Artificial Intelligent Final Report
Path Finding

Lee Jui Chi
26001803777

January 21, 2020

Computer specifications:

- Name: MacBook Pro(16-inch, 2019)
- Memory:16GM 2667 MHz DDR4
- Processor: 2.3GHz 8-Core Intel Core i9

1 Data Preparation

Construct the undirected graph represented by the matrix using Visone, shown on Figure 1.

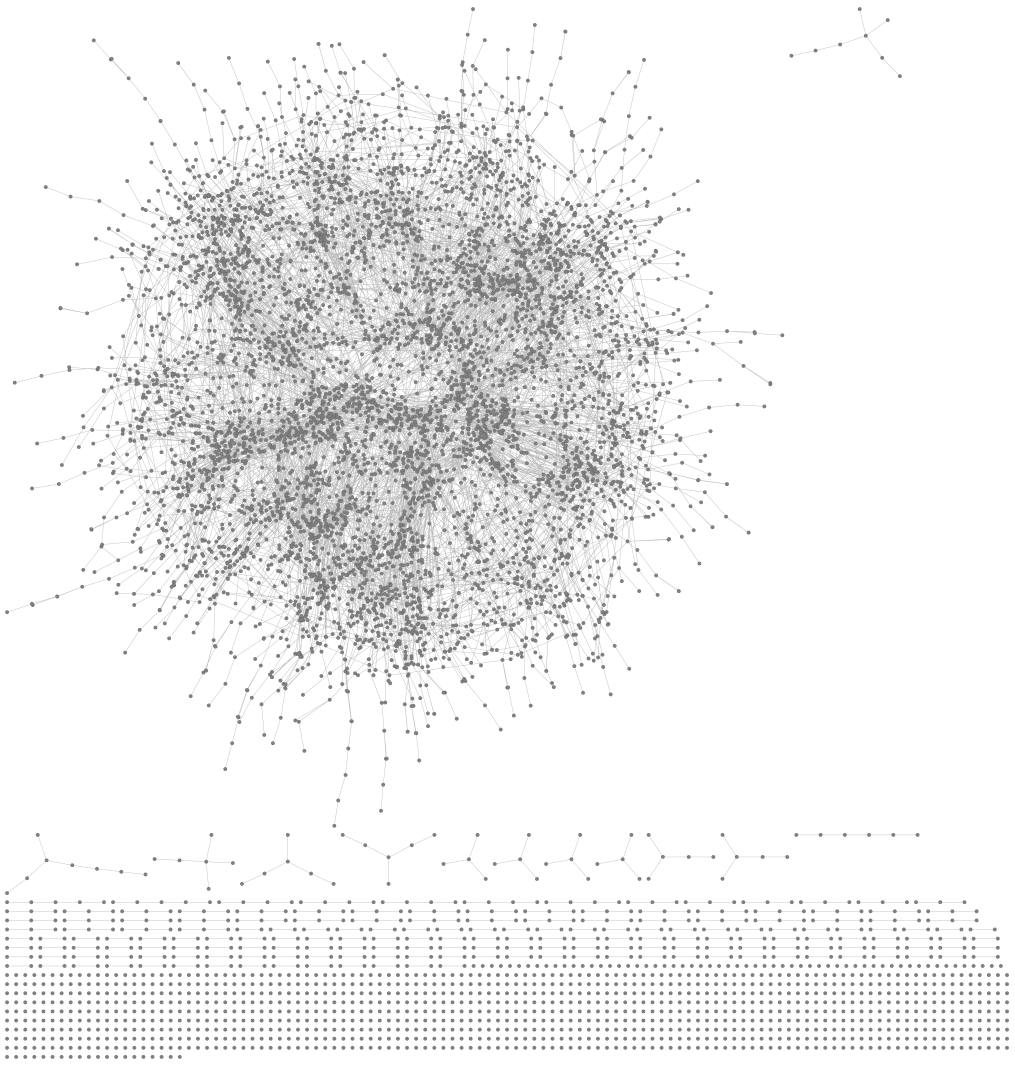

Figure 1: The undirected graph of Data1.csv

Number of nodes Visone found: 8000, Number of links Visone found: 8000

Next, to implement searching algorithms, I need to get the edge list from the data and construct a Graph class. First, I used a double for loop to read the data in each row and column and save the edges which have connections with others into a list. Second, I prepared a Graph class for mapping the edges, it uses the dictionaries to store all possible next nodes and all the weights between two nodes.

