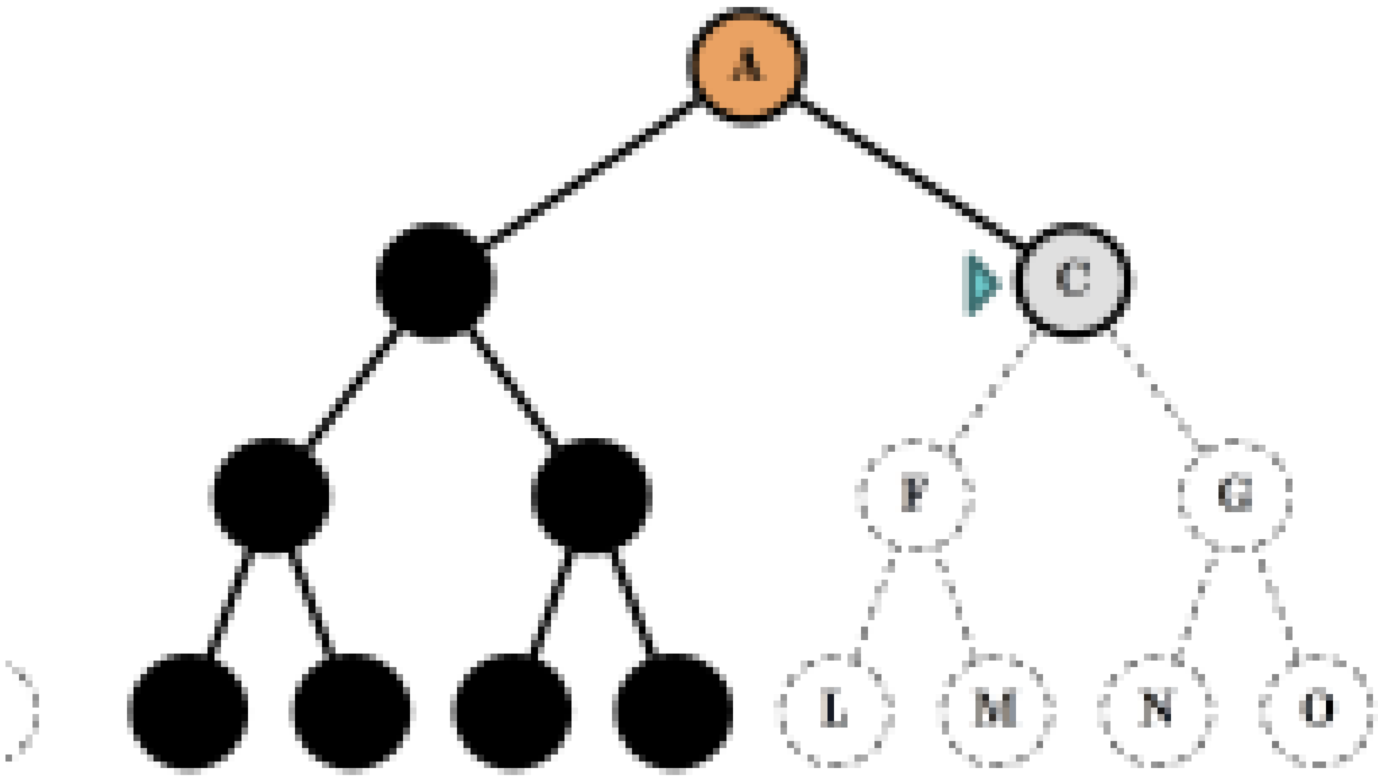

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
I
J
K
L
M
N
O


A
B
C
E
F
G
J
K
L
M
N
O


A
B
C
E
F
G
J
K
L
M
N
O


A
C
F
G
L
M
N
O

THIS SEARCH STRATEGY IS CALLED DEPTH FIRST SEARCH

(DFS)

ANALYSIS OF DFS

- **Complete** No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path.
  $\Rightarrow$ complete in finite spaces
- **Time** $O(b^m)$: $1 + b + b^2 + b^3 + \ldots + b^m = O(b^m)$
  bad if m is much larger than d
  but if solutions are dense, may be much faster than BFS.
- **Space** $O(bm)$ linear space complexity! (needs to store only a single path from the root to a leaf node, **along with the remaining unexpanded sibling nodes for each node on the path, hence the m factor.**)
- **Optimal** No
- **Implementation:** fringe: LIFO (Stack)

DFS PERFORMANCE

- Assume that branching factor b is 10
- Computer can process One million node per second. Means 110 nodes in .11 milliseconds
- And to store a node it requires 1000 bytes. Means 107 kilobytes for 110 nodes.
- Construct a four column (depth, Nodes, time required and space requirement) up to depth of 16

TIME AND SPACE REQUIREMENT OF BFS

Depth	Nodes	Time	Space
2			
4			
6			
8			
10			
12			
14			
16			

TIME AND SPACE COMPLEXITY

- We go down from 10 exabytes in BFS to . . . in DFS?
- We go down from 10 exabytes in BFS to 156 kilobytes in DFS!

UCS: UNIFORM COST SEARCH

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE : /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

HEURISTIC BASED SEARCH

WHERE TO GO

- Let we have following search problem. How many states we are required to explore before reaching at the goal state ?

