2. Problem Solving

Artificial Intelligence and Neural Network (AINN)

Part II

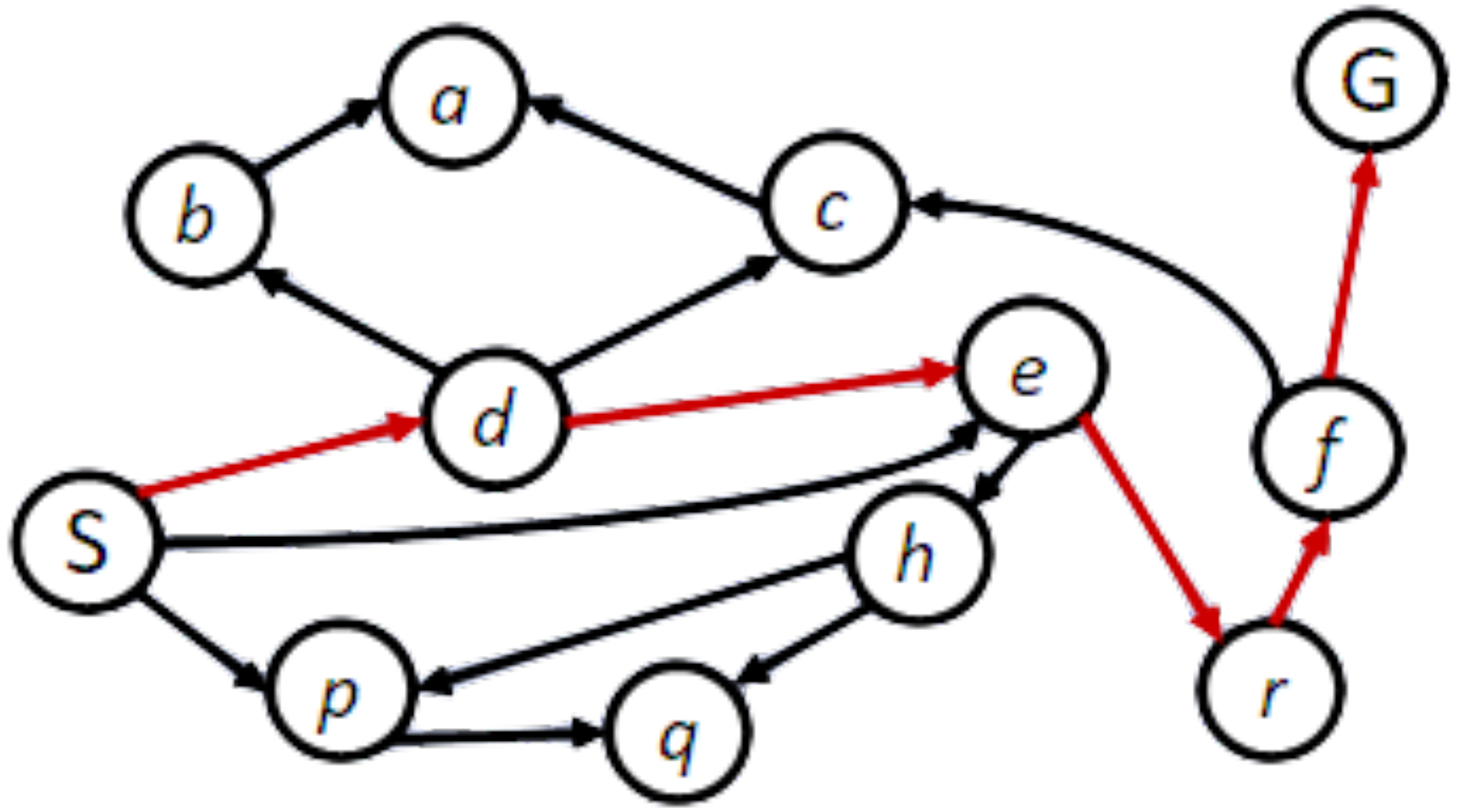
Dr. Udaya Raj Dhungana
Assist. Professor
Pokhara University, Nepal
Guest Faculty
Hochschule Darmstadt University of Applied Sciences, Germany
E-mail: udaya@pu.edu.np and udayas.epost@gmail.com

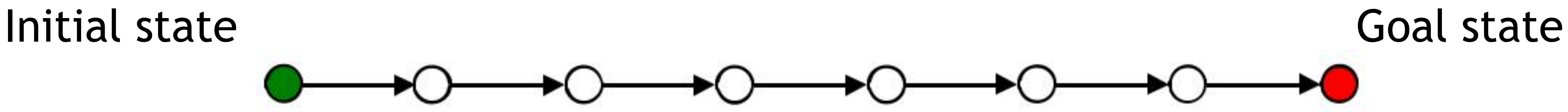
Overview

- Solving Problems by Searching
- Example Problems
- Searching for Solutions
- Uninformed Search Strategies
- Informed Search Strategies

Solving Problem by Searching

- A wide range of problems can be formulated as *searches*
  – as the process of searching for a sequence of actions that take you from an initial state to a goal state





Solving Problem by Searching

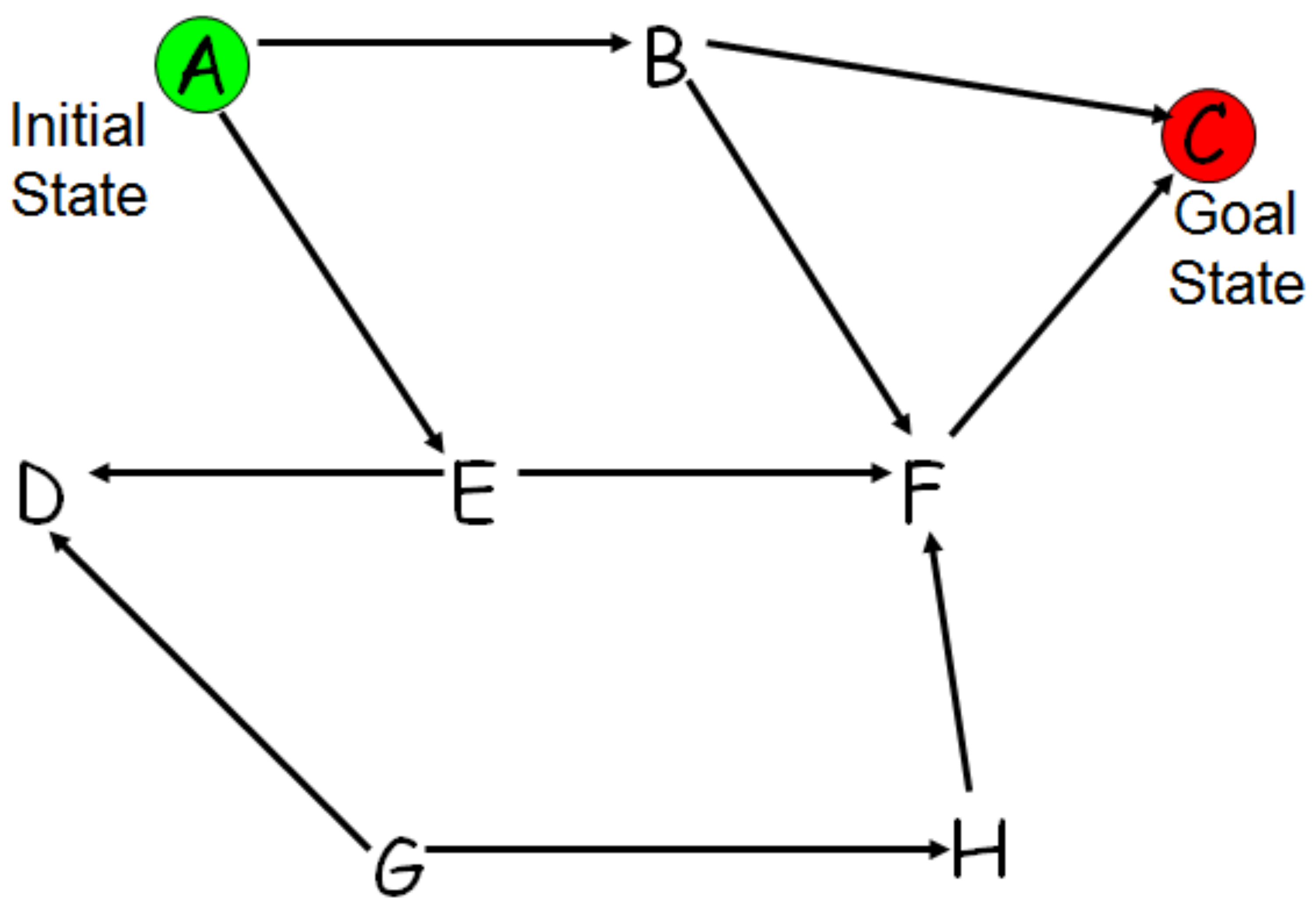
- State-Space Search
  - The ***states*** might be
    - legal board configurations in a game,
    - towns and cities in some sort of route map,
    - collections of mathematical propositions, etc.
  - The ***state-space*** is
    - the configuration of the possible states and how they connect to each other e.g. the legal moves between states.
  - Need to search the state-space to
    - find an optimal path from a start state to a goal state.
  - Example
    - Chess

Solving Problem by Searching

- State-Space Search

```
link(g,h).
link(g,d).
link(e,d).
link(h,f).
link(e,f).
link(a,e).
link(a,b).
link(b,f).
link(b,c).
link(f,c).
```

State-Space




```
go(X,X,[X]).
go(X,Y,[X|T]):-
            link(X,Z),
            go(Z,Y,T).
```

Simple search algorithm

```
| ?- go(a,c,X).
X = [a,e,f,c] ? ;
X = [a,b,f,c] ? ;
X = [a,b,c] ? ;
no
```

Consultation

Check in prolog

Searching for Solutions

- A Search Problem is defined by
  - a *state space*
    - *(i.e., an initial state or set of initial states and a set of* operators)
  - a *set of goal states*

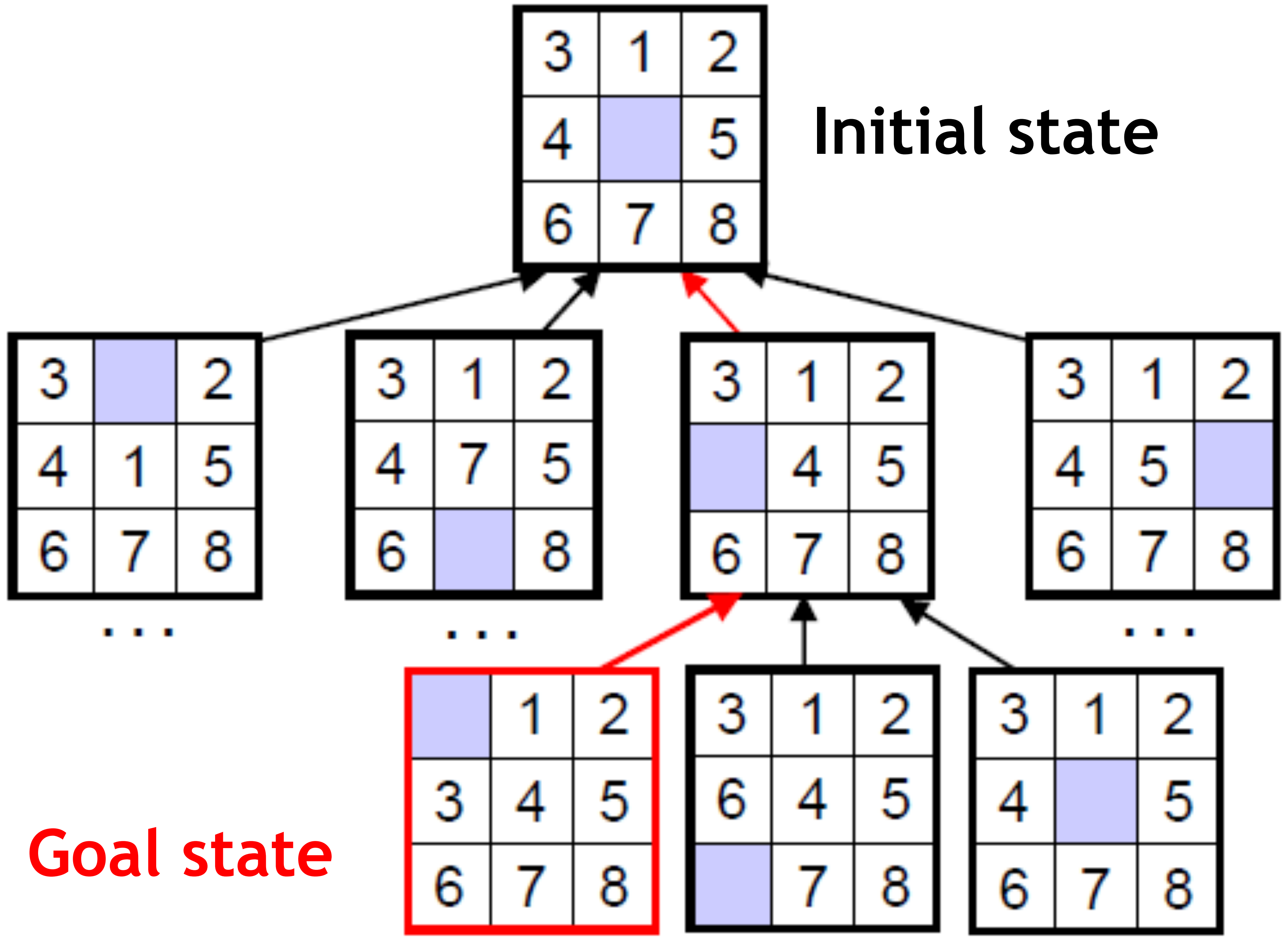
- A solution
  - is a path in the state space from an initial state to a goal state

Searching for Solutions

- *Exploring the state space*
  - *Search is the process of exploring the state space to find a solution*
  - exploration starts from the **initial state**
  - the search procedure applies operators to the initial state to generate one or more new states which are hopefully nearer to a solution
  - the search procedure is then applied recursively to the newly generated states
  - the procedure terminates when either a solution is found, or no operators can be applied to any of the current states

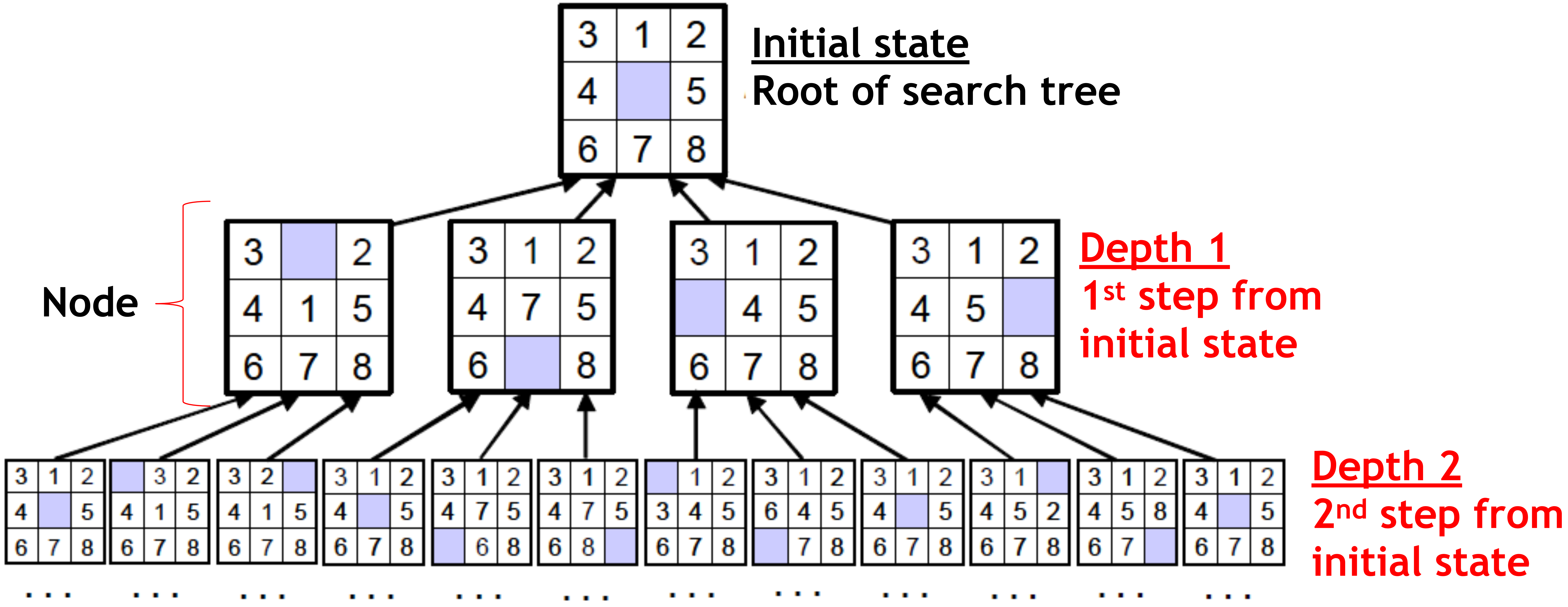
Searching for Solutions

- Search Tree
  - The possible action sequences starting at the initial state form a **search tree**



Searching for Solutions

- Search Tree
  - the part of the state space that has been explored by a search procedure can be represented as a *search tree*



Searching for Solutions

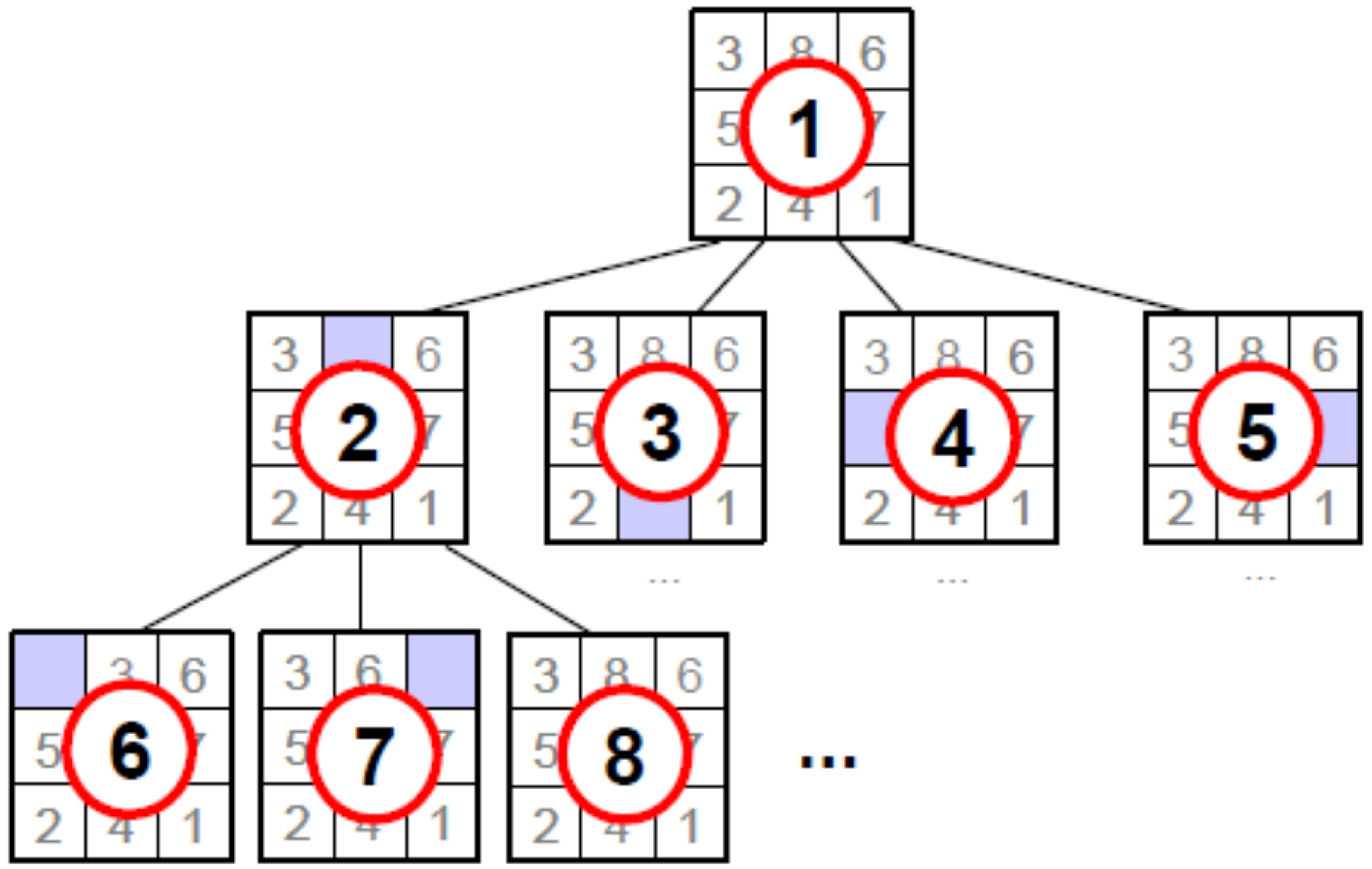
- the process of generating the children of a node by applying operators is called ***expanding*** *the node*

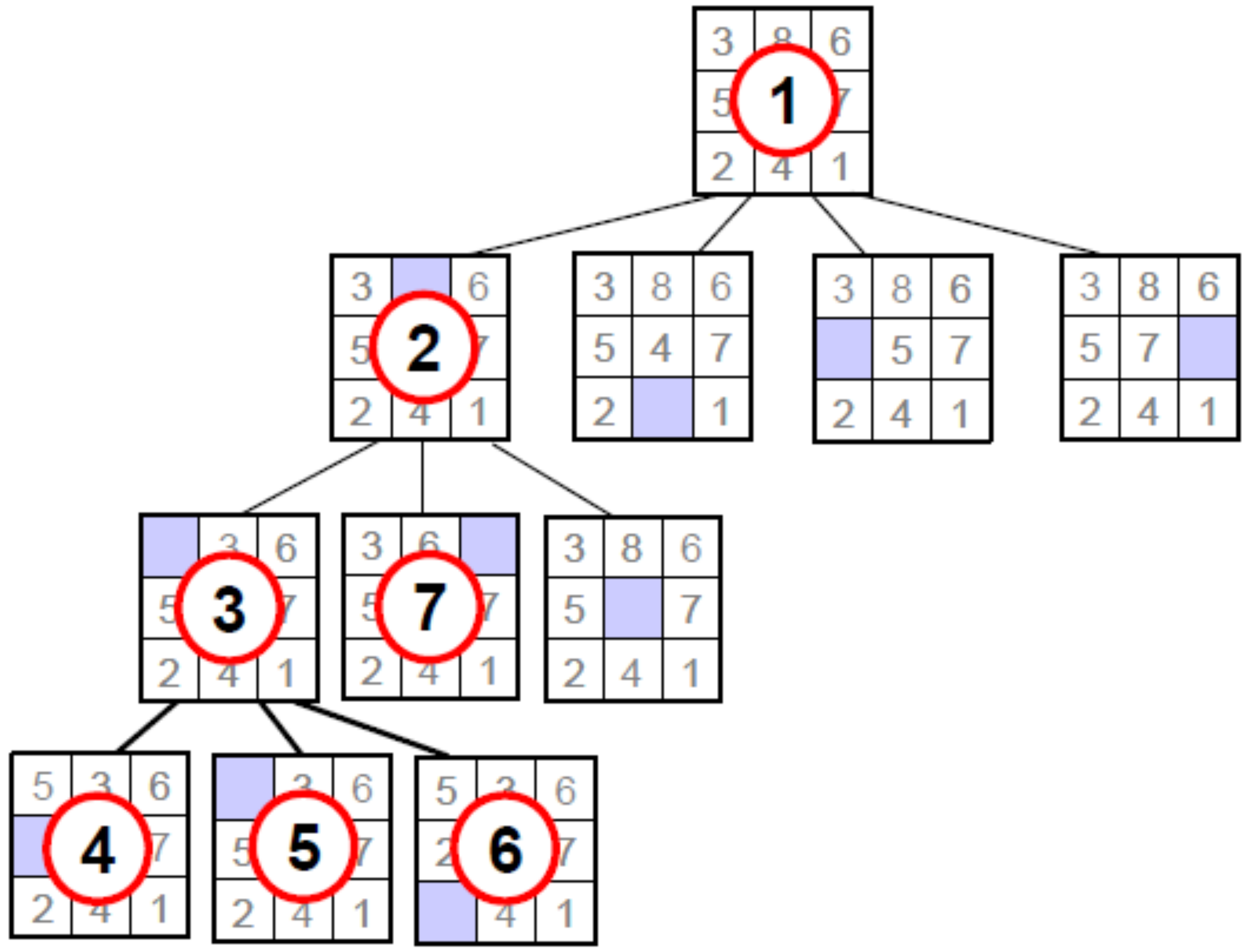
- the **branching factor** of a search tree is the average number of children of each non-leaf node

- if the **branching factor** is b, ***the number of nodes*** *at* ***depth*** d *is* b^d

Searching for Solutions

- Search strategies
  - They vary primarily according to how they choose which state (node) to expand next in the search tree
  - Examples:





Searching for Solutions

- Search strategies
  - are evaluated along the following dimensions:
    - completeness: does it always find a solution if one exists?
    - time complexity: number of nodes generated
    - space complexity: maximum number of nodes in memory
    - optimality: does it always find a least-cost solution?

  - Time and space complexity are measured in terms of
    - *b*: maximum branching factor of the search tree
    - *d*: depth of the least-cost solution
    - *m*: maximum depth of the state space (may be ∞)

Uninformed Search Strategies

- **Uninformed** search strategies
  - use only the information available in the problem definition
  - Also known as **blind search**

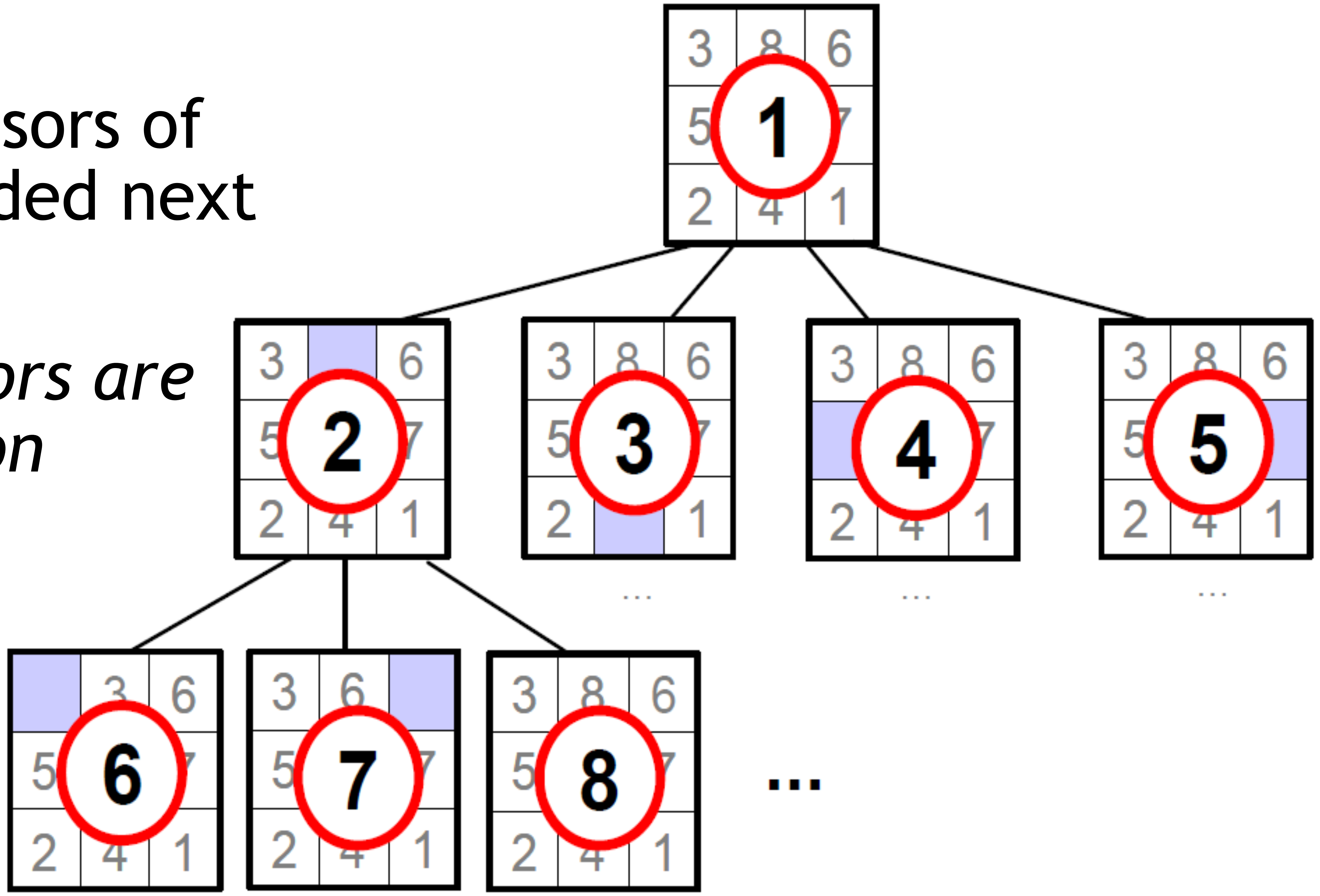
- Some uninformed strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

Breadth First Search

1. Root node is expanded first

3. Then all the successors of the root are expanded next

5. Then *their successors are expanded, and so on*



Breadth First Search

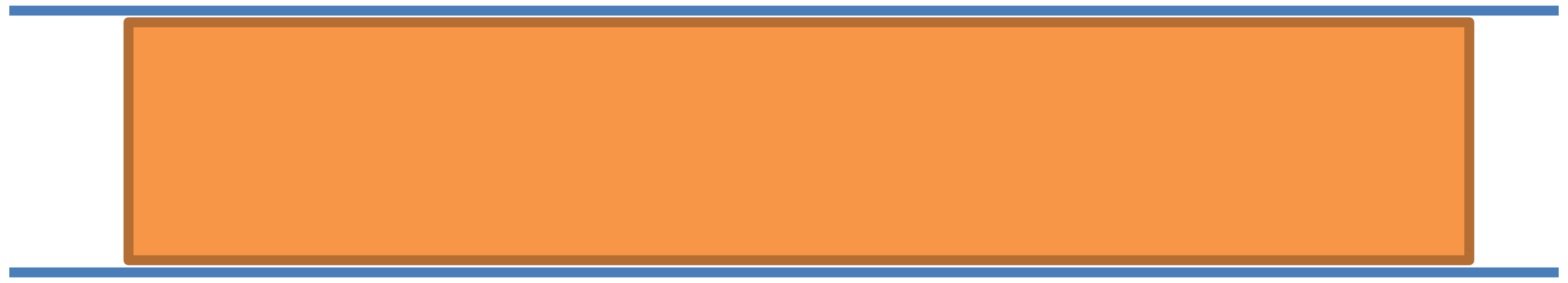
- In BFS, the shallowest unexpanded node is chosen for next expansion next

- This is achieved very simply by using a FIFO **queue**, i.e., new successors go at end

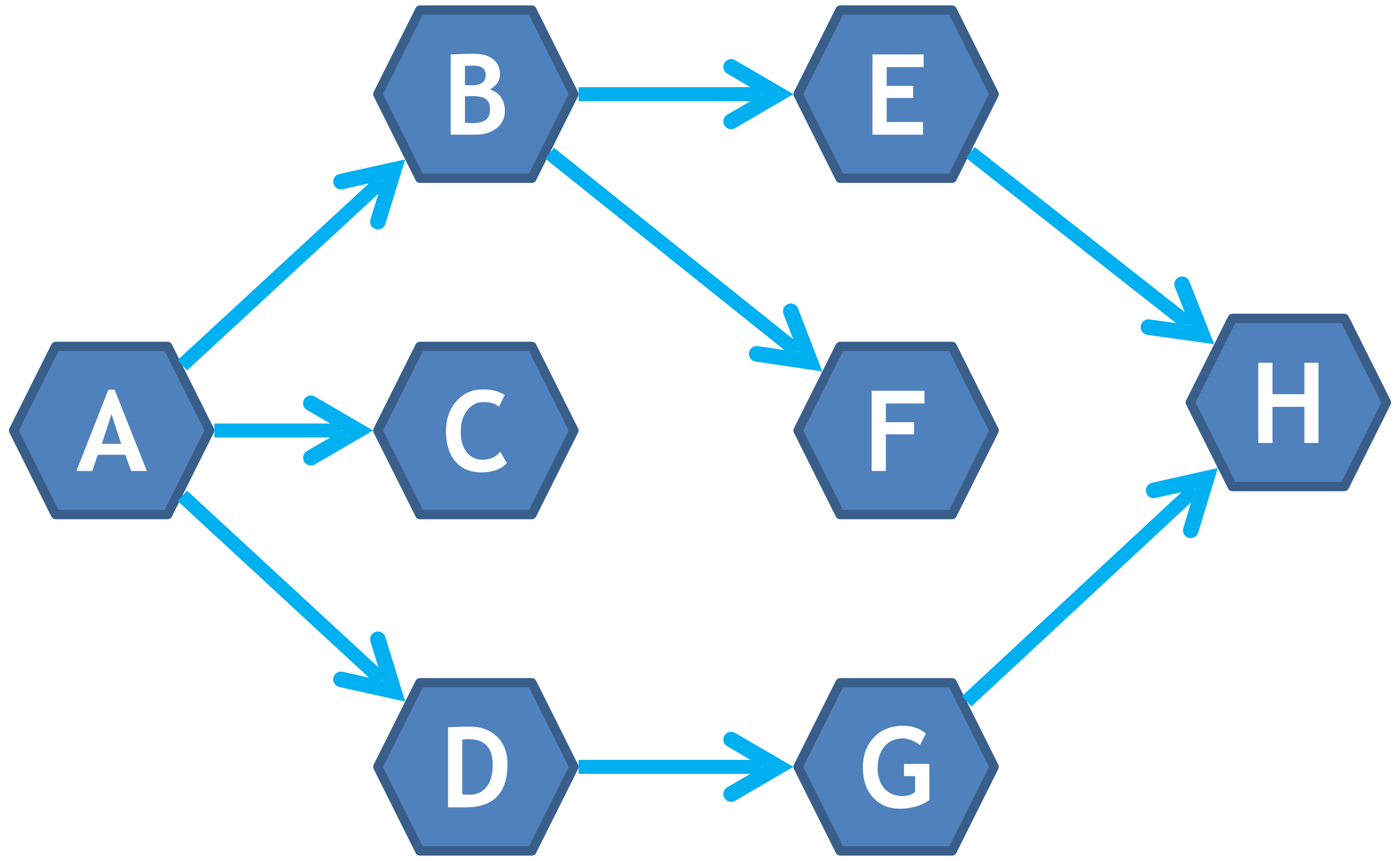
- BFS uses a Queue to remember to get the next node to start a search when a dead end occurs in any iteration

