National University of Computer and Emerging Sciences



Lab Manual

CL461-Artificial Intelligence Lab

Department of Computer Science

FAST-NU, Lahore, Pakistan

Table of Contents

[1](#_heading=h.1ksv4uv) Objectives 3

[2](#_heading=h.44sinio) Task Distribution 3

[3](#_heading=h.2bn6wsx) A-Star Algorithm 3

[3.1](#_heading=h.2jxsxqh) History 3

[3.2](#_heading=h.z337ya) What is A-star Algorithm (A\*)? 3

[3.3](#_heading=h.3j2qqm3) A\* Algorithm Flow 4

[Step by Step Approach: 5](#_heading=h.1y810tw)

[3.4](#_heading=h.4i7ojhp) A\* Search and its Hueristic 5

[3.5 Importance of Scale: 6](#_heading=h.2xcytpi)

[Problem Solving using A-StarAlgorithm 7](#_heading=h.1ci93xb)

4 DFS and BFS

# Objectives

After performing this lab, students shall be able to understand the following Python concepts and applications:

* A\* Algorithm
* Problem solving using A\* Algorithm

# Task Distribution

| **Total Time** | **170 Minutes** |
| --- | --- |
| A\* Algorithm Introduction | 25 Minutes |
| Application of A\* Algorithm | 25 Minutes |
| Exercise | 120 Minutes |
| Online Submission | 10 Minutes |

# A-Star Algorithm

## History

* As early as 1962, John Holland's work on adaptive systems laid the foundation for later developments.
* In 1964 Nils Nilsson invented a heuristic based approach to increase the speed of Dijkstra’s algorithm. This algorithm was called A1.
* In 1967 Bertram Raphael made dramatic improvements upon this algorithm, but failed to show optimality. He called this algorithm A2.
* Then in 1968 Peter E.Hart introduced an argument that proved A2 was optimal when using a consistent heuristic with only minor changes. His proof of the algorithm also included a section that showed that the new A2 algorithm was the best algorithm possible given the conditions. He thus named the algorithm in kleene star

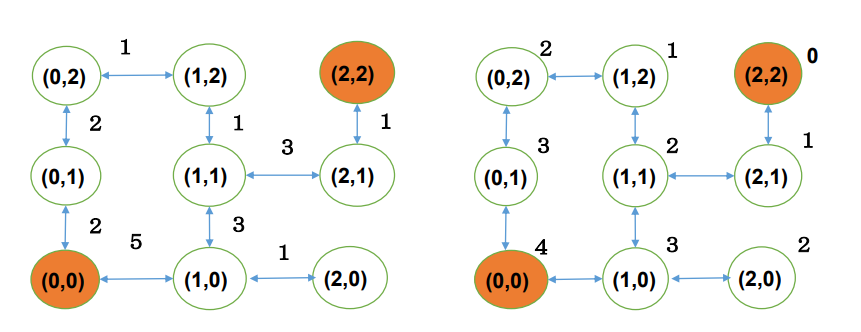
## What is A-star Algorithm (A\*)?

* It is a searching algorithm that is used to find the shortest path between an initial and a final point.
* It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A\* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.
* It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

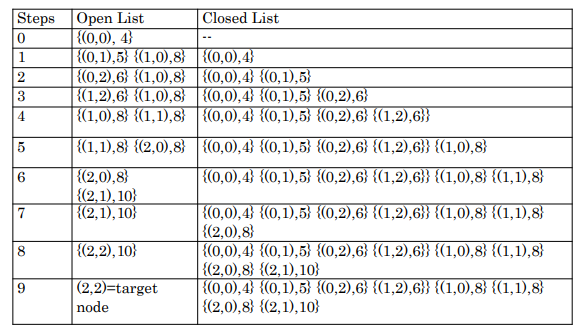
## A\* Algorithm Flow

Step 1: Put the initial node x0 and its cost F(x0)=H(x0) to the open list. • Step 2: Get a node x from the top of the open list. If the open list is empty, stop with failure. If x is the target node, stop with success. • Step 3: Expand x to get a set S of child nodes. Put x to the closed list.

