VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DISCRETE MATHEMATICS

ASSIGNMENT REPORT

Advisor:
Students:

HO CHI MINH CITY, NOVEMBER 2021

Contents

1 Member list & Workload

No.	Fullname	Student ID	Problems	Percentage of work
1			- Text Text. - Text.	40%
2			- Relation & Counting: 4, 5, 6 Bonus: 4, 5, 6. - Graph: 1, 2, 3, Bonus: 1, 2, 3.	20%

2 Introduction To Assignment

We will discuss several common problems such as Shortest-Path Algorithms and Network Flow Problem in graph theory in this report. Not only are these algorithms useful in practice, they are also interesting because in many real-life applications.

3 Solving Assignment Problem

3.1 Problem 1

3.1.1 Kth Shortest Path Algorithm

3.1.1.a Algorithm

The idea is using DFS (Deep-First Search) algorithm to find all possible paths from the source to the destination and adding loops to path if graph contains loops then storing them in the array. Then we will calculate the length of each path and sort them in ascending order. The k-th shortest path is the k-th element in the sorted array.
Example:

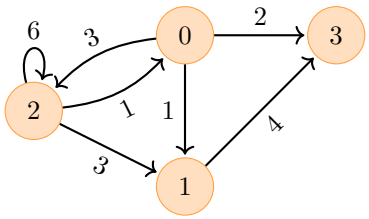

Figure 1: Print 2nd Shortest Path from 2 to 3

We set the THRESHOLD to be 3, which means we will only consider the paths with the maximum loop length less than 3. After running the algorithm, we will get the following paths (We consider several paths):

- 2, 0, 1, 3 Length: 6
- 2, 0, 3 Length: 3
- 2, 1, 3 Length: 7

- 2, 2, 0, 1, 3 Length: 12

Then we sort the paths in ascending order and get the resulting paths from the sorted array:

- 2, 0, 3 Length: 3
- 2, 0, 1, 3 Length: 6
- 2, 1, 3 Length: 7
- 2, 2, 0, 1, 3 Length: 12

So the second Shortest Path from 2 to 3 is the path 2, 0, 1, 3 with length 6.

3.1.1.b Code Input

```cpp
#include "GRAPH.h"
//THRESHOLD of loop has been set in GRAPH.h

int main()
{
   // Create a graph given in the above diagram
   Graph g(4, false); //have loops
   g.addEdge(0, 1, 1);
   g.addEdge(0, 2, 3);
   g.addEdge(0, 3, 2);
   g.addEdge(2, 0, 1);
   g.addEdge(2, 1, 3);
   g.addEdge(1, 3, 4);
   g.addEdge(2, 2, 6);
   int s = 2, d = 3;
   g.printAllPaths(s, d);
   Path p = g.kthPath(s, d, 2);
   p.printPath();
   return 0;
}
```

Figure 2: Input for example in Problem 1a

3.1.2 Kth Shortest Path Without Loops

3.1.2.a Algorithm

By using DFS algorithm and mark visited node, we can find the shortest path from the source to the destination without loops.
From the example of Fig. 1, we get three shortest paths without loops from the source to the destination:

- 2, 0, 1, 3 Length: 6
- 2, 0, 3 Length: 3
- 2, 1, 3 Length: 7

Then we sort the paths in ascending order and get the resulting paths from the sorted array:

- 2, 0, 3 Length: 3
- 2, 0, 1, 3 Length: 6
- 2, 1, 3 Length: 7

So the second Shortest Path from 2 to 3 without loops is the path 2, 0, 1, 3 with length 6.

3.1.2.b Code Input

```cpp
#include "GRAPH.h"
//THRESHOLD of loop has been set in GRAPH.h

int main()
{
   // Create a graph given in the above diagram
   Graph g(4, false); //doesn't have loops
   g.addEdge(0, 1, 1);
   g.addEdge(0, 2, 3);
   g.addEdge(0, 3, 2);
   g.addEdge(2, 0, 1);
   g.addEdge(2, 1, 3);
   g.addEdge(1, 3, 4);
   g.addEdge(2, 2, 6);
   int s = 2, d = 3;
   g.printAllPaths(s, d);
   Path p = g.kthPath(s, d, 2);
   p.printPath();
   return 0;
}
```

Figure 3: Input for example in Problem 1b

3.1.3 Shortest Path With Constraint Edges

3.1.3.a Algorithm

We construct a path from the source to the destination with some edge must be included in the path by breakdown the path into smaller paths. For example, we want to create a path from 2 to 3 with (0,1) and (1,3) edges must be included. Then we will find the shortest path from 2 to 0, 0 to 1, 1 to 1 (which length of path is 0 if there is no loop), 1 to 3 and 3 to 3 (which length of path is 0 if there is no loops) and combine them to get the shortest path from 2 to 3. So the resulting path is 2, 0, 1, 3.