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as **VISITED**. Display it. Insert it in a queue.
2. If no adjacent node is found, **de-queue** the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

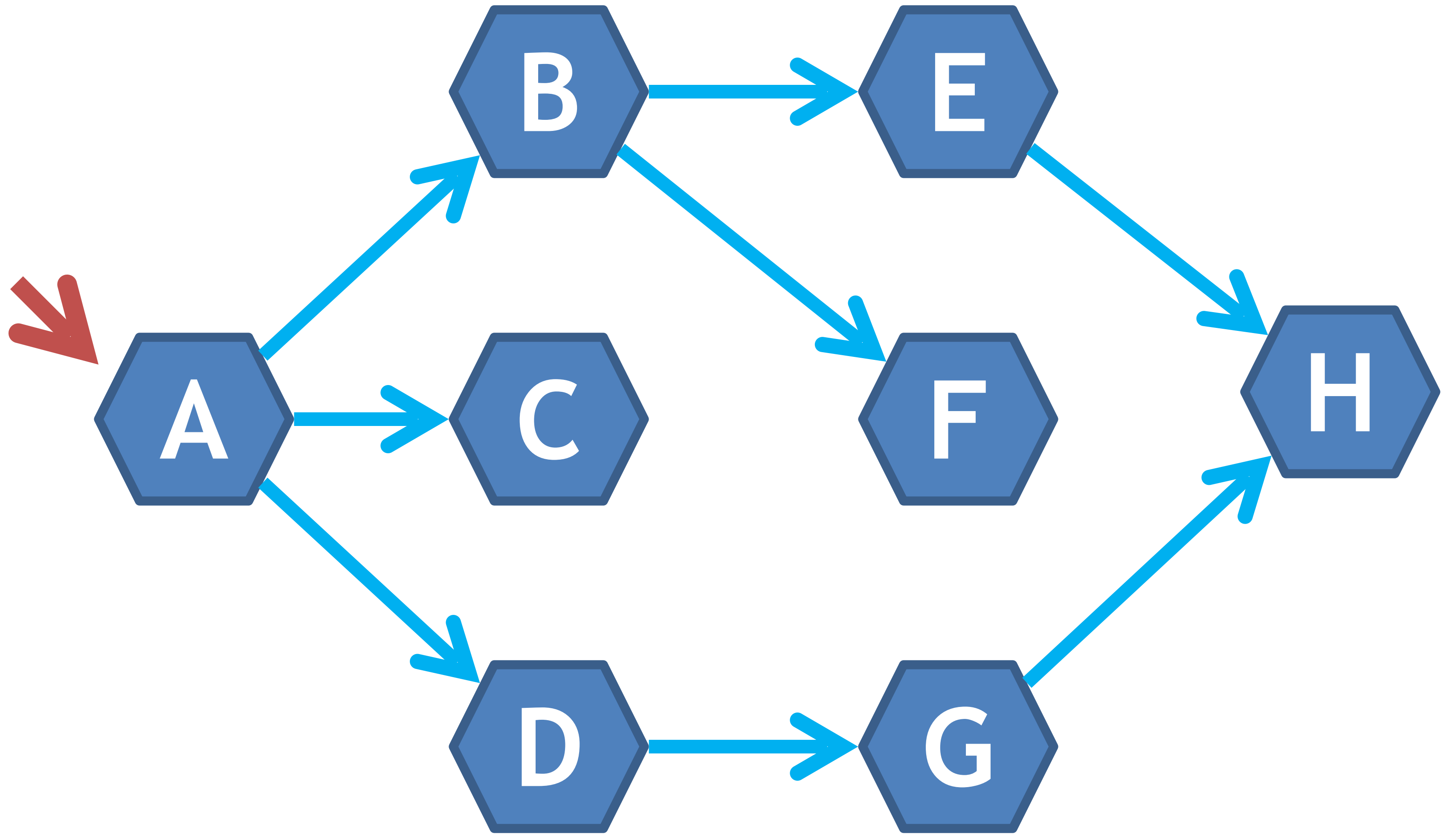


Rule applied

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

Rule applied

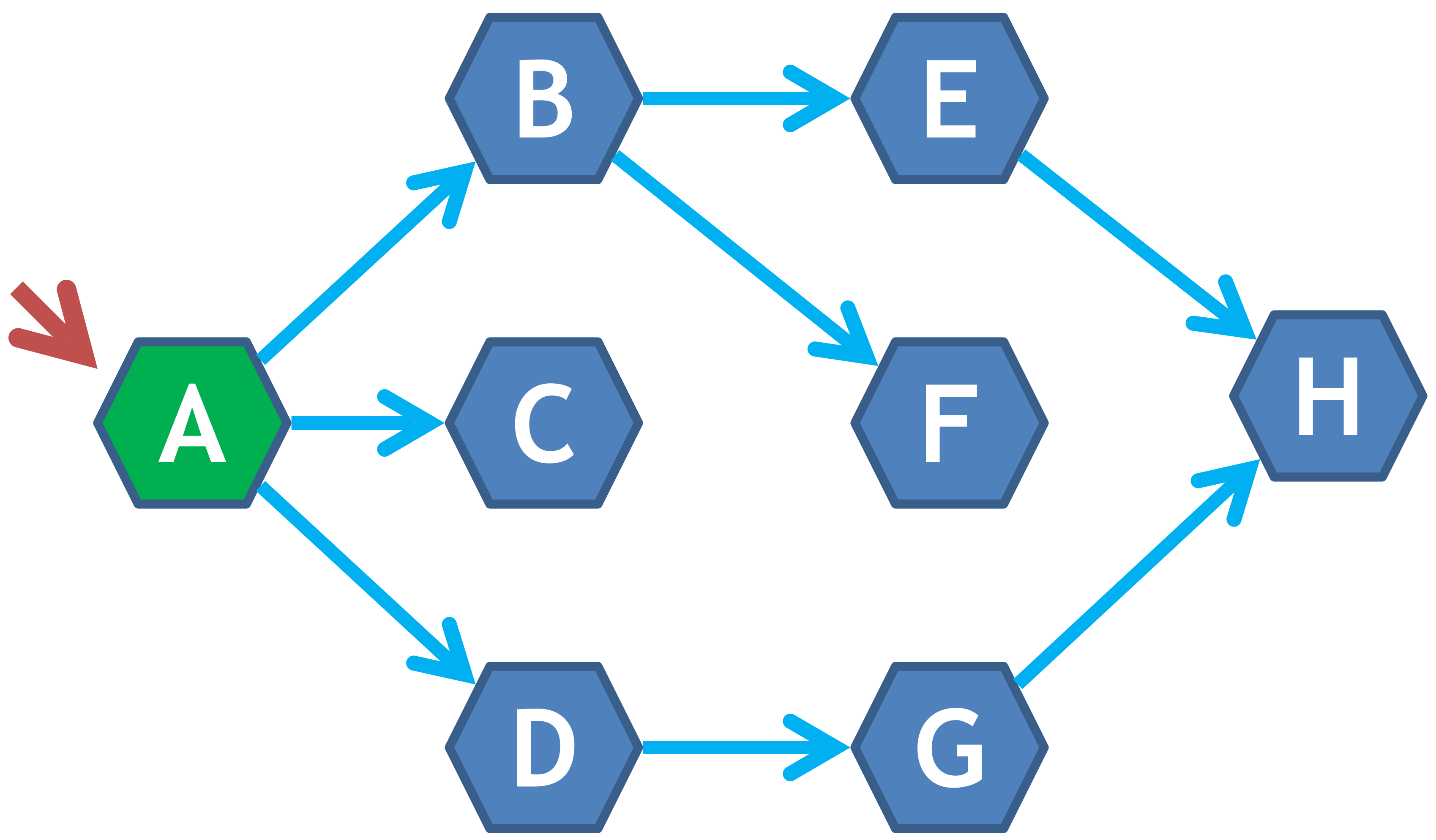
Start with Node A

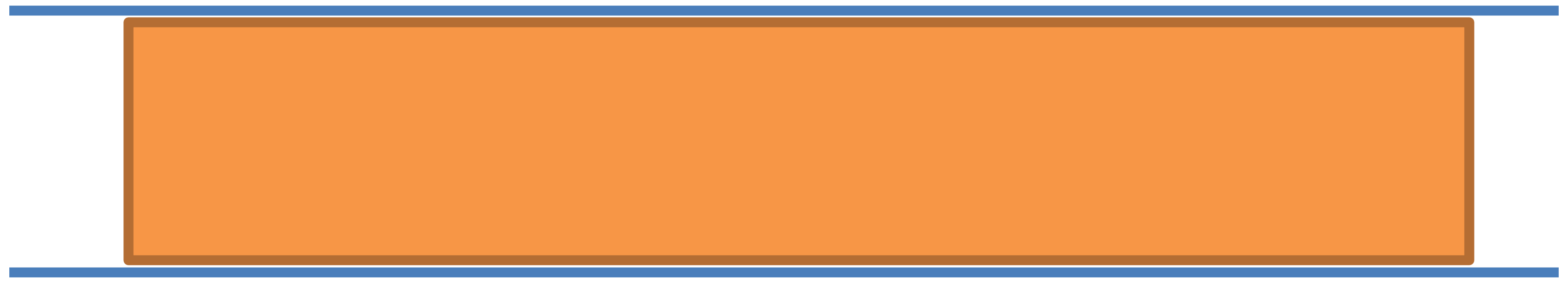
- **Visit node A**

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

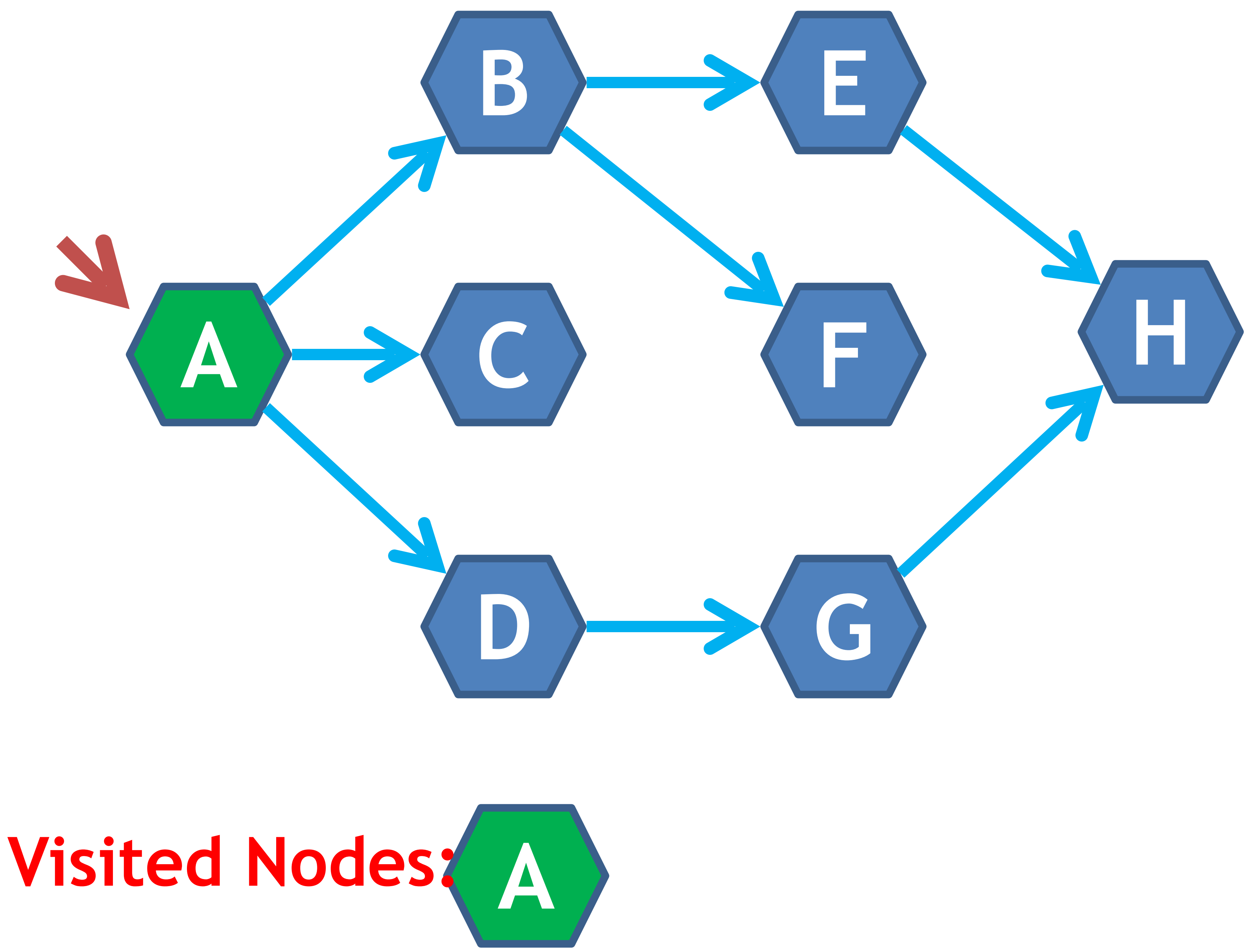
Rule applied

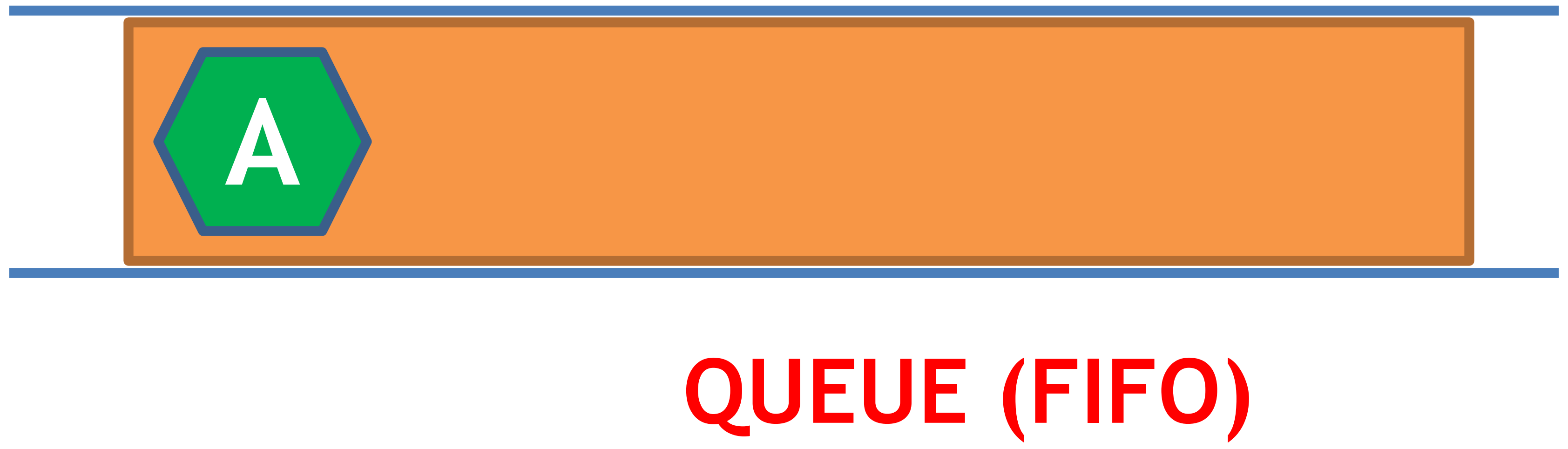
•Mark A as visited

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



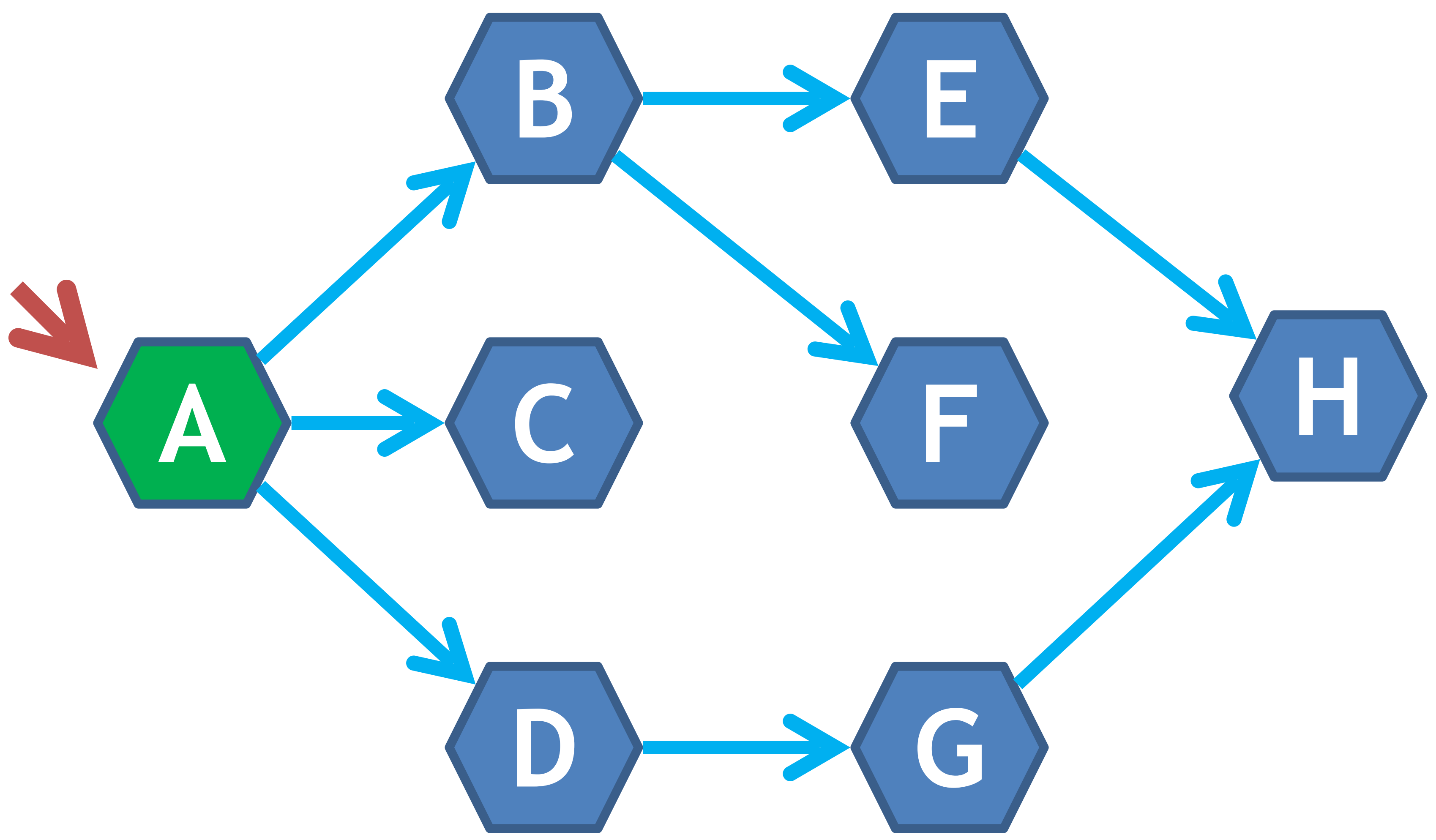


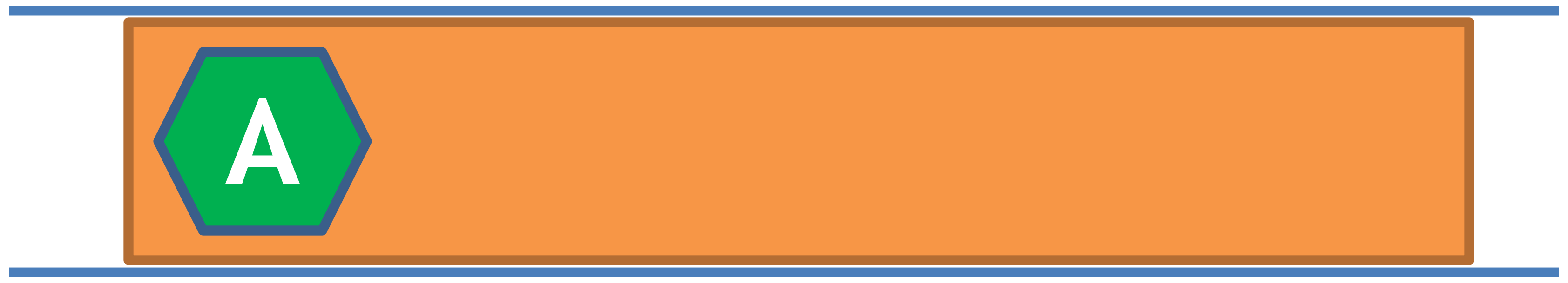
Rule applied

- Insert A into QUEUE

Breadth First Search

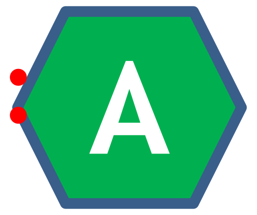
1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





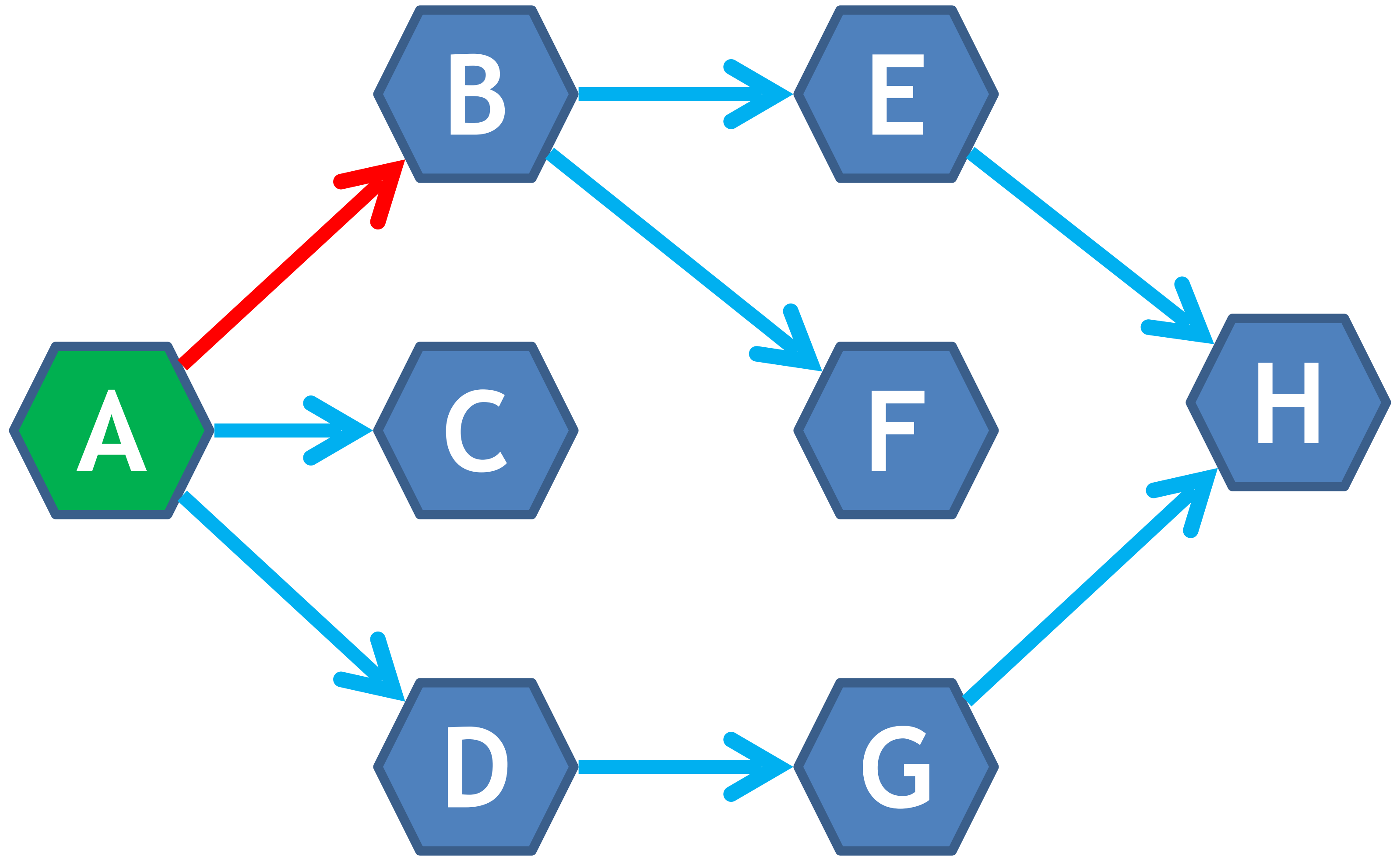
QUEUE (FIFO)

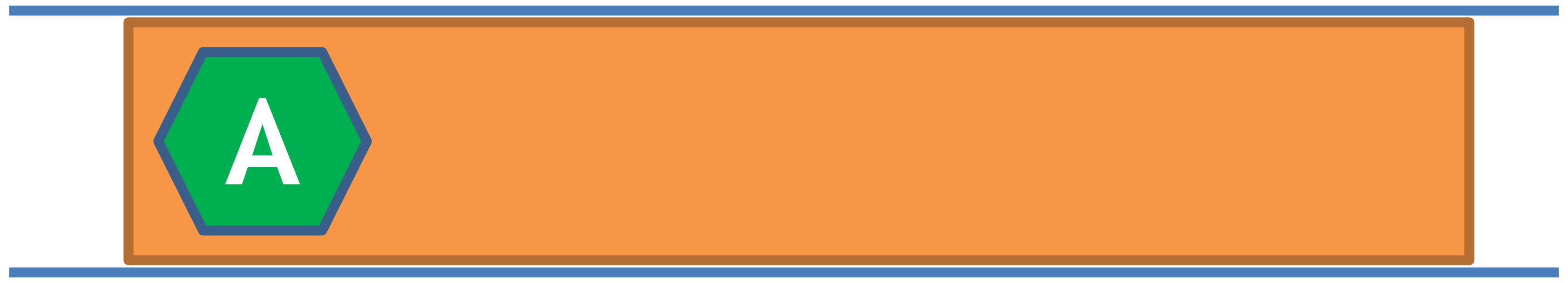
Rule applied
Now, visit the adjacent unvisited nodes of A one by one

Visited Nodes: 

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

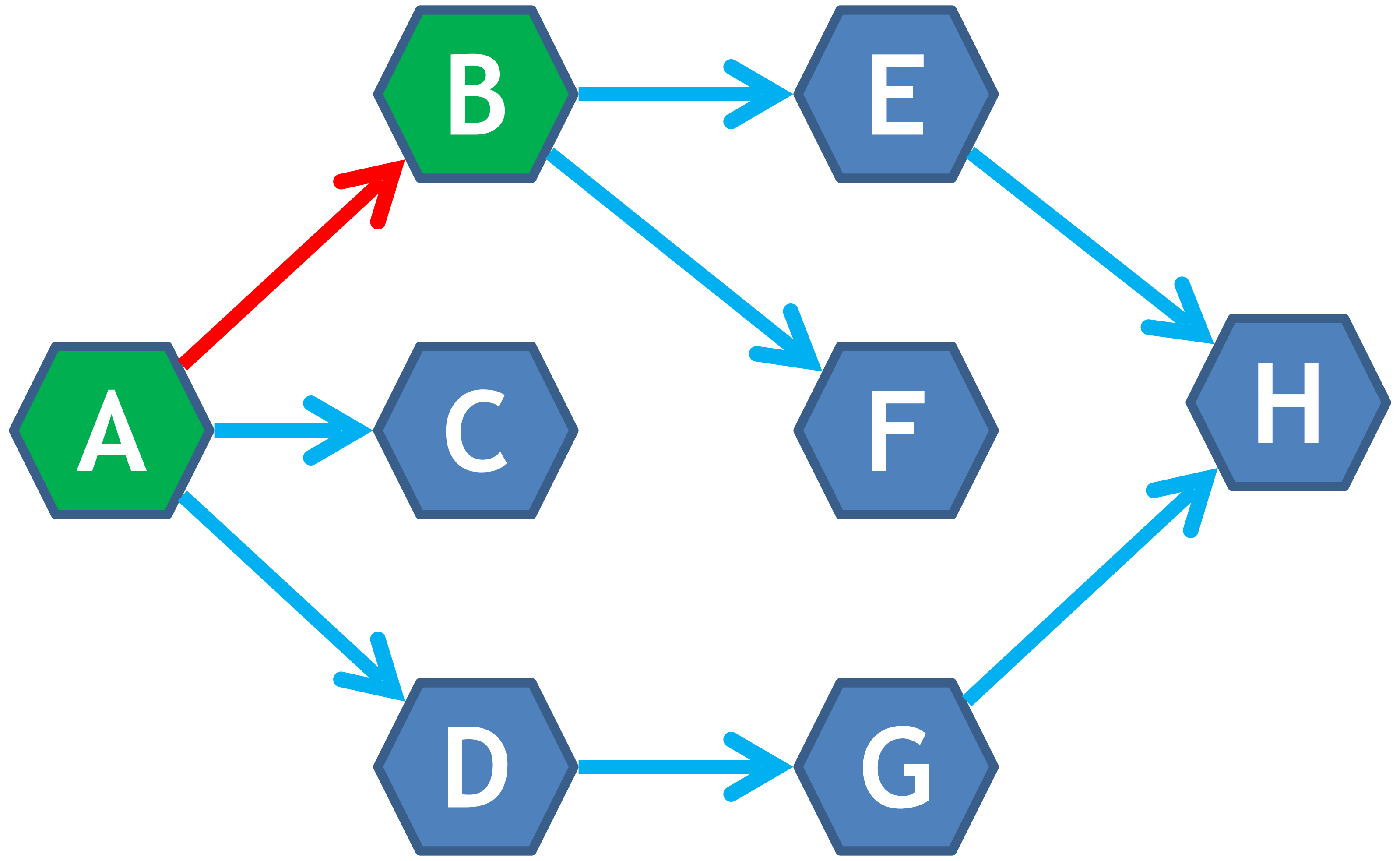
Rule applied

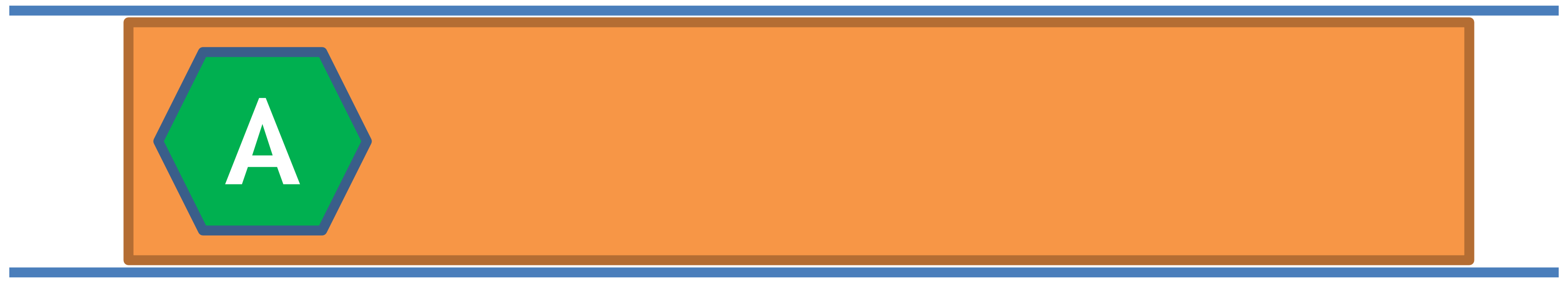
- Alphabetically choose B
  - Visit node B

Visited Nodes: A

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty

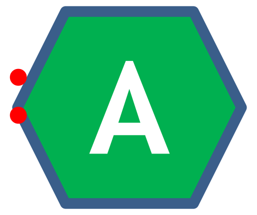




QUEUE (FIFO)

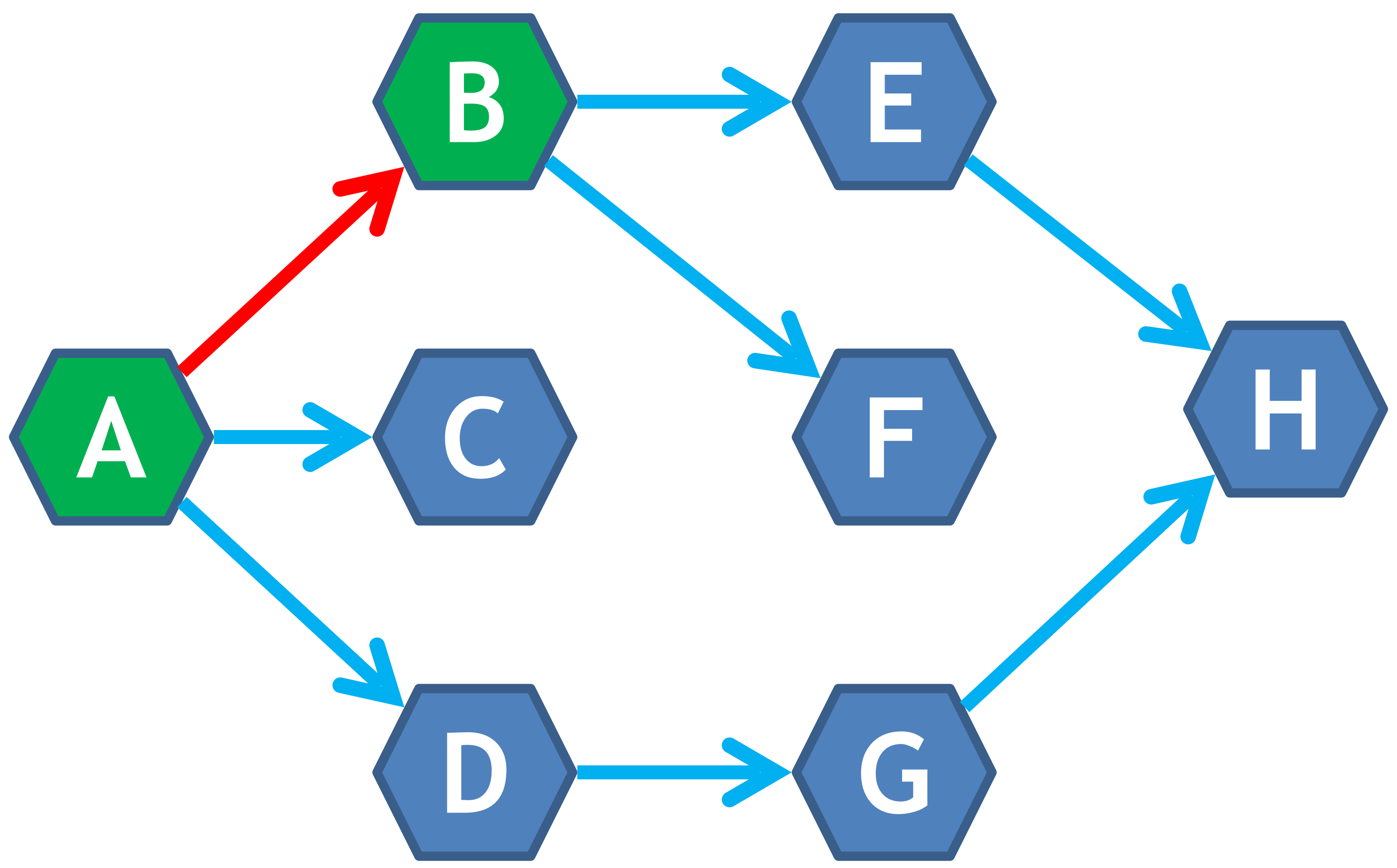
Rule applied

- Mark node B as visited

Visited Nodes: 