Step 4: For each x’ in S, find its cost – If x’ is in the closed list but the new cost is smaller than the old one, move x’ to the open list and update the edge (x,x’) and the cost. – Else, if x’ is in the open list, but the new cost is smaller than the old one, update the edge (x,x’) and the cost. – Else (if x’ is not in the open list nor in the closed list), put x’ along with the edge (x,x’) and the cost F to the open list.



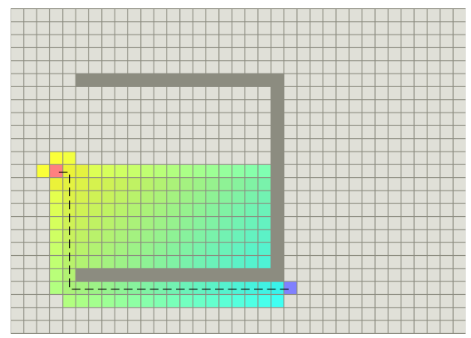
## Step by Step Approach:



## A\* Search and its Hueristic

1. A\* is like Dijkstra’s algorithm in that it can be used to find a shortest path. A\* is like Greedy
2. Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as
3. Greedy Best-First-Search.
4. The secret to its success is that it combines the pieces of information that Dijkstra’s algorithm
5. uses (favouring vertices that are close to the starting point) and information that Greedy Best-
6. First-Search uses (favouring vertices that are close to the goal). In the standard terminology
7. used when talking about A\*, g(n) represents the exact cost of the path from the starting point
8. to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal.
9. In the above diagram, the yellow (h) represents vertices far from the goal and teal (g)
10. represents vertices far from the starting point. A\* balances the two as it moves from the
11. starting point to the goal. Each time through the main loop, it examines the vertex n that has
12. the lowest f(n) = g(n) + h(n).
13. The heuristic can be used to control A\*’s behaviour.
14. ∙At one extreme, if h(n) is 0, then only g(n) plays a role, and A\* turns into Dijkstra’s
15. algorithm, which is guaranteed to find a shortest path.
16. If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then
17. A\* is guaranteed to find a shortest path. The lower h(n) is, the more node A\* expands,
18. making it slower.
19. If h(n) is exactly equal to the cost of moving from n to the goal, then A\* will only
20. follow the best path and never expand anything else, making it very fast. Although
21. you can’t make this happen in all cases, you can make it exact in some special cases.
22. It’s nice to know that given perfect information, A\* will behave perfectly.
23. If h(n) is sometimes greater than the cost of moving from n to the goal, then A\* is not
24. guaranteed to find a shortest path, but it can run faster.
25. At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role,
26. and A\* turns into Greedy Best-First-Search
27. A\* is like Dijkstra’s algorithm in that it can be used to find a shortest path. A\* is like Greedy
28. Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as
29. Greedy Best-First-Search.
30. The secret to its success is that it combines the pieces of information that Dijkstra’s algorithm
31. uses (favouring vertices that are close to the starting point) and information that Greedy Best-
32. First-Search uses (favouring vertices that are close to the goal). In the standard terminology
33. used when talking about A\*, g(n) represents the exact cost of the path from the starting point
34. to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal.
35. In the above diagram, the yellow (h) represents vertices far from the goal and teal (g)
36. represents vertices far from the starting point. A\* balances the two as it moves from the
37. starting point to the goal. Each time through the main loop, it examines the vertex n that has
38. the lowest f(n) = g(n) + h(n).
39. The heuristic can be used to control A\*’s behaviour.
40. At one extreme, if h(n) is 0, then only g(n) plays a role, and A\* turns into Dijkstra’s
41. algorithm, which is guaranteed to find a shortest path.
42. If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then
43. A\* is guaranteed to find a shortest path. The lower h(n) is, the more node A\* expands,
44. making it slower.
45. If h(n) is exactly equal to the cost of moving from n to the goal, then A\* will only
46. follow the best path and never expand anything else, making it very fast. Although
47. you can’t make this happen in all cases, you can make it exact in some special cases.
48. It’s nice to know that given perfect information, A\* will behave perfectly.
49. If h(n) is sometimes greater than the cost of moving from n to the goal, then A\* is not
50. guaranteed to find a shortest path, but it can run faster.
51. At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role,
52. and A\* turns into Greedy Best-First-Search
53. A\* is like Dijkstra’s algorithm in that it can be used to find a shortest path. A\* is like Greedy
54. Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as
55. Greedy Best-First-Search.
56. The secret to its success is that it combines the pieces of information that Dijkstra’s algorithm
57. uses (favouring vertices that are close to the starting point) and information that Greedy Best-
58. First-Search uses (favouring vertices that are close to the goal). In the standard terminology
59. used when talking about A\*, g(n) represents the exact cost of the path from the starting point
60. to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal.
61. In the above diagram, the yellow (h) represents vertices far from the goal and teal (g)
62. represents vertices far from the starting point. A\* balances the two as it moves from the
63. starting point to the goal. Each time through the main loop, it examines the vertex n that has
64. the lowest f(n) = g(n) + h(n).
65. The heuristic can be used to control A\*’s behaviour.
66. At one extreme, if h(n) is 0, then only g(n) plays a role, and A\* turns into Dijkstra’s
67. algorithm, which is guaranteed to find a shortest path.
68. If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then
69. A\* is guaranteed to find a shortest path. The lower h(n) is, the more node A\* expands,
70. making it slower.
71. If h(n) is exactly equal to the cost of moving from n to the goal, then A\* will only
72. follow the best path and never expand anything else, making it very fast. Although
73. you can’t make this happen in all cases, you can make it exact in some special cases.
74. It’s nice to know that given perfect information, A\* will behave perfectly.
75. If h(n) is sometimes greater than the cost of moving from n to the goal, then A\* is not
76. guaranteed to find a shortest path, but it can run faster.
77. At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role,
78. and A\* turns into Greedy Best-First-Search
79. A\* is like Dijkstra’s algorithm in that it can be used to find a shortest path. A\* is like Greedy
80. Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as
81. Greedy Best-First-Search

