CSE 551 PROGRAMMING ASSIGNMENT

SAI AJITESH KROVVIDI

1219320388

Introduction:

Any two paths A_1, A_2 are said to be Vertex-disjoint if the two paths do not have a common internal vertex. The problems that require finding out the number of vertex disjoint paths can be converted to a max-flow problem. A flow network can be constructed by splitting each vertex c into c_{in} and c_{out} and adding a super source and super sink to the network. The maximum flow can be calculated by using Edmonds-Karp Algorithm and once we obtain the maximum flow we will be in a position to determine whether or not there exist m vertex disjoint paths. As we move further we would be discussing in detail the problem statement, the algorithm, the pseudo-code for implementation, the format for input along with the results and observations.

Problem Statement:

Given an n x n grid which is an undirected graph with n rows and n columns of vertices. Vertex in the i^{th} row and j^{th} column is given by (i,j). For a given m less than or equal to n^2 starting points (x_1,y_1), (x_2,y_2), (x_3,y_3), (x_4,y_4), ...(x_m,y_m) in the grid. The objective of the Funny Problem is to determine if there are m vertex disjoint paths from starting points to any m different points on the boundary. If there exist m such paths what is the vertex-disjoint path that is a part of the solution.

Algorithm:

Step 1. Two vertices which can be called a super-source (s) and a super-sink (t) have to be added

// This step is to convert the given funny problem into a maximum-flow problem with a super source (s) and a super sink (t)

Step 2. Each vertex 'e' (except s and t) in the graph should be split into two vertices e_{in} and e_{out}

// This step converts the given undirected graph into a directed graph for applying the max-flow algorithm.

Step 3. Add an edge from s to each e_{in} with a capacity of one where e is the set of starting vertices for the Funny Problem

Step 4. For every single vertex e, add an edge from e_{in} to e_{out} with a capacity of one

Step 5. Every single edge (e,f) has to be replaced with e_{out} to f_{in} and f_{out} to e_{in} with a capacity of one

Step 6. For every boundary vertex f add an edge from f_{out} to t with a capacity of one

Step 7. Now calculate the maximum-flow using Edmonds-Karp Algorithm

Step 8. If the maximum s-t flow is equal to the cardinality of the set of start vertices, then Yes a solution exists with m vertex disjoint paths from the start points to the boundary

Step 8(a). The paths that constitute the solution are as follows

Step 9. Else, No solution does not exist.

Functions used:

1) path_augmenting_function(path, Graph)

   This function takes in two inputs path and graph respectively and returns a graph.

2) max_flow_cal(graph)

   This function takes a graph as an input and returns maximum flow, shortest path respectively

3) obtain_path(sink, predecessor_array)

   This function takes in sink and array of predecessors obtained from BFS as an input and returns the shortest paths in terms of nodes

4) funny_problem(first_points)

   This function takes in a list of start vertices as input and returns max-flow values and vertex-disjoint paths

Pseudocode

1) Function: path_augmenting_function(Input: path, Graph): Output Graph

```
        edge_weight =1
        i = 1
        while i < length of shortest paths
            first = first vertex of that edge
            last = last vertex of that edge
            Graph[first][last] -= edge_weight
            Graph[last][first] += edge_weight
        return Graph
```

2) Function max_flow_cal(Input: Graph): Output max_flow and shortest_paths with nodes

```
        Different_paths =[] //list to store shortest paths
        Nodes, predecessor = bfs(csr_matrix(Graph), directed=True, return_predecessors=True)
```

Function call obtain_path(Input: sink, predecessor_array): Output shortest path as nodes

different_paths.append(shortest_path) // adds shortest path for the input converted to graph

while source is a part of the shortest path

Function call path_augmenting_function(Input path, Graph): Output Graph

Nodes, predecessor = bfs(csr_matrix(Output Graph),directed=True, return_predecessors=True)

Function call obtain_path(Input: sink, predecessor_array): Output shortest path as nodes

different_paths.append(shortest_path) // adds shortest path for the residual graph

return flow, paths with nodes

3) Function: obtain_path(Input: sink, predecessor_array): Output shortest path as nodes

temp = [] // list to store nodes

while node exists in bfs:

temp.append(that node)

assign this node value as index for predecessor_array

temp.reverse()

return temp

4) Function funny_problem(Input: List): Outputs max_flow and vertex disjoint paths

graph = np.zeros(2*d*d + 2) //Initialize a graph of this size with all 0s

Set the capacity of e_{in} to e_{out}=1

Set the capacity of e_{out} to f_{in} = 1

Set the capacity of f_{out} to e_{in} = 1

Set the capacity of s to e_{in} = 1

Set the capacity of f_{out} to t = 1

Function call max_flow_cal(Input: Graph): Output flow and shortest_paths with nodes

while i < length of the shortest paths

Track each disjoint path with a list

```
            while j < length of each disjoint path
                if( j + 1 < length of each disjoint path)
                    Find out the vertices in that disjoint path
                j = j+2
          Add all the vertices of the vertex-disjoint paths to a list
    return max-flow, vertex-disjoint paths vertices list
```

5) Function main()

```
          open(inputfile_from_file_path)
          n,m = line[0] // obtain n,m from line 0
          i =0
          while i<m:
             (x,y) = line[i+1]  //obtain start vertices by iterating though the input file
```

Function call for function: funny_problem(Input: List): Output max flow and vertex disjoint paths with vertices

```
          if(max_flow ! = m)
               print( No solution)
          else
               print(Yes)
               while i < m
                    print(vertices[0]: vertices[i])
```

Programming Language

I have used Python for the programming assignment. Necessary modules such as numpy, scipy were imported for the functioning of the program.