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

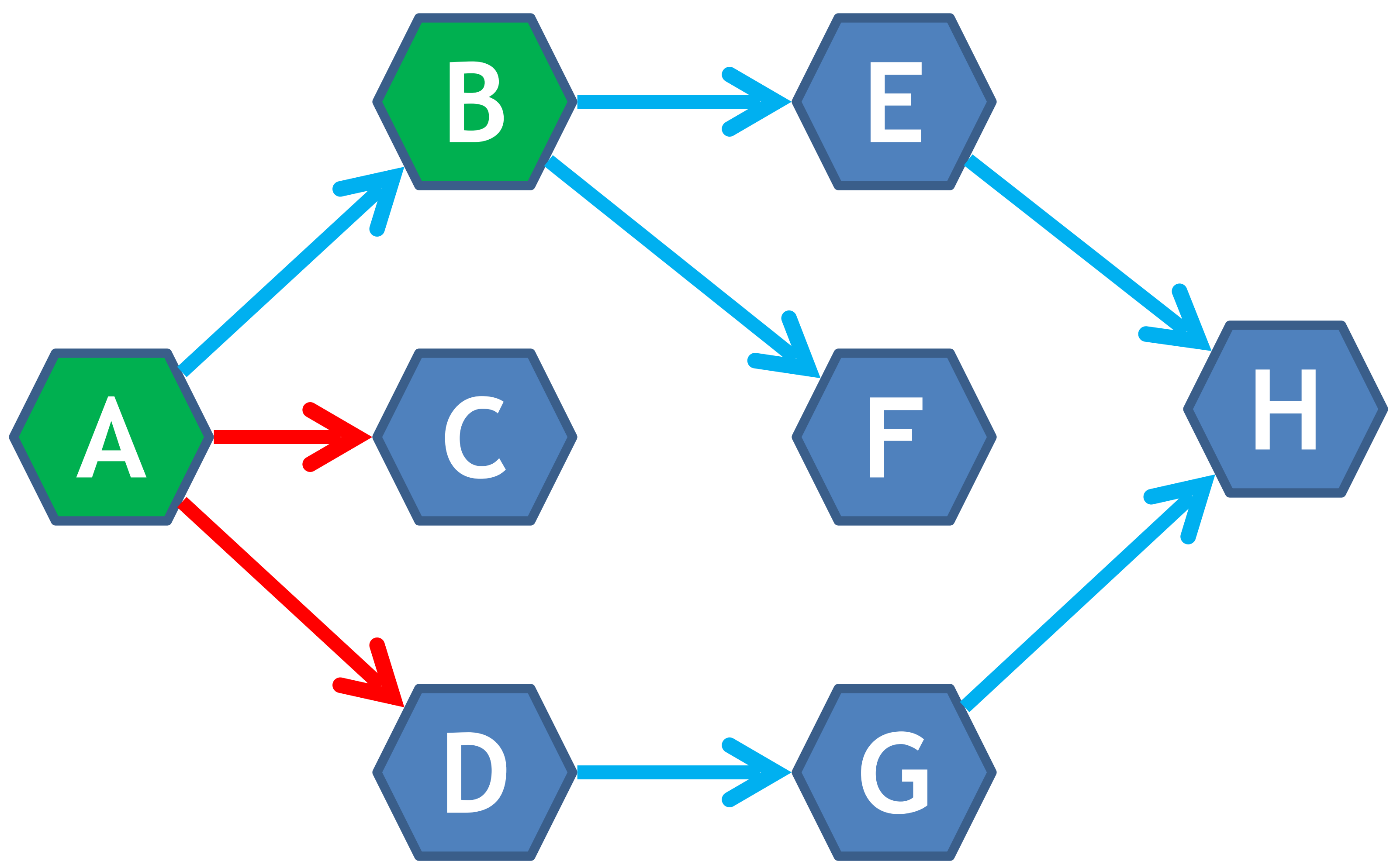
Rule applied

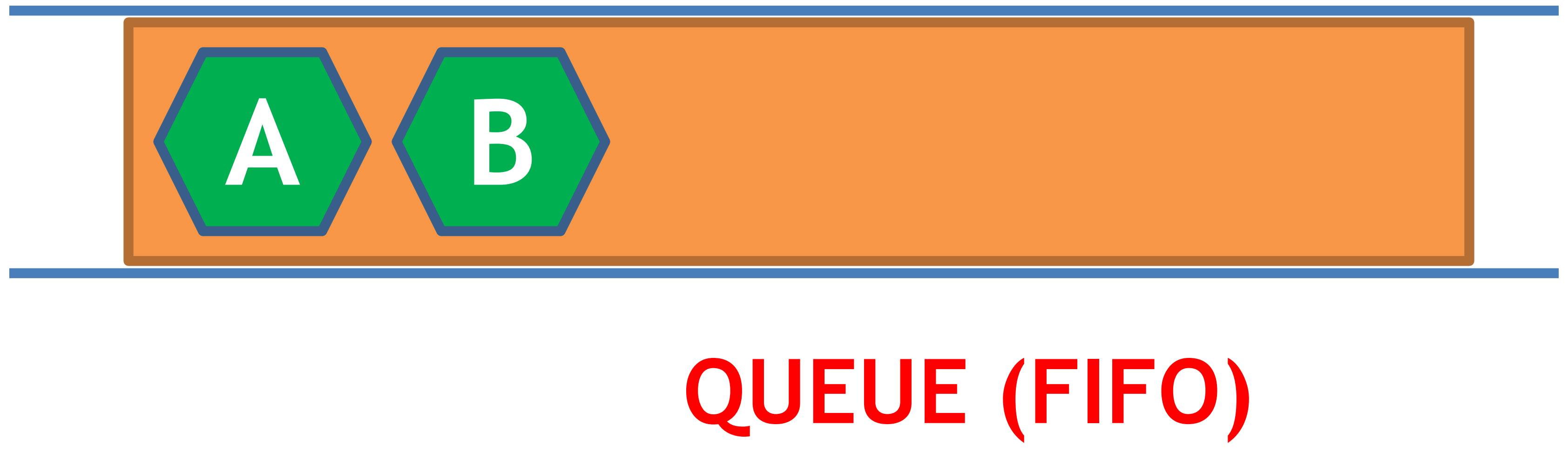
- Insert B into QUEUE

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





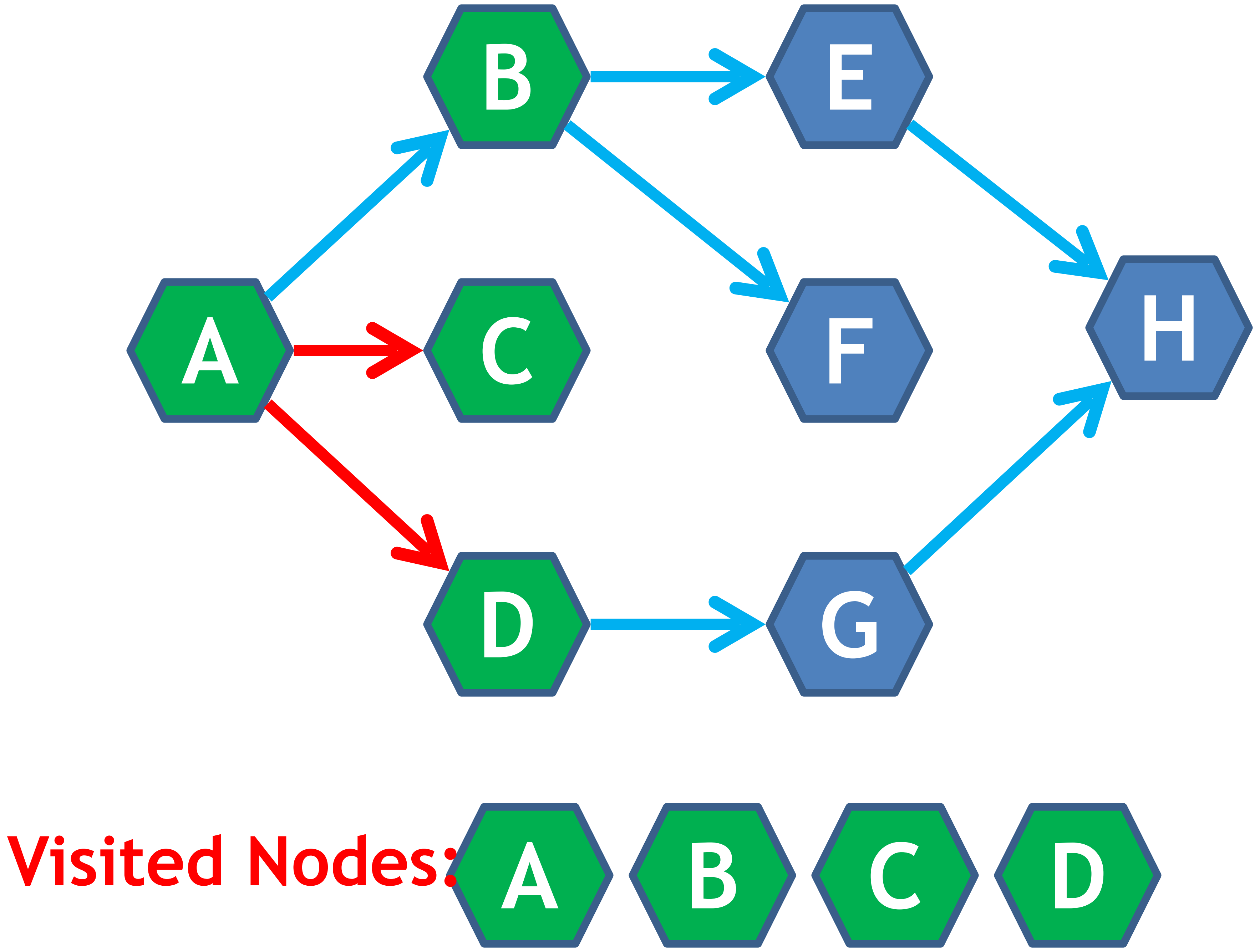
Rule applied

- **Similarly visit all the adjacent nodes of A, i.e. C and D**

Visited Nodes: A B

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty

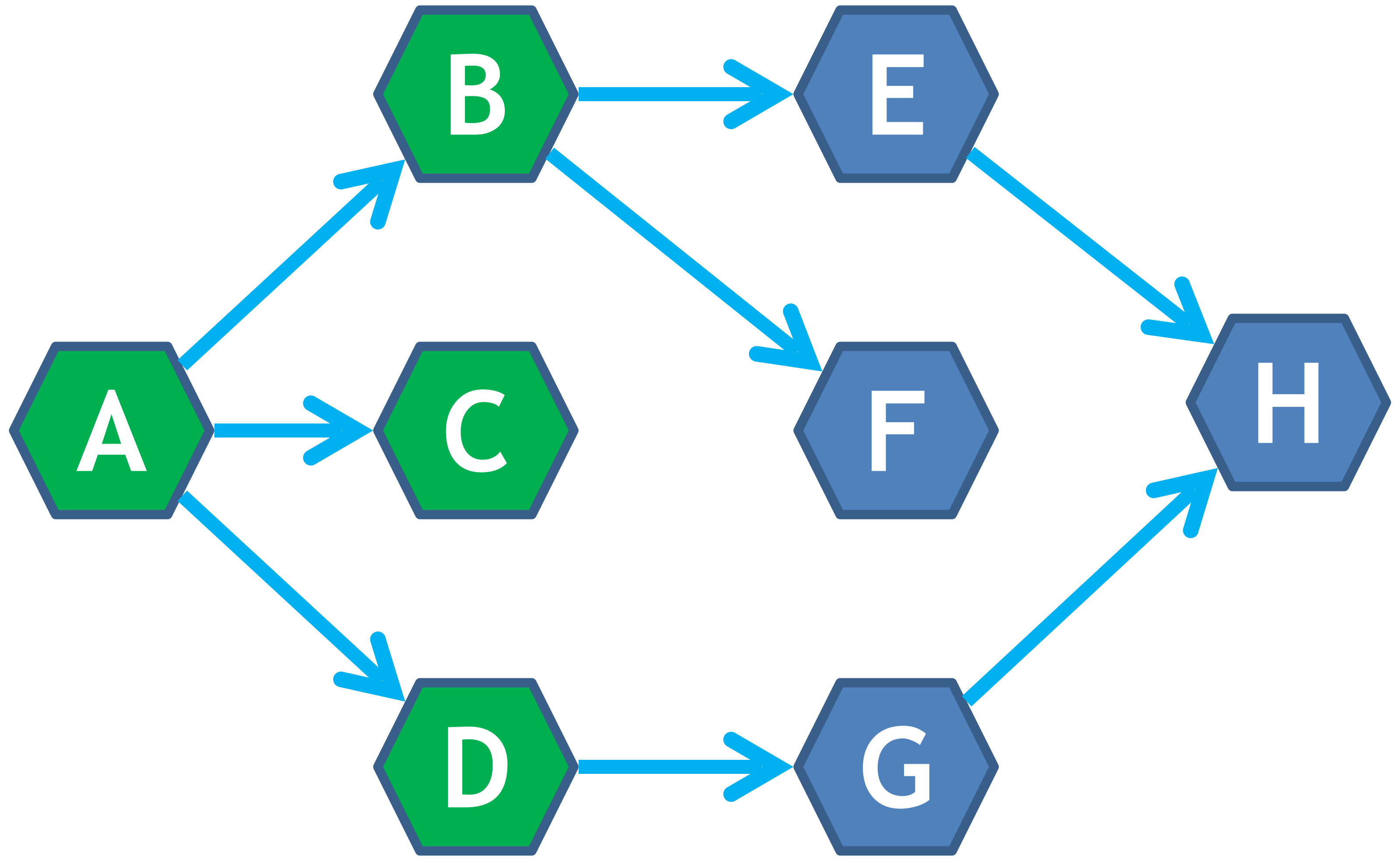


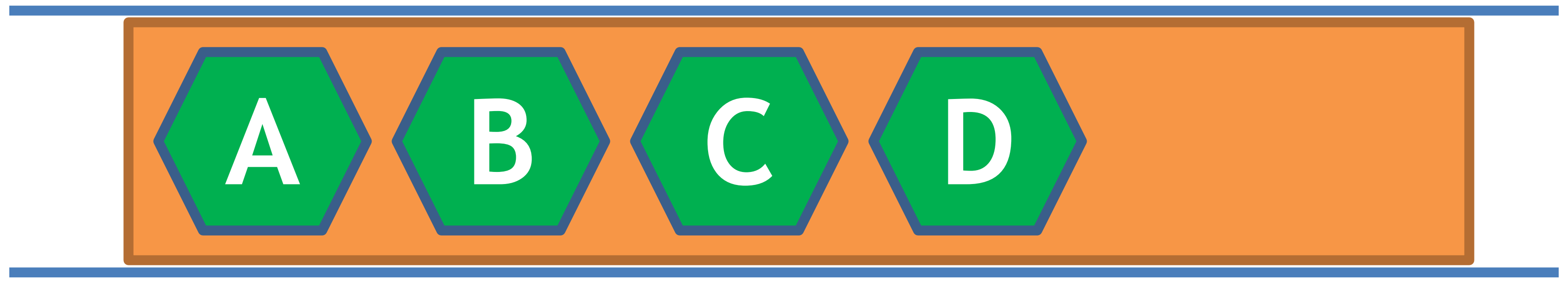
Rule applied

- Mark C and D visited
- Insert them into QUEUE

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

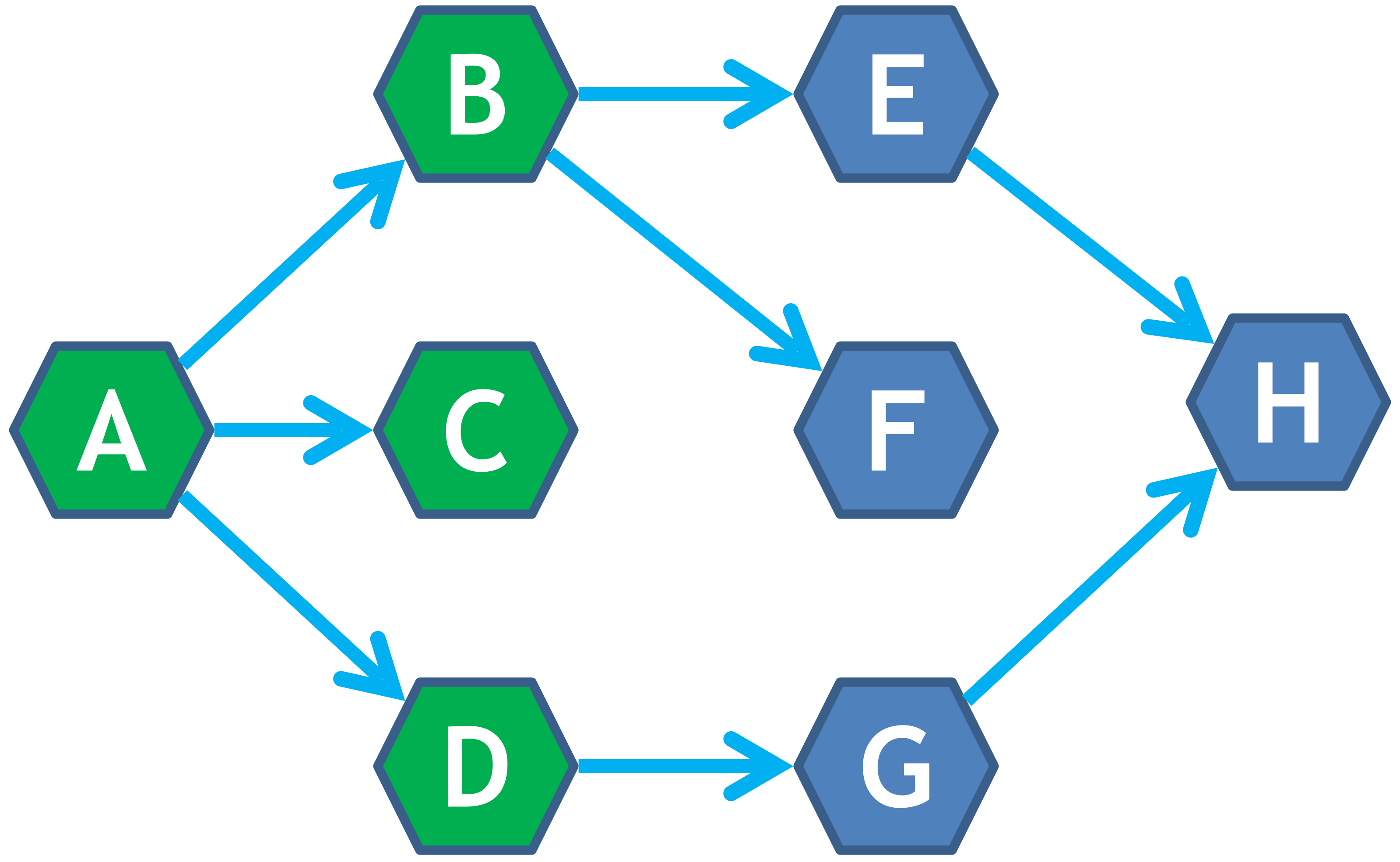
Rule applied

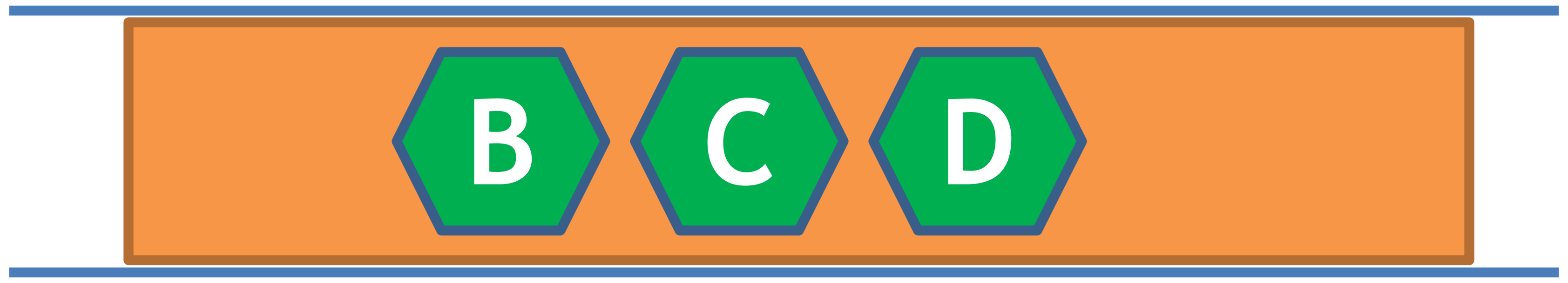
- Now, there are no adjacent unvisited nodes of A

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

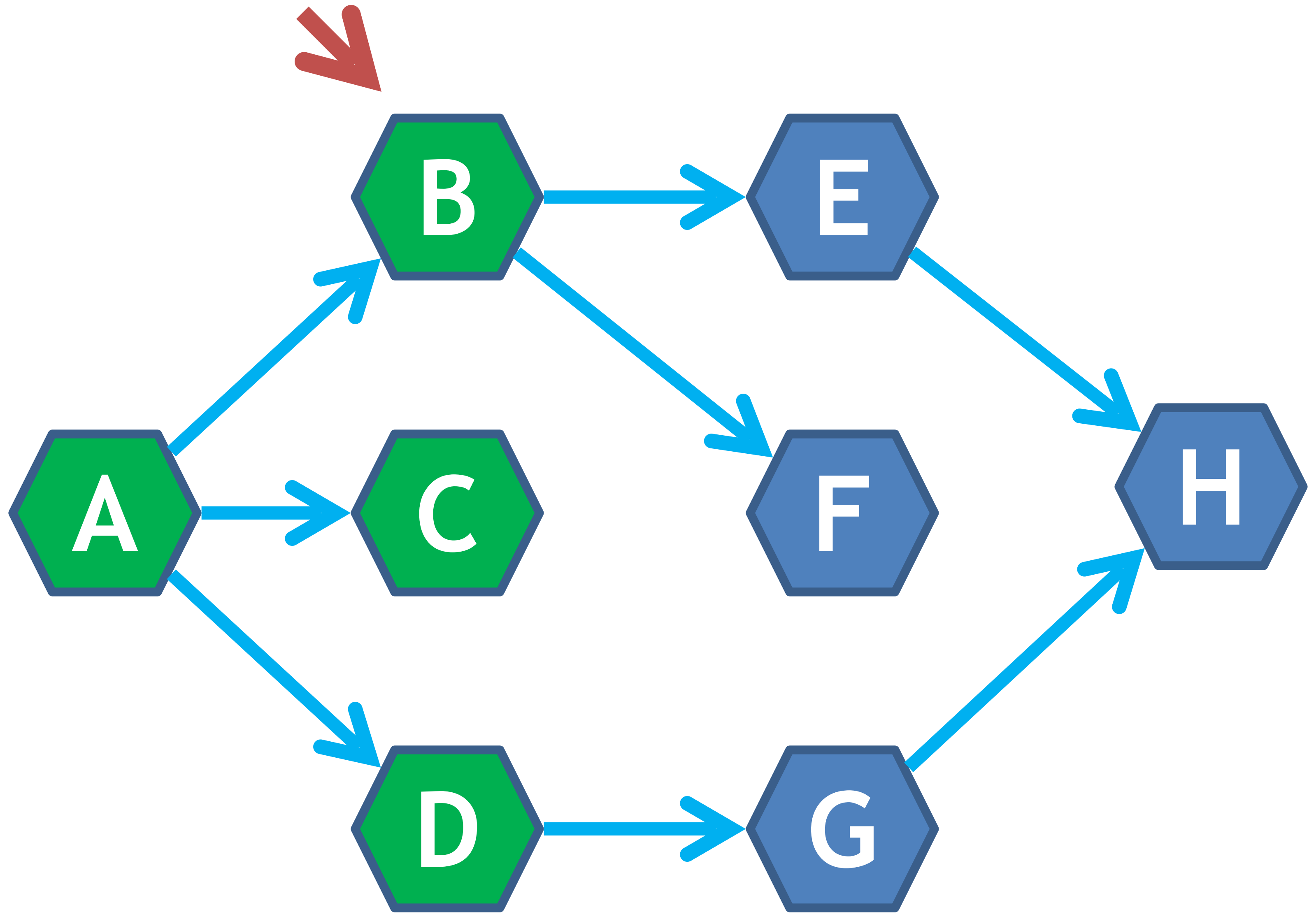
Rule applied

- Now, there are no adjacent unvisited nodes of A
  - So de-queue node A

Visited Nodes: A B C D

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

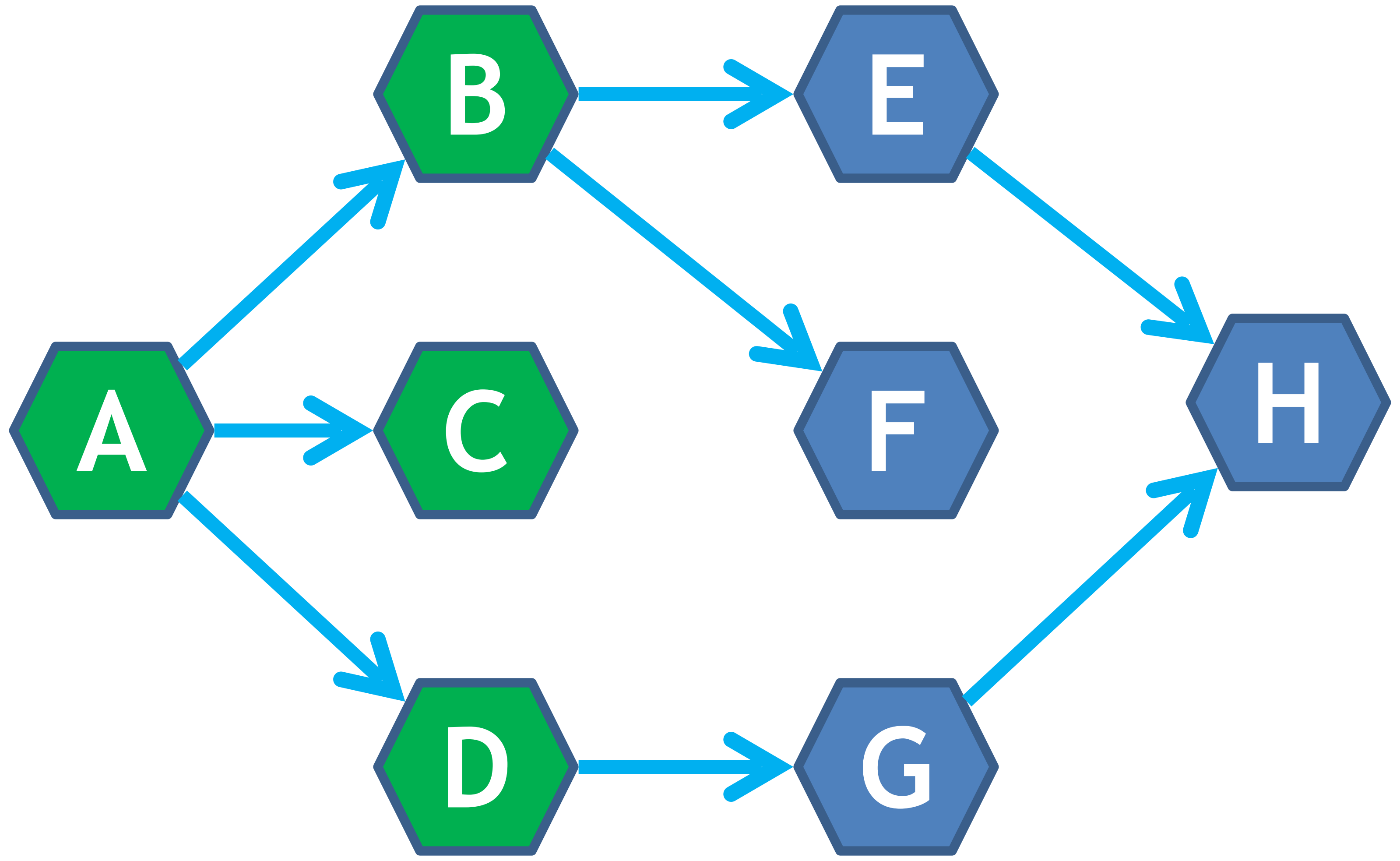
<u>**Rule applied**</u>

- **Now the first node in the queue is B**

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



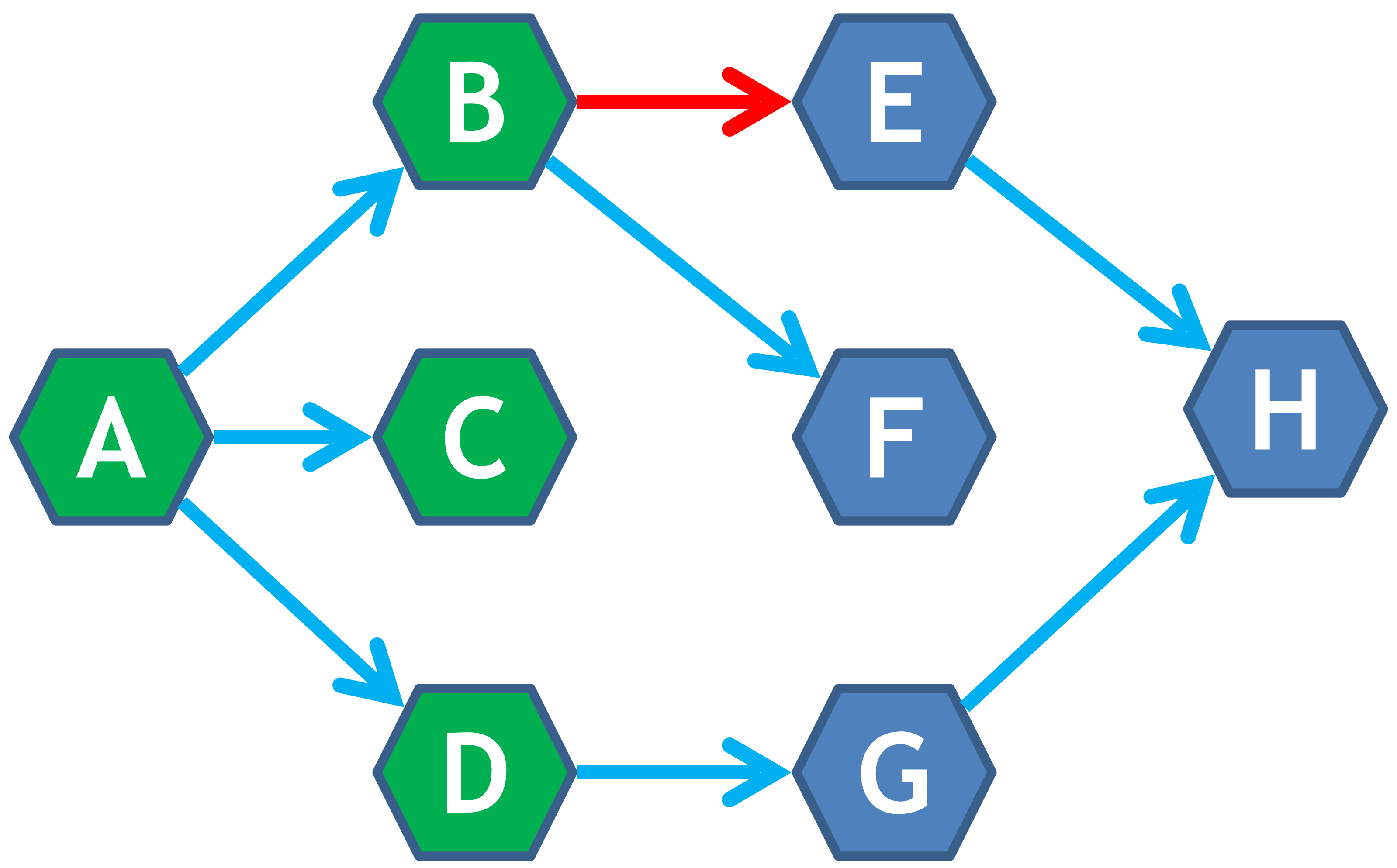
QUEUE (FIFO)

<u>**Rule applied**</u>

- **Visit all the adjacent unvisited nodes of B**

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



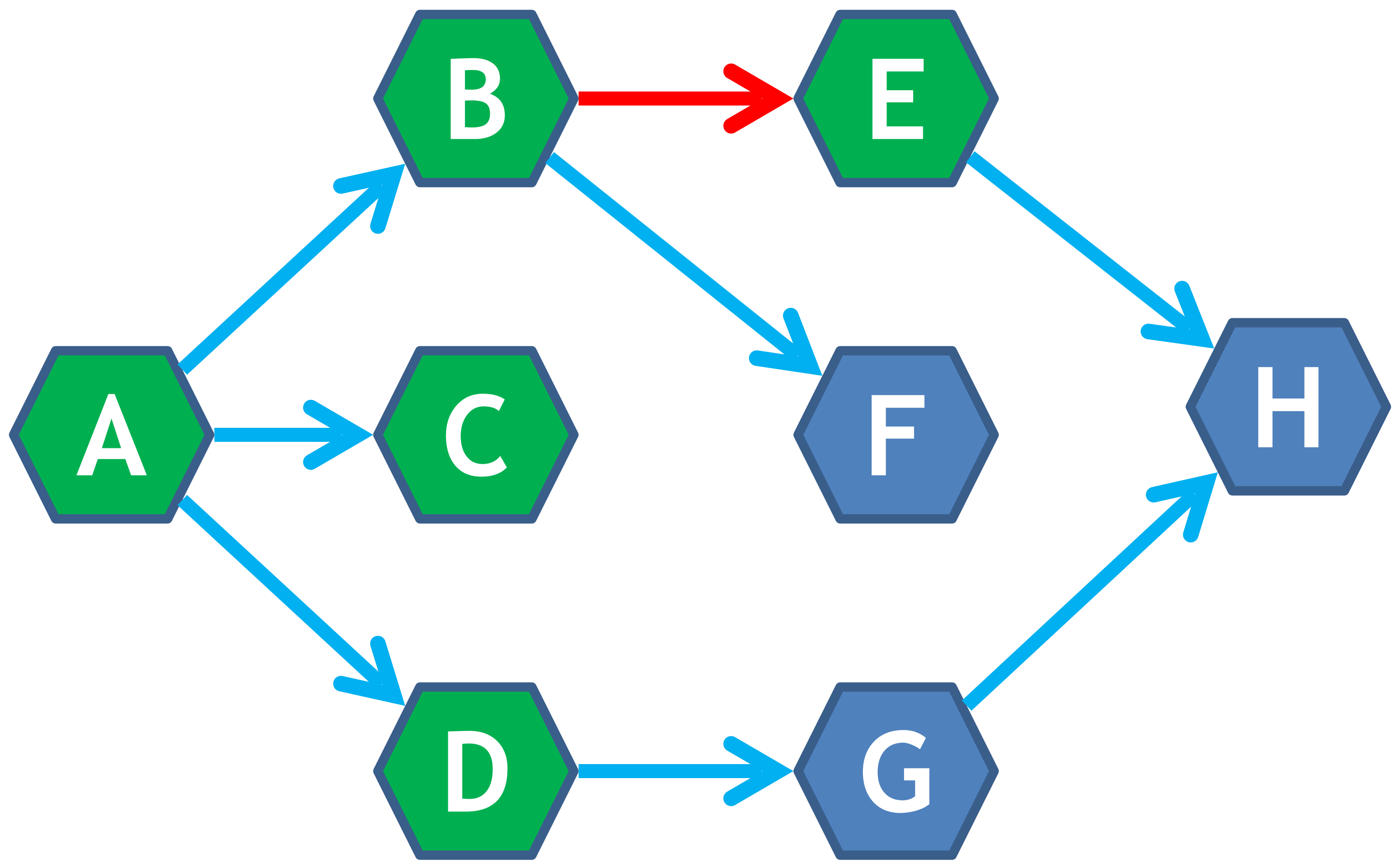
QUEUE (FIFO)