A\* is like Dijkstra’s algorithm in that it can be used to find a shortest path. A\* is like GreedyBest-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast asGreedy Best-First-Search.



The secret to its success is that it combines the pieces of information that Dijkstra’s algorithmuses (favouring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favouring vertices that are close to the goal). In the standard terminologyused when talking about A\*, g(n) represents the exact cost of the path from the starting pointto any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal.

In the above diagram, the yellow (h) represents vertices far from the goal and teal (g)represents vertices far from the starting point. A\* balances the two as it moves from thestarting point to the goal. Each time through the main loop, it examines the vertex n that hasthe lowest f(n) = g(n) + h(n)

The heuristic can be used to control A\*’s behaviour.

* At one extreme, if h(n) is 0, then only g(n) plays a role, and A\* turns into Dijkstra’salgorithm, which is guaranteed to find a shortest path.
* If h(n) is always lower than (or equal to) the cost of moving from n to the goal, thenA\* is guaranteed to find a shortest path. The lower h(n) is, the more node A\* expands,making it slower.
* If h(n) is exactly equal to the cost of moving from n to the goal, then A\* will onlyfollow the best path and never expand anything else, making it very fast. Althoughyou can’t make this happen in all cases, you can make it exact in some special cases.It’s nice to know that given perfect information, A\* will behave perfectly.
* If h(n) is sometimes greater than the cost of moving from n to the goal, then A\* is notguaranteed to find a shortest path, but it can run faster.
* At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role,and A\* turns into Greedy Best-First-Search.

You need to arrange three boxes labeled as A, B and C. State space

graph of the problem is shown below.

So we have an interesting situation in that we can decide what we want to get out of A\*. Atexactly the right point, we’ll get shortest paths really quickly. If we’re too low, then we’ll continue to get shortest paths, but it’ll slow down. If we’re too high, then we give up shortestpaths, but A\* will run faster.