Reading the the data into the edge list cost: 54.893192753 seconds, and found 8000 edges (as same as the number of the links Visone found).

Example of an edge list:
adj matrix
0, 1, 0, 1
1, 0, 1, 0
0, 1, 0, 0
1, 0, 0, 0
The edgeList(from node, to node, weight=1) will be [(0, 1, 1), (0, 3, 1), (1, 2, 1)]

Reading the data and getting the edge list

```
import numpy as np

start_readDataIntoEdgeList = timer()
the_file = 'Data1.csv'
# read matrix without head.
# load the csv file using numpy library
a = np.loadtxt(the_file, delimiter=',', dtype=int)  # set the delimiter and
    he datatype
print(type(a)) #<class 'numpy.ndarray'>

print ('shape:',a.shape[0] ,"*", a.shape[1]) # shape: 8000 * 8000

num_edges = 0

# the edge list store the connection
edgeList = []

# Getting the connection
# double for loop untill 8000 * 8000
for row in range(a.shape[0]):
    for column in range(a.shape[1]):
        # get rid of repeat edge
        # (column, row, weight=1)
        if (a.item(row,column) == 1 and (column,row, 1) not in edgeList):
            num_edges += 1
            edgeList.append((row, column, 1))
# read the adjacency matrix file delimited by comma, and output the edgelist
    , for example

# adj matrix          edge list (from_node, to_node, weight=1)
# 0, 1, 0, 1          0,1,1
# 1, 0, 1, 0          0,3,1
# 0, 1, 0, 0 ->       1,2,1
# 1, 0, 0, 0
```

```
end_readDataIntoEdgeList = timer()
print("Read Data Into Edge List cost: " + str(end_readDataIntoEdgeList -
    start_readDataIntoEdgeList)) # Time in seconds, e.g. 5.38091952400282
print ('\nnum_edges:', num_edges) # num_edges: 8000, as same as the number
    of links Visone found
```

Prepare the Graph class

```
from collections import defaultdict

# the Graph class for mapping the edges
class Graph():
    def __init__(self):
        """
        self.edges is a dict of all possible next nodes
        e.g. {'2': ['5', '6', '9', '10'], ...}
        self.weights has all the weights between two nodes,
        with the two nodes as a tuple as the key
        e.g. {('2', '6'): 2, ('2', '10'): 4, ...}
        """
        # using default dictionary to store the list of the edges
        self.edges = defaultdict(list)
        # the weights between nodes
        self.weights = {}

    # the function for adding the edges
    def add_edge(self, from_node, to_node, weight):
        # edges are bi-directional
        # adding the edges and weights for both sides
        self.edges[from_node].append(to_node)
        self.edges[to_node].append(from_node)
        self.weights[(from_node, to_node)] = weight
        self.weights[(to_node, from_node)] = weight
```

2 Depth First Search path

To find three different paths from node number 2837("Start") to node number 7721("Destination")on the graph, I used Depth-First Search (DFS) as my main searching algorithm. It explores possible vertices (from a supplied root) down each branch before backtracking. Figure 2 shows a diagram of Depth First Search path algorithm.