Rule applied
- Visit node E

Visited Nodes: A B C D

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

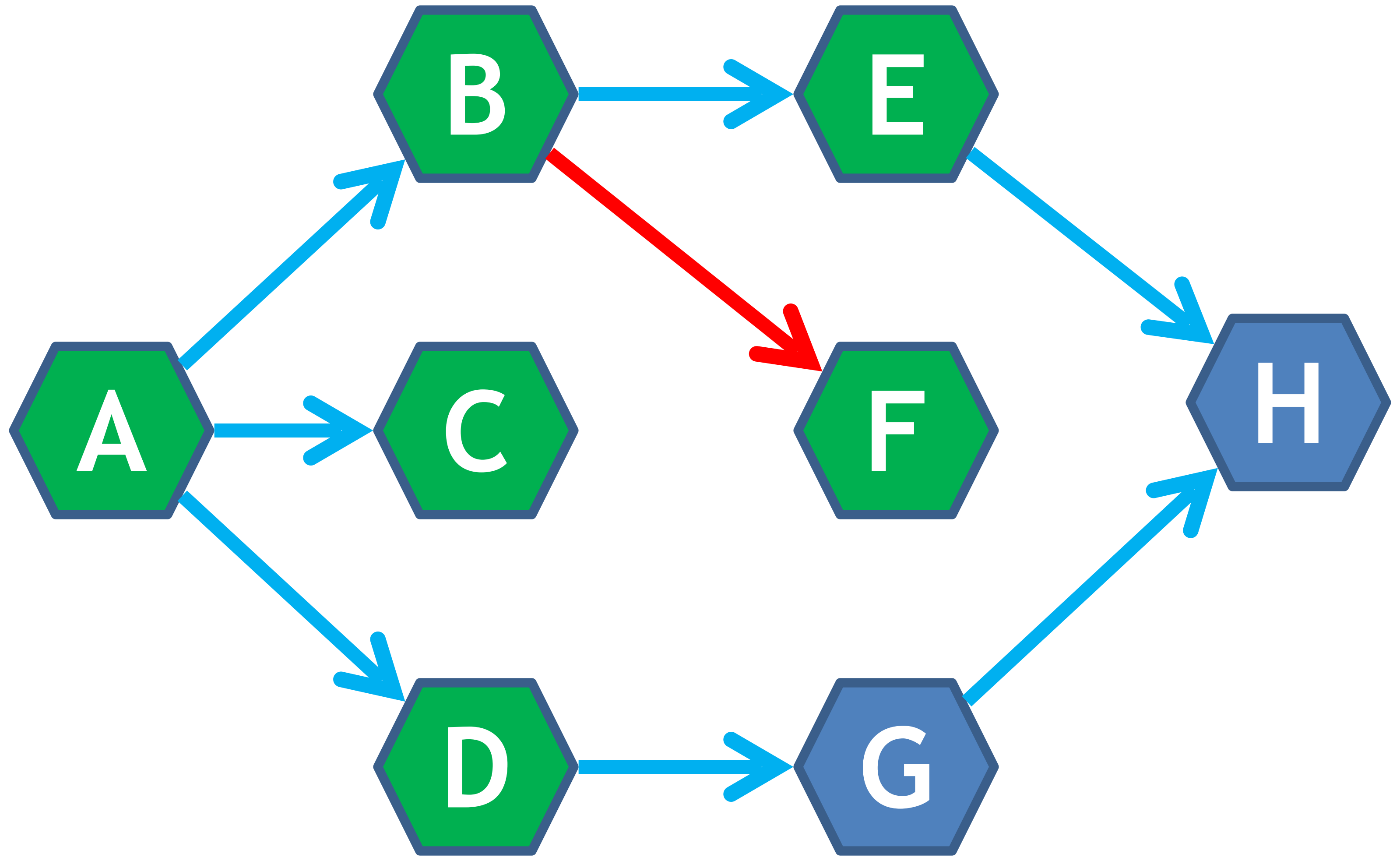
Rule applied

- Mark node E visited
- Insert it into QUEUE

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

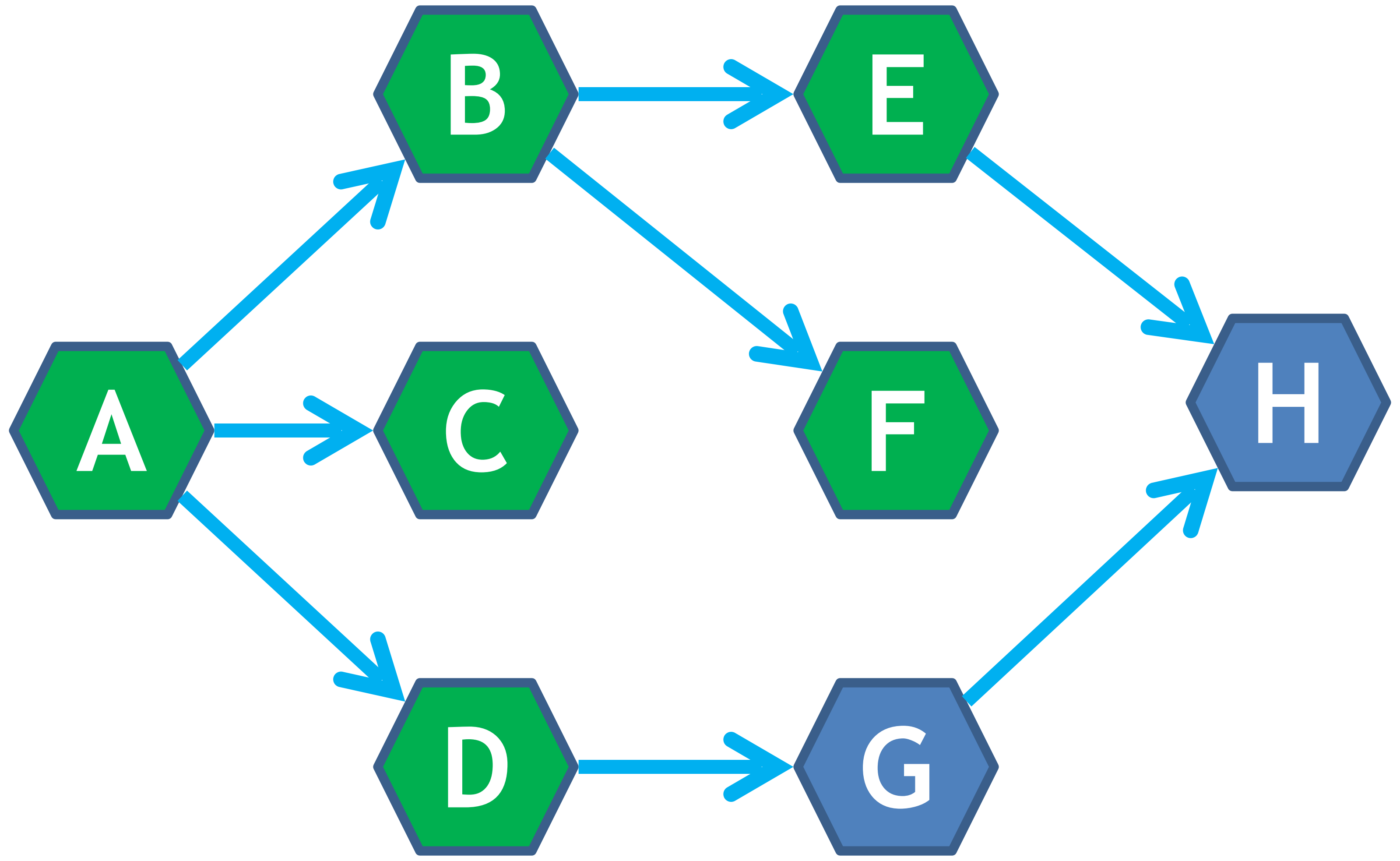
<u>Rule applied</u>

- Do same for F

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

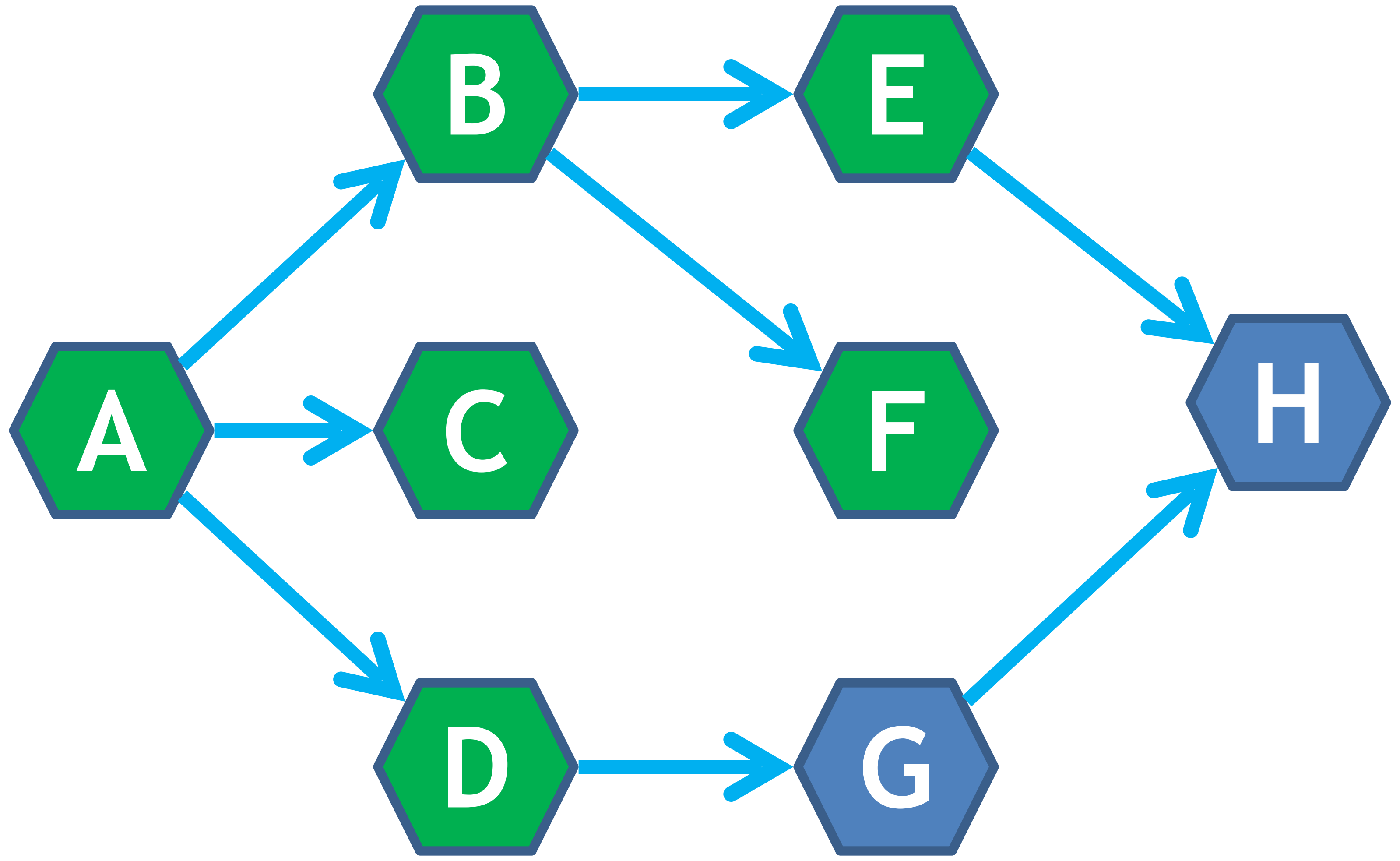
Rule applied

- **Now, B does not have any further unvisited nodes**

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

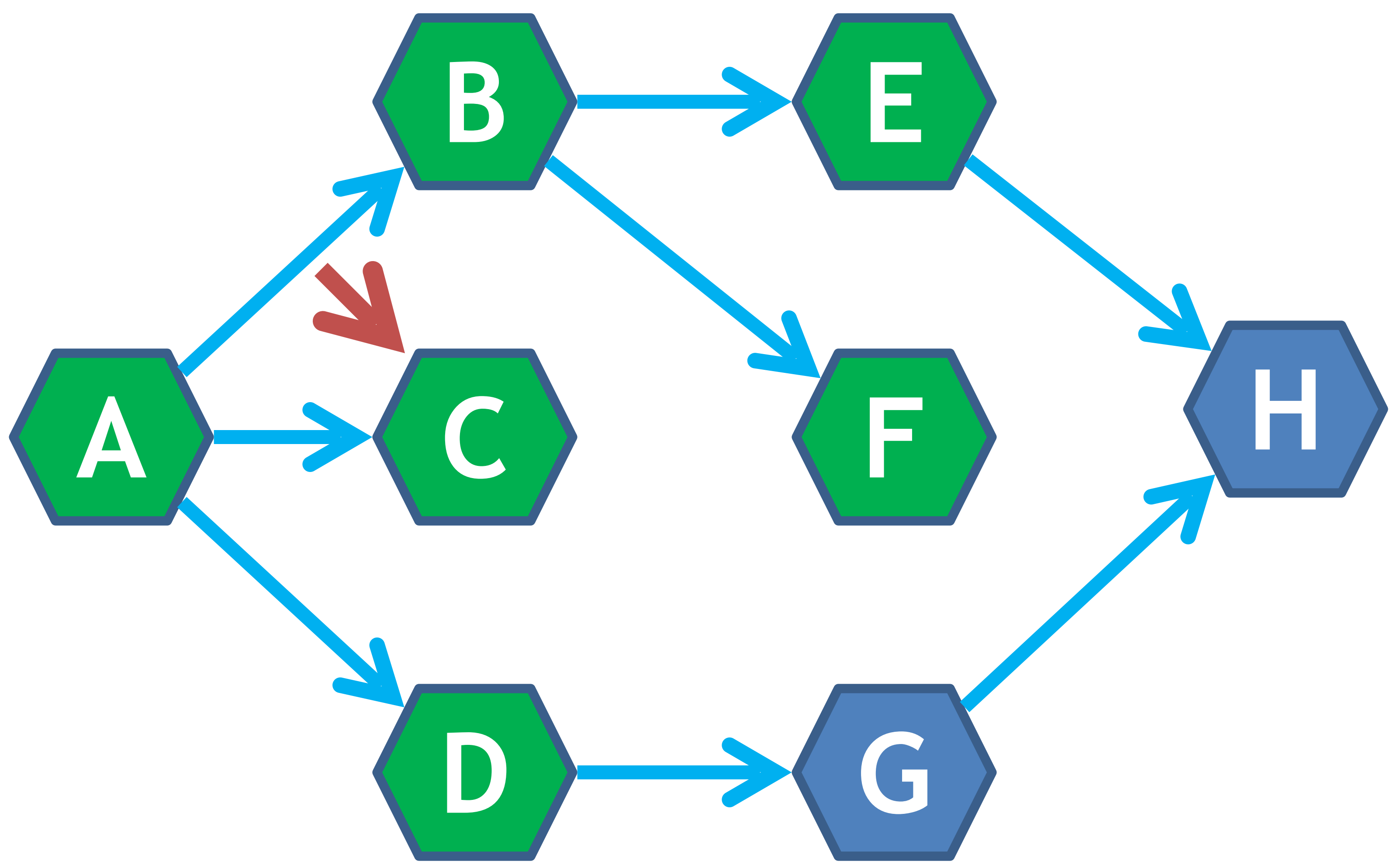
Rule applied

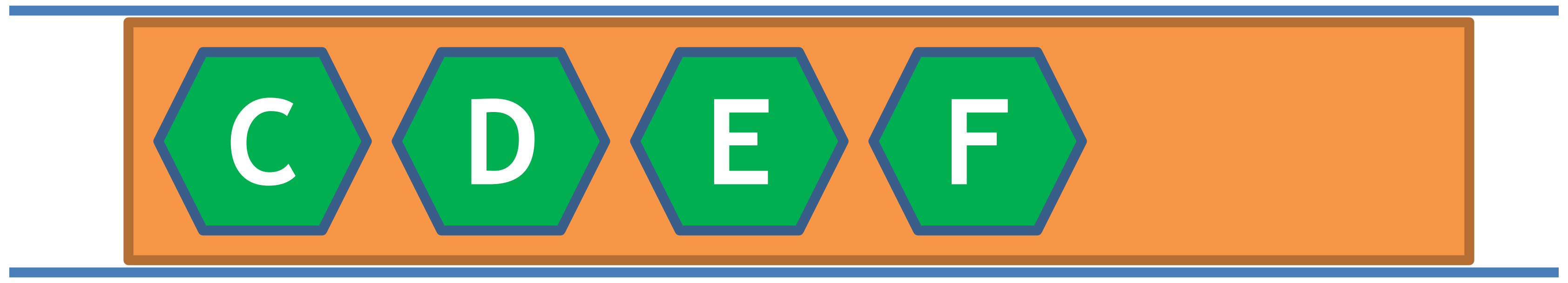
- Now, B does not have any further unvisited nodes
  - So de-queue node B

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

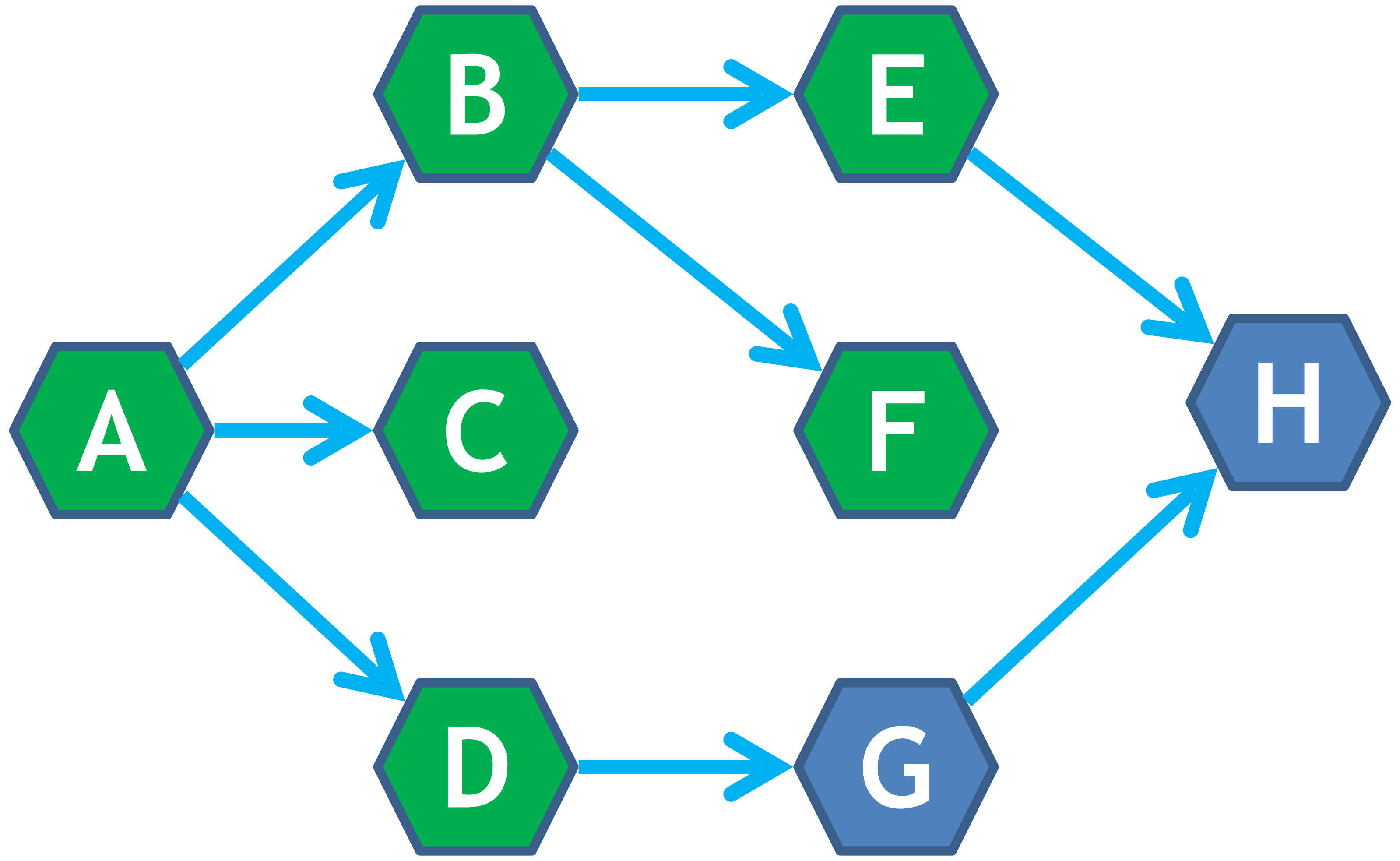
Rule applied

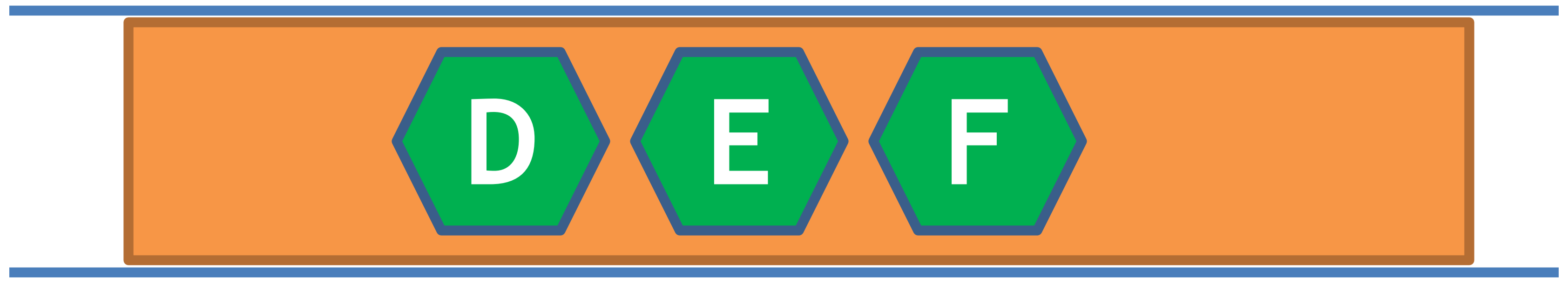
- C also doesn't have any adjacent nodes

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

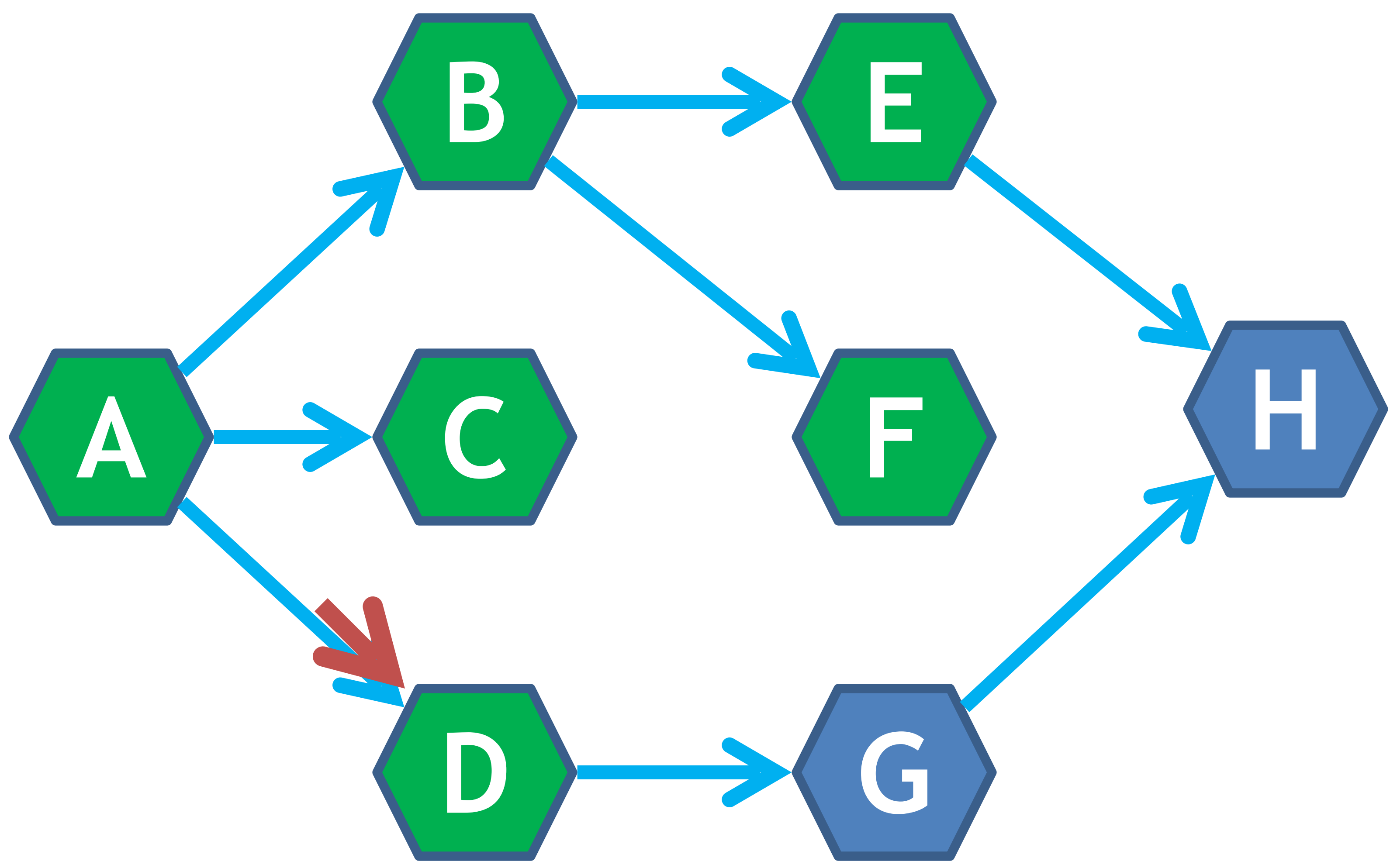
Rule applied

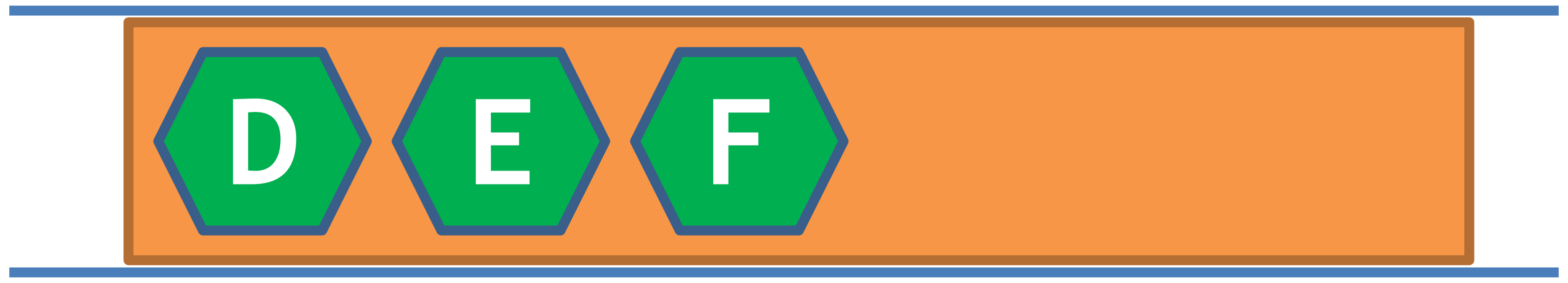
- C also doesn't have any adjacent nodes
    - So de-queue node C

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

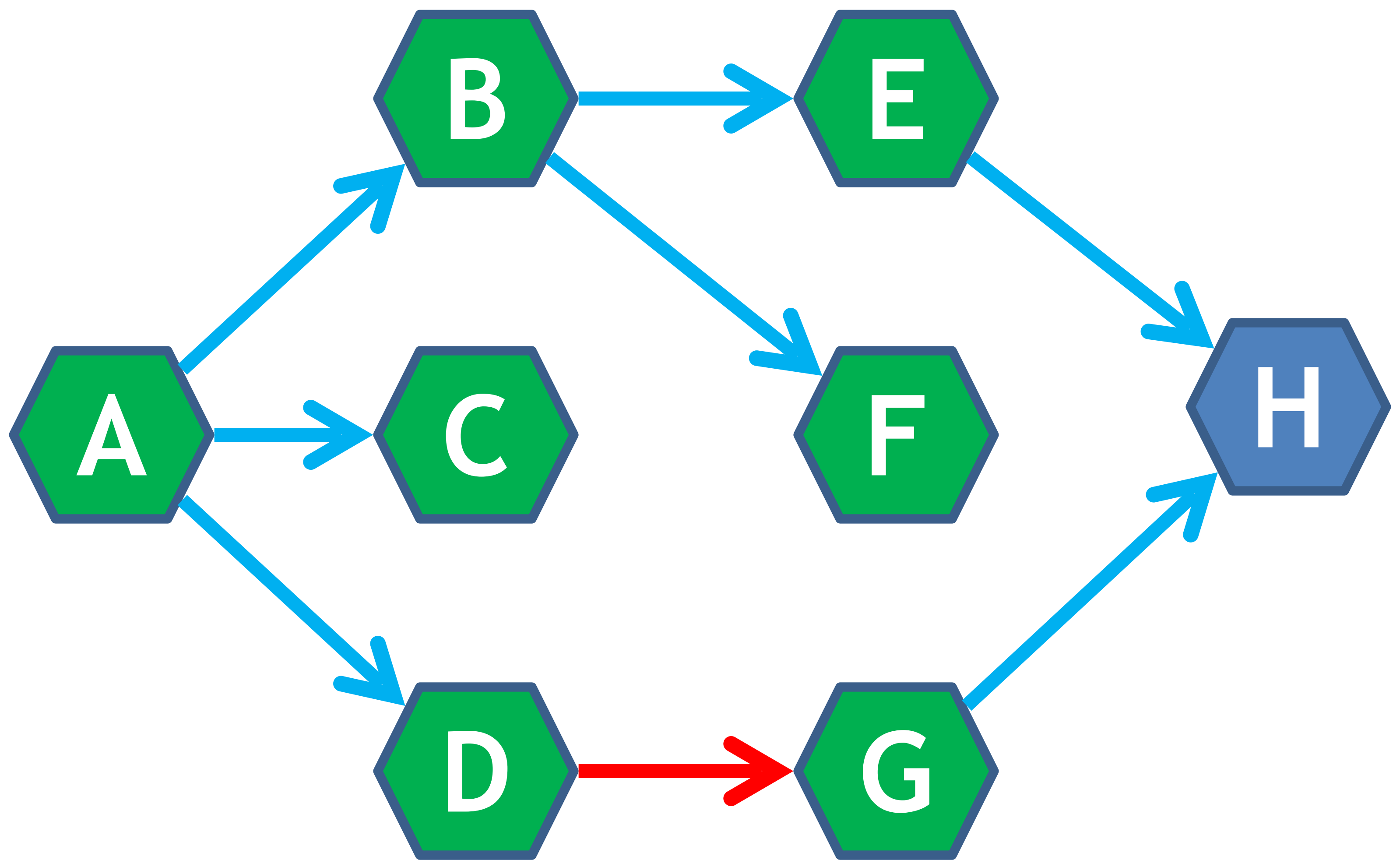
Rule applied

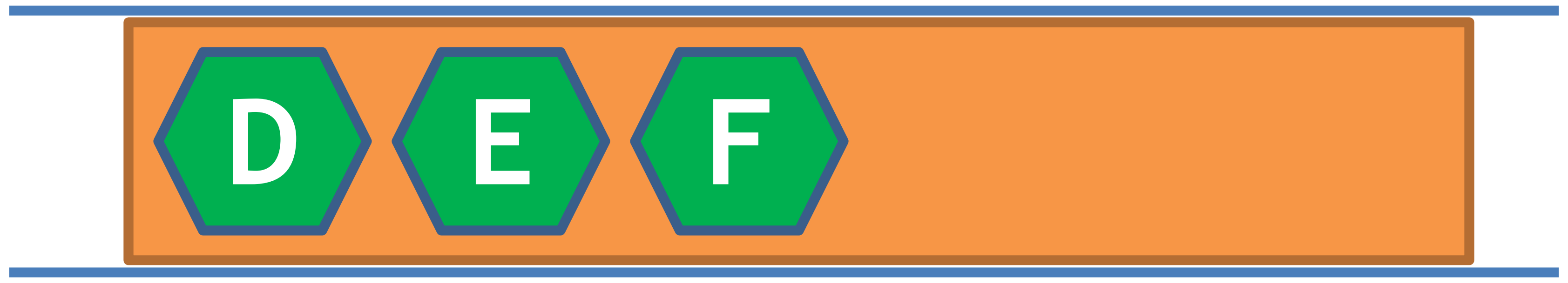
- Now, visit adjacent unvisited nodes of D
  - Its node G