## 3.5 Importance of Scale:

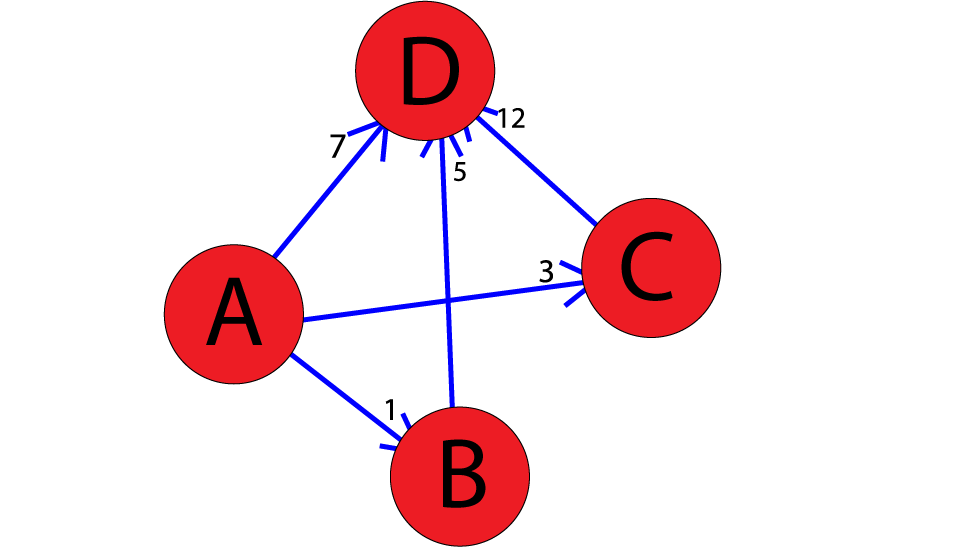
A\* computes f(n) = g(n) + h(n). To add two values, those two values need to be at the samescale. If g(n) is measured in hours and h(n) is measured in meters, then A\* is going toconsider g or h too much or too little, and you either won’t get as good paths or you A\* willrun slower than it could

## Problem Solving using A-StarAlgorithm

Famous problems which use A\*Algorithm:

**Shortest Path Problem**

The graph is represented with an adjacency list, where the keys represent graph nodes, and the values contain a list of edges with the the corresponding neighboring nodes.



Here you'll find the A\* algorithm implemented

from collections import deque

class Graph:

# example of adjacency list (or rather map)

# adjacency\_list = {

# 'A': [('B', 1), ('C', 3), ('D', 7)],

# 'B': [('D', 5)],

# 'C': [('D', 12)]

# }

def \_\_init\_\_(self, adjacency\_list):

self.adjacency\_list = adjacency\_list

def get\_neighbors(self, v):

return self.adjacency\_list[v]

# heuristic function with equal values for all nodes

def h(self, n):

H = {

'A': 1,

'B': 1,

'C': 1,

'D': 1

}

return H[n]

def a\_star\_algorithm(self, start\_node, stop\_node):

# open\_list is a list of nodes which have been visited, but who's neighbors

# haven't all been inspected, starts off with the start node

# closed\_list is a list of nodes which have been visited

# and who's neighbors have been inspected

open\_list = set([start\_node])

closed\_list = set([])

# g contains current distances from start\_node to all other nodes

# the default value (if it's not found in the map) is +infinity

g = {}

g[start\_node] = 0

# parents contains an adjacency map of all nodes

parents = {}

parents[start\_node] = start\_node

while len(open\_list) > 0:

n = None

# find a node with the lowest value of f() - evaluation function

for v in open\_list:

if n == None or g[v] + self.h(v) < g[n] + self.h(n):

n = v;

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path))

return reconst\_path

# for all neighbors of the current node do

for (m, weight) in self.get\_neighbors(n):

# if the current node isn't in both open\_list and closed\_list

# add it to open\_list and note n as it's parent

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

# otherwise, check if it's quicker to first visit n, then m

# and if it is, update parent data and g data

# and if the node was in the closed\_list, move it to open\_list

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

To run this code,

adjacency\_list = {

'A': [('B', 1), ('C', 3), ('D', 7)],

'B': [('D', 5)],

'C': [('D', 12)]

}

graph1 = Graph(adjacency\_list)

graph1.a\_star\_algorithm('A', 'D')

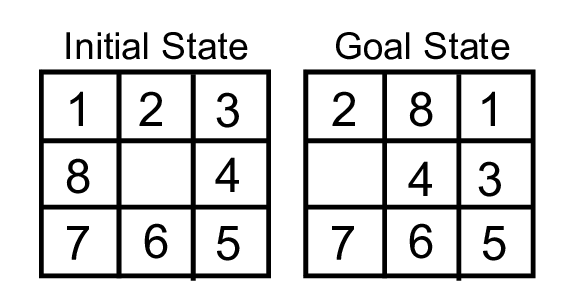
Output will be:

Path found: ['A', 'B', 'D']

['A', 'B', 'D']

## Exercise: You have to implement the 8-Puzzle problem using A\* Algorithm.

[**N-Puzzle**](https://en.wikipedia.org/wiki/15_puzzle) or **sliding puzzle** is a popular puzzle that consists of N tiles where N can be 8, 15, 24, and so on. In our example N = 8. The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns. Eg. 15-Puzzle will have 4 rows and 4 columns and an 8-Puzzle will have 3 rows and 3 columns. The puzzle consists of N tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle are provided. You have to solve the puzzle by moving the tiles one by one in the single empty space and thus achieving the Goal configuration.



