Part 1 – Implement Breadth First Search Algorithm using a Queue

```
from queue import Queue

def bfs_traversal(adj_list, start_vertex):
    visited = set()
    q = Queue()
    q.put(start_vertex)

    while not q.empty():
        current_vertex = q.get()
        if current_vertex not in visited:
            print(f"Visited vertex {current_vertex}")
            visited.add(current_vertex)
            for neighbor in adj_list.get(current_vertex, []):
                q.put(neighbor)

if __name__ == "__main__":
    adjacency_list = {
        0: [1, 3],
        1: [0, 2, 3],
        2: [1, 4, 5],
        3: [1],
        4: [2],
        5: [2]
    }
    starting_vertex = 0
    bfs_traversal(adjacency_list, starting_vertex)
```

output:

```
Visited vertex 0
Visited vertex 1
Visited vertex 3
Visited vertex 2
Visited vertex 4
Visited vertex 5
```

Part 2 – Implement Depth First Search Algorithm using a Stack

```
def dfs(g, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start)
```

```
    for neighbor in g [start]:
        if neighbor not in visited:
            dfs(g, neighbor, visited)

g= {
  'A':['B','S'],
  'B':['A'],
  'S':['A','C','G'],
  'C':['D','E','F','S'],
  'D':['C'],
  'E':['C','H'],
  'F':['C','G'],
  'G':['S','F','H'],
  'H':['E','G']


}

starting_vertex = 'A'

dfs(g, starting_vertex)
```

output:

```
A
B
S
C
D
E
H
G
F
```

Part 3 – Implement A* Algorithm using Numpy

```
from copy import deepcopy
import numpy as np
import time

def bestsolution(state):
    bestsol = np.array([], int).reshape(-1, 9)
    count = len(state) - 1
    while count != -1:
        bestsol = np.insert(bestsol, 0,
state[count]['puzzle'], 0)
        count = (state[count]['parent'])
    return bestsol.reshape(-1, 3, 3)
```

```
# checks for the uniqueness of the iteration(it).
def all(checkarray):
    set=[]
    for it in set:
        for checkarray in it:
            return 1
        else:
            return 0


# number of misplaced tiles
def misplaced_tiles(puzzle,goal):
    mscost = np.sum(puzzle != goal) - 1
    return mscost if mscost > 0 else 0


def coordinates(puzzle):
    pos = np.array(range(9))
    for p, q in enumerate(puzzle):
        pos[q] = p
    return pos


# start of 8 puzzle evaluvation, using Misplaced tiles
heuristics
def evaluvate_misplaced(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7,
8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],  1)],
                dtype =  [('move',  str, 1),('position',
list),('head', int)])

    dtstate = [('puzzle',  list),('parent',
int),('gn',  int),('hn',  int)]

    costg = coordinates(goal)
    # initializing the parent, gn and hn, where hn is
misplaced_tiles  function call
    parent = -1
    gn = 0
    hn = misplaced_tiles(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

   #priority queues with position as keys and fn as value.
    dtpriority = [('position', int),('fn', int)]

    priority = np.array([(0, hn)], dtpriority)
```

```
    while 1:
        priority = np.sort(priority, kind='mergesort',
order=['fn', 'position'])
        position, fn = priority[0]
        # sort priority queue using merge sort,the first
element is picked for
exploring.
        priority = np.delete(priority, 0,
0)
        puzzle, parent, gn, hn = state[position]
        puzzle = np.array(puzzle)

        blank = int(np.where(puzzle == 0)[0])

        gn = gn + 1
        c = 1
        start_time = time.time()
        for s in steps:
            c = c + 1
            if blank not in s['position']:
                openstates = deepcopy(puzzle)
                openstates[blank], openstates[blank +
s['head']] = openstates[blank + s['head']], openstates[blank]

                if ~(np.all(list(state['puzzle']) ==
openstates, 1)).any():
                    end_time = time.time()
                    if (( end_time - start_time ) > 2):
                        print(" The 8 puzzle is unsolvable
\n")
                        break

                    hn =
misplaced_tiles(coordinates(openstates), costg)
                    # generate and add new state in the
list
                    q = np.array([(openstates, position, gn,
hn)], dtstate)
                    state = np.append(state, q, 0)
                    # f(n) is the sum of cost to reach node
                    fn = gn +
hn

                    q = np.array([(len(state) - 1, fn)],
dtpriority)
                    priority = np.append(priority, q, 0)
```

```
                    if np.array_equal(openstates,
goal):
                        print(' The 8 puzzle is solvable \n')
                        return state, len(priority)

    return state, len(priority)


# initial state
puzzle = []

puzzle.append(2)
puzzle.append(8)
puzzle.append(3)
puzzle.append(1)
puzzle.append(6)
puzzle.append(4)
puzzle.append(7)
puzzle.append(0)
puzzle.append(5)

#goal state
goal = []

goal.append(1)
goal.append(2)
goal.append(3)
goal.append(8)
goal.append(0)
goal.append(4)
goal.append(7)
goal.append(6)
goal.append(5)


state, visited = evaluvate_misplaced(puzzle, goal)
bestpath = bestsolution(state)
print(str(bestpath).replace('[', ' ').replace(']', ''))
totalmoves = len(bestpath) - 1
print('\nSteps to reach goal:',totalmoves)
visit = len(state) - visited
print('Total nodes visited: ',visit,"\n")

output:
The 8 puzzle is solvable

   2 8 3
   1 6 4
   7 0 5
```

```
    2 8 3
    1 0 4
    7 6 5

    2 0 3
    1 8 4
    7 6 5

    0 2 3
    1 8 4
    7 6 5

    1 2 3
    0 8 4
    7 6 5

    1 2 3
    8 0 4
    7 6 5

Steps to reach goal: 5
Total nodes visited:  6
```