	2	3
1	4	5
8	7	6

Start state

1	2	3
8		4
7	6	5

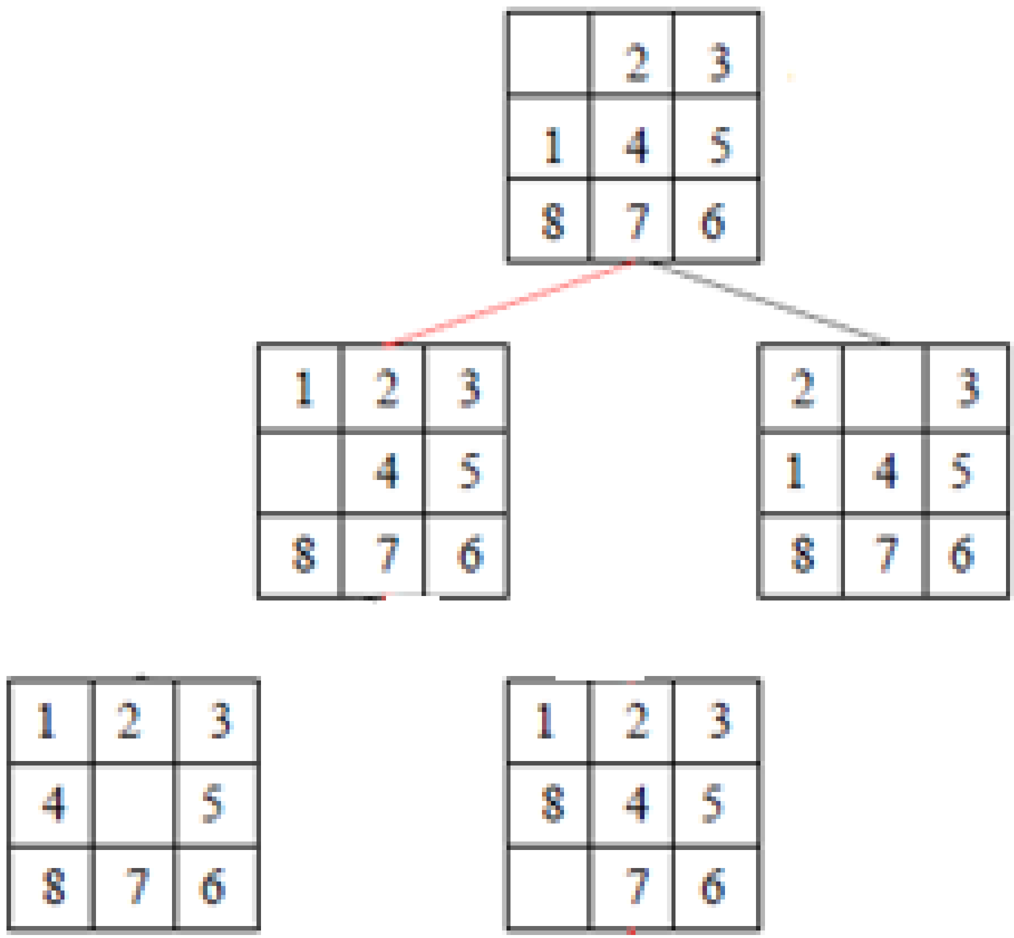
Goal state

WHICH SIDE WE SHOULD NEED TO MOVE:

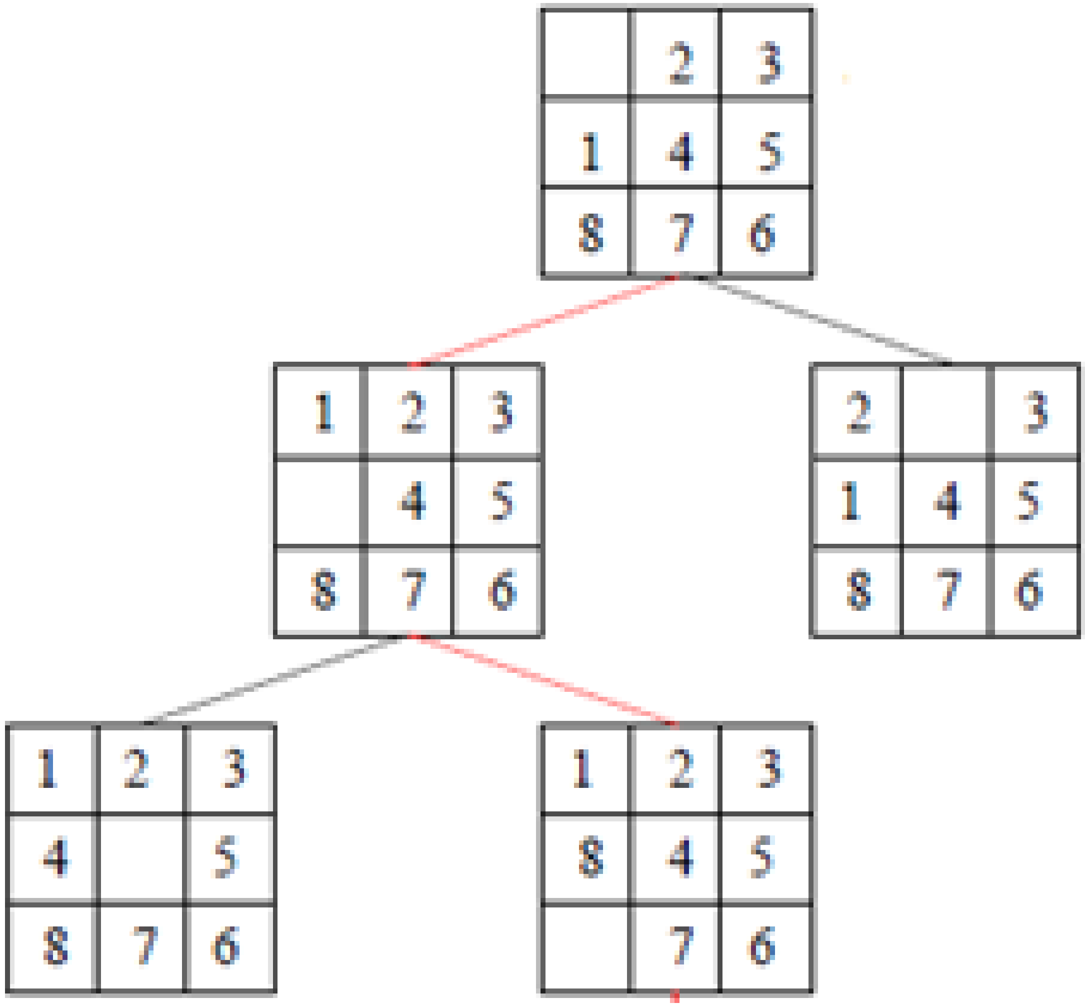
	2	3
1	4	5
8	7	6

1	2	3
	4	5
8	7	6

2		3
1	4	5
8	7	6



2 3
1 4 5
8 7 6
1 2 3
4 5
8 7 6
2 3
1 4 5
8 7 6
1 2 3
4 5
8 7 6
1 2 3
8 4 5
7 6


2 3
1 4 5
8 7 6
1 2 3
4 5
8 7 6
2 3
1 4 5
8 7 6
1 2 3
4 5
8 7 6
1 2 3
8 4 5
7 6

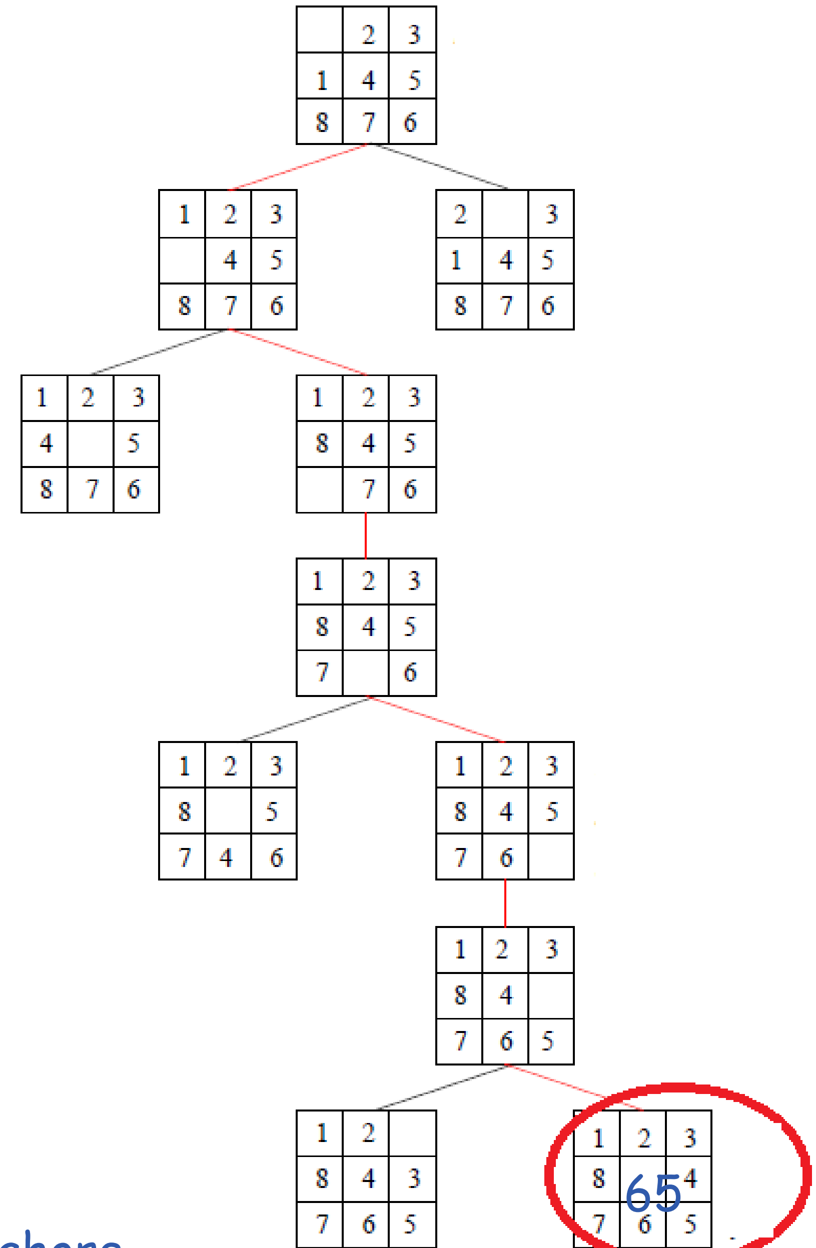
ONE POSSIBLE SOLUTION

	2	3
1	4	5
8	7	6

Start state

1	2	3
8		4
7	6	5

Goal state



How many state algorithm have to explore before reaching to goal state ?

Can we guide algorithm in which direction it should first move to find the goal ?