Visited Nodes: A B C D E F

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

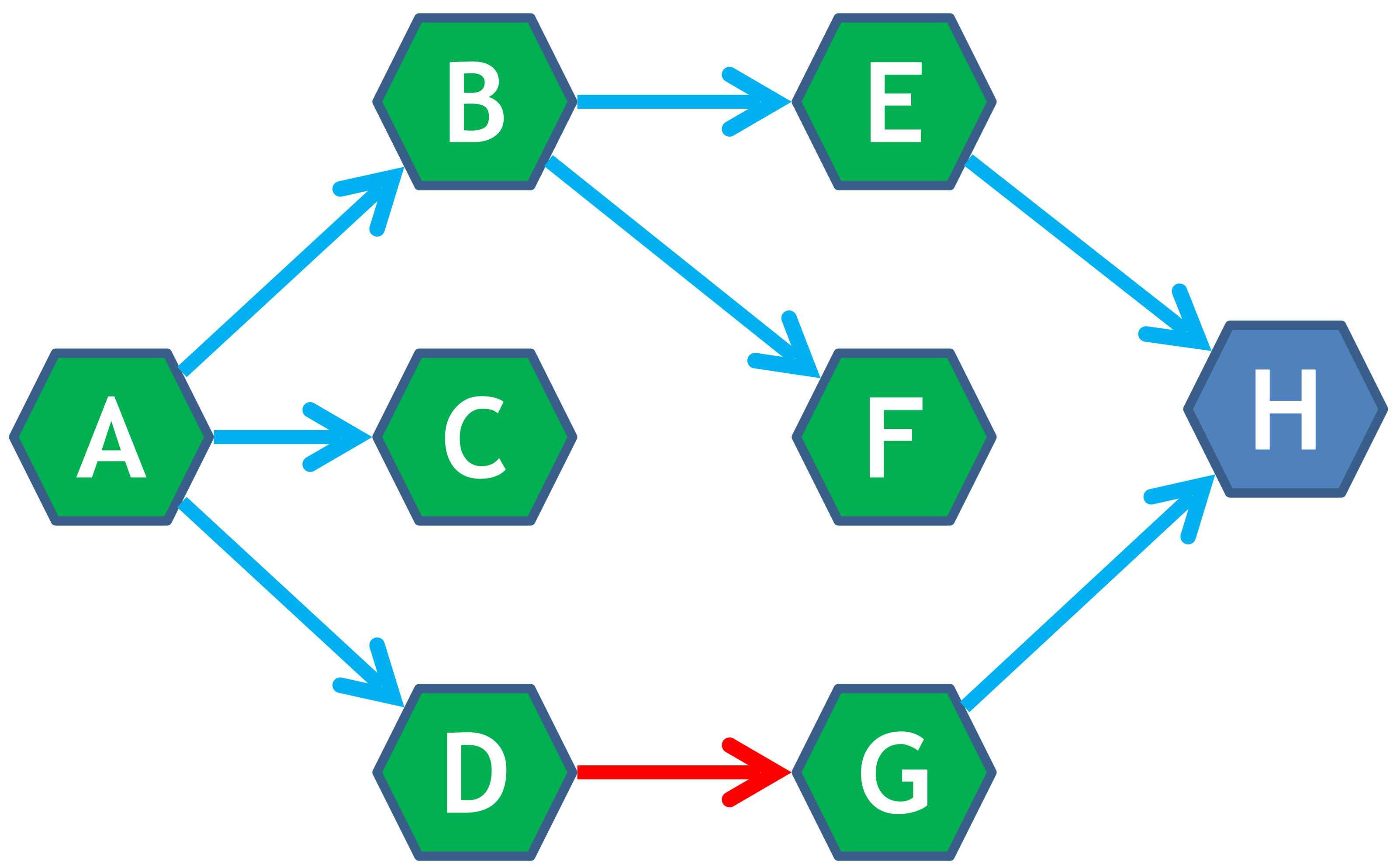
Rule applied

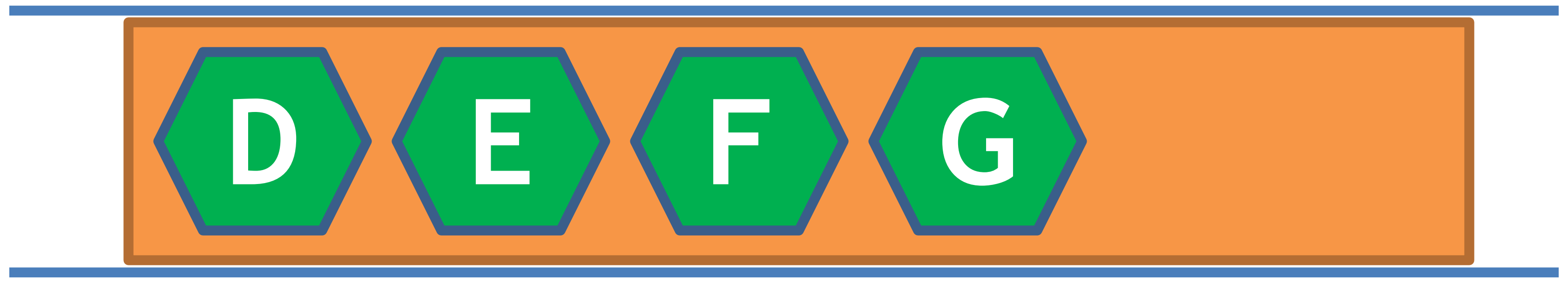
- **Visit node G**
- **Mark it as visited**

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

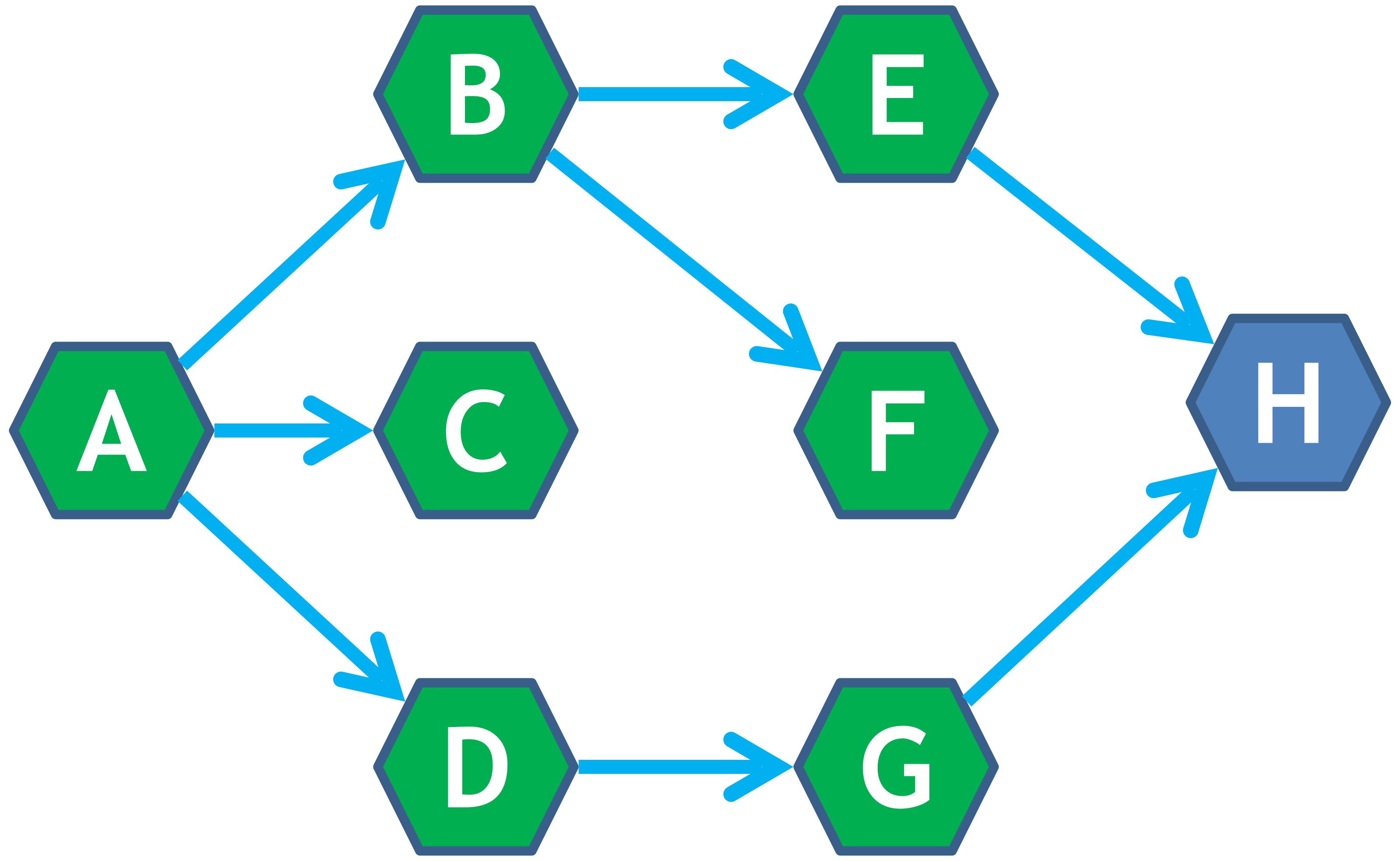
Rule applied

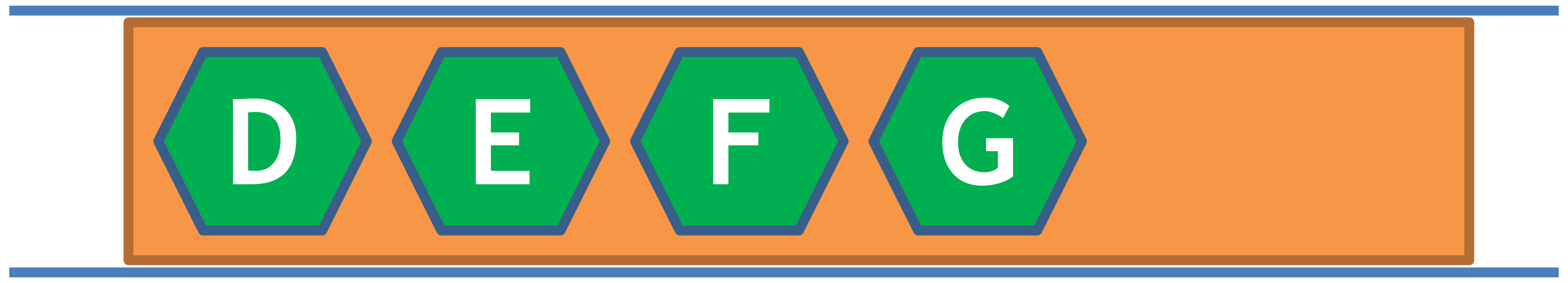
- Add G to QUEUE

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

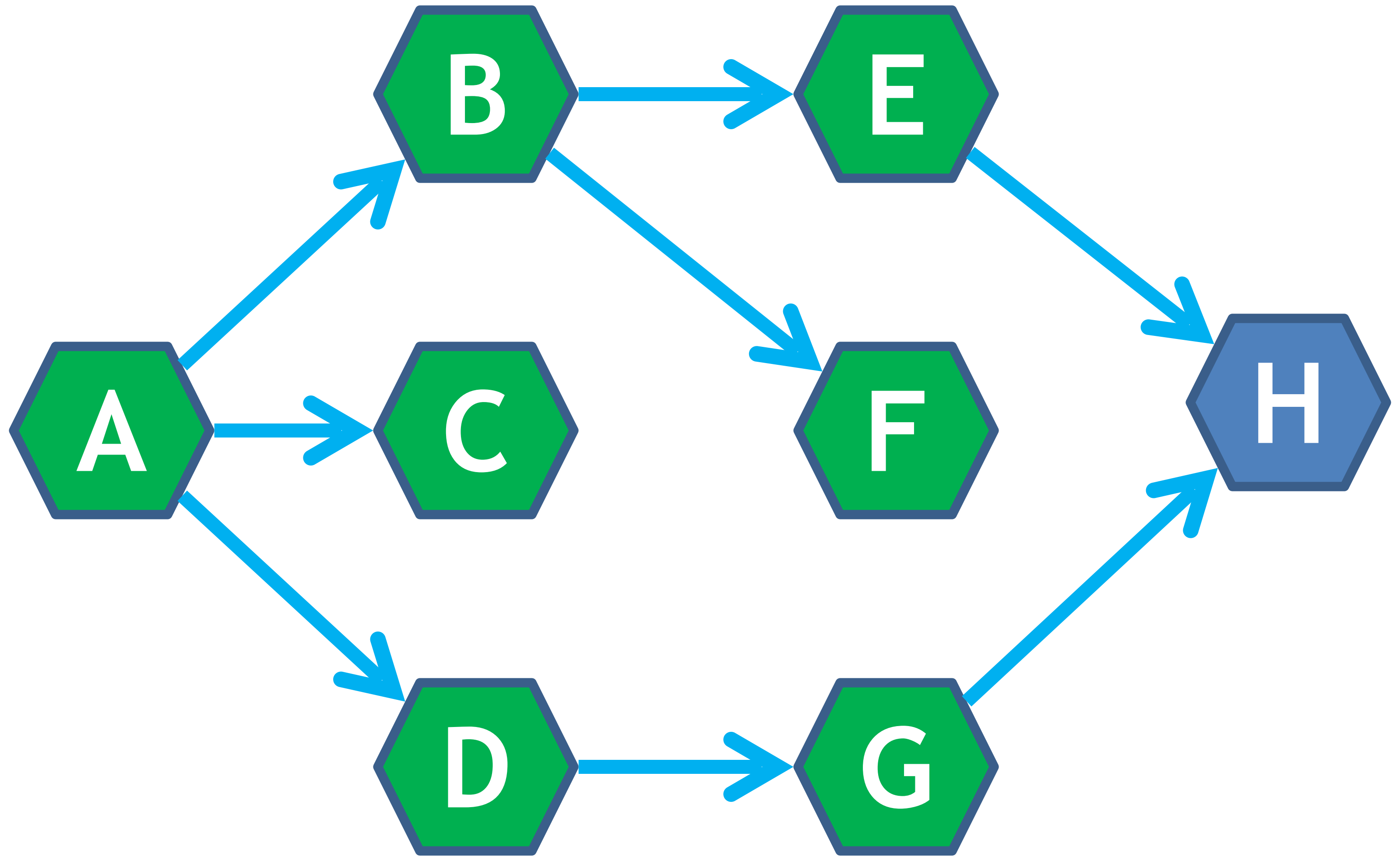
Rule applied

- Now, D doesn't have any adjacent unvisited node

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

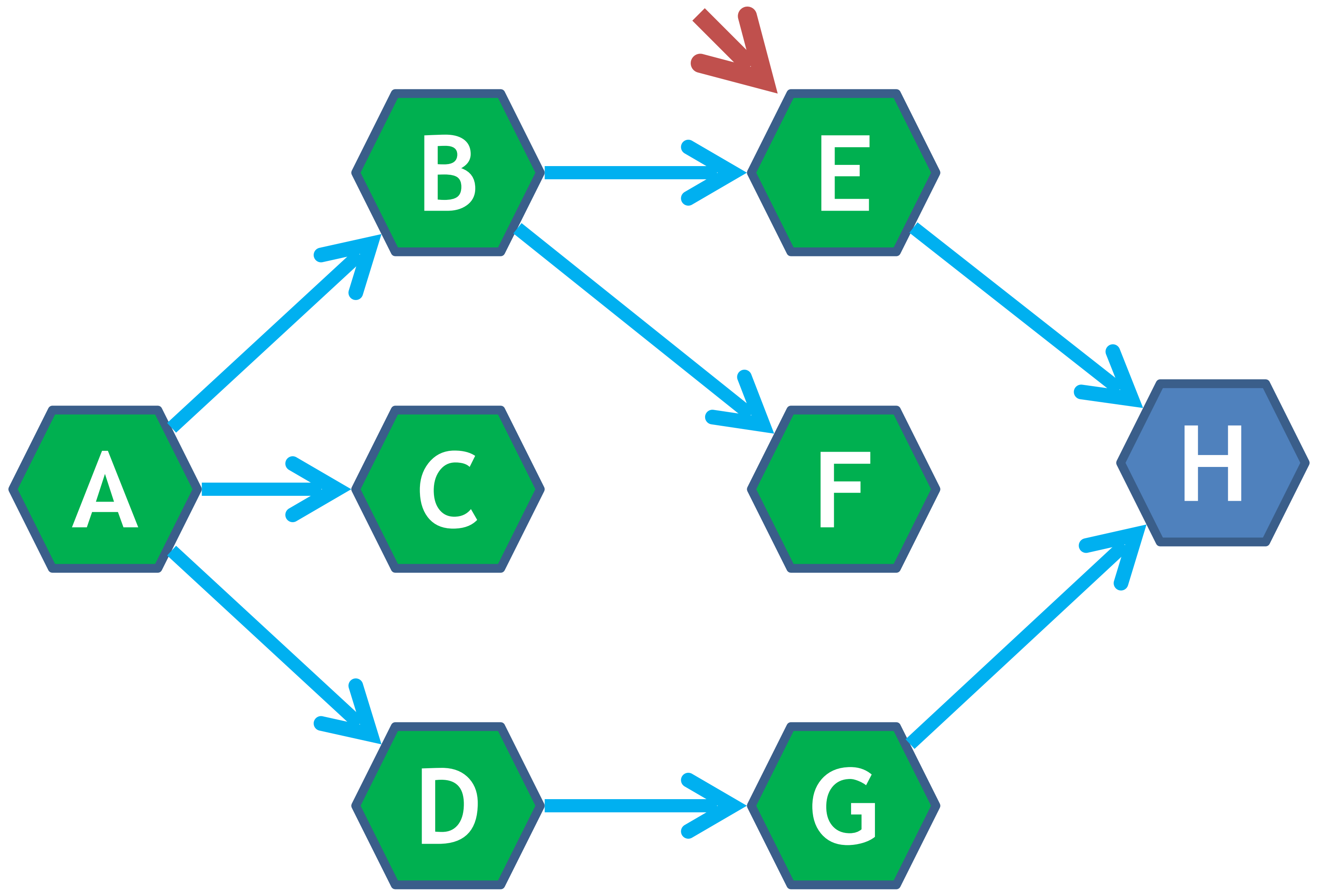
Rule applied

- Now, D doesn't have any adjacent unvisited node
- So, de-queue node D

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

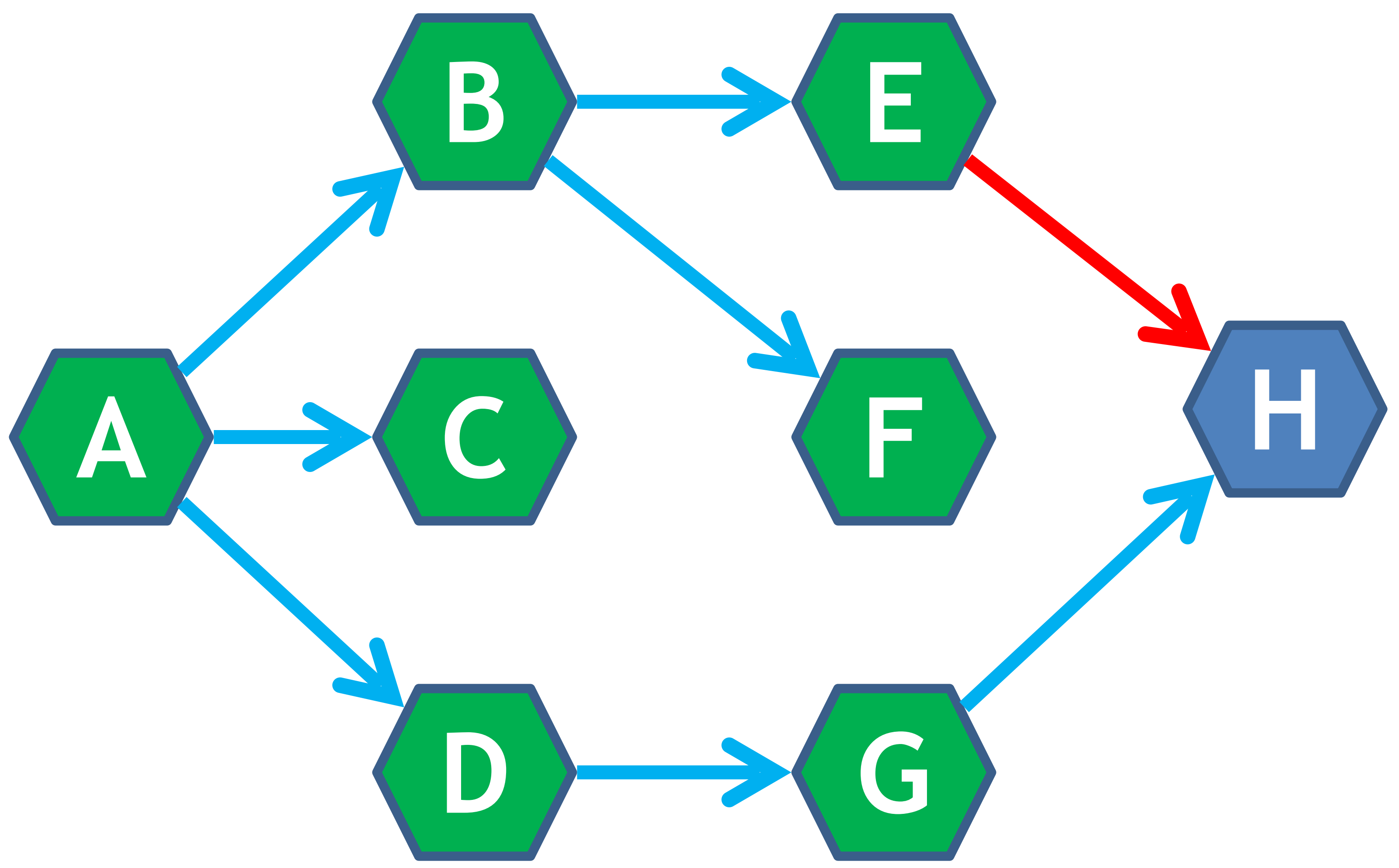
Rule applied

- Now, visit the adjacent unvisited nodes of E
  - i.e. node H

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

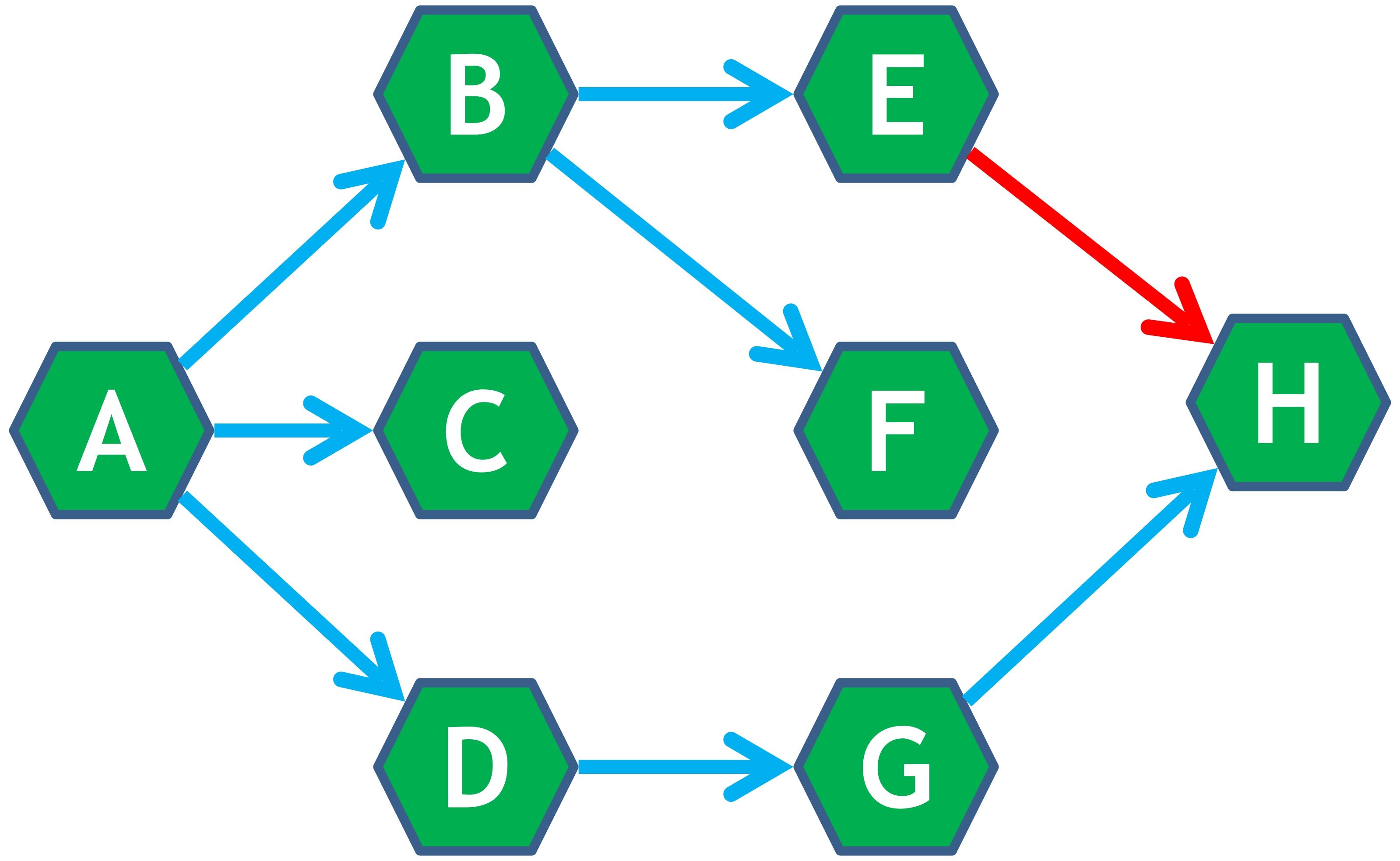
Rule applied

- Visit node H

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



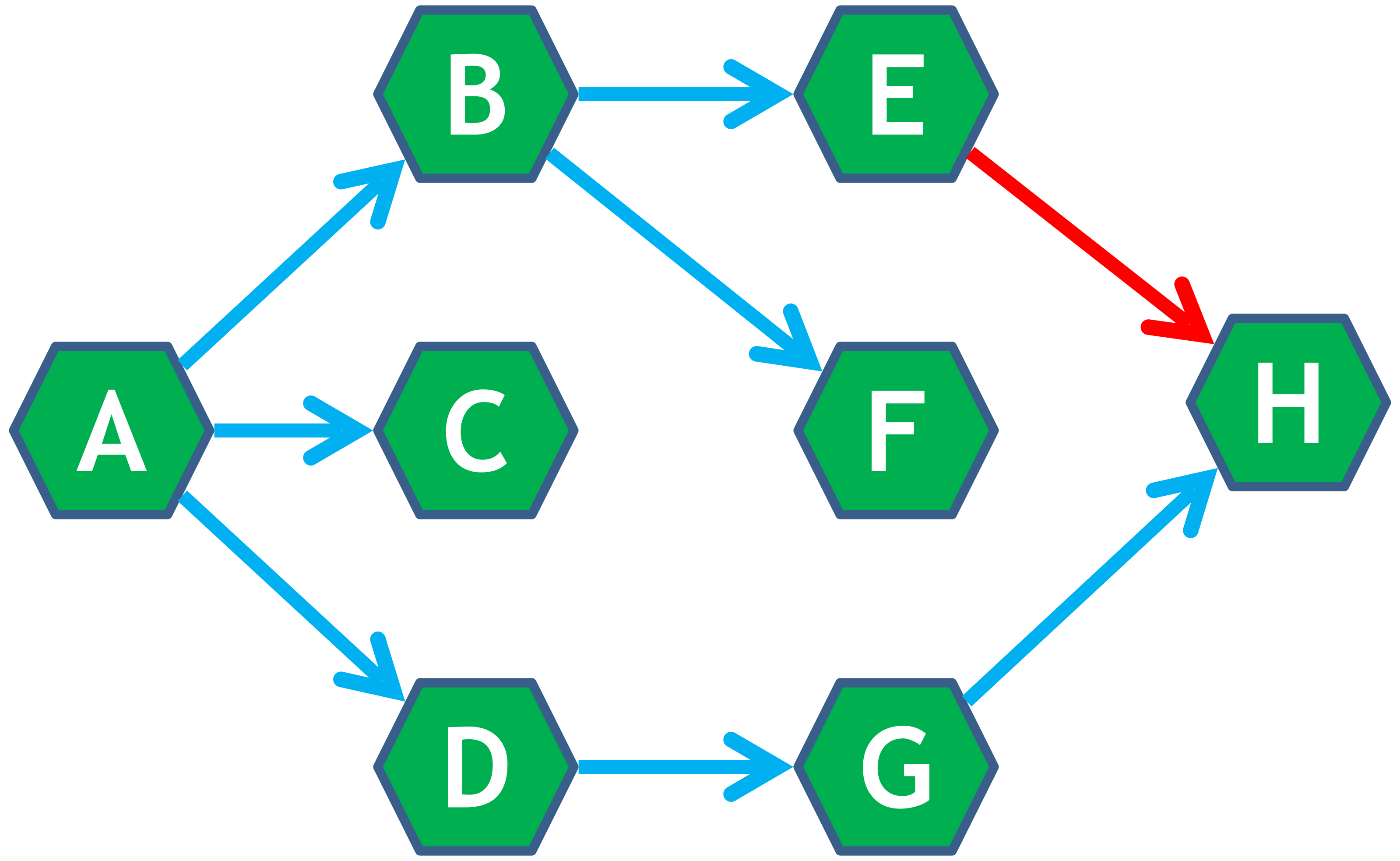
QUEUE (FIFO)

Rule applied
- Mark H as visited

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

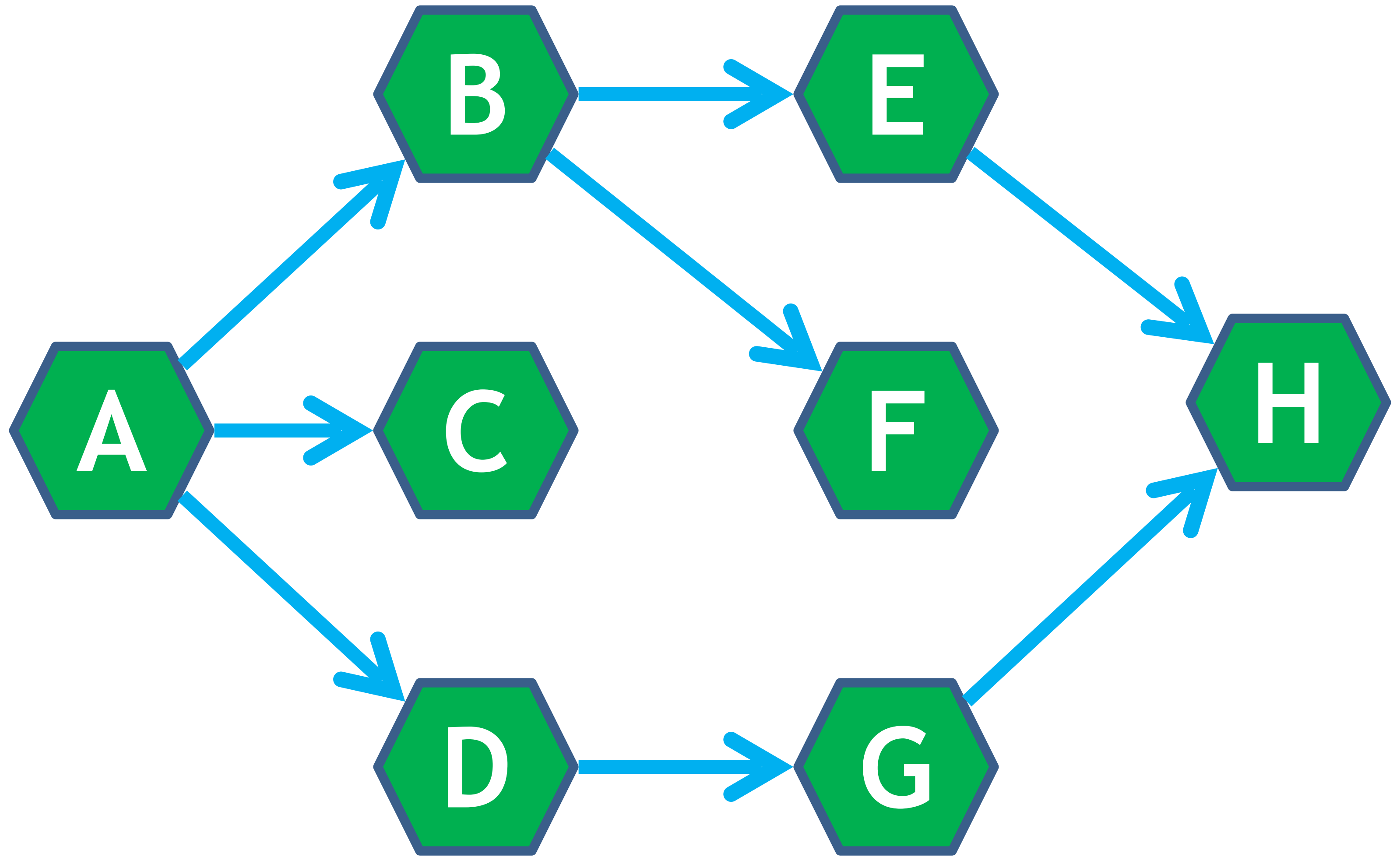
Rule applied

- Add node H into QUEUE

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

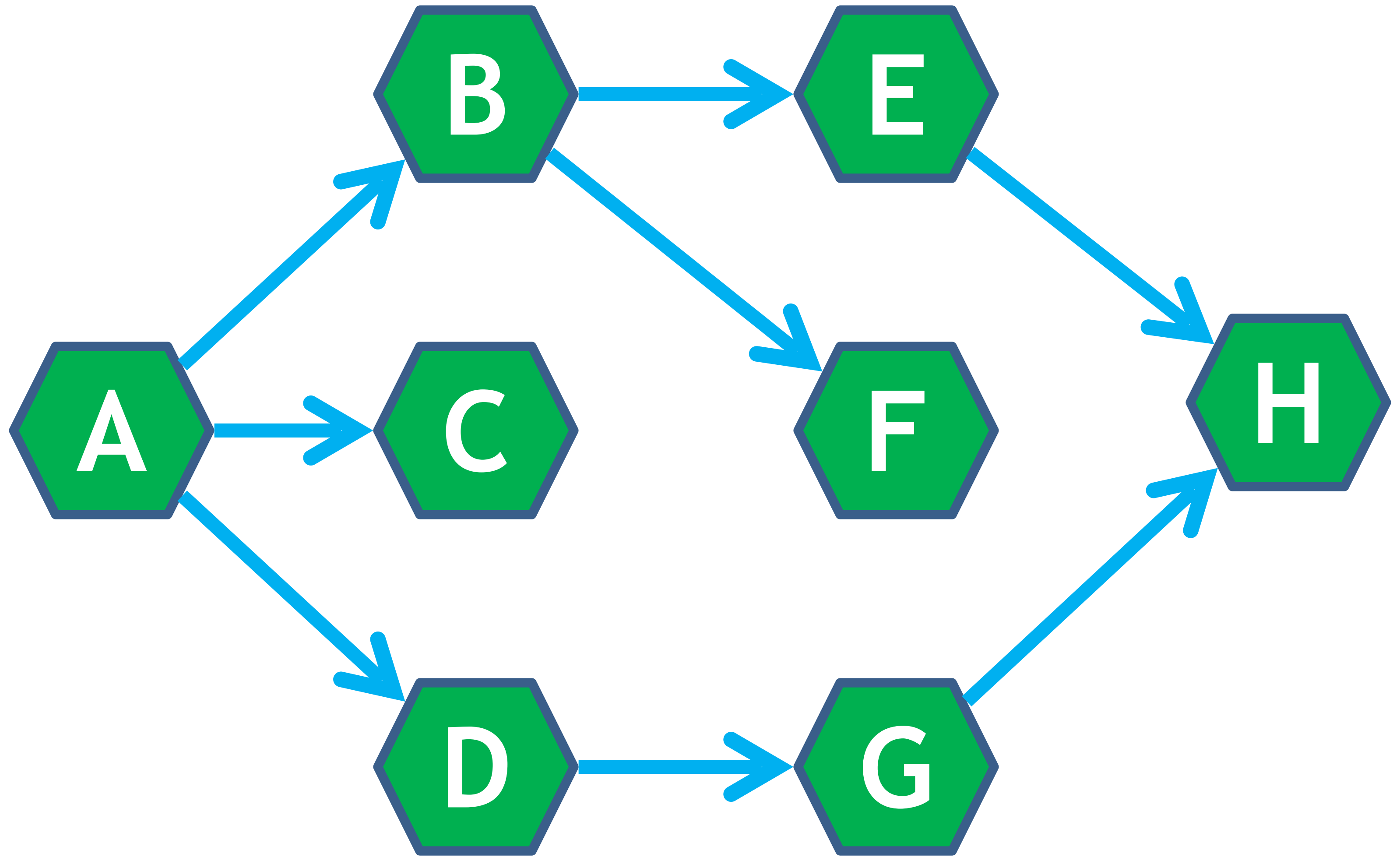
Rule applied

- Now, E doesn't have any adjacent unvisited nodes

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty



QUEUE (FIFO)