Input

The input for this program is given from a file and this is read from the main function of the program

The format for the input is as follows

1) The first line of the file will contain n and length of m (otherwise number of start points) values separated by a ‘,’

2) From the second line onwards the input file contains the x,y coordinates of the start vertices in the format as shown: (x,y)

3) The input files are present in the SampleTestCases folder of the zip file uploaded

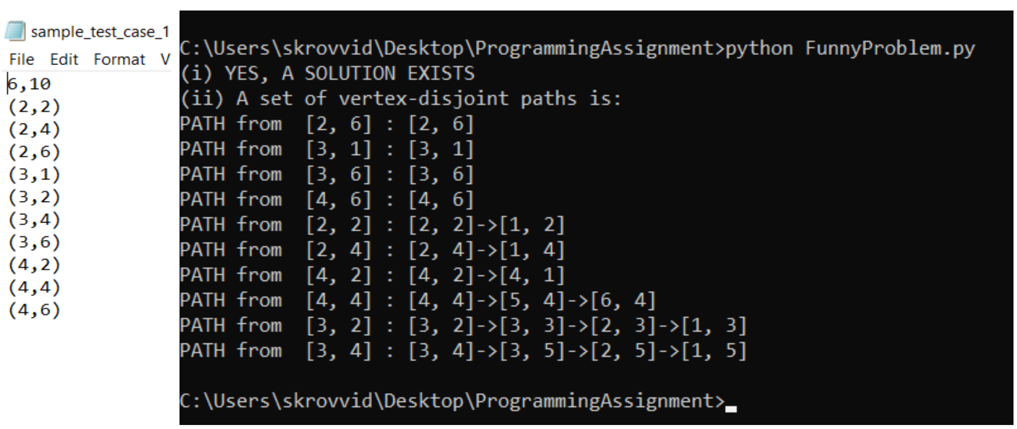
4) The input files I have submitted are sample_test_case_1.txt, sample_test_case_2.txt, and sample_test_case_3.txt

5) The input file is opened using this command

   file_path = open("SampleTestCases\sample_test_case_1.txt", 'r')

Results:

The results obtained are in line with the assignment requirements and are as shown below in Fig.1, Fig.2, and Fig.3 along with their corresponding inputs taken.


```
sample_test_case_1
File  Edit  Format  V
6,10
(2,2)
(2,4)
(2,6)
(3,1)
(3,2)
(3,4)
(3,6)
(4,2)
(4,4)
(4,6)
```

```
C:\Users\skrovvid\Desktop\ProgrammingAssignment>python FunnyProblem.py
(i) YES, A SOLUTION EXISTS
(ii) A set of vertex-disjoint paths is:
PATH from  [2, 6] : [2, 6]
PATH from  [3, 1] : [3, 1]
PATH from  [3, 6] : [3, 6]
PATH from  [4, 6] : [4, 6]
PATH from  [2, 2] : [2, 2]->[1, 2]
PATH from  [2, 4] : [2, 4]->[1, 4]
PATH from  [4, 2] : [4, 2]->[4, 1]
PATH from  [4, 4] : [4, 4]->[5, 4]->[6, 4]
PATH from  [3, 2] : [3, 2]->[3, 3]->[2, 3]->[1, 3]
PATH from  [3, 4] : [3, 4]->[3, 5]->[2, 5]->[1, 5]

C:\Users\skrovvid\Desktop\ProgrammingAssignment>
```

Fig.1 Corresponds to sample_test_case_1.txt

```
sample_test_case_2
File  Edit  Format  V
6,11
(2,2)
(2,4)
(2,6)
(3,1)
(3,2)
(3,4)
(3,5)
(3,6)
(4,2)
(4,4)
(4,6)
```

```
C:\Users\skrovvid\Desktop\ProgrammingAssignment>python FunnyProblem.py
NO, A SOLUTION DOES NOT EXIST

C:\Users\skrovvid\Desktop\ProgrammingAssignment>
```

Fig.2 Corresponds to sample_test_case_2.txt

```
sample_test_case_3
File  Edit  Format  V
6,11
(2,2)
(2,4)
(2,6)
(3,1)
(3,2)
(3,4)
(3,6)
(4,1)
(4,2)
(4,4)
(4,6)
```

```
C:\Users\skrovvid\Desktop\ProgrammingAssignment>python FunnyProblem.py
(i) YES, A SOLUTION EXISTS
(ii) A set of vertex-disjoint paths is:
PATH from  [2, 6] : [2, 6]
PATH from  [3, 1] : [3, 1]
PATH from  [3, 6] : [3, 6]
PATH from  [4, 1] : [4, 1]
PATH from  [4, 6] : [4, 6]
PATH from  [2, 2] : [2, 2]->[1, 2]
PATH from  [2, 4] : [2, 4]->[1, 4]
PATH from  [4, 2] : [4, 2]->[5, 2]->[5, 1]
PATH from  [4, 4] : [4, 4]->[5, 4]->[6, 4]
PATH from  [3, 2] : [3, 2]->[3, 3]->[2, 3]->[1, 3]
PATH from  [3, 4] : [3, 4]->[3, 5]->[2, 5]->[1, 5]

C:\Users\skrovvid\Desktop\ProgrammingAssignment>
```

Fig.3 Corresponds to sample_test_case_3.txt

References:

[1] https://stackoverflow.com/questions/9833516/how-to-find-all-vertex-disjoint-paths-in-a-graph

[2] https://walkccc.me/CLRS/Chap26/Problems/26-1/