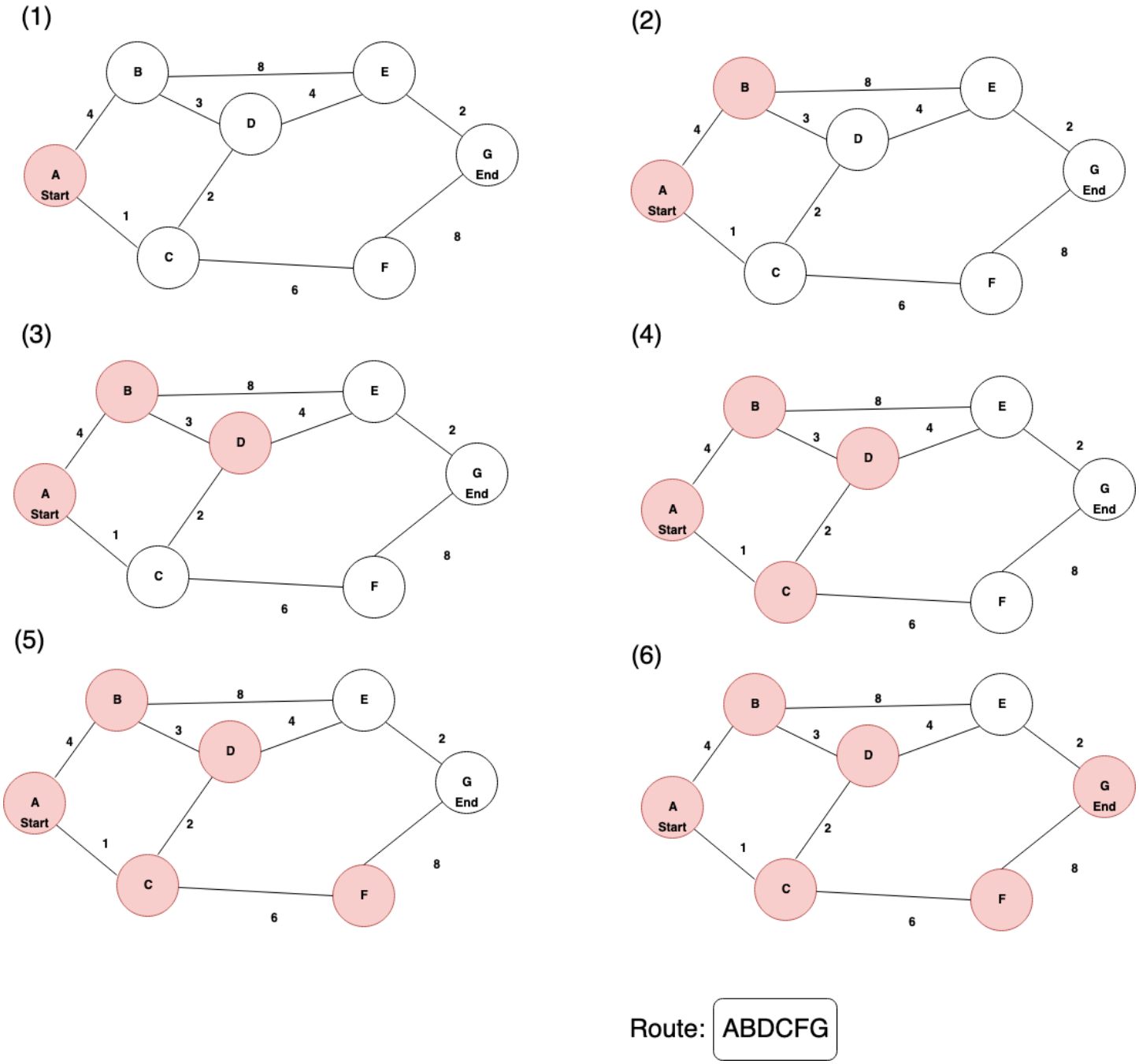

Figure 2: Depth First Search path algorithm diagram

Since the size of the map is huge, so I used the iterative way, not using recursive way to solve it. Iterative Depth First Search:

- Add root node to the stack
- Loop on the stack until it is empty
  - Get the node and the path at the top of the stack(current), mark it as visited, and remove it
  - For every non-visited child of the current node, check if it's the goal node, If so, then yeild this path and this child node. Otherwise, push it and the path to the stack.

