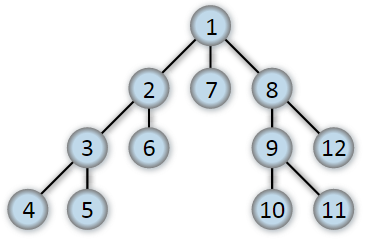
**Lab1-Implement Depth First Search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.**

Depth first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root for a graph) and explore as far as possible along each branch before backtracking.

The following graph shows the order in which the nodes are discovered in DFS:

**[](https://commons.wikimedia.org/wiki/File%3ADepth-first-tree.svg)**

**Depth First Search In Trees**:-A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any acyclic connected graph is a tree. For a tree, we have the following traversal methods:

* [Preorder](https://www.techiedelight.com/preorder-tree-traversal-iterative-recursive/): visit each node before its children.
* [Postorder](https://www.techiedelight.com/postorder-tree-traversal-iterative-recursive/): visit each node after its children.
* [Inorder](https://www.techiedelight.com/inorder-tree-traversal-iterative-recursive/) : visit left subtree, node, right subtree.

A Depth first search (DFS) is a way of traversing graphs closely related to the preorder traversal of a tree. Following is the recursive implementation of preorder traversal.

Depth first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

Below are steps to solve the problem:

* Create a recursive function that takes the index of the node and a visited array.
* Mark the current node as visited and print the node.
* Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

**Advantages of Depth First Search:**

Memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.

The time complexity of a depth-first Search to depth d is O(bd) since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.

If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

DFS requires less memory since only the nodes on the current path are stored. By chance DFS may find a solution without examining much of the search space at all.

**Breadth First Search (BFS)** is an algorithm that is used to graph data or searching tree or traversing structures. The full form of BFS is the Breadth-first search.

The algorithm efficiently visits and marks all the key nodes in a graph in an accurate breadthwise fashion. This algorithm selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. Remember, BFS accesses these nodes one by one.

Once the algorithm visits and marks the starting node, then it moves towards the nearest unvisited nodes and analyses them. Once visited, all nodes are marked. These iterations continue until all the nodes of the graph have been successfully visited and marked.

The **Breadth First Search (BFS)** algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree’s root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory.

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree.

Unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

Visited and not visited.

A Boolean visited array is used to mark the visited vertices. it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

First move horizontally and visit all the nodes of the current layer

Move to the next layer

A graph traversal is a commonly used methodology for locating the vertex position in the graph. It is an advanced search algorithm that can analyze the graph with speed and precision along with marking the sequence of the visited vertices. This process enables you to quickly visit each node in a graph without being locked in an infinite loop.

In the various levels of the data, we can mark any node as the starting or initial node to begin traversing. The BFS will visit the node and mark it as visited and places it in the queue.

Now the BFS will visit the nearest and unvisited nodes and marks them. These values are also added to the queue. The queue works on the FIFO model.

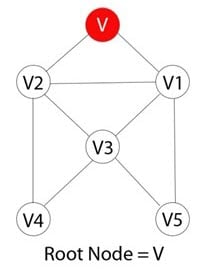
In a similar manner, the remaining nearest and unvisited nodes on the graph are analyzed marked and added to the queue. These items are deleted from the queue as receive and printed as the result.

Graph traversal requires the algorithm to visit, check, and/or update every single un-visited node in a tree-like structure. Graph traversals are categorized by the order in which they visit the nodes on the graph.

BFS algorithm starts the operation from the first or starting node in a graph and traverses it thoroughly. Once it successfully traverses the initial node, then the next non-traversed vertex in the graph is visited and marked.

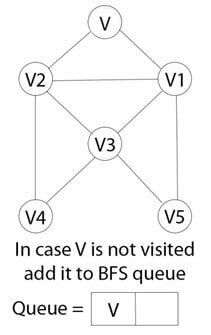
All the nodes adjacent to the current vertex are visited and traversed in the first iteration. A simple queue methodology is utilized to implement the working of a BFS algorithm, and it consists of the following steps:

**Step 1)**



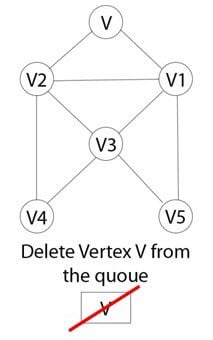
Each vertex or node in the graph is known. For instance, you can mark the node as V.