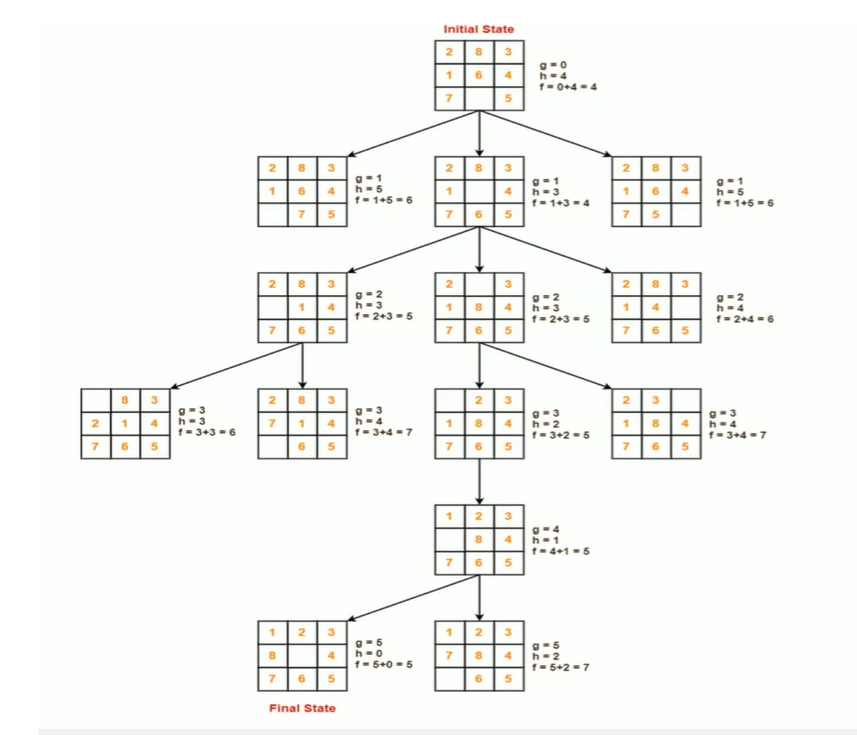
## Rules for solving the puzzle.

Instead of moving the tiles in the empty space, we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions viz.,

1. Up  
2.Down  
3. Right or  
4. Left

The empty space cannot move diagonally and can take **only one step at a time**(i.e. move the empty space one position at a time).

You can read more about solving the 8-Puzzle problem [here](https://www.geeksforgeeks.org/8-puzzle-problem-using-branch-and-bound/).



**Learning Outcome**

Students will learn how to use search algorithms on graphs, including understanding the workings of DFS and BFS. They will also gain knowledge of the pros and cons associated with both DFS and BFS.

**Search** in AI is the process of navigating from a **starting state** to a **goal state** by **transitioning** through **intermediate states**. Almost any AI problem can be defined in these terms.

* **State** — A potential outcome of a problem
* **Transition** — The act of moving between states.
* **Starting State —** Where to start searching from.
* **Intermediate State**- The states between the starting state and the goal state that we need to transition to.
* **Goal State —** The state to stop searching.
* **Search Space** — A collection of states.