For getting only 3 different paths, I used a combination of while loop (for only get 3 paths, not all paths) and yeild (instead of return). The yield statement suspends function's execution and sends a value back to the caller, but retains enough state to enable function to resume where it is left off, so that I can prevent it getting three same paths.

Since the paths DFS found have went through too many nodes, it is better to represent them as three-number sequences, I ran three for loops to find the differences of them. The represented middle node is the node which the other two paths don't have.
DFS path 1: 2837, ..., 4682, ..., 7721, went through 2115 nodes
DFS path 2: 2837, ..., 278, ..., 7721, went through 2132 nodes
DFS path 3: 2837, ..., 529, ..., 7721, went through 2121 nodes
DFS search for 3 paths cost: 0.3444683060000031 seconds

Iterative Depth First Search

```
# Depth First Search path finding function
def dfs_paths(graph, start, goal):
    # for counting the time we run the while loop
    count = 0
    # uses the stack data-structure to iteratively solve the problem,
    # stack list starting from the start
    # [starting node, [the path list]]
    stack = [(start, [start])]
    while(stack):
        # set the limitation for the number of the paths we want (ed. only
           run 3 times while loop)
        if(count == 3):
            # get out the loop
            break
        # get the vertex and the path from the stack
        (vertex, path) = stack.pop()
        # Using the overloading of the subtraction operator to remove items
           from a set, to add only the unvisited adjacent vertices.
        for next in set(graph.edges[vertex]) - set(path):
            # if we find the end node
            if next == goal:
                # add 1 to the count
                count += 1
                # yielding each possible path when we locate the goal. Using
                    a generator allows the user to only compute the desired
                   amount of alternative paths

                yield path + [next]
            else:
                # keep finding the nodes
                stack.append((next, path + [next]))
# Initialize the graph
graph = Graph()
# Starting edge
start = 2837
# Ending edge
end = 7721


# adding edges from the edge list into the graph
for edge in edgeList:
    graph.add_edge(*edge)

# Find three different DFS paths
```

```
print("Depth_first_path: ")
start_dfs = timer()
dfs_list = list(dfs_paths(graph, start, end))
end_dfs = timer()
print("DFS search for 3 paths cost: " + str(end_dfs - start_dfs)) # Time in
    seconds, e.g. 5.38091952400282
# print("DFS path no.1: ", dfs_list[0]) # DFS path no.1: [2837, ..., 7721]
# print("\nDFS path no.2: ", dfs_list[1]) # DFS path no.2: [2837, ..., 7721]
# print("\nDFS path no.3: ", dfs_list[2]) # DFS path no.3: [2817, ..., 7721]
# Finding the nodes is the node which the other two paths don't have
for i in dfs_list[0]:
    if(i not in dfs_list[1]):
        print(i)
for i in dfs_list[0]:
    if(i not in dfs_list[2]):
        print(i)
for i in dfs_list[1]:
    if(i not in dfs_list[2]):
        print(i)

# Finding the number of nodes DFS went through
print("DFS Lenth 1")
print(len(dfs_list[0]))
print("DFS Lenth 2")
print(len(dfs_list[1]))
print("DFS Lenth 3")
print(len(dfs_list[2]))
```

---

3 Djikstras Shortest Path Algorithm

Dijkstra's algorithm is a path-finding algorithm, it can be used to find the shortest path between two nodes in a graph. From the starting node, it keeps finding the next node of the lowest weight and continues to do so until the next node of the lowest weight is the end node.