3.1.3.b Code Input

```cpp
#include "GRAPH.h"


int main()
{
   // Create a graph given in the above diagram
   Graph g(4, true);
   g.addEdge(0, 1, 1);
   g.addEdge(0, 2, 3);
   g.addEdge(0, 3, 2);
   g.addEdge(2, 0, 1);
   g.addEdge(2, 1, 3);
   g.addEdge(1, 3, 4);
   g.addEdge(2, 2, 6);
   g.addConEdge(0, 1); // add constraint edge
   g.addConEdge(1, 3); // add constraint edge
   int s = 2, d = 3;
   Path p = g.pathWithEdge(s, d);
   p.printPath();
   return 0;
}
```

Figure 4: Input for example in Problem 1b

3.2 Problem 2

3.2.1 Maximum Flow With Ford-Fulkerson Algorithm

3.2.1.a Algorithm

We create a residual graph from the original graph. Then we will use Ford-Fulkerson algorithm to find the maximum flow from the source to the destination. To find an augmenting path, we can either do a BFS (Breadth-first search) or DFS (Depth-first search) of the residual graph. We have used BFS in our implementation. Using BFS, we can find out if there is a path from source to sink. The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges. We need to add path flow along reverse edges because may later need to send flow in reverse direction.

Example:

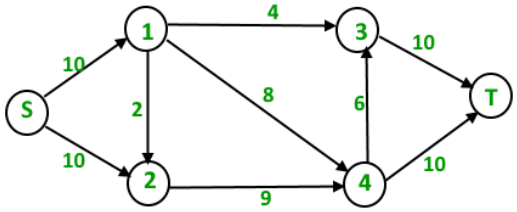

Figure 5: Example in Problem 2

After running the algorithm, we get the result is 19.

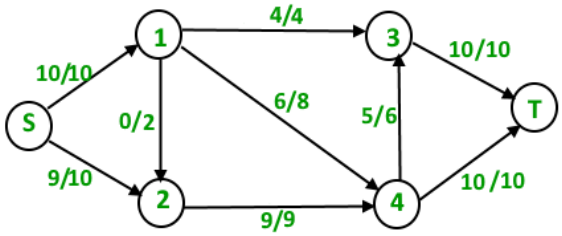

Figure 6: Example in Problem 2

3.2.1.b Code Input

According to the example in Fig. 5, we construct adjacency matrix which is graph[i][j] is the initial capacity of edge (i , j):

```
#include "fordFulkerson.h"
// Driver Code
int main()
{
  // EXAMPLE FF ALGORITHM
   int graph[V][V] = {{0, 10, 10, 0, 0, 0},
                      {0, 0, 2, 4, 8, 0},
                      {0, 0, 0, 0, 9, 0},
                      {0, 0, 9, 0, 0, 10},
                      {0, 0, 0, 6, 0, 10},
                      {0, 0, 0, 0, 0, 0}};
   int ff_source = 0;
   int ff_des = 5;
   cout << "Maximum flow of Ford-Fulkerson Algorithm from " << ff_source << " to " << ff_des << " is " <<
fordFulkerson(graph, 0, 5) << endl;
}
```

Figure 7: Maximum Flow With Ford-Fulkerson Algorithm

3.2.2 Maximum Flow With Dinic Algorithm

3.2.2.a Algorithm

In Dinic's algorithm, we use BFS to check if more flow is possible and to construct level graph. In level graph, we assign levels to all nodes, level of a node is shortest distance (in terms of number of edges) of the node from source. Once level graph is constructed, we send multiple flows using this level graph. After each iteration, we also check if more flow is possible and proceed only if possible.
In example of Fig. 5, we create a level graph with first iteration.