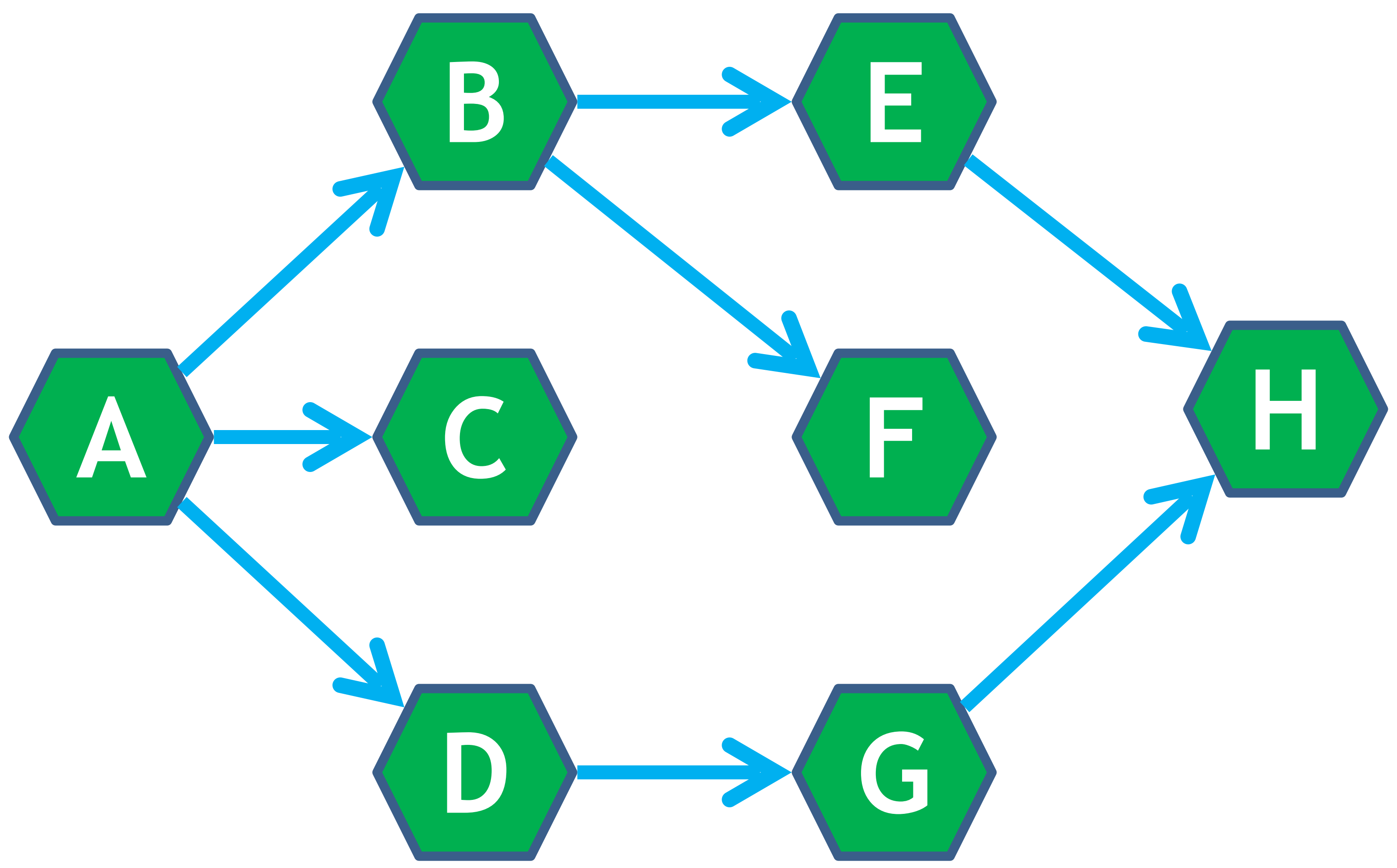
Rule applied

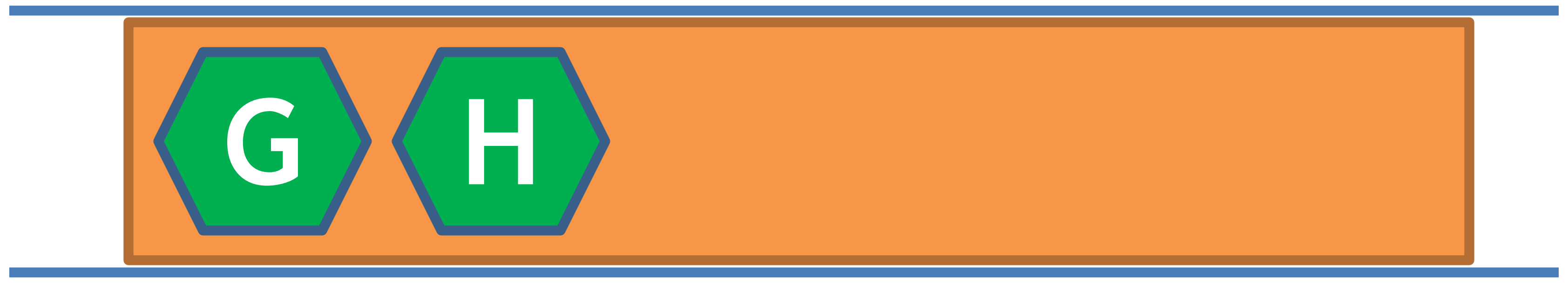
- Now, E doesn't have any adjacent unvisited nodes
  - So, de-queue E

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

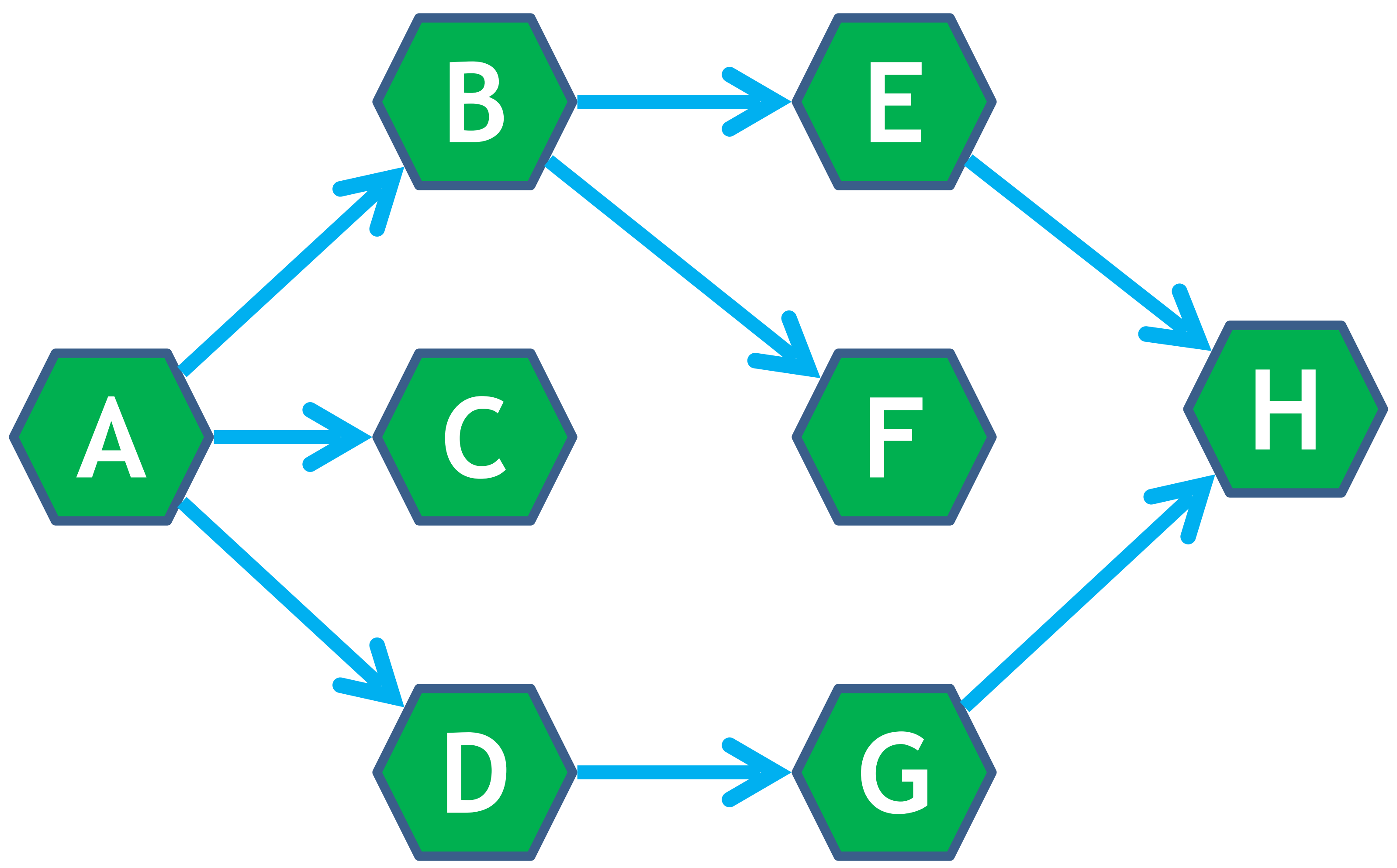
Rule applied

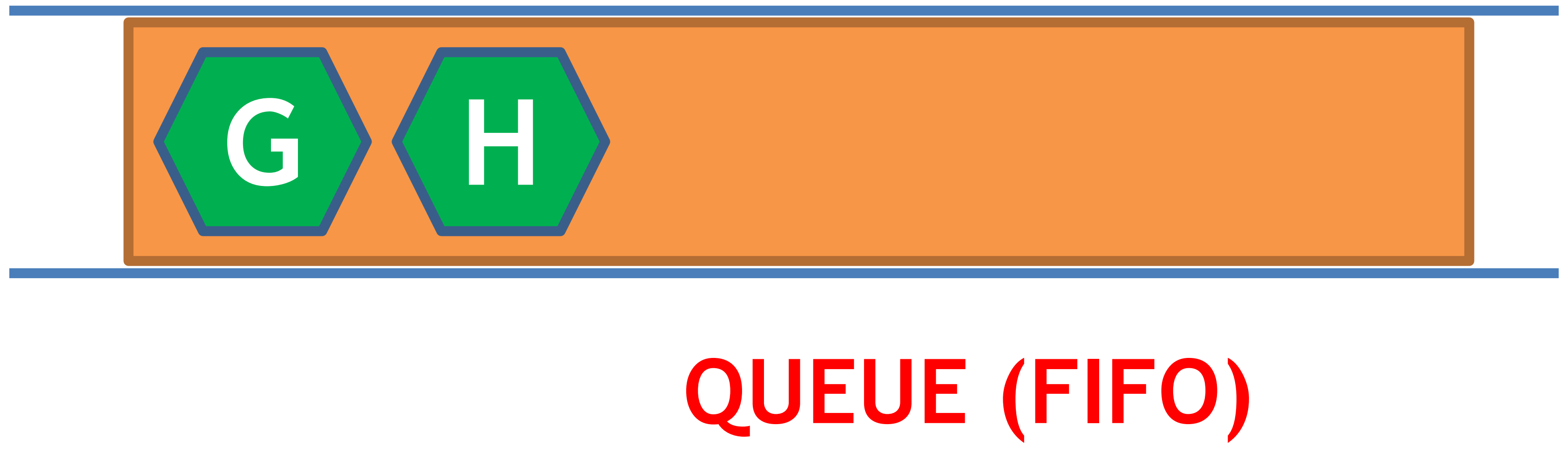
- Now, F doesn't have any adjacent unvisited nodes
  - So, de-queue F

Visited Nodes:

Breadth First Search

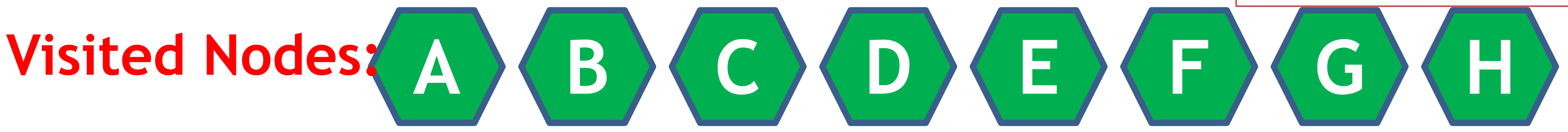
1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





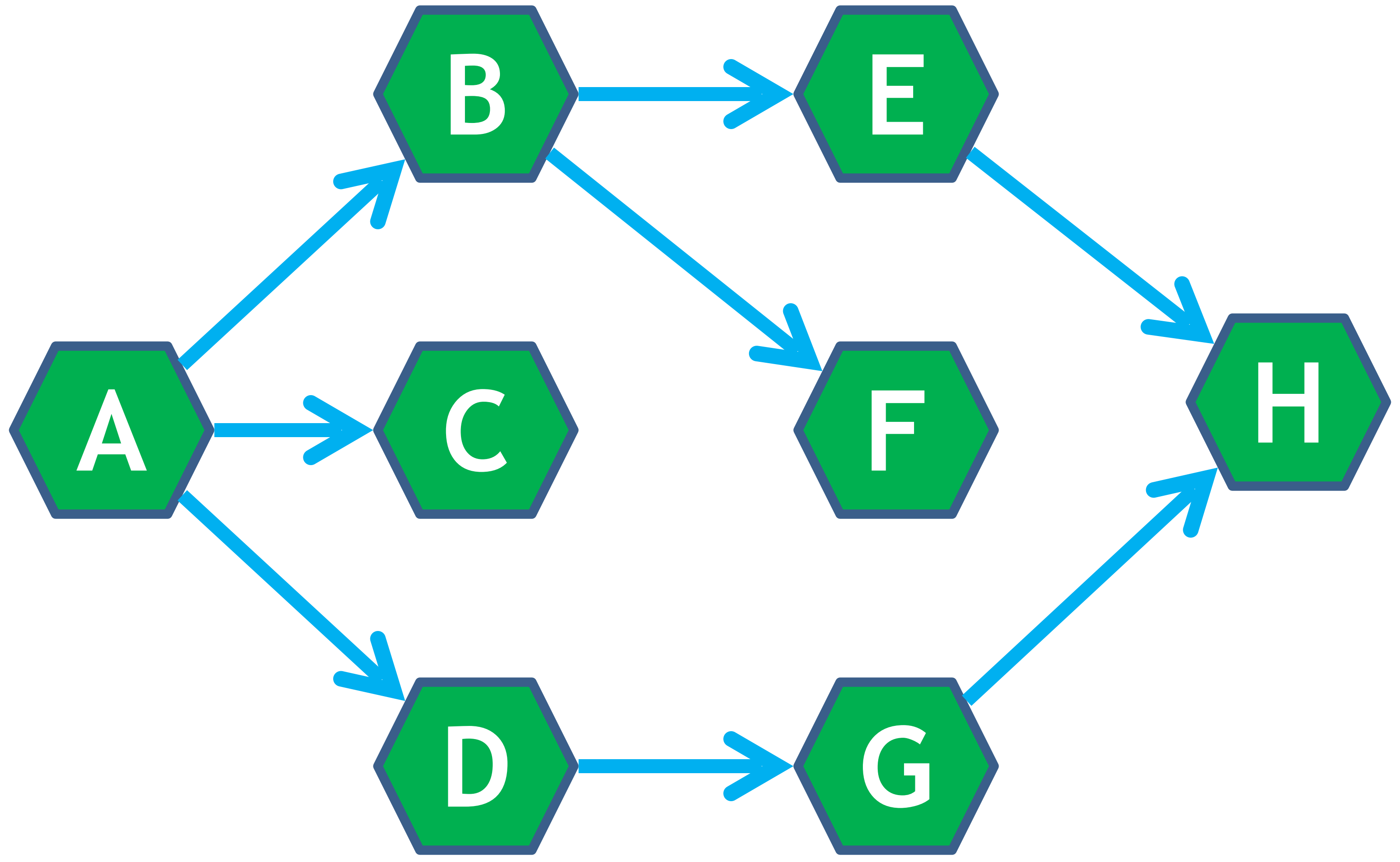
Rule applied

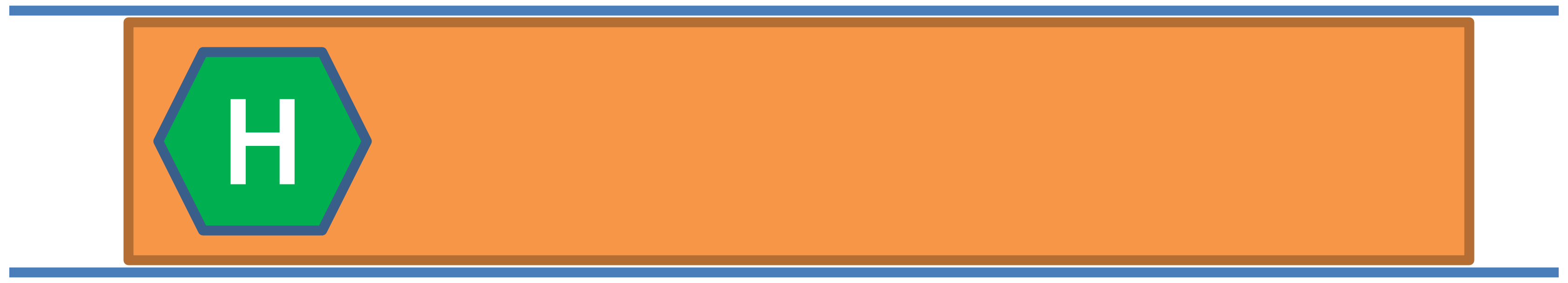
- Similarly, G and H also don't have any adjacent unvisited nodes, So, de-queue G and H



Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

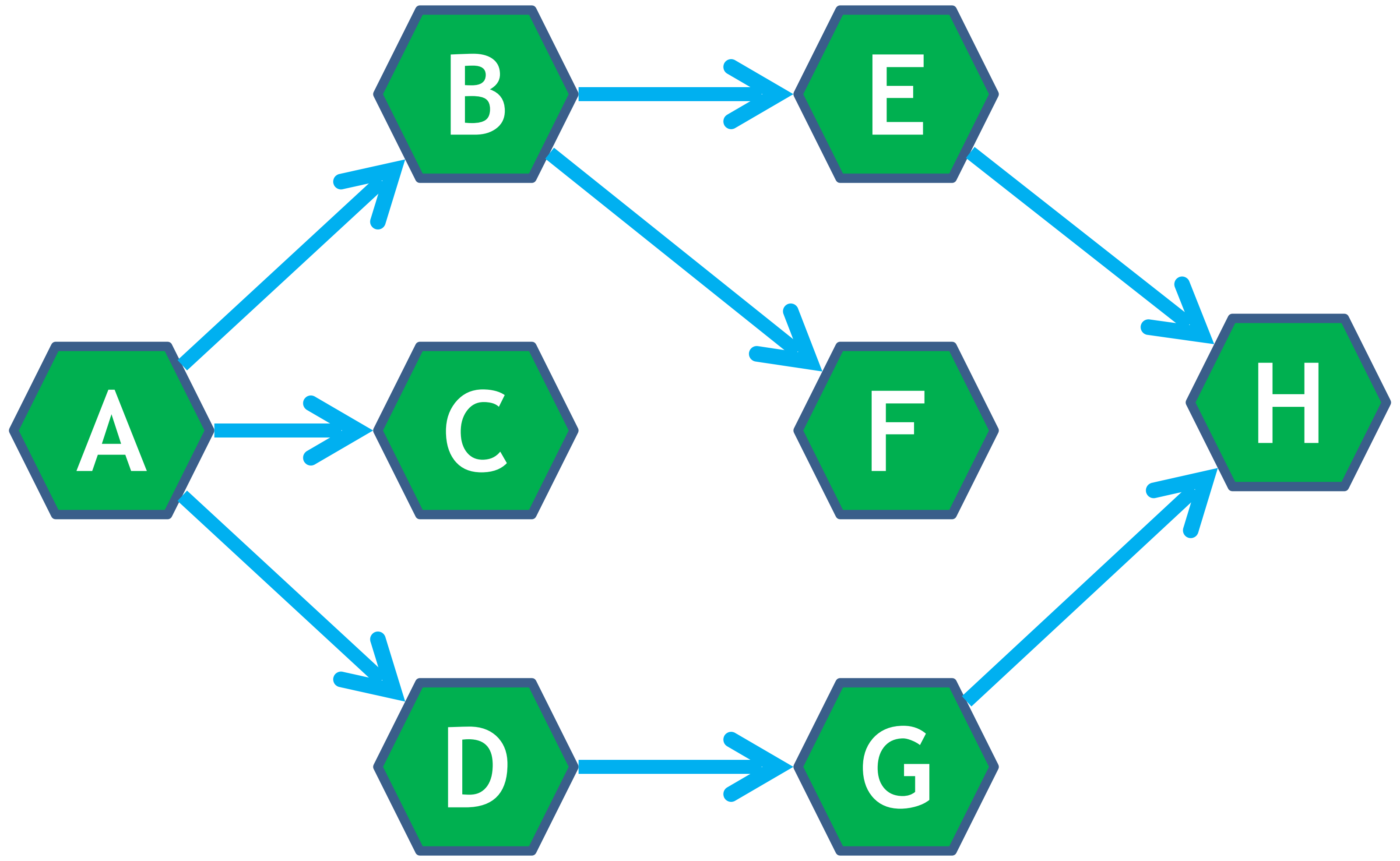
Rule applied

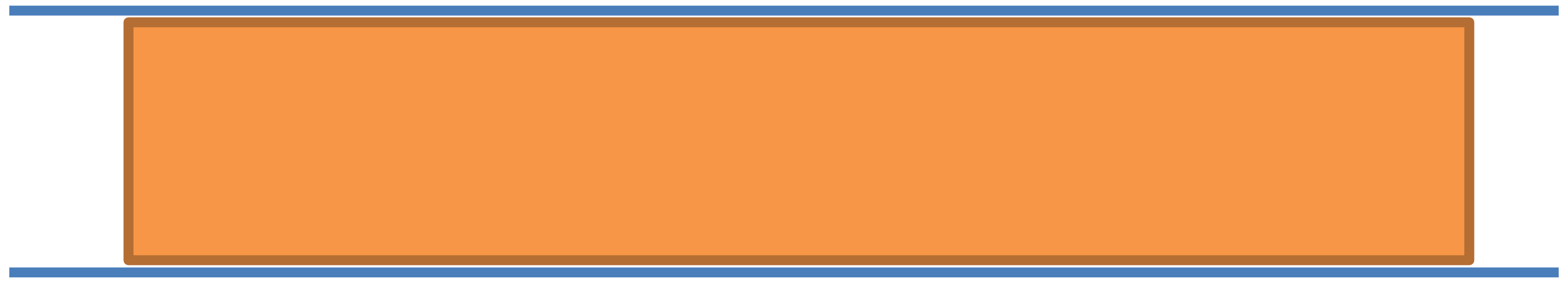
- **Similarly, G and H also don’t have any adjacent unvisited nodes, So, de-queue G and H**

Visited Nodes:

Breadth First Search

1. Visit all the adjacent unvisited nodes. Mark it as VISITED. Insert it in a queue.
2. If no adjacent node is found, de-queue the queue.
3. Repeat 1 and 2 until the queue is empty





QUEUE (FIFO)

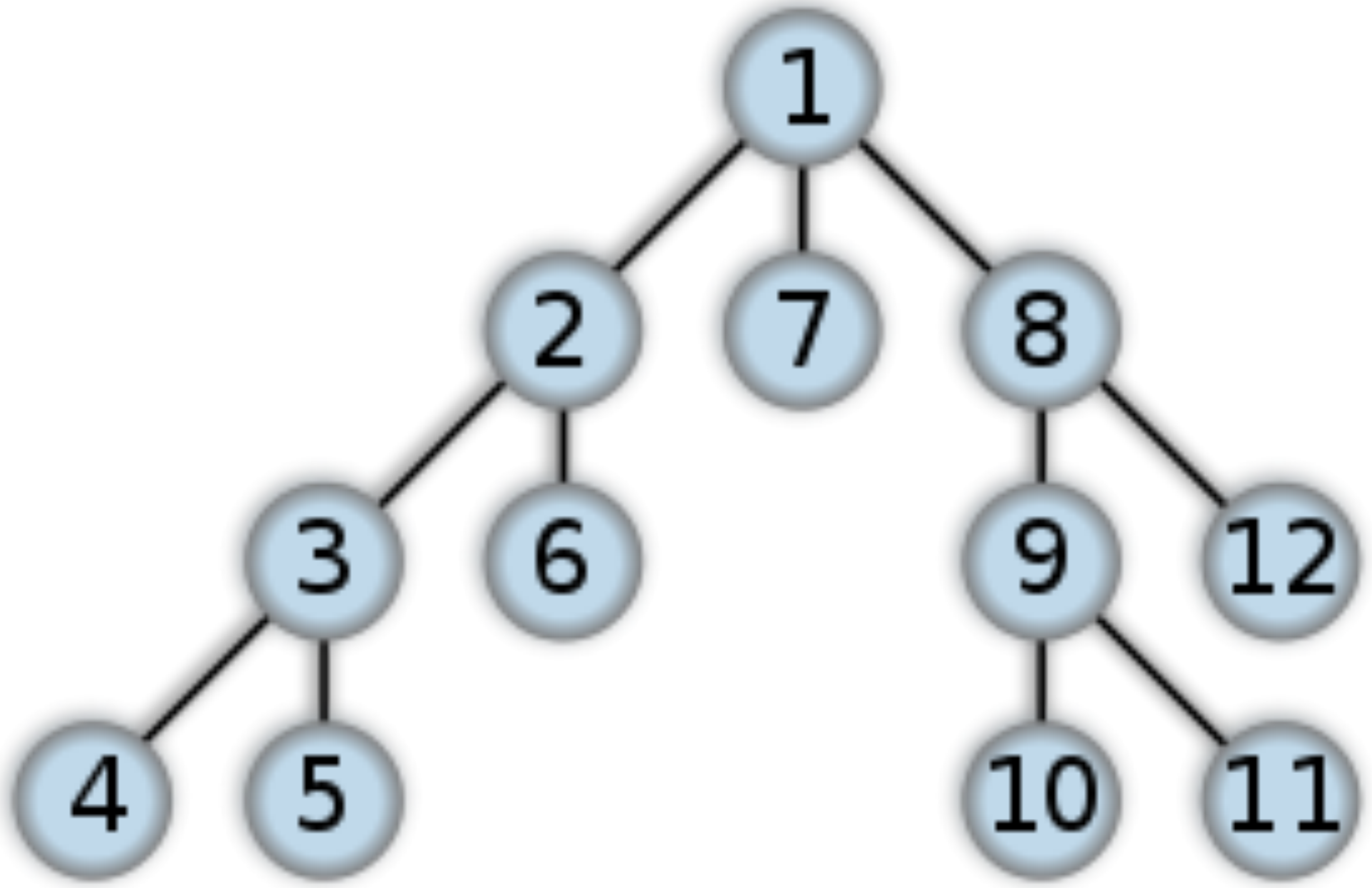
Rule applied

- Now the queue is empty
- All the nodes are visited
- Algorithm is terminated

Visited Nodes: A B C D E F G H

Breadth First Search

1. Practice: Perform BFS on the given tree with all detail steps



QUEUE (FIFO)

Rule applied

Visited Nodes:

Properties of Breadth First Search

- It is *complete*:
  - if b (max branching factor) is finite
  - *if there is a solution, BFS will find it.*
- Time
  - $1 + b + b^2 + \ldots + b^d + b(b^d-1) = O(b^{d+1})$
  - exponential in d
- Space
  - $O(b^{d+1})$
  - Keeps every node in memory
- Optimal
  - Yes (if cost is 1 per step); not optimal in general

Lessons from Breadth First Search

- The **memory requirements** are **bigger problems** for breadth-first search than its execution time
- **Exponential-complexity search problems** cannot be solved by un-informed methods for any but the smallest instances

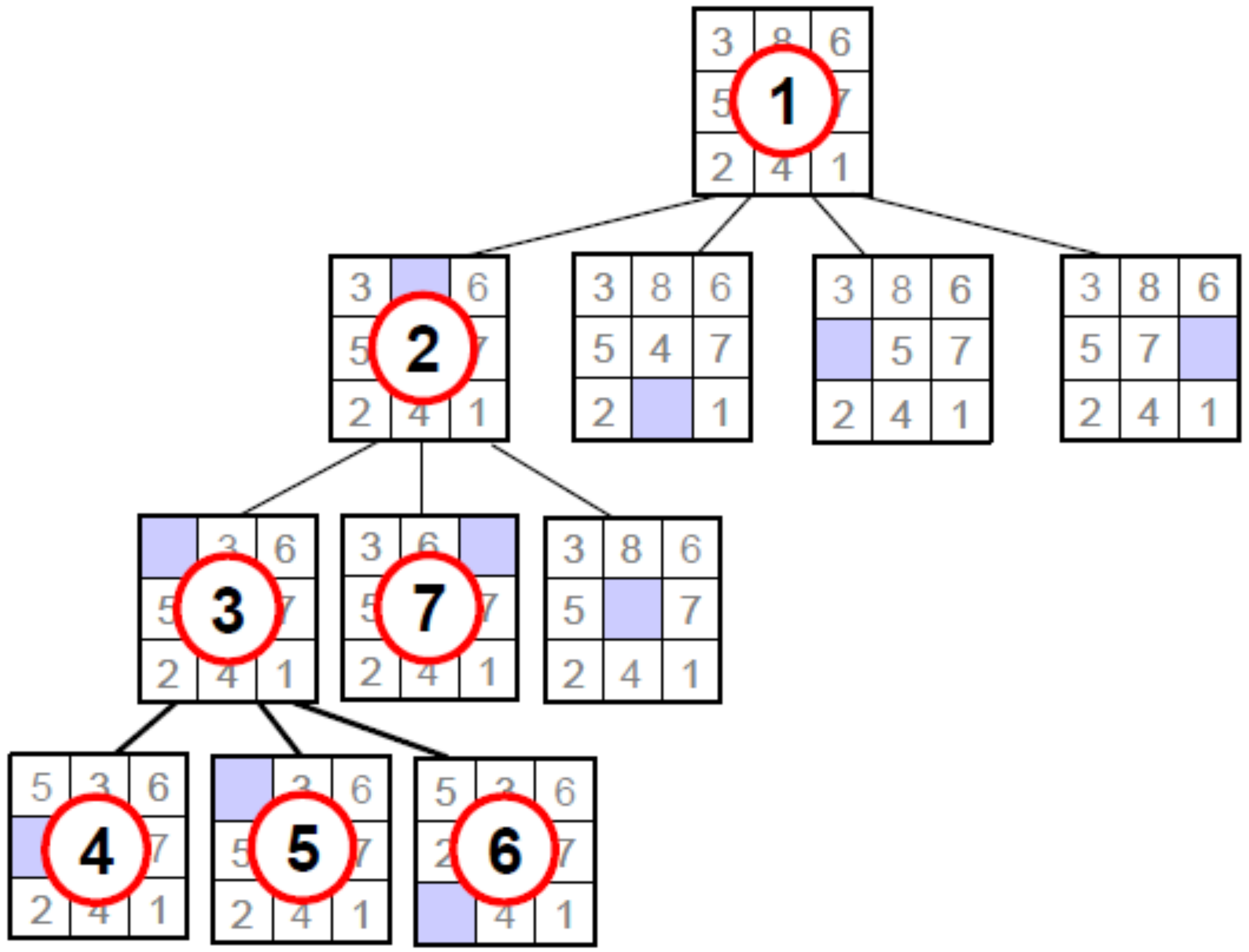
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

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

An exponential complexity $O(b^d)$ is scary

Depth First Search

- Expand the ***deepest*** unexpanded node
- Implementation- Using **stack**
- Unexplored successors are placed on a stack until fully explored



Depth First Search

- Algorithm:
  1. Visit the adjacent unvisited nodes. Mark it as visited. Push into stack.
  2. If no adjacent node is found, pop up a node from the stack.
  3. Repeat steps 1 and 2 until the stack is empty.