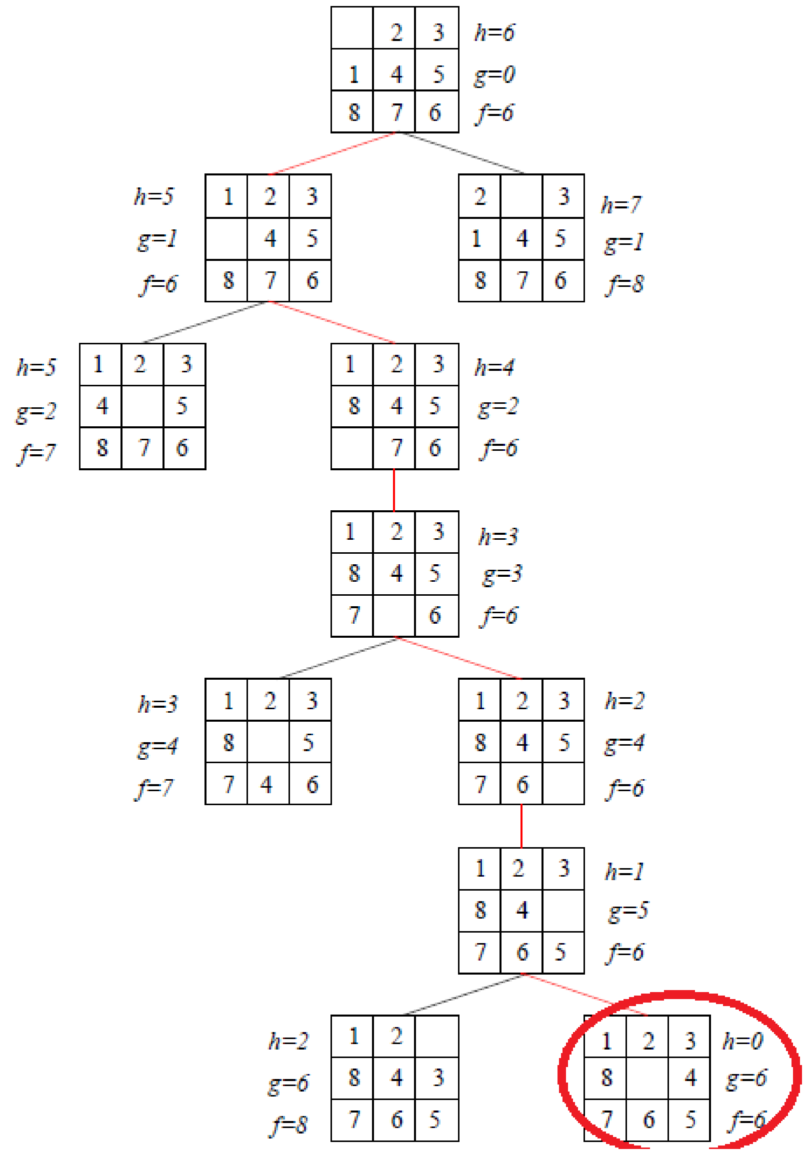
POSSIBLE HINTS

- The number of tiles in wrong position:

	2	3
1	4	5
8	7	6

Start state

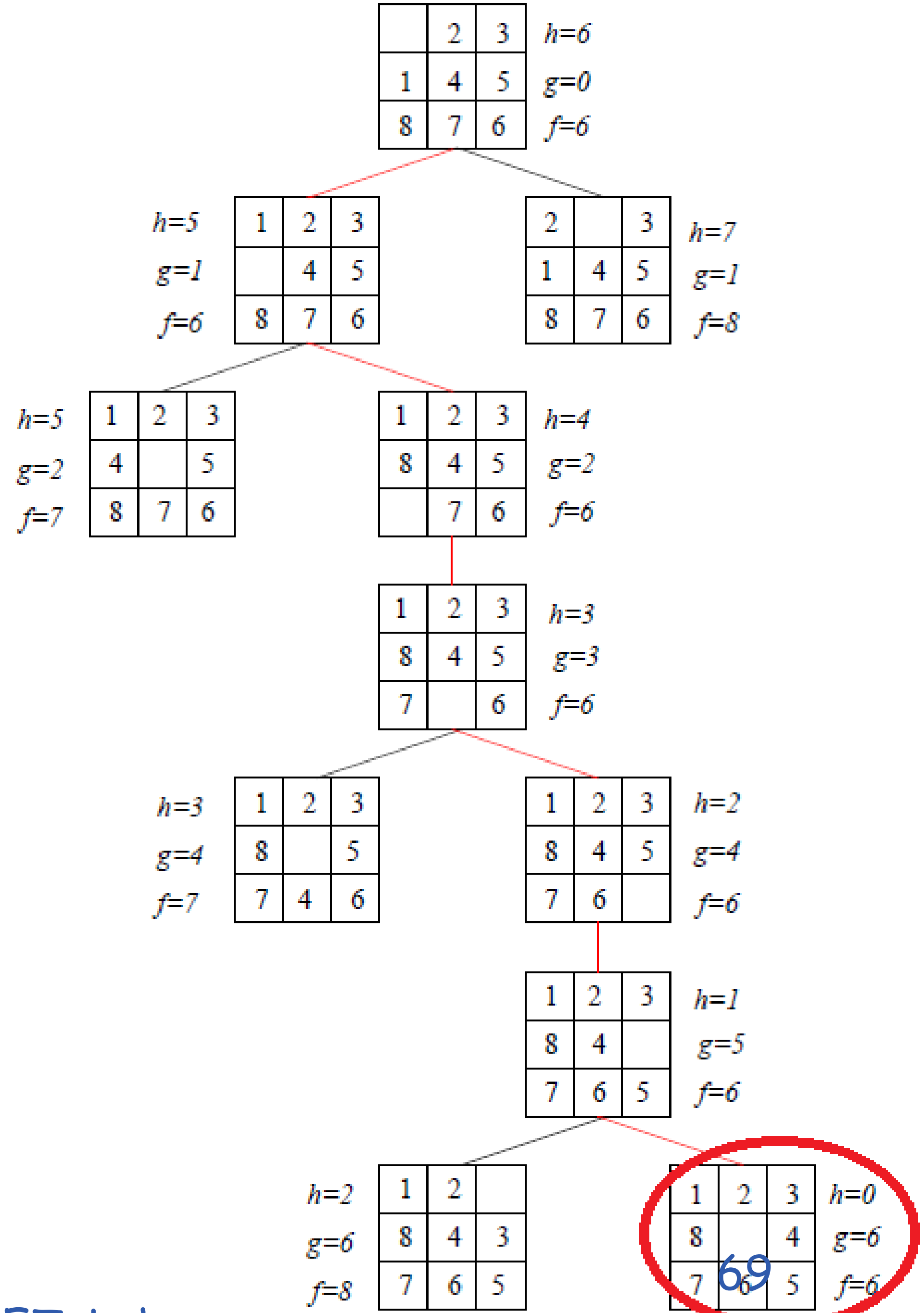
- h(start) = 6
- (Algorithm may prefer to search toward node that has less number of wrong tile in position)


h=6 g=0 f=6
h=5 g=1 f=6
h=7 g=1 f=8
h=5 g=2 f=7
h=4 g=2 f=6
h=3 g=3 f=6
h=3 g=4 f=7
h=2 g=4 f=6
h=1 g=5 f=6
h=2 g=6 f=8
h=0 g=6 f=6

A*- A SPECIAL BEST-FIRST SEARCH

- **Notation:**
  - c(n,n') - cost of arc (n,n')
  - g^(n) = cost of current path from start to node n in the search tree.
  - h^(n) = estimate of the cheapest cost of a path from n to a goal.
  - Special evaluation function: f = g+h
- **f(n) estimates the cheapest cost solution path that goes through n.**
  - h(n) is the true cheapest cost from n to a goal.
  - g(n) is the true shortest path from the start s, to n.