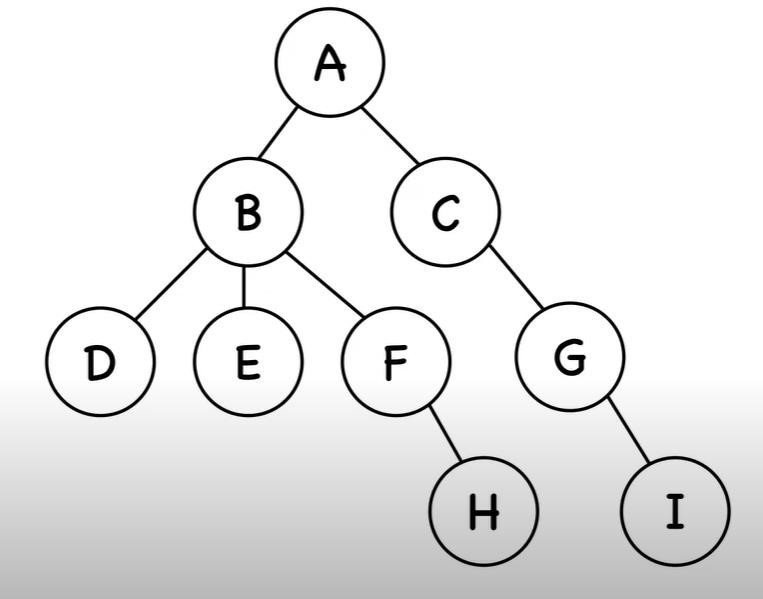
Breadth-first search

* Breadth-first search is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.

**Data Structure use in** Breadth-first search

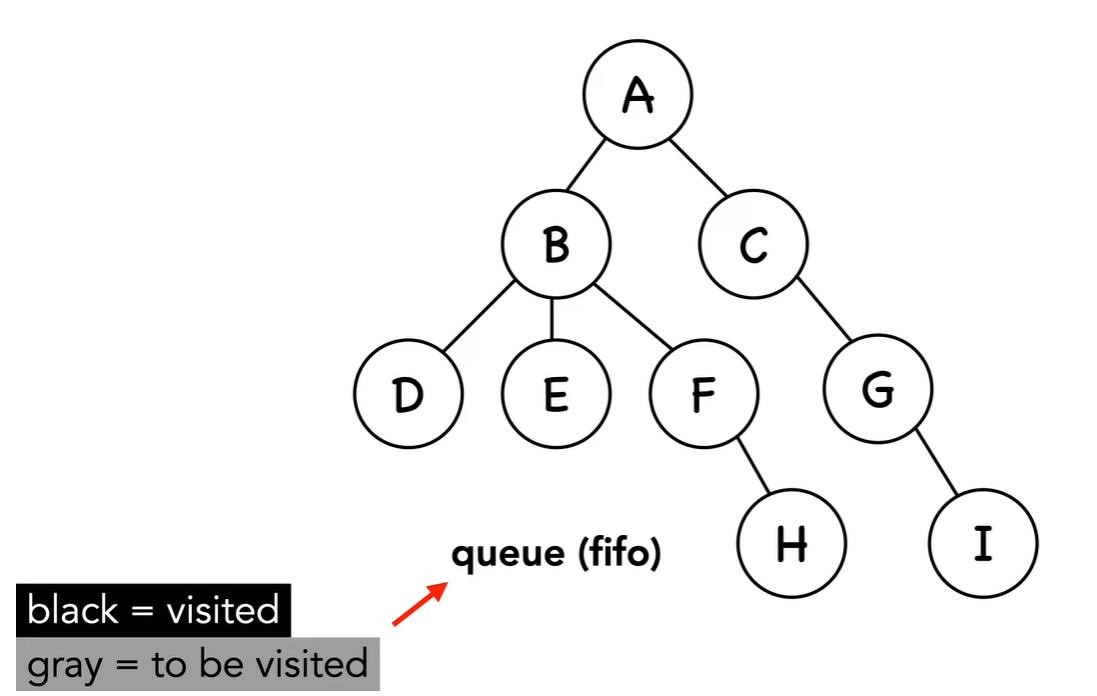
* Important data structure to keep track of vertices is FIFO QUEUE

Breadth-first search Example Demonstration is given below.