Depth First Search

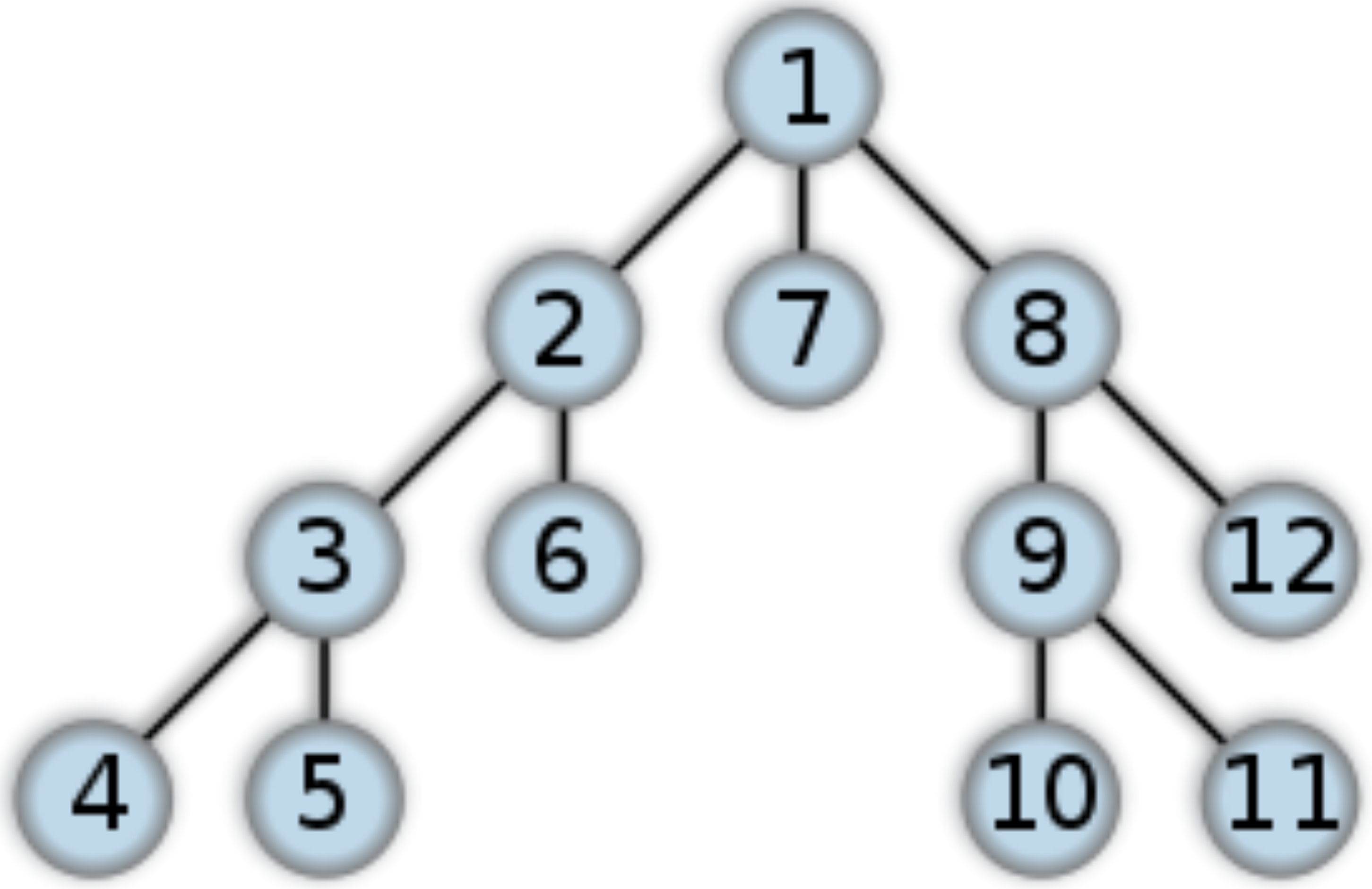
Link: https://www.youtube.com/watch?v=oL5J5il9pFg&t=528s

Depth-First Search

- **Depth-first search (DFS)** is an algorithm for traversing or searching a finite graph data structures.
- Search starts from the root(choose any random vertex) and explores as far as possible along each branch before backtracking.
- DFS uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Depth First Search

1. Practice: Perform DFS on the given tree with all detail steps



STACK

Rule applied

Visited Nodes:

Depth First Search

- Complete
  - No: fails in infinite-depth spaces, spaces with loops
    - Modify to avoid repeated spaces along path
  - Yes: in finite spaces

- Time
  - $O(b^m)$
  - Not great if m is much larger than d
  - But if the solutions are dense, this may be faster than breadth-first search

- Space
  - $O(bm)$…linear space

- Optimal
  - No

Iterative deepening Search

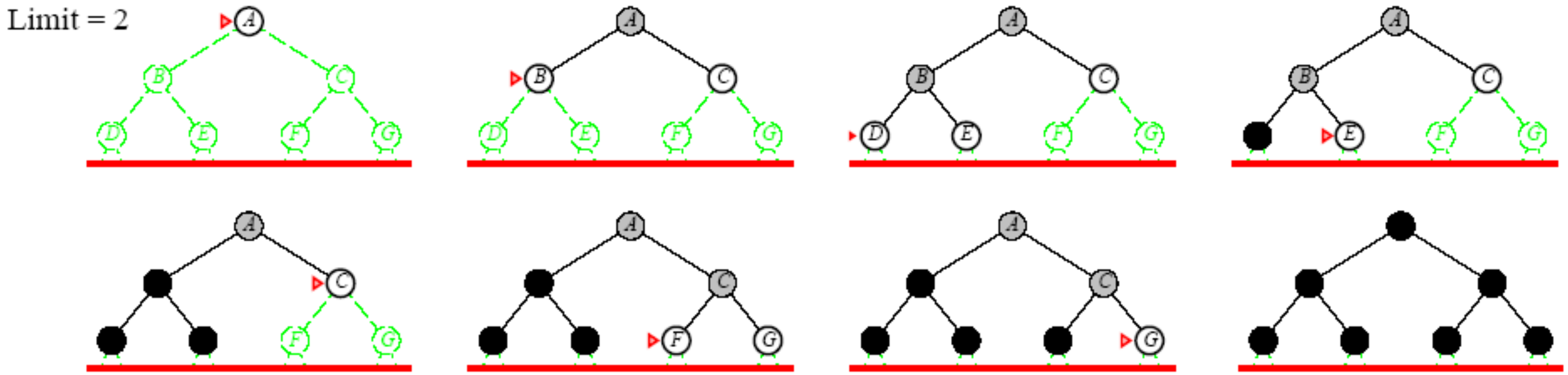
- Iterative deepening search
  - Uses depth-first search
  - Finds the best depth limit by
    - Gradually increases the depth limit; 0, 1, 2, … until a goal is found
    - It provides breadth-first search property

Iterative deepening Search

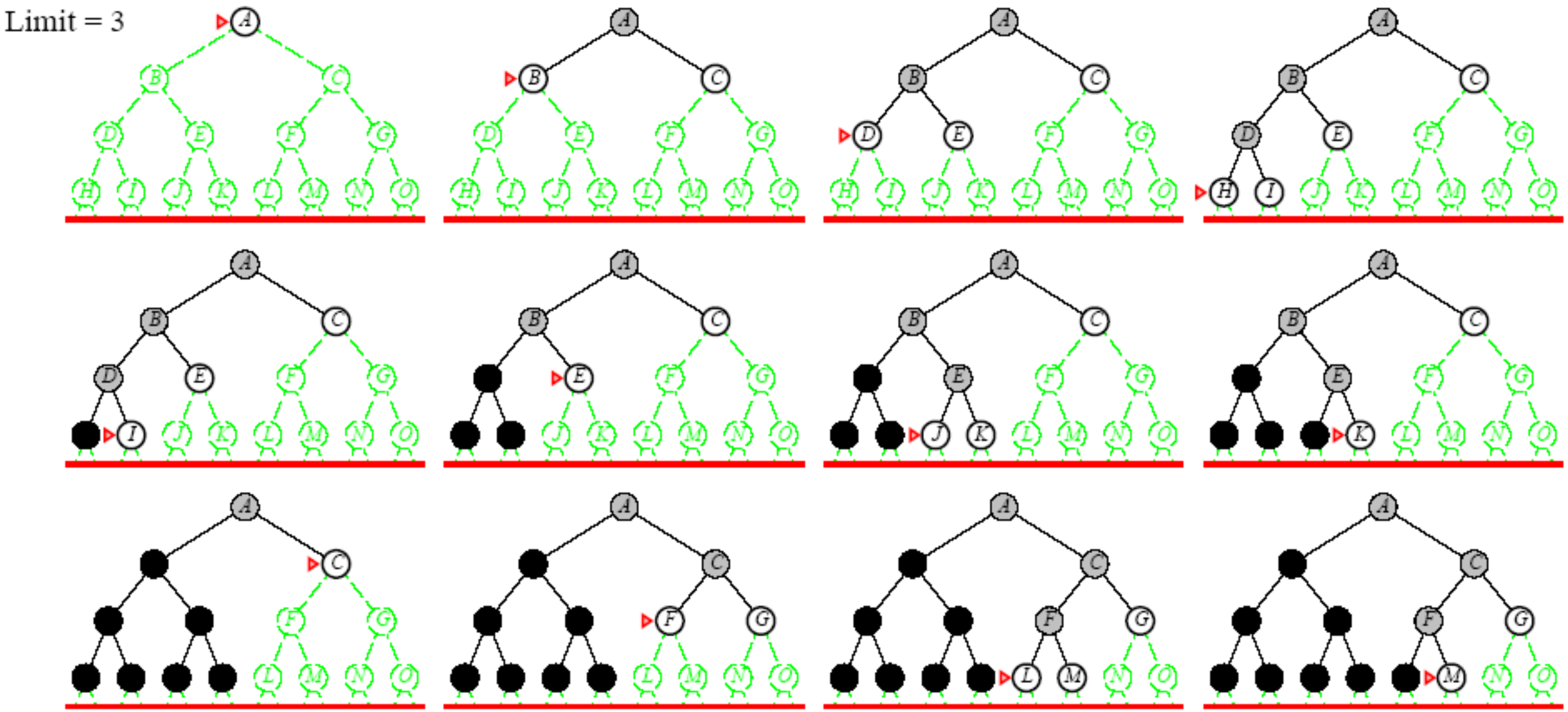
Limit = 0

Iterative deepening Search

Iterative deepening Search

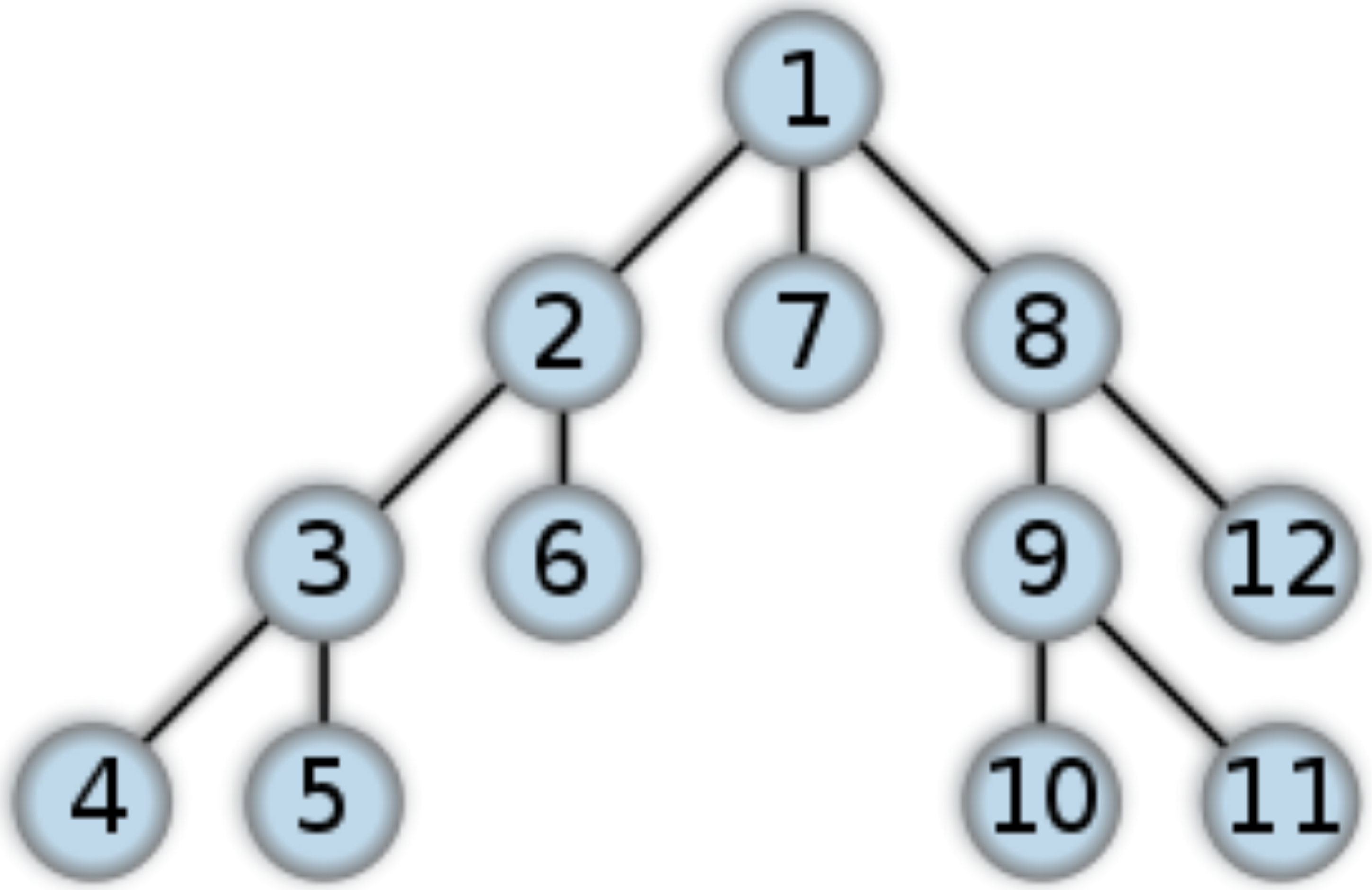


Iterative deepening Search



Iterative deepening Search

1. Practice: Perform **Iterative Deepening Search** on the given tree with all possible depth limit.



Visited Nodes:

Iterative deepening Search

- Combines the benefits of depth-first and breadth-first search
  - Like depth-first search, its memory requirements are modest: O(bd)
  - Like breadth-first search, it is
    - complete when the branching factor is finite and
    - optimal when the path cost is a non-decreasing function of the depth of the node.
  - time complexity: $O(b^d)$

Iterative deepening Search

- In general, iterative deepening is preferred
  - When the search space is large and
  - When the depth of the solution is not known.

Comparing Uninformed Search Strategies

Criterion	Breadth-First	Depth-First	Iterative Deepening
Complete?	Yes	No	Yes
Time	$O(b^d)$	$O(b^m)$	$O(b^d)$
Space	$O(b^d)$	$O(bm)$	$O(bd)$
Optimal?	Yes	No	Yes

Where,

- *b*: maximum branching factor (average number of child nodes for a given node)
- *d*: depth of the least-cost solution
- *m*: maximum depth of the state space (tree) (may be ∞)

Can we do better?

- Yes!

- BFS, DFS, and IDS are all examples of *uninformed search* algorithms –
  – they always consider the states in a certain order, without regard to how close a given state is to the goal.

- There exist other *informed search algorithms that*
  – *consider* (estimates of) how close a state is from the goal when deciding which state to consider next.

Informed Search Strategies

- uses **problem-specific knowledge**
  - beyond the definition of the problem itself

- can find solutions
  - **more efficiently** than uninformed strategy

- Some informed search strategies
  - Greedy Best first search
  - A* search

Informed Search Strategies

- a node is selected for expansion based on an **evaluation function, f(n)**

- The evaluation function is construed as a **cost estimate**
  - A node with the ***lowest evaluation*** *is expanded first*.

Informed Search Strategies

- The choice of evaluation function **f(n)** determines the search strategy.

  - Each **f(n)** has a heuristic function **h(n)**
    - The **heuristic function** is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbor of a node will lead to a goal with a lowest cost.

  - **h(n)** = estimated ***cost of the cheapest path*** from the state at **node *n*** *to a* ***goal state***

  - if n is a goal node, then **h(n)=0.**

Informed Search Strategies

- Evaluation function:

$$f(n) = g(n) + h(n)$$

- **g(n)** = cost from the initial state to the current state n

- **h(n)** = estimated cost of the cheapest path from node n to a goal node

    - If $f(n) = g(n) + h(n)$. ——> A* search
    - If $f(n) = h(n)$ ——> best first search

Greedy Best First Search

- tries to expand the node
  - that is closest to the goal
  - assuming it will lead to a solution quickly

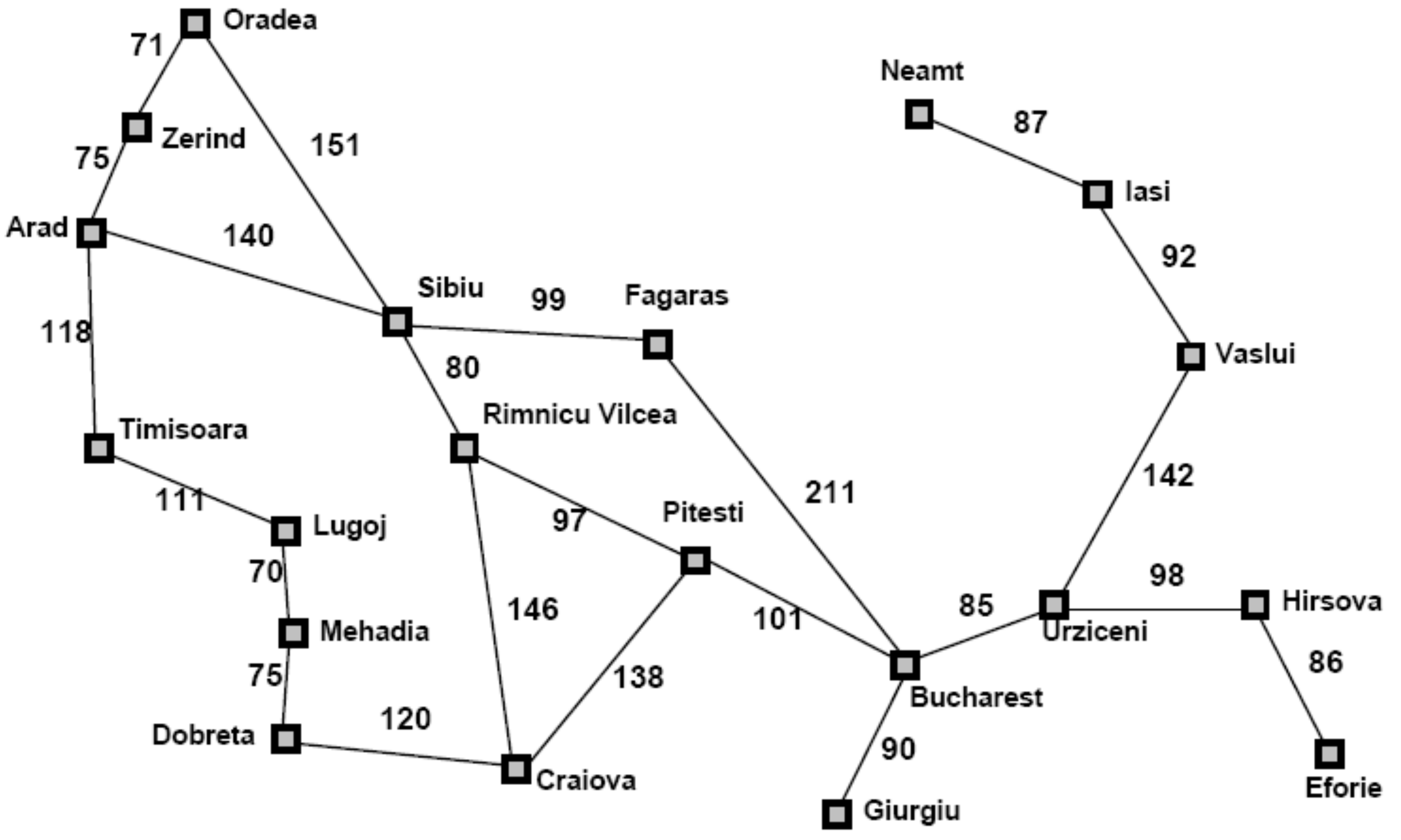
- evaluates nodes by using just the heuristic function

$$f(n) = h(n)$$

where, $h(n) = h_{SLD}$ = the straight-line distance heuristic

Greedy Best First Search

Suppose you are at **Arad** and you want to go to **Bucharest**.

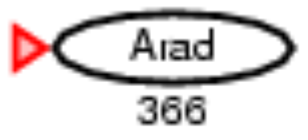


Map of Romania

Heuristic, h(n)

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

Greedy Best First Search



Heuristic, h(n)

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

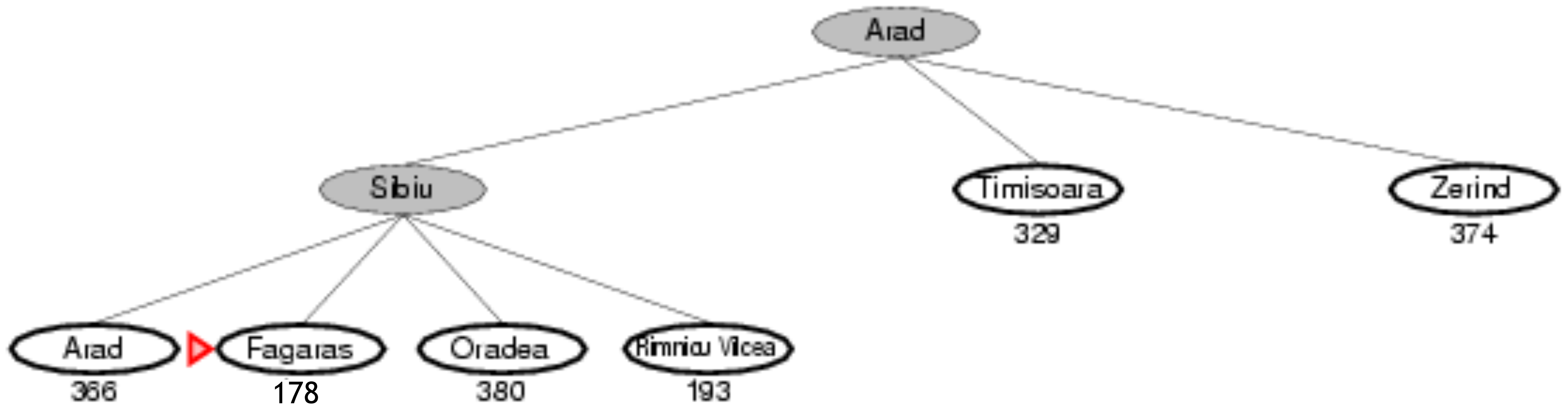
Greedy Best First Search

Heuristic, h(n)

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

Greedy Best First Search

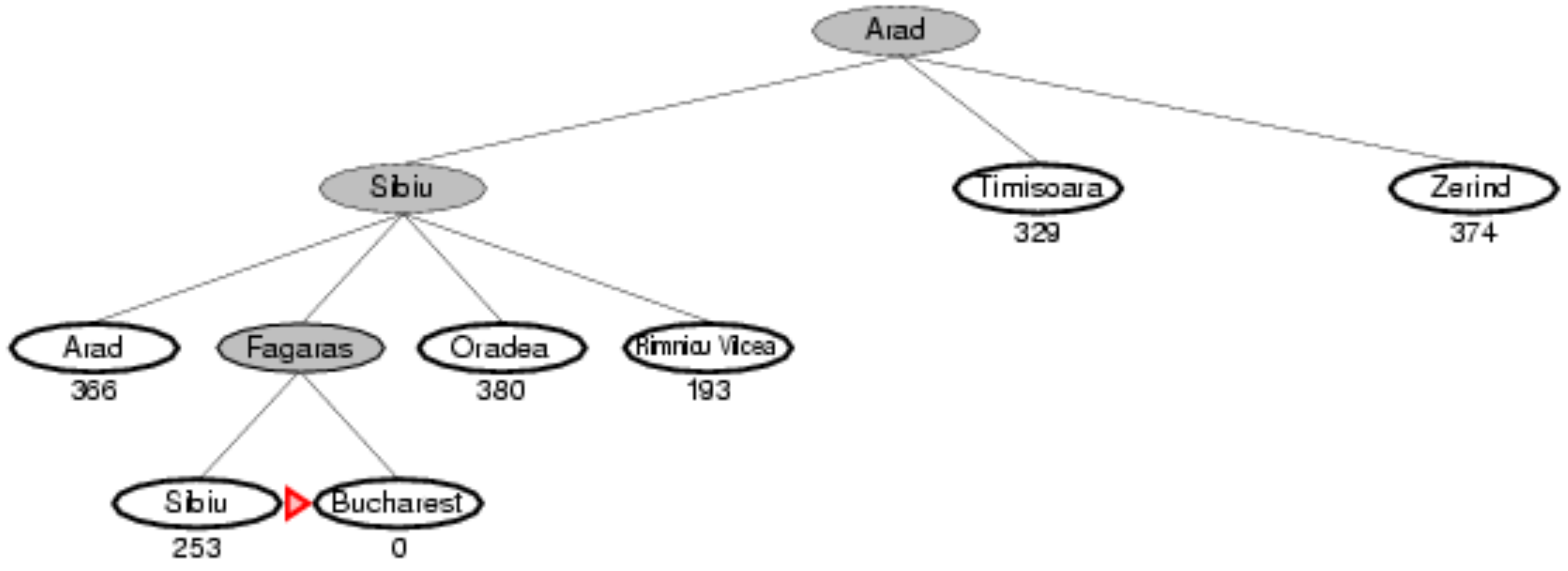


Heuristic, h(n)

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

Greedy Best First Search



Heuristic, h(n)

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

Greedy Best First Search

- Complete
  - No, GBFS can get stuck in loops (e.g. bouncing back and forth between cities)
  - e.g., Iasi → Neamt → Iasi → Neamt →
- Time
  - $O(b^m)$ but a good heuristic can have dramatic improvement
- Space
  - $O(b^m)$ – keeps all the nodes in memory
- Optimal
  - No!

A* Search

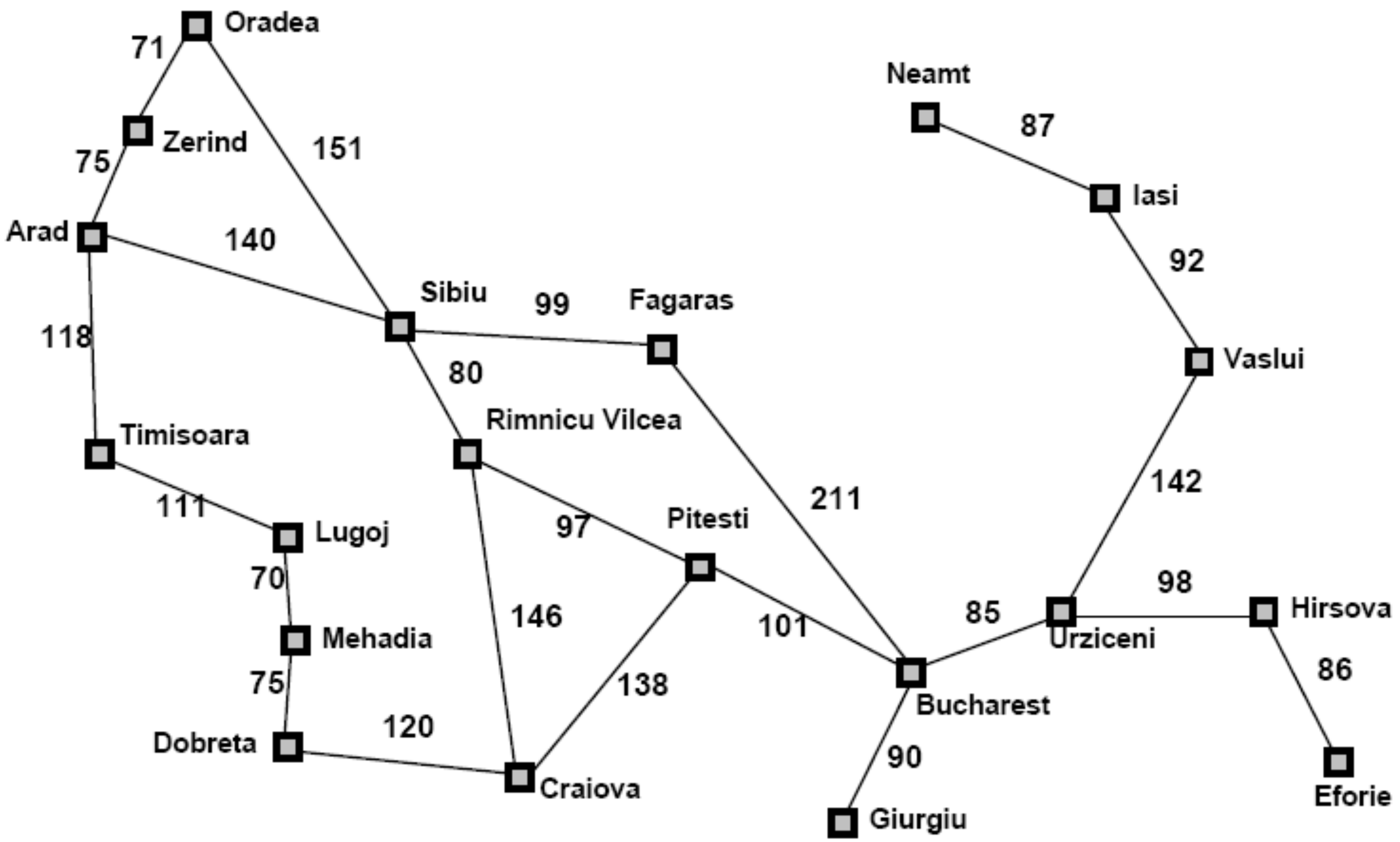
- Pronounced as “A-star search”
- Idea: avoid expanding paths that are already expensive
- Evaluation function:

$$f(n) = g(n) + h(n)$$

Where,

- $g(n)$ = cost so far to reach n (from start node)
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal

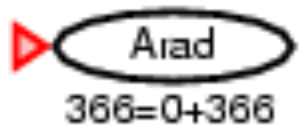
A* Search



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

A* Search

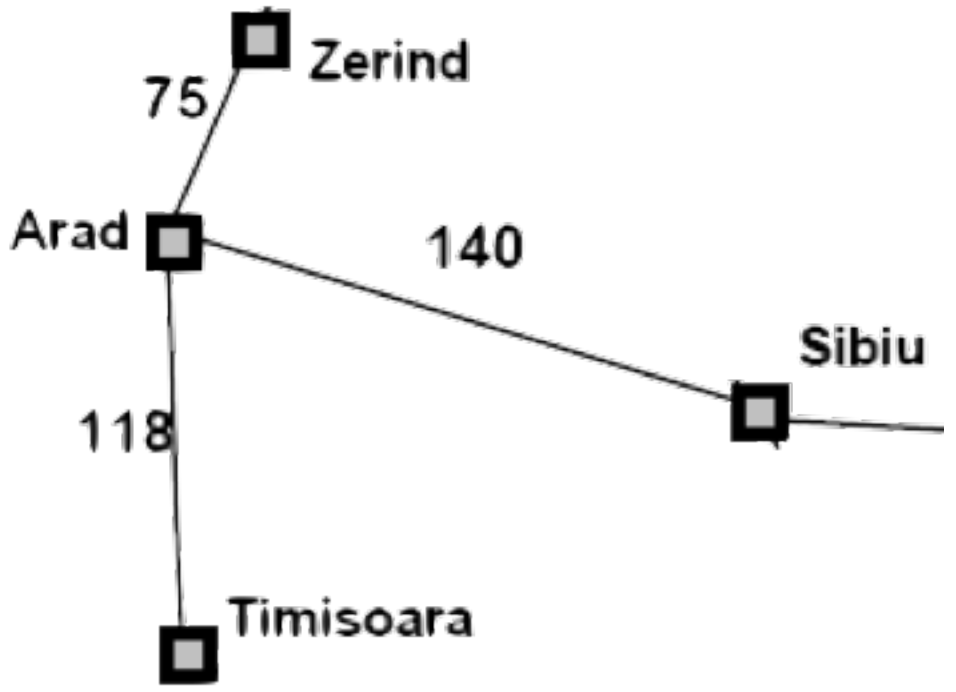


Heuristic, h(n)

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

A* Search

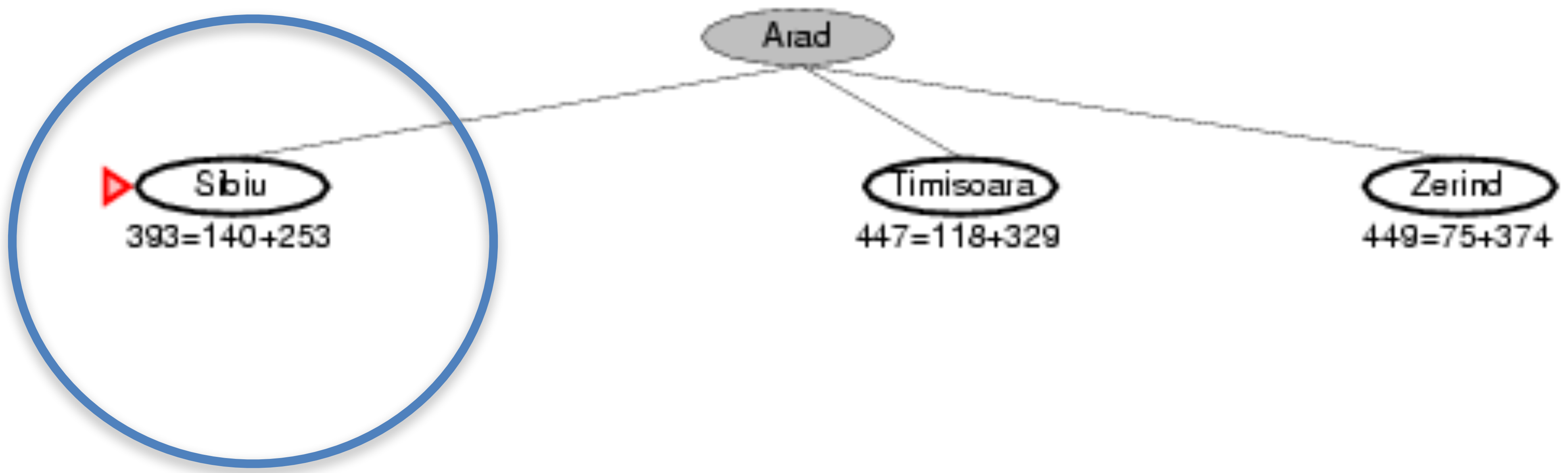


Heuristic, h(n)