- Let' say we have some good heuristic which calculate the cost to reach goal than how can we implement the code ?



Dr. Muhammad Awais Hassan
Department of Computer Science UET, Lahore

- Greedy Search (move to the direction of less $h(n)$)
- A* Search considers both cost to reach the node from start ($g(n)$) and estimated cost to reach to goal from this node ($h(n)$).
- $\text{Cost} = g(n) + h(n)$

TO IMPLEMENT THIS STRATEGY, WHAT CHANGES YOU SHALL NEED TO MAKE IN ALGORITHM ?

UCS: UNIFORM COST SEARCH

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE : /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

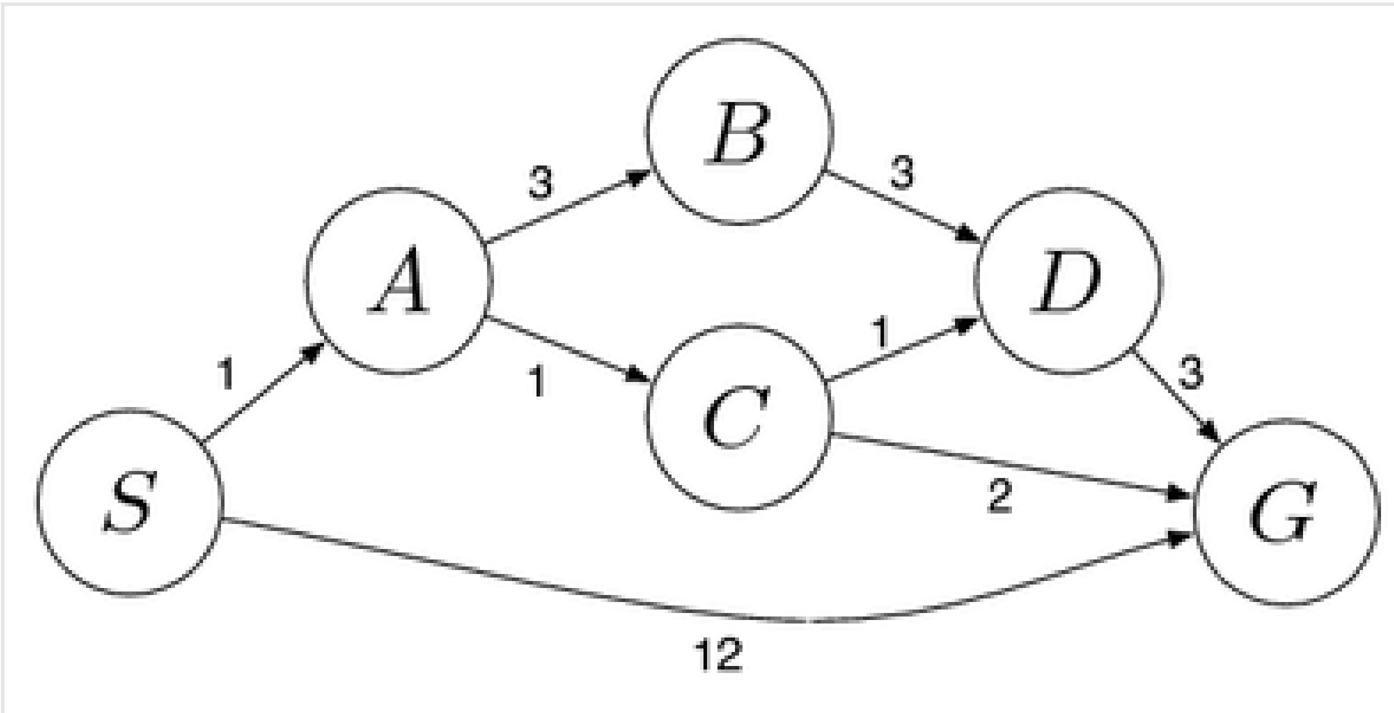
Effectiveness of A* Search Algorithm

Average number of nodes expanded

d	IDS	A*(h1)	A*(h2)
2	10	6	6
4	112	13	12
8	6384	39	25
12	364404	227	73
14	3473941	539	113
20	------------	7276	676

Average over 100 randomly generated 8-puzzle problems
h1 = number of tiles in the wrong position
h2 = sum of Manhattan distances

DOES HEURISTIC WORK



ITERATIONS BASED ON UCS

Initialization: { [S , 0] }

Iteration1: { [S->A , 1] , [S->G , 12] }

Iteration2: { [S->A->C , 2] , [S->A->B , 4] , [S->G , 12] }

Iteration3: { [S->A->C->D , 3] , [S->A->B , 4] , [S->A->C->G , 4] , [S->G , 12] }

Iteration4: { [S->A->B , 4] , [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->G , 12] }