For each destination node that we visited, we note the possible next destinations and the total weight to visit that destination. If a destination is the one we have seen before and the weight to visit is lower than it was previously, this new weight will take its place. This can work even all the weight are the same. For example:

- Visiting 200 from 1 via 29, 409 is a cost of 3
- Visiting 200 from 1 via 50 is a cost of 2
- Therefore we note that the shortest route to 200 is via 50, and the new weight will take its place

We keep the record of the previous destination node and the total weight to get there, and continue evaluating until the destination node weight is the lowest total weight of all possible options. Then, continue searching until there are no more path shorter than this path, and finalize that it is the shortest path. Figure 3 shows a diagram of Djikstras Shortest Path Algorithm

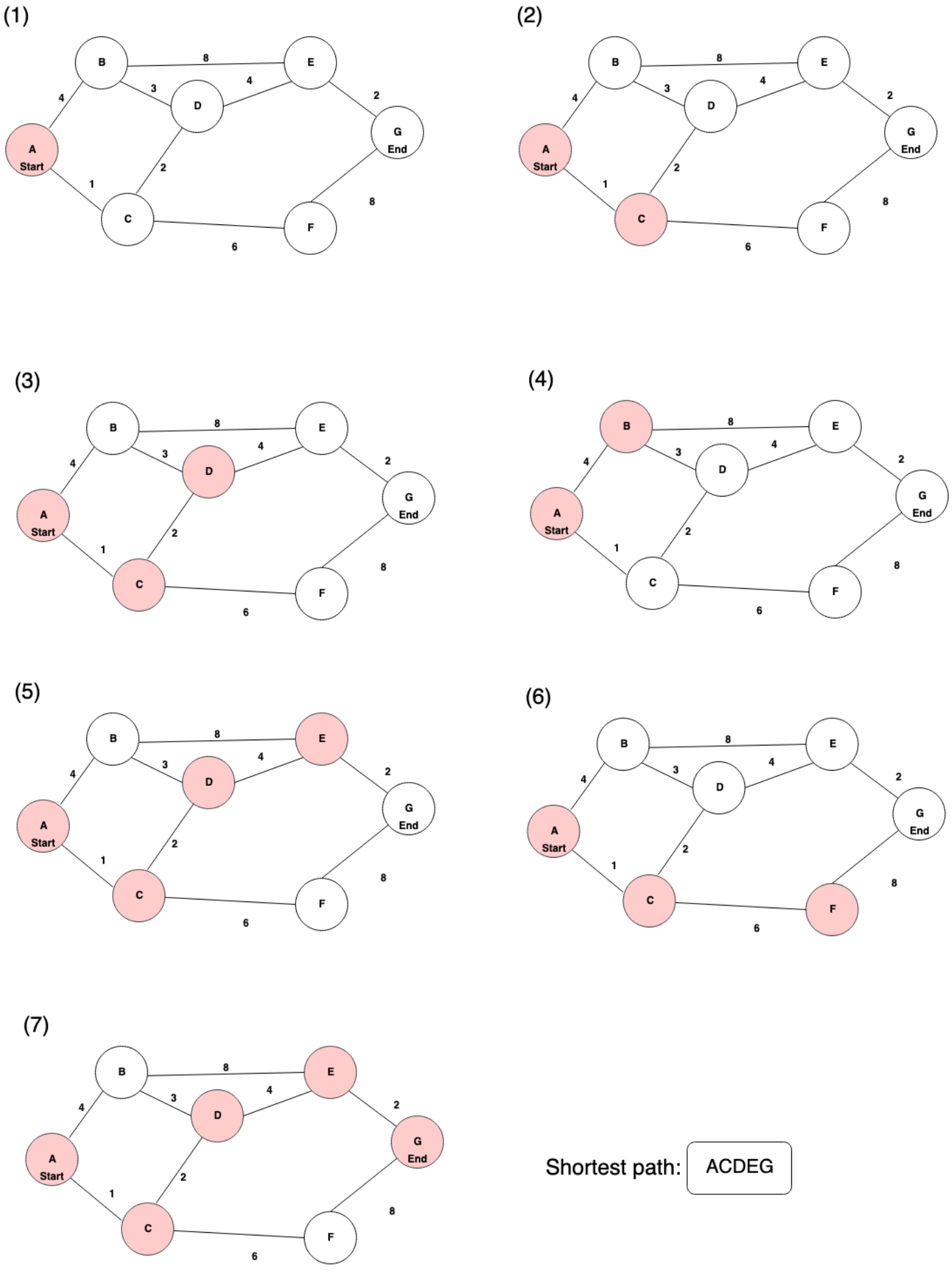

Figure 3: Depth First Search path algorithm diagram

Shortest Path: [2837, 7169, 7574, 7993, 6824, 4574, 2018, 6612, 2830, 7791, 16, 7721]

Shortest Distance: 11

Time cost: 1.0892150219999976 seconds

Djikstras shortest path

```
# Dijsktra algorithm function for finding the shortest path
def dijsktra(graph, initial, end):
    # shortest paths is a dict of nodes
    # whose value is a tuple of (previous nodes, weight)
    shortest_paths = {initial: (None, 0)}
    current_node = initial
    visited = set()

    # Find nodes until it is the end node
    while current_node != end:
        # Record the node which are visited
        visited.add(current_node)
        # Find the dict of all possible next nodes for current node in graph
        destinations = graph.edges[current_node]
        # Record the weight to the current node from the tuple of (previous
            node, weight)
        weight_to_current_node = shortest_paths[current_node][1]

        # Loop through all possible next nodes
        for next_node in destinations:
            # Calculate the possible weight sum
            weight = graph.weights[(current_node, next_node)] +
                weight_to_current_node
            # Record the (current_node, weight) into the path if it doesn't
                find the next node already in the path
            if next_node not in shortest_paths:
                shortest_paths[next_node] = (current_node, weight)