**Step 2)**



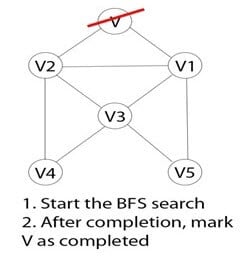
In case the vertex V is not accessed then add the vertex V into the BFS Queue

**Step 3)**



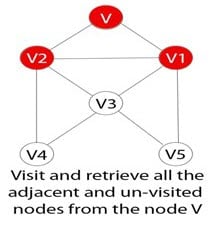
Start the BFS search, and after completion, Mark vertex V as visited.

**Step 4)**



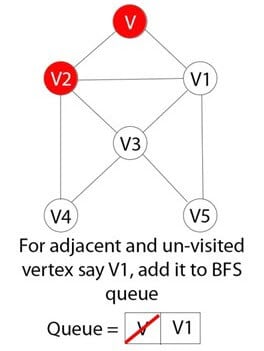
The BFS queue is still not empty, hence remove the vertex V of the graph from the queue.

**Step 5)**



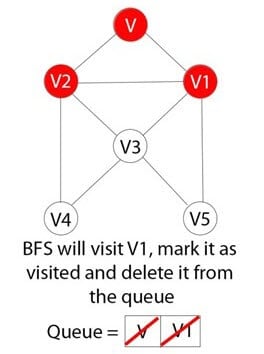
Retrieve all the remaining vertices on the graph that are adjacent to the vertex V

**Step 6)**



For each adjacent vertex let’s say V1, in case it is not visited yet then add V1 to the BFS queue

**Step 7)**



BFS will visit V1 and mark it as visited and delete it from the queue.

**Lab2- Implement A star(A\*) Algorithm for any game search problem.**

## ****A\* Algorithm-****

* A\* Algorithm is one of the best and popular techniques used for path finding and graph traversals.
* A lot of games and web-based maps use this algorithm for finding the shortest path efficiently.
* It is essentially a best first search algorithm.

**Working-**

 A\* Algorithm works as-

* It maintains a tree of paths originating at the start node.
* It extends those paths one edge at a time.
* It continues until its termination criterion is satisfied.

 A\* Algorithm extends the path that minimizes the following function-

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

 Here,

* ‘n’ is the last node on the path
* g(n) is the cost of the path from start node to node ‘n’
* h(n) is a heuristic function that estimates cost of the cheapest path from node ‘n’ to the goal node

**Algorithm-**

 The implementation of A\* Algorithm involves maintaining two lists- OPEN and CLOSED.

* OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet.
* CLOSED contains those nodes that have already been visited.

 The algorithm is as follows-

**Step-01:**

 Define a list OPEN.

* Initially, OPEN consists solely of a single node, the start node S.

**Step-02:**

 If the list is empty, return failure and exit.

**Step-03:**

 Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.

* If node n is a goal state, return success and exit.

**Step-04:**

 Expand node n.

**Step-05:**

 If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.

* Otherwise, go to Step-06.

**Step-06:**

 For each successor node,

* Apply the evaluation function f to the node.
* If the node has not been in either list, add it to OPEN.

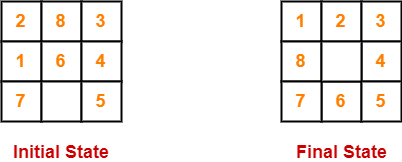
**Step-07:**

 Go back to Step-02.

