Part 1 – Implement Breadth First Search Algorithm using a Queue.

```
from collections import deque

def bfs(graph, start):
    visited = set()  # To keep track of visited vertices
    queue = deque([start])  # Initialize the queue with the starting vertex

    while queue:
        current_vertex = queue.popleft()  # Dequeue the front vertex
        if current_vertex not in visited:
            print(current_vertex, end=" ")  # Print the current vertex
            visited.add(current_vertex)

            # Enqueue all neighbors of the current vertex
            queue.extend(neighbor for neighbor in graph[current_vertex] if neighbor not in visited)

# Example graph represented as an adjacency list
graph = {
    0: [1, 3],
    1: [0, 2, 5],
    2: [1],
    3: [0, 4],
    4: [3, 5],
    5: [1, 4]
}

# Starting vertex
start_vertex = 0

print("BFS Traversal:")
bfs(graph, start_vertex)
```

```
BFS Traversal:
0 1 3 2 5 4
```

Part 2 – Implement Depth First Search Algorithm using a Stack.

```
def dfs(graph, start):
    visited = set()  # To keep track of visited vertices
    stack = [start]  # Initialize the stack with the starting vertex

    while stack:
        current_vertex = stack.pop()  # Pop the top vertex from the stack
        if current_vertex not in visited:
            print(current_vertex, end=" ")  # Print the current vertex
            visited.add(current_vertex)

            # Push all unvisited neighbors of the current vertex onto the stack
            stack.extend(neighbor for neighbor in reversed(graph[current_vertex]) if neighbor not in visited)

# Example graph represented as an adjacency list
graph = {
    'A': ['B', 'S'],
    'B': ['A'],
    'C': ['D', 'E', 'F', 'S'],
    'D': ['C'],
    'E': ['C', 'H', 'F'],
    'F': ['C', 'E', 'G'],
    'G': ['F', 'H'],
    'H': ['E', 'G'],
    'S': ['A', 'C']
}

# Starting vertex
start_vertex = 'A'

print("DFS Traversal:")
dfs(graph, start_vertex)
```

```
DFS Traversal:
A B S C D E H G F
```

Part 3 – Implement A* Algorithm using Numpy.

```
#Part 3 – Implement A* Algorithm using Numpy
```

```
from heapq import heappush, heappop
initial_state = ((2, 8, 3), (1, 6, 4), (7, 0, 5))
final_state = ((1, 2, 3), (8, 0, 4), (7, 6, 5))
def heuristic(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_row, goal_col = (value - 1) // 3, (value - 1) % 3
                distance += abs(i - goal_row) + abs(j - goal_col)
    return distance
def astar(initial_state, final_state):
    open_list = []
    closed_set = set()
    heappush(open_list, (0 + heuristic(initial_state), 0, initial_state))
    while open_list:
        _, g_score, current_state = heappop(open_list)
        if current_state == final_state:
            return current_state
        closed_set.add(current_state)
        zero_row, zero_col = -1, -1
        for i in range(3):
            for j in range(3):
                if current_state[i][j] == 0:
                    zero_row, zero_col = i, j
                    break
            if zero_row != -1:
                break
        for dr, dc in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            new_row, new_col = zero_row + dr, zero_col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_state = list(map(list, current_state))
                new_state[zero_row][zero_col], new_state[new_row][new_col] = new_state[new_row][new_col], new_state[zero_row][zero_col]
                new_state = tuple(map(tuple, new_state))
                if new_state not in closed_set:
                    heappush(open_list, (g_score + 1 + heuristic(new_state), g_score + 1, new_state))
    return None
result = astar(initial_state, final_state)
if result is not None:
    print("Solution Found:")
    for row in result:
        print(row)
else:
    print("No Solution Found.")
```

```
Solution Found:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```