            # if it already found the next node in the path
            else:
                # Refresh the current shortest weight as the total weight to
                    the next node
                current_shortest_weight = shortest_paths[next_node][1]
                # if the current shortest weight is bigger then the possible
                    weight sum
                if current_shortest_weight > weight:
                    # Record the (current_node, weight) into the path
                    shortest_paths[next_node] = (current_node, weight)
        # the potential next desitnations
        next_destinations = {node: shortest_paths[node] for node in
            shortest_paths if node not in visited}
        if not next_destinations:
            return "Route Not Possible"
        # next node is the destination with the lowest weight
        current_node = min(next_destinations, key=lambda k:
            next_destinations[k][1])

    # Work back through destinations in shortest path
```

```
    path = []
    while current_node is not None:
        path.append(current_node)
        next_node = shortest_paths[current_node][0] # previous nodes
        current_node = next_node
    # Reverse path
    path = path[::-1]
    return path, current_shortest_weight # (path, total weight cost for this
        path)


# Initialize the graph
graph = Graph()
# Starting edge
start = 2837
# Ending edge
end = 7721

# adding edges from the edge list into the graph
for edge in edgeList:
    graph.add_edge(*edge)

start_dijsktra = timer()
# Find the shortest path
shortest_path = dijsktra(graph, start, end)
end_dijsktra = timer()
print("Dijsktra search for shortest path cost: " + str(end_dijsktra -
   start_dijsktra)) # Time in seconds, e.g. 5.380919524002 82
print("Shortest path: ")
print(shortest_path) # [(path , total weight cost for this path)]
```

4 Reference

Visone: https://visone.info/
Djikstras Shortest path: http://benalexkeen.com/implementing-djikstras-shortest-path-algorithm-with-python/
Depth First Search path: https://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/
Reading the data: https://stackoverflow.com/questions/15283803/read-in-matrix-from-file-make-edgelist-and-write-edgelist-to-file
The Graph class: https://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/
Path Finding Algorithm: https://medium.com/omarelgabrys-blog/path-finding-algorithms-f65a8902eb40