**PRACTICE PROBLEMS BASED ON A\* ALGORITHM-**

**Problem-01:**

 Given an initial state of a 8-puzzle problem and final state to be reached-



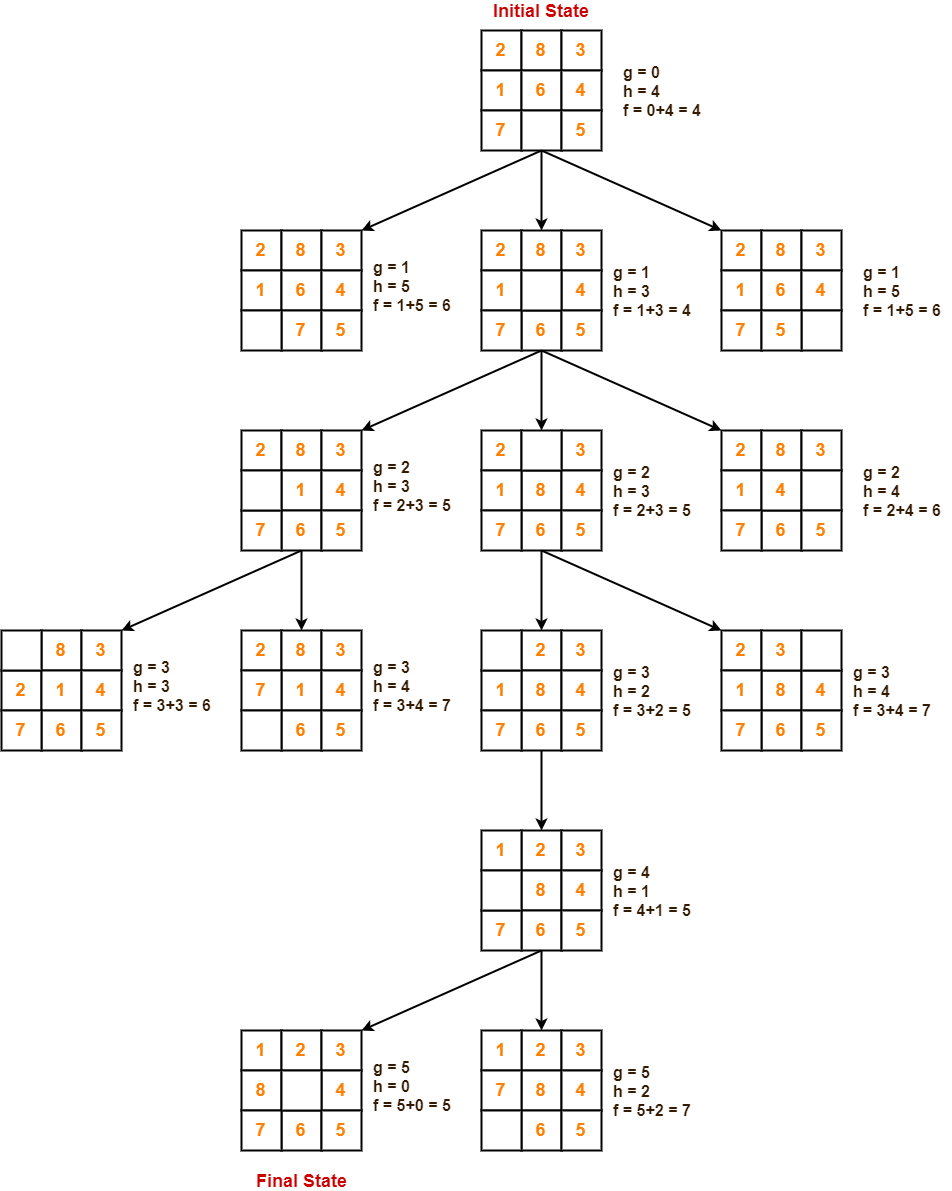
 Find the most cost-effective path to reach the final state from initial state using A\* Algorithm.

Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

**Solution-**

 A\* Algorithm maintains a tree of paths originating at the initial state.

* It extends those paths one edge at a time.
* It continues until final state is reached.



import copy

# Importing the heap functions from python

# library for Priority Queue

from heapq import heappush, heappop

# This variable can be changed to change

# the program from 8 puzzle(n=3) to 15

# puzzle(n=4) to 24 puzzle(n=5)...

n = 3

# bottom, left, top, right

row = [ 1, 0, -1, 0 ]

col = [ 0, -1, 0, 1 ]

# A class for Priority Queue

class priorityQueue:

# Constructor to initialize a

# Priority Queue

def \_\_init\_\_(self):

self.heap = []

# Inserts a new key 'k'

def push(self, k):

heappush(self.heap, k)

# Method to remove minimum element

# from Priority Queue

def pop(self):

return heappop(self.heap)

# Method to know if the Queue is empty

def empty(self):

if not self.heap:

return True

else:

return False

# Node structure

class node:

def \_\_init\_\_(self, parent, mat, empty\_tile\_pos,

cost, level):