A diagram of a triangle with circles and letters

Description automatically generated



A diagram of a triangle with letters and numbers

Description automatically generated

A diagram of a triangle with letters and numbers

Description automatically generated

A diagram of a diagram

Description automatically generated

A diagram of a diagram

Description automatically generated

A diagram of a diagram

Description automatically generated

A diagram of a tree

Description automatically generated

A diagram of a diagram of a diagram

Description automatically generated

A diagram of a diagram of a diagram

Description automatically generated with medium confidence

A diagram of a triangle with circles and letters

Description automatically generated

A diagram of a diagram of a diagram

Description automatically generated

A diagram of a diagram of a diagram

Description automatically generated

A diagram of a diagram of a triangle

Description automatically generated with medium confidence

A diagram of a diagram of a diagram

Description automatically generated with medium confidence

A diagram of a diagram of a triangle

Description automatically generated with medium confidence

A diagram of a diagram

Description automatically generated

A diagram of a diagram of a triangle

Description automatically generated with medium confidence

A diagram of a network

Description automatically generated

A diagram of a triangle with letters and numbers

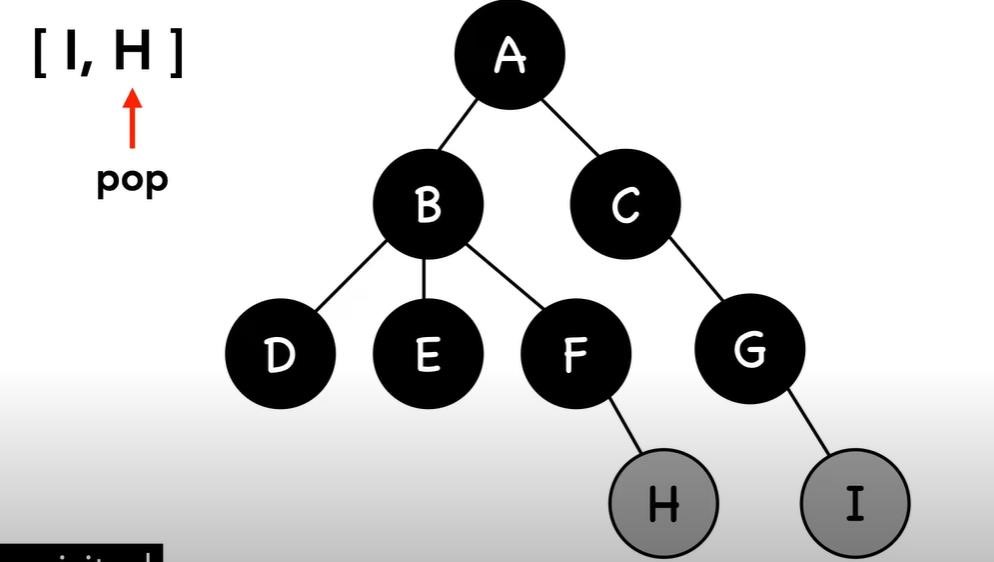
Description automatically generated

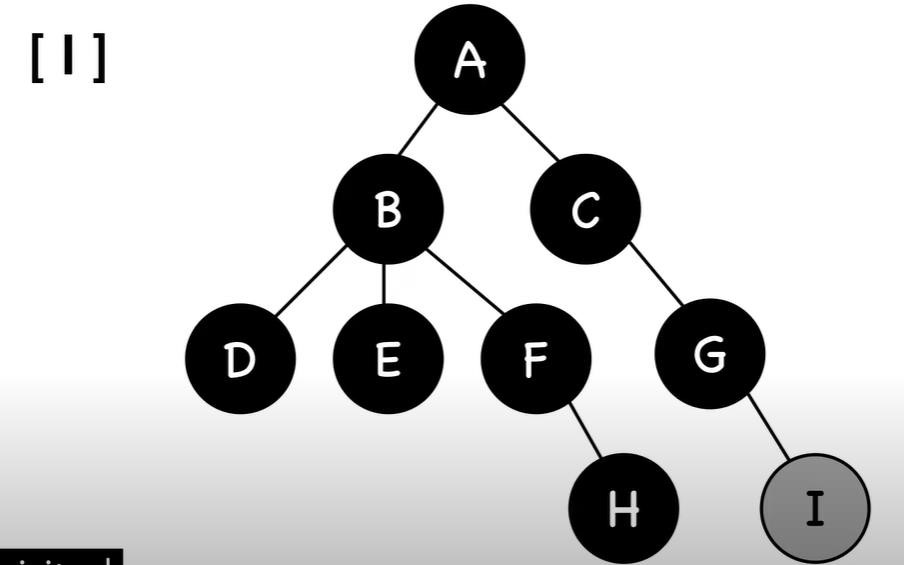
A diagram of a triangle with letters and numbers