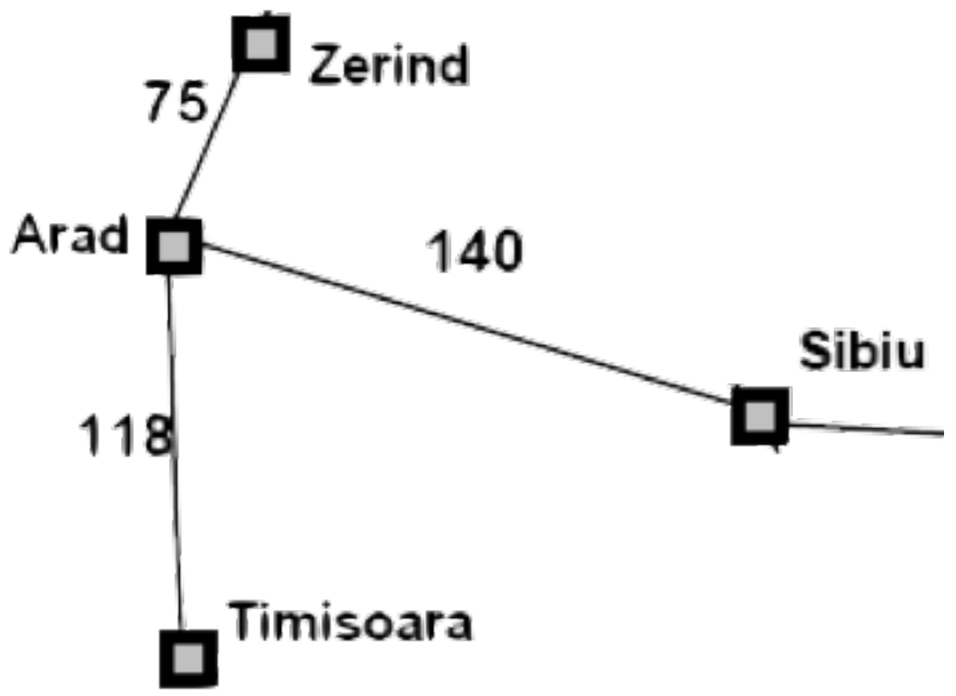
Straight−line distance
to Bucharest

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

A* Search



$$f(n) = g(n) + h(n)$$
$$= 140 + 253$$
$$= 393$$

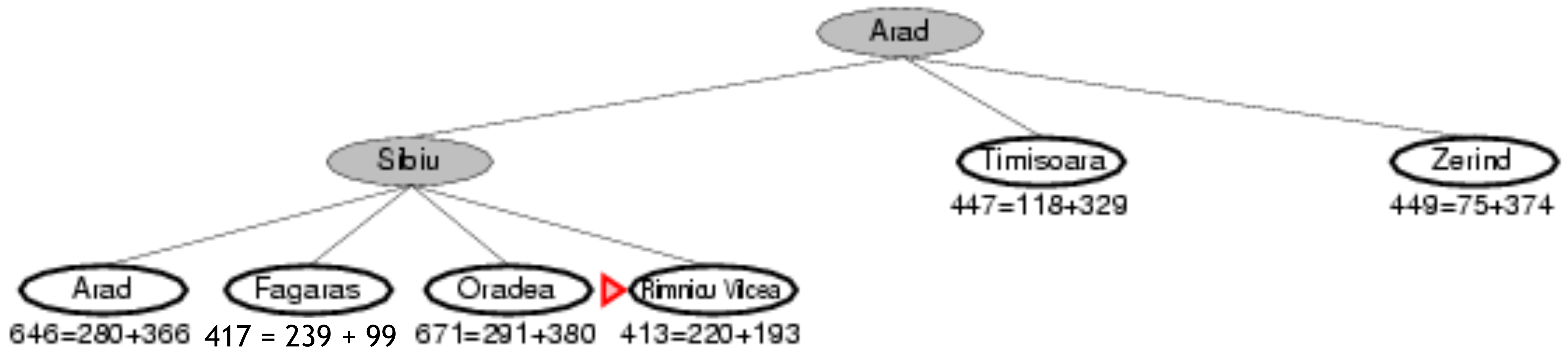


Heuristic, h(n)

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

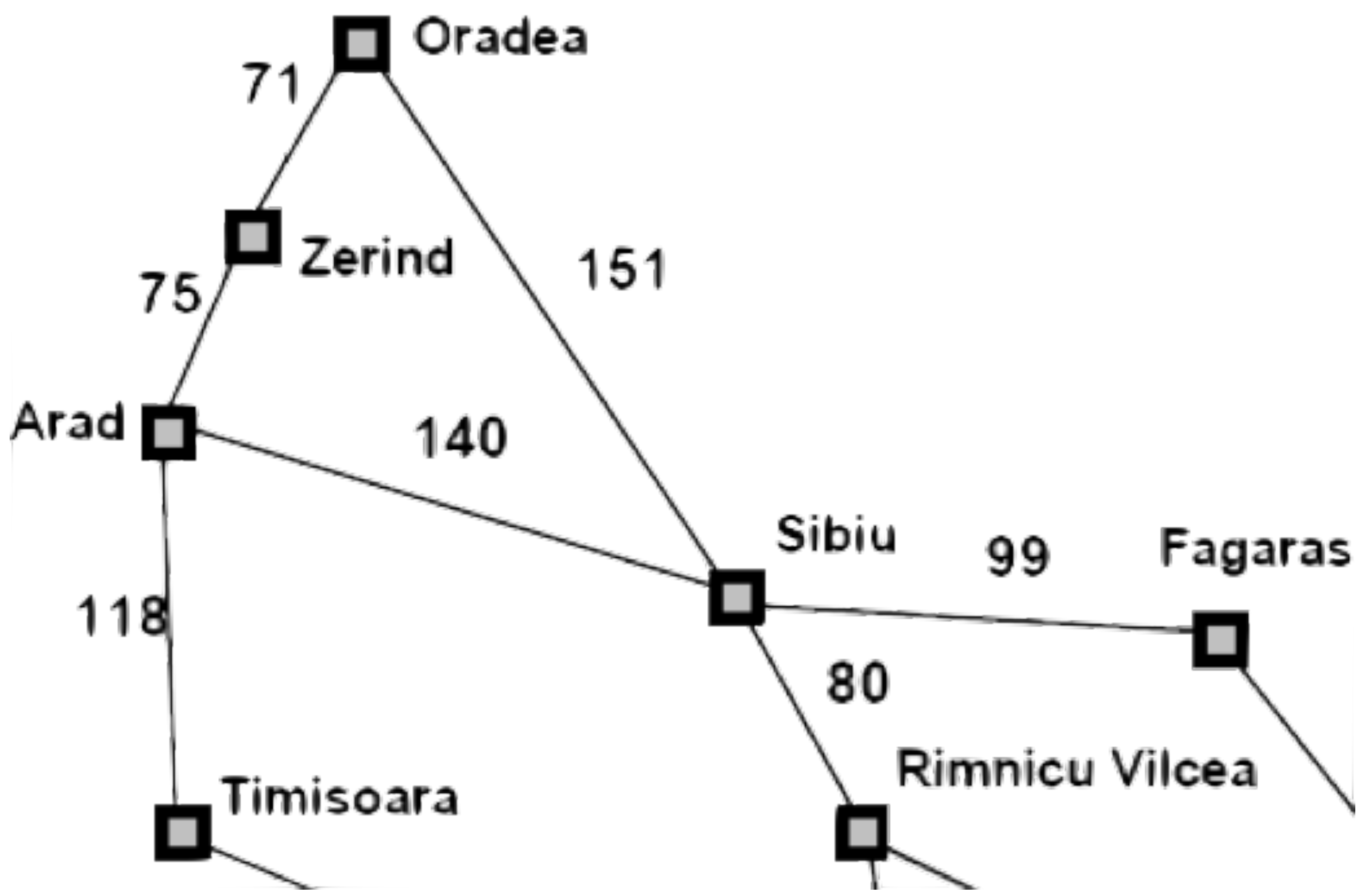
A* Search



For Arad:
f(n) = g(n) + h(n)
= (140 +140) + 366
= 646

For Fagaras:
f(n) = (140 +99) + 178
= 417

For **Rimnicu Vilcea:**
f(n) = (140 + 80) + 178
= 413

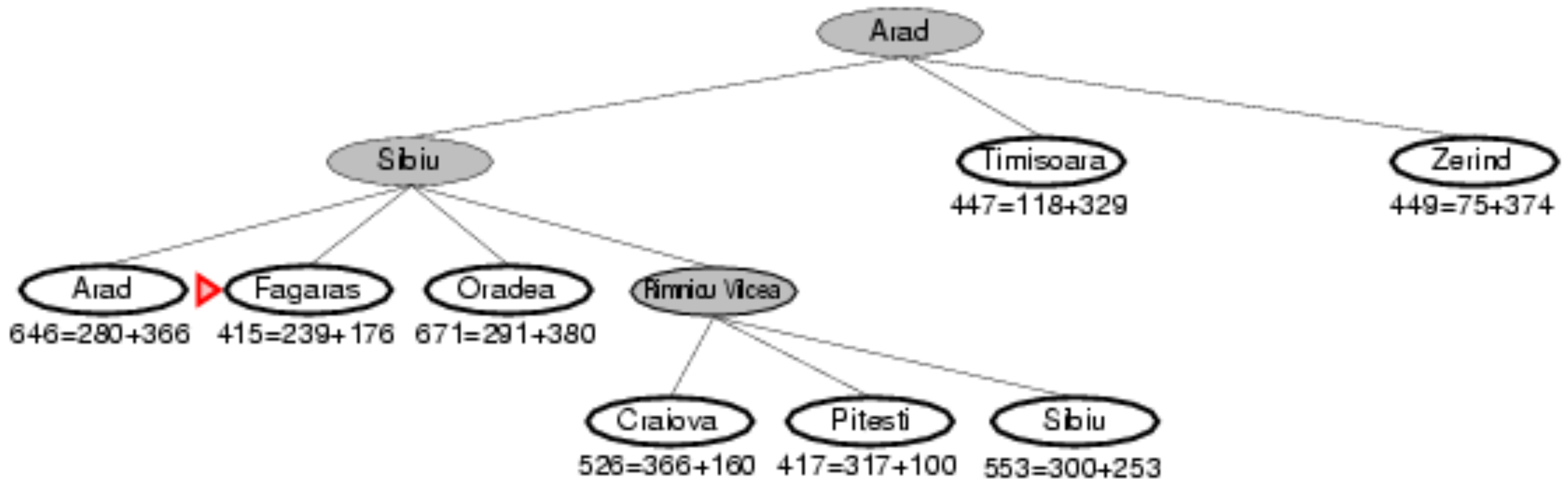


Heuristic, h(n)

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

A* Search

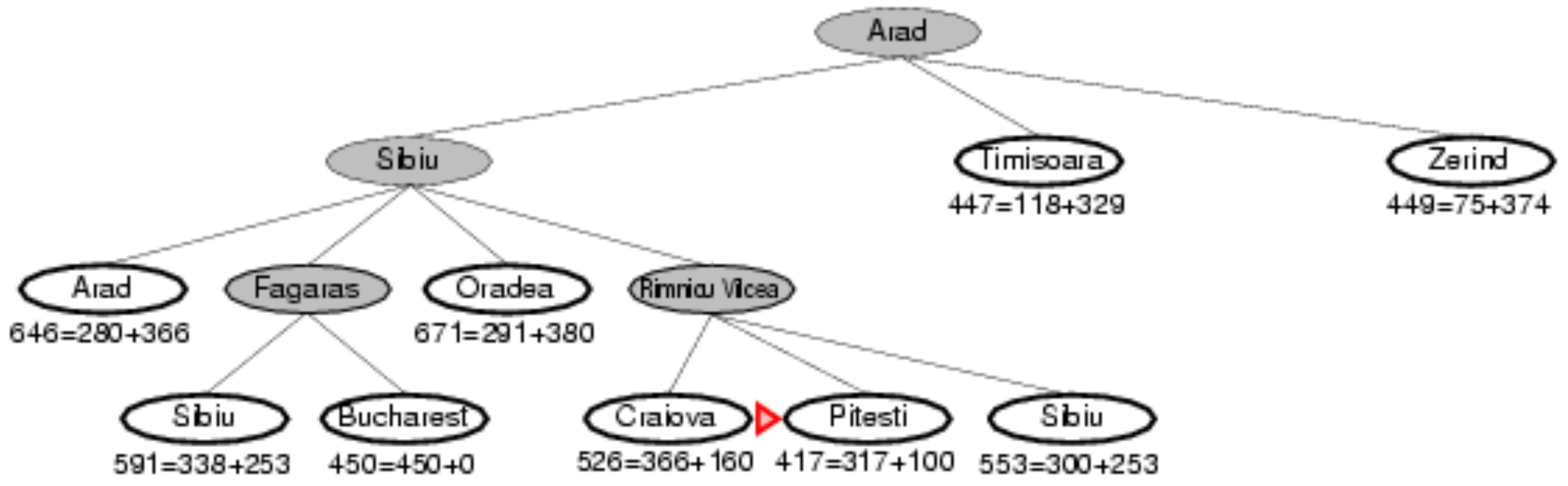


Heuristic, h(n)

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

A* Search

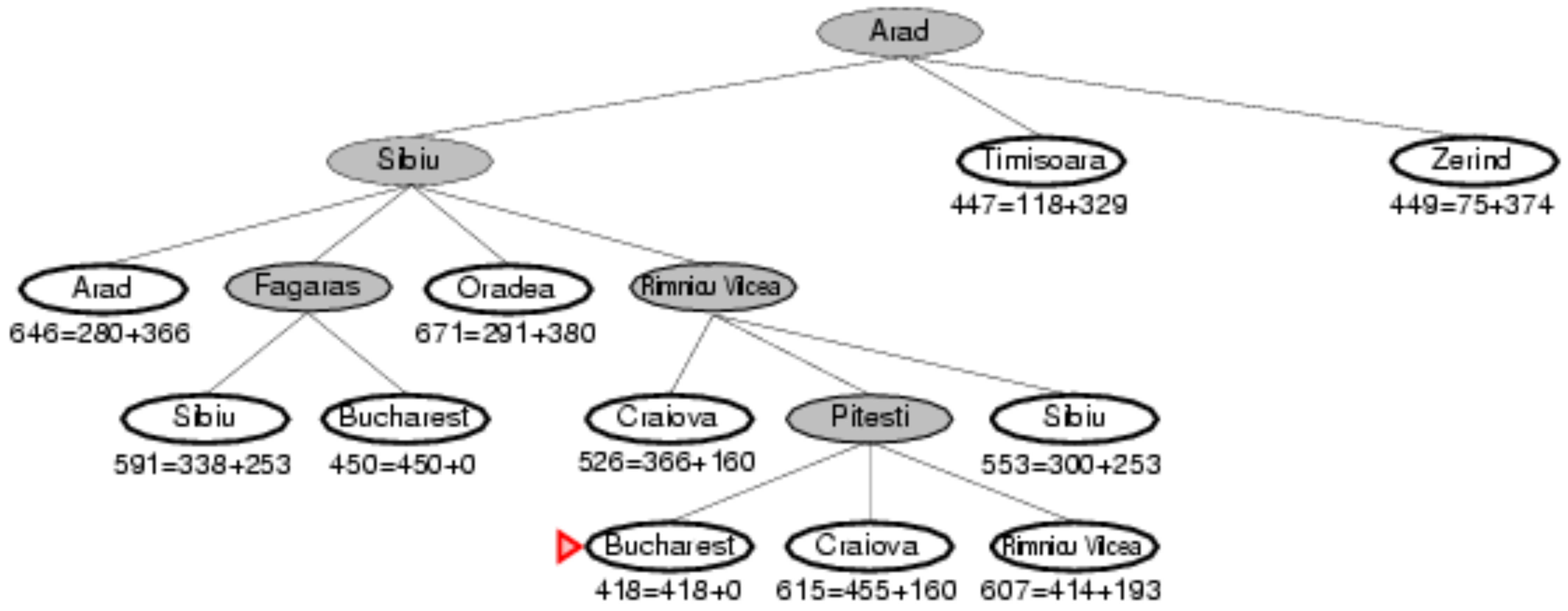


Heuristic, h(n)

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

A* Search

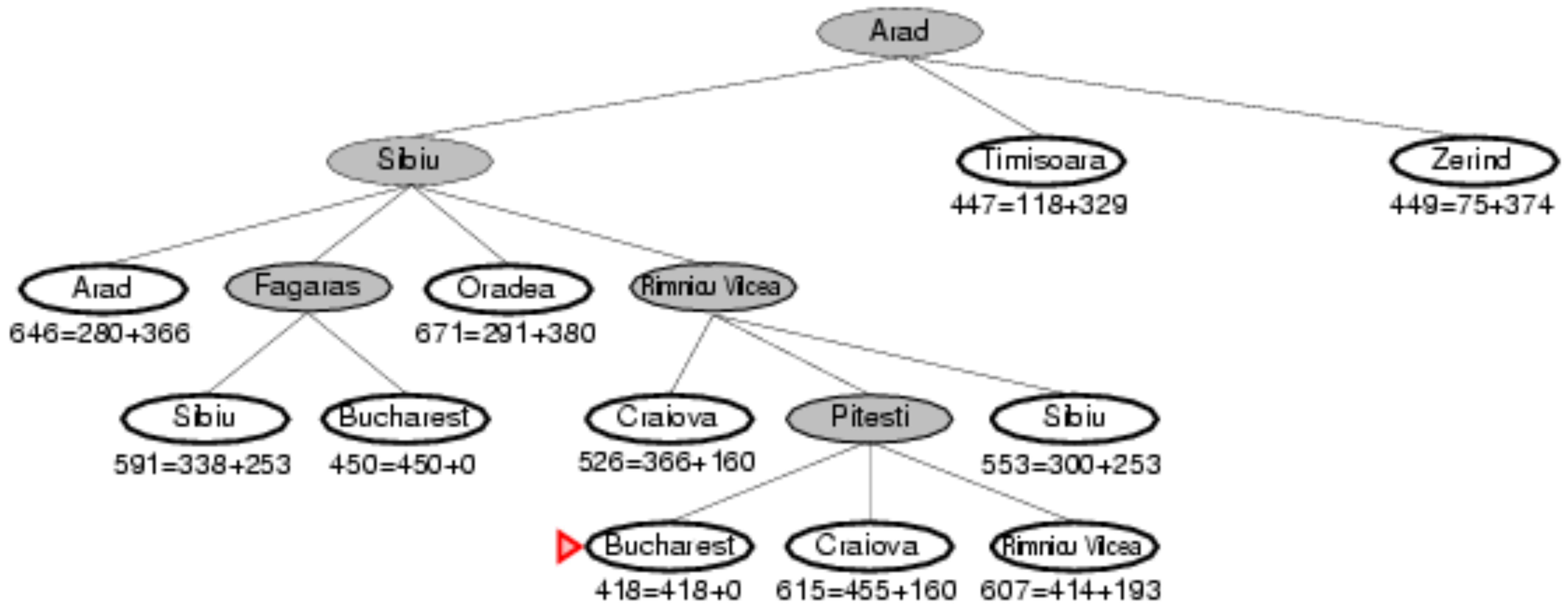


Heuristic, h(n)

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

A* Search



Heuristic, h(n)

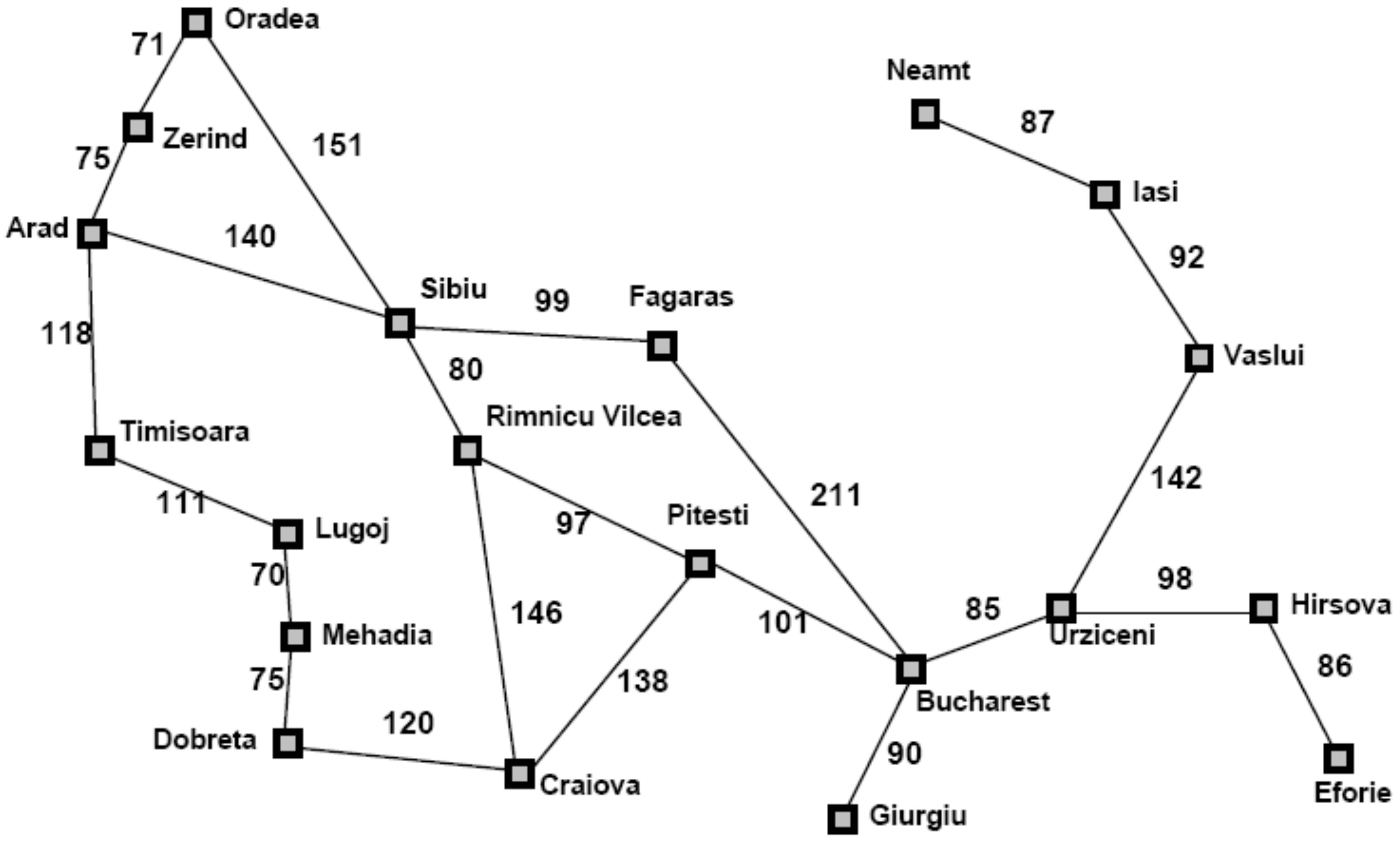
Straight−line distance to Bucharest

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

Arad -> sibiu -> Rimicu Vilcea -> Pitesti -> Bucharest

A* Search

Perform A* Search to reach to Bucharest from (a) Oradea and (b) Timisoara.



Heuristic, h(n)

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

A* Search

- Complete
  - Yes

- Time
  - The better the heuristic, the better the time
    - Best case h is perfect, $O(d)$
    - Worst case $h = 0$, $O(b^d)$ same as BFS

- Space
  - Keeps all nodes in memory and save in case of repetition
  - This is $O(b^d)$ or worse
  - A* usually runs out of space before it runs out of time

- Optimal
  - Yes

Local Search Algorithms

- The search algorithms that we have seen so far
  - are designed to explore search spaces systematically.
  - When a goal is found, the *path to that goal also constitutes a solution to the problem*

- In many optimization problems, the path to the goal is irrelevant;
  - For example, in the 8-queens problem
  - what matters is the final configuration of queens, not the order in which they are added.
  - In such cases, we can use local search algorithms

Local Search Algorithms

- **operate using** a single **current node and** Try to improve it

- generally move only to neighbors of that node

- key advantages:
  - They use very little memory
  - they can often find reasonable solutions in large or infinite (continuous) state spaces

Local Search Algorithms

- are useful for solving pure **optimization problems**
  - in which the aim is to find the best state according to an **objective function**

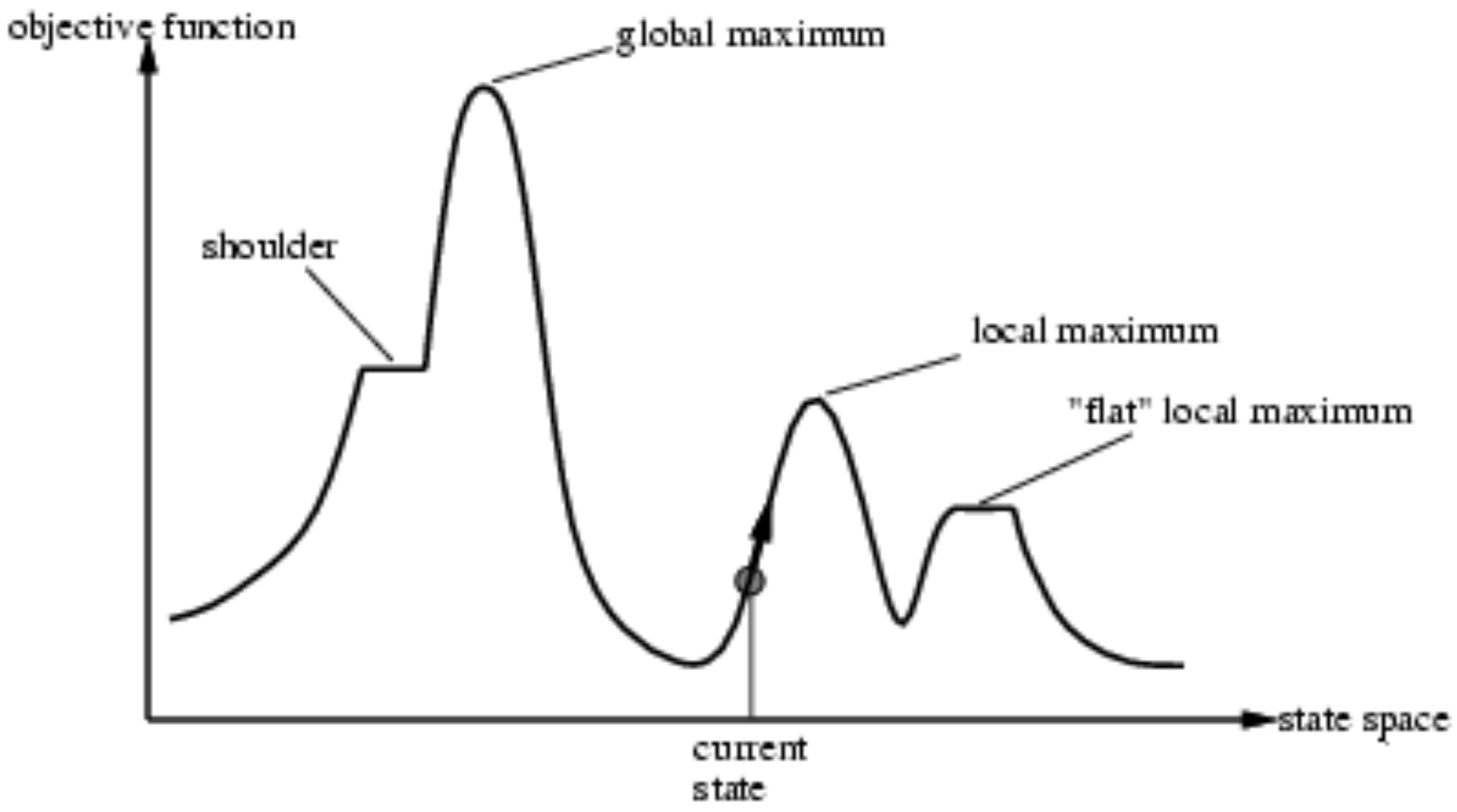
Hill-climbing search

- It is simply a loop that
  - continually moves in the direction of increasing value—that is, uphill
  - It terminates when it reaches a “peak” where no neighbor has a higher value

- Hill climbing
  - does not look ahead beyond the immediate neighbors of the current state (so called **greedy local search)**
  - Like trying to find the top of Mount Everest in a thick fog with amnesia

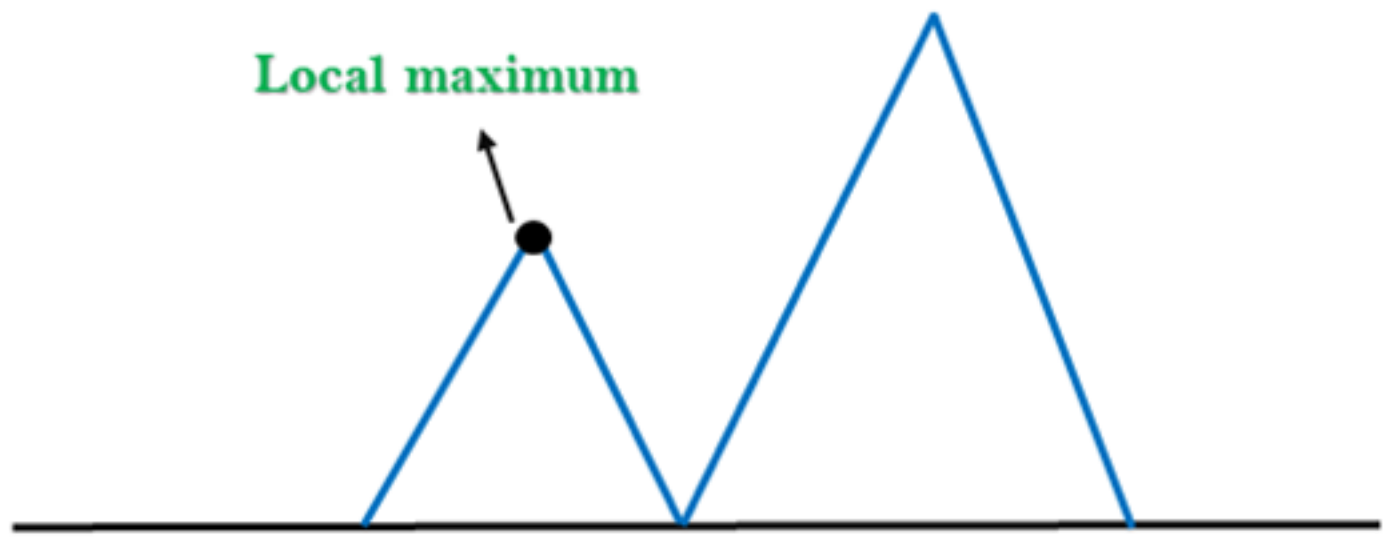
Hill-climbing search

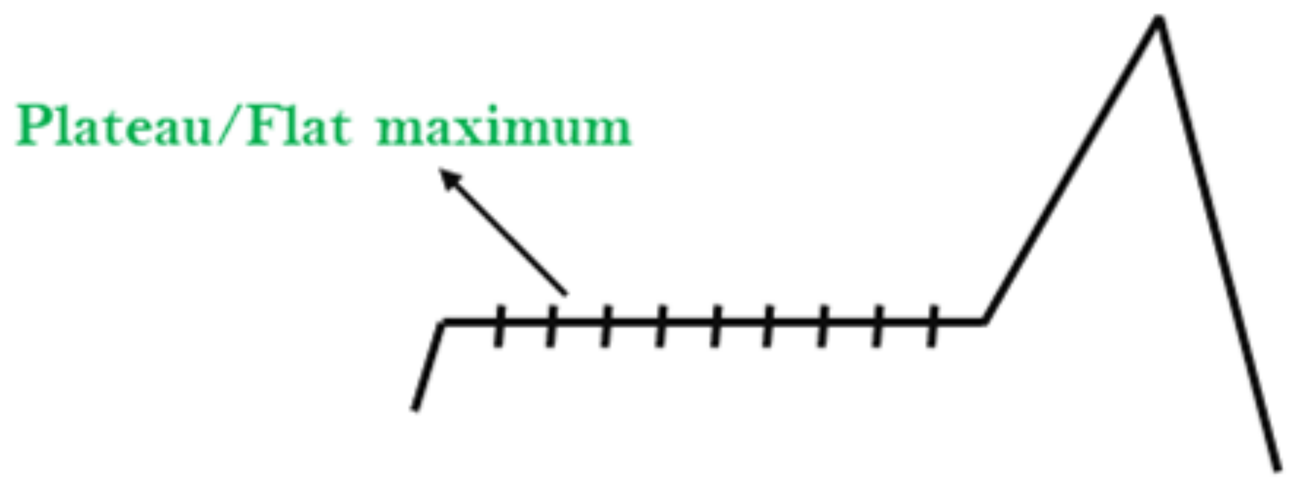
- Problem: depending on initial state, can get stuck in local maxima

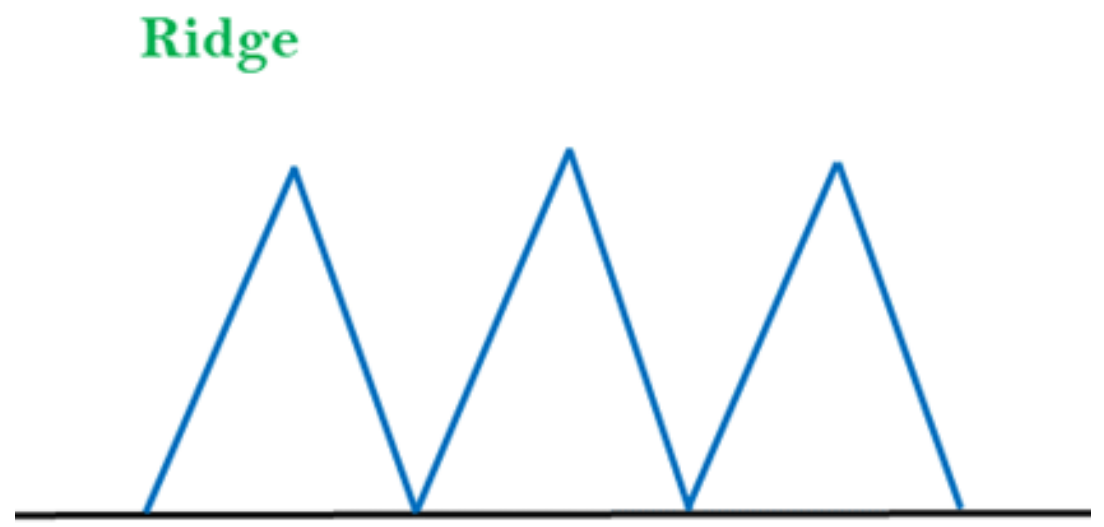


Hill-climbing search

- hill climbing often gets stuck for the following reasons:
  - Local maxima = no uphill step
    - a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
  - Plateau = all steps equal (flat or shoulder)
    - Must move to equal state to make progress, but no indication of the correct direction
  - Ridge
    - Ridges result in a sequence of local maxima
      - that is very difficult for greedy algorithms to navigate.







Questions

- What is the propose of searching in AI?
- What is state-space?
- What is problem solving agent?
- How do you formulate a problem?
- Formulate a water jug problem.
- Differentiate between informed and uninformed search?
- How does Breadth First Search work? Explain with an example.
- How does A* search work? Explain with example.

THANK YOU

End of Chapter