# Stores the parent node of the

# current node helps in tracing

# path when the answer is found

self.parent = parent

# Stores the matrix

self.mat = mat

# Stores the position at which the

# empty space tile exists in the matrix

self.empty\_tile\_pos = empty\_tile\_pos

# Stores the number of misplaced tiles

self.cost = cost

# Stores the number of moves so far

self.level = level

# This method is defined so that the

# priority queue is formed based on

# the cost variable of the objects

def \_\_lt\_\_(self, nxt):

return self.cost < nxt.cost

# Function to calculate the number of

# misplaced tiles ie. number of non-blank

# tiles not in their goal position

def calculateCost(mat, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mat[i][j]) and

(mat[i][j] != final[i][j])):

count += 1

return count

def newNode(mat, empty\_tile\_pos, new\_empty\_tile\_pos,

level, parent, final) -> node:

# Copy data from parent matrix to current matrix

new\_mat = copy.deepcopy(mat)

# Move tile by 1 position

x1 = empty\_tile\_pos[0]

y1 = empty\_tile\_pos[1]

x2 = new\_empty\_tile\_pos[0]

y2 = new\_empty\_tile\_pos[1]

new\_mat[x1][y1], new\_mat[x2][y2] = new\_mat[x2][y2], new\_mat[x1][y1]

# Set number of misplaced tiles

cost = calculateCost(new\_mat, final)

new\_node = node(parent, new\_mat, new\_empty\_tile\_pos,

cost, level)

return new\_node

# Function to print the N x N matrix

def printMatrix(mat):

for i in range(n):

for j in range(n):

print("%d " % (mat[i][j]), end = " ")

print()

# Function to check if (x, y) is a valid

# matrix coordinate

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

# Print path from root node to destination node

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatrix(root.mat)

print()

# Function to solve N\*N - 1 puzzle algorithm

# using Branch and Bound. empty\_tile\_pos is

# the blank tile position in the initial state.

def solve(initial, empty\_tile\_pos, final):

# Create a priority queue to store live

# nodes of search tree

pq = priorityQueue()

# Create the root node

cost = calculateCost(initial, final)

root = node(None, initial,

empty\_tile\_pos, cost, 0)

# Add root to list of live nodes

pq.push(root)

# Finds a live node with least cost,

# add its children to list of live

# nodes and finally deletes it from

# the list.

while not pq.empty():

# Find a live node with least estimated

# cost and delete it from the list of

# live nodes

minimum = pq.pop()

# If minimum is the answer node

if minimum.cost == 0:

# Print the path from root to

# destination;

printPath(minimum)

return

# Generate all possible children

for i in range(4):

new\_tile\_pos = [

minimum.empty\_tile\_pos[0] + row[i],

minimum.empty\_tile\_pos[1] + col[i], ]

if isSafe(new\_tile\_pos[0], new\_tile\_pos[1]):

# Create a child node

child = newNode(minimum.mat,

minimum.empty\_tile\_pos,

new\_tile\_pos,

minimum.level + 1,

minimum, final,)

# Add child to list of live nodes

pq.push(child)

# Driver Code

# Initial configuration

# Value 0 is used for empty space

initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

# Solvable Final configuration

# Value 0 is used for empty space

final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

# Blank tile coordinates in

# initial configuration

empty\_tile\_pos = [ 1, 2 ]

# Function call to solve the puzzle

solve(initial, empty\_tile\_pos, final)

**Lab3- Implement Greedy search algorithm for any of the following application:**

**I. Selection Sort IV.Job Scheduling Problem**

## ****Selection Sort-****

 Selection sort is one of the easiest approaches to sorting.

* It is inspired from the way in which we sort things out in day to day life.
* It is an in-place sorting algorithm because it uses no auxiliary data structures while sorting.

**How Selection Sort Works?**

 Consider the following elements are to be sorted in ascending order using selection sort-

6, 2, 11, 7, 5

 Selection sort works as-

* It finds the first smallest element (2).
* It swaps it with the first element of the unordered list.
* It finds the second smallest element (5).
* It swaps it with the second element of the unordered list.
* Similarly, it continues to sort the given elements.