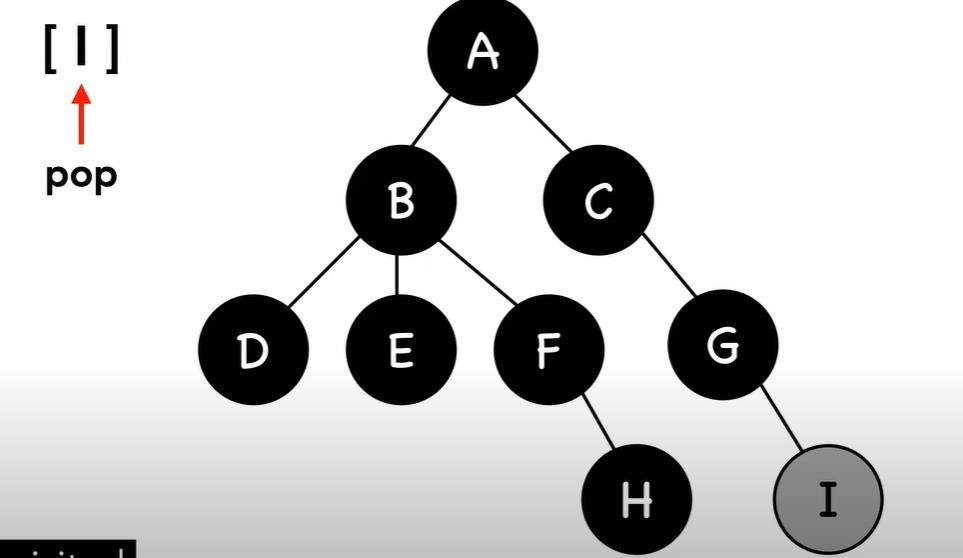
Description automatically generated

A diagram of a triangle with letters and numbers

Description automatically generated



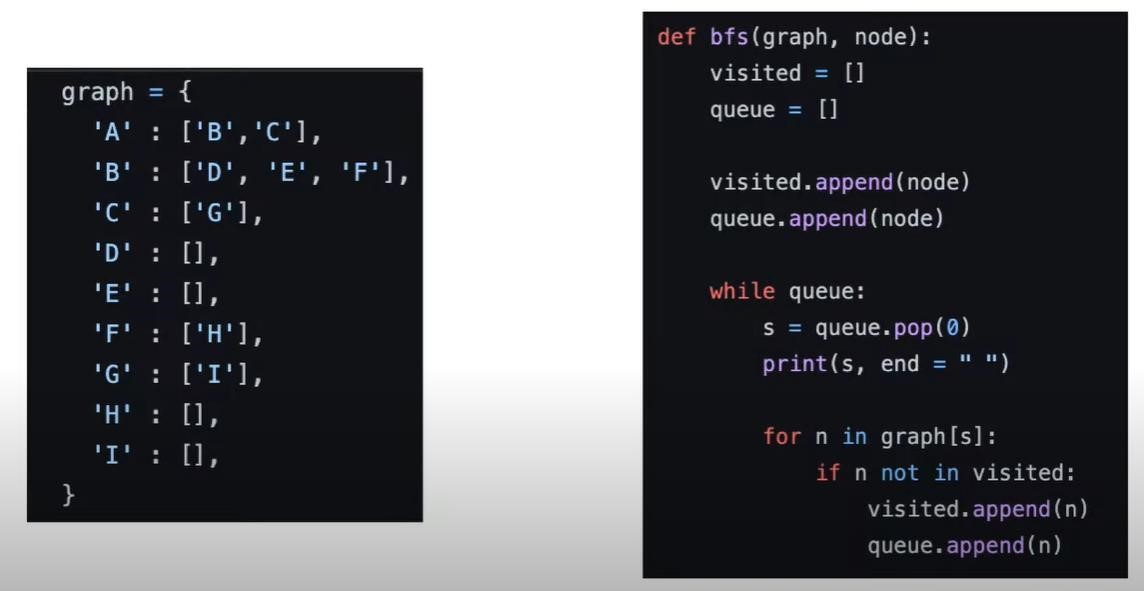




A diagram of a diagram

Description automatically generated

Code of BFS is given below.



* Explore another algorithm known as Depth-First Search (DFS) using the internet. Then, write code for DFS (Show your visualization work on sheet)