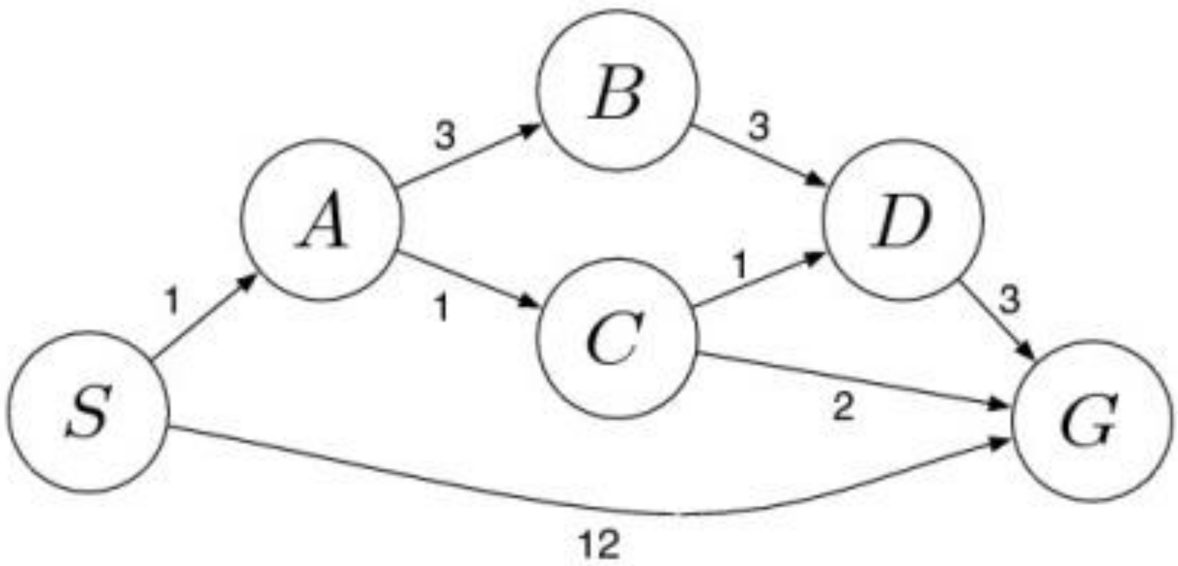
Iteration5: { [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->A->B->D , 7] , [S->G , 12] }

Iteration6 gives the final output as S->A->C->G.

- The UCS return optimal path but its search in all directions to find the optimal solution.
- This is called blind search or uninformed search.

- Let's say, we have some information about the search. Such as guess distance from any node to goal state.
- Previously in UCS, we have cost to reach at node n from start state.

Let h1 is guess distance of a node from Goal state. Can this information help us to perform some kind of informed search.



State	h1	h2
S	4	3
A	2	2
B	6	5
C	2	1
D	3	2
G	0	0

DRY RUN (A*)

Initialization: { [S , 4] }

Iteration1: { [S->A , 3] , [S->G , 12] }

Iteration2: { [S->A->C , 4] , [S->A->B , 10] , [S->G , 12] }

Iteration3: { [S->A->C->G , 4] , [S->A->C->D , 6] , [S->A->B , 10] , [S->G , 12] }

Iteration4 gives the final output as S->A->C->G.

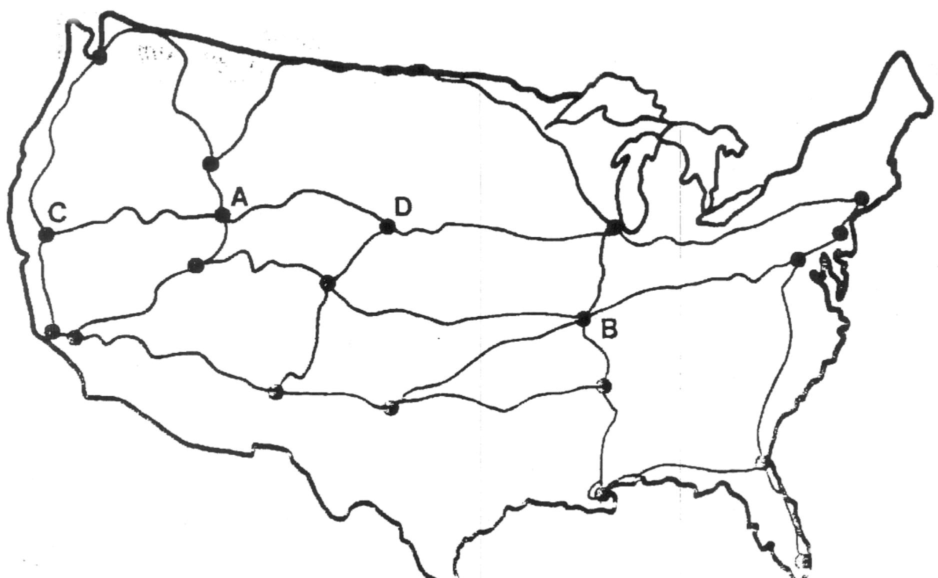
- What the advantage you are seeing ?
- Comparative less number of node exploration.