As a result, sorted elements in ascending order are-

2, 5, 6, 7, 11

**Selection Sort Algorithm-**

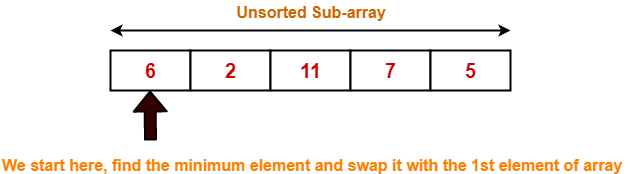
## ****Selection Sort Example-****

 Consider the following elements are to be sorted in ascending order-

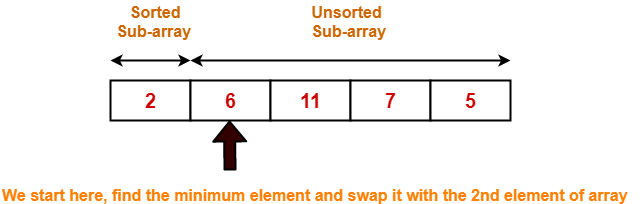
6, 2, 11, 7, 5

 The above selection sort algorithm works as illustrated below-

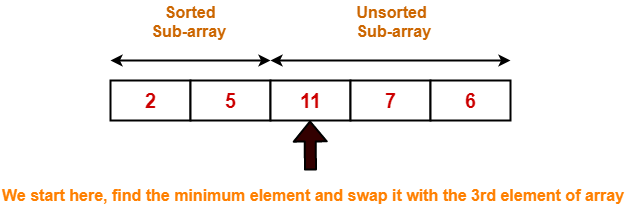
**Step-01: For i = 0**



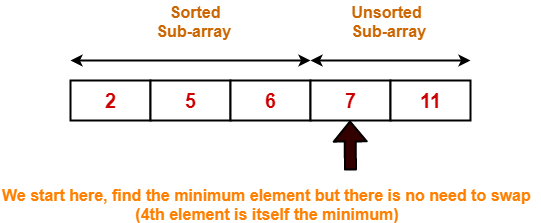
### ****Step-02: For i = 1****



### ****Step-03: For i = 2****



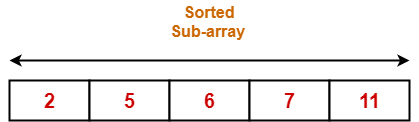
### ****Step-04: For i = 3****



### ****Step-05: For i = 4****

Loop gets terminated as ‘i’ becomes 4.

The state of array after the loops are finished is as shown-



With each loop cycle,

* The minimum element in unsorted sub-array is selected.
* It is then placed at the correct location in the sorted sub-array until array A is completely sorted.

## ****Time Complexity Analysis-****

* Selection sort algorithm consists of two nested loops.
* Owing to the two nested loops, it has O(n2) time complexity.

|  |  |
| --- | --- |
|  | **Time Complexity** |
| Best Case | n2 |
| Average Case | n2 |
| Worst Case | n2 |

## ****Space Complexity Analysis-****

 Selection sort is an in-place algorithm.

* It performs all computation in the original array and no other array is used.
* Hence, the space complexity works out to be O(1).

**Code**

I. Selection Sort

defselectionSort(arr):

for i in range(len(arr)):

min = float('-inf')

for j in range(i + 1, len(arr)):

ifarr[i] >arr[j]:

arr[i],arr[j] = arr[j], arr[i]

returnarr

print(selectionSort([59,56,45,34,65,16]))

IV.Job Scheduling Problem

# Jobs, Profit, Slot

profit = [15,27,10,100, 150]

jobs = ["j1", "j2", "j3", "j4", "j5"]

deadline = [2,3,3,3,4]

profitNJobs = list(zip(profit,jobs,deadline))

profitNJobs = sorted(profitNJobs, key = lambda x: x[0], reverse = True)

slot = []

for \_ in range(len(jobs)):

slot.append(0)

profit = 0

ans = []

for i in range(len(jobs)):

ans.append('null')

for i in range(len(jobs)):

job = profitNJobs[i]

#check if slot is occupied

for j in range(job[2], 0, -1):

if slot[j] == 0:

ans[j] = job[1]

profit += job[0]

slot[j] = 1

break

print("Jobs scheduled buddy:",ans[1:])

print(profit)