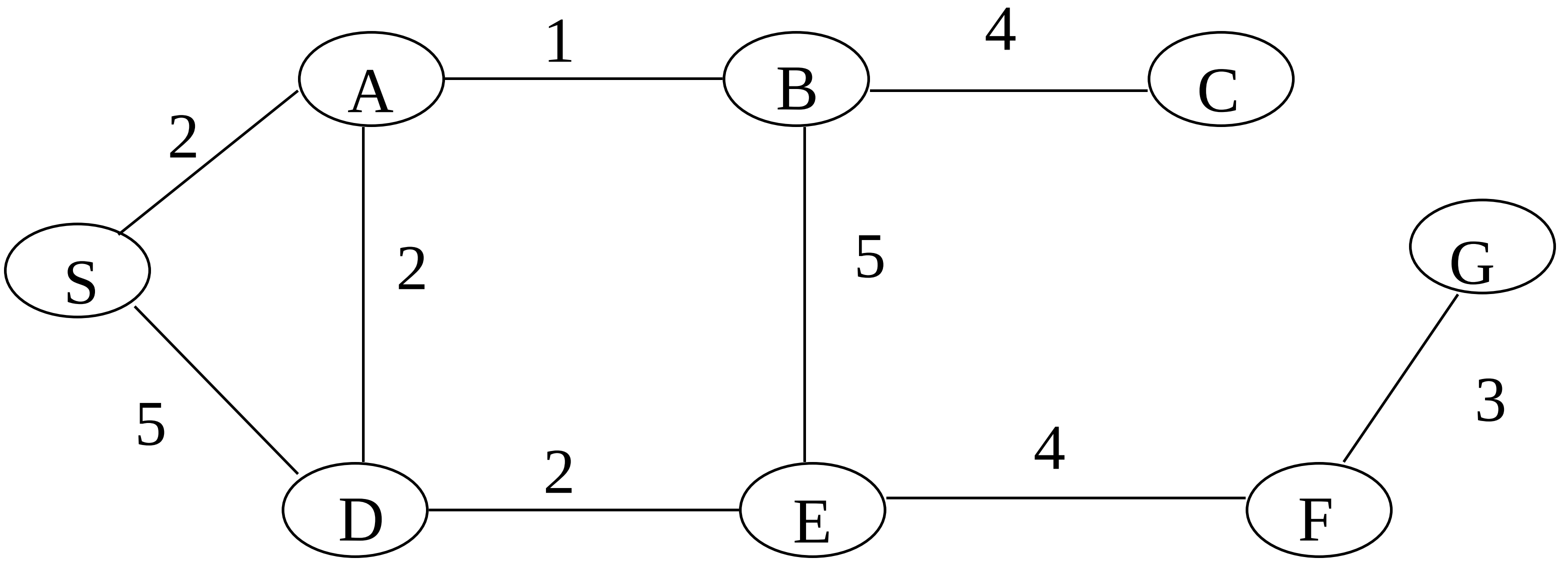
THE ROAD-MAP

- Find shortest path between city *A* and B
- Possible Heuristic

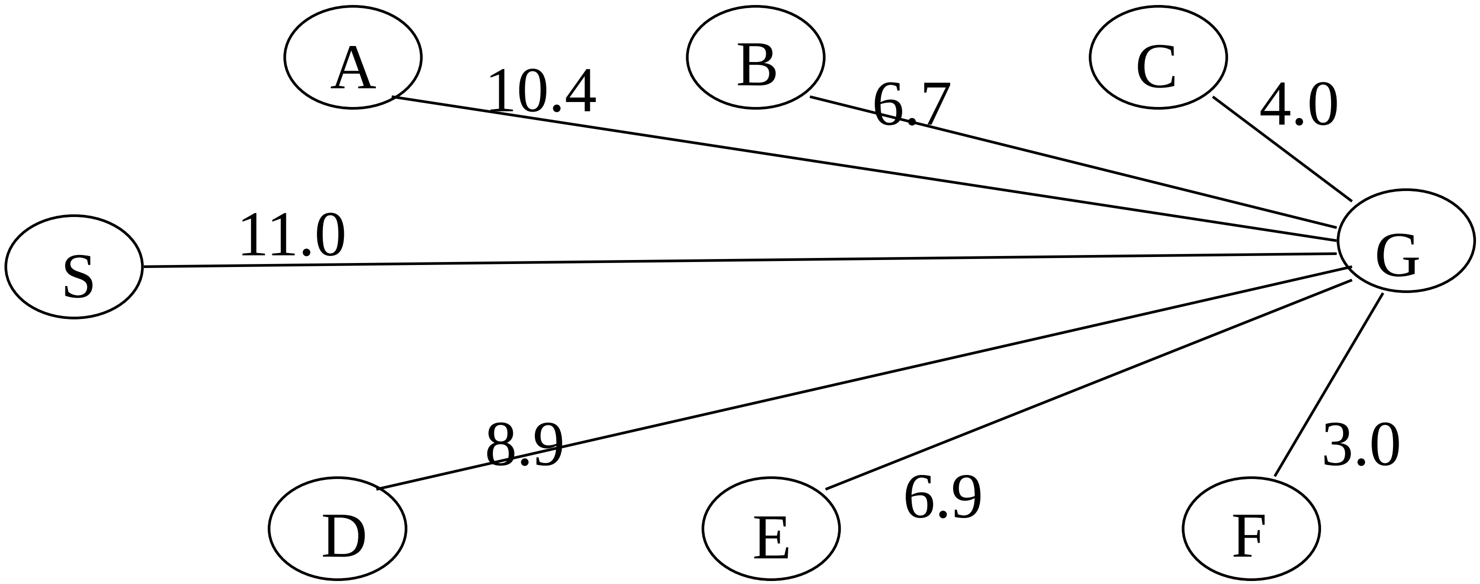
$$h(i) \equiv \text{air distance from city } i \text{ to } B$$

$$\underbrace{d(A,D)+h(D)}, \; \underbrace{d(A,D)+h(C)}$$



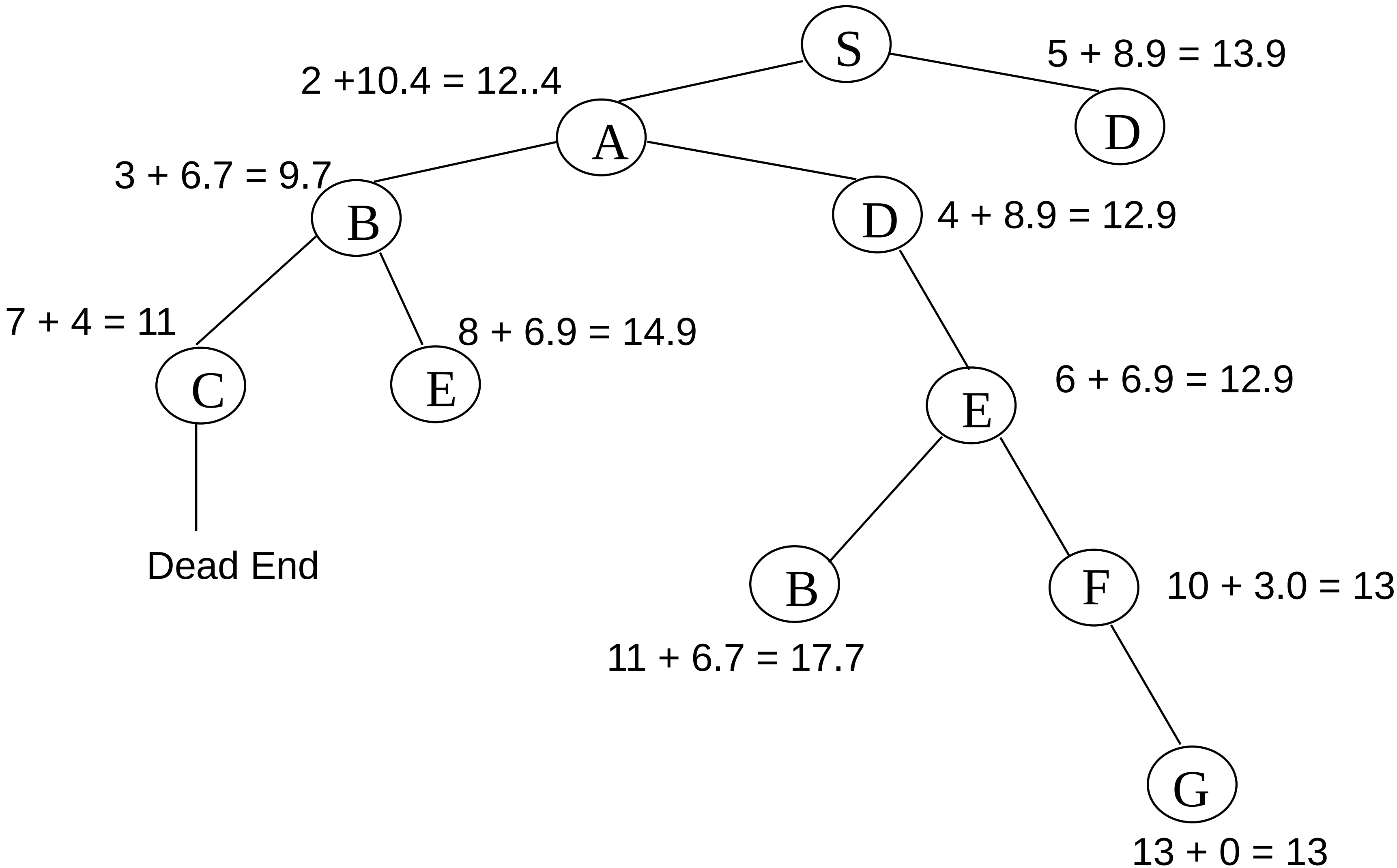


1
4
A
B
C
2
S
2
5
G
3
5
4
2
D
E
F

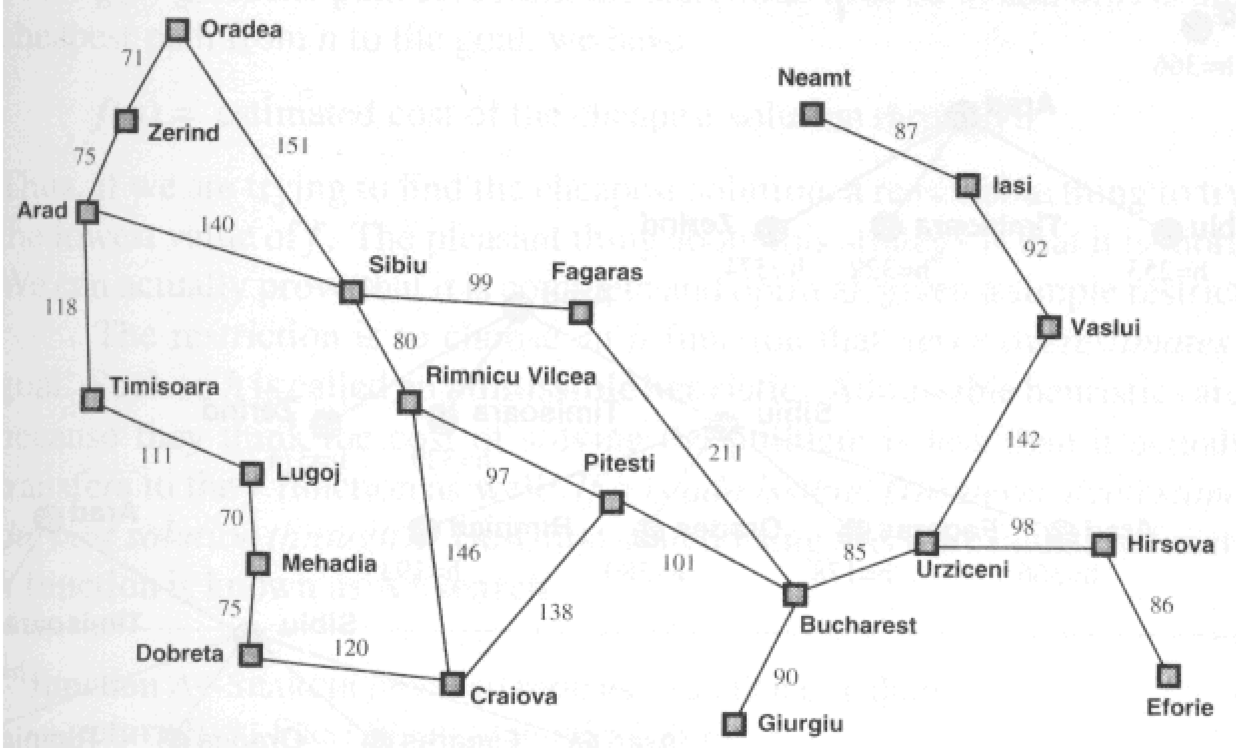


A
B
C
10.4
6.7
4.0
11.0
S
G
8.9
3.0
6.9
D
E
F

EXAMPLE OF A* ALGORITHM IN ACTION



FIND PATH FROM ARAD TO BUCHAREST



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Figure 4.2 Map of Romania with road distances in km, and straight-line distances to Bucharest.

ASSIGNMENT 03

- Implement PACMEN with BFS, DFS and Heuristic to Reach the Goal

CREDITS

- https://www.ics.uci.edu/~dechter/courses/ics-270a/winter-03/lecture-notes/4-class-notes.ppt
- http://web.mat.bham.ac.uk/S.Z.Nemeth/ho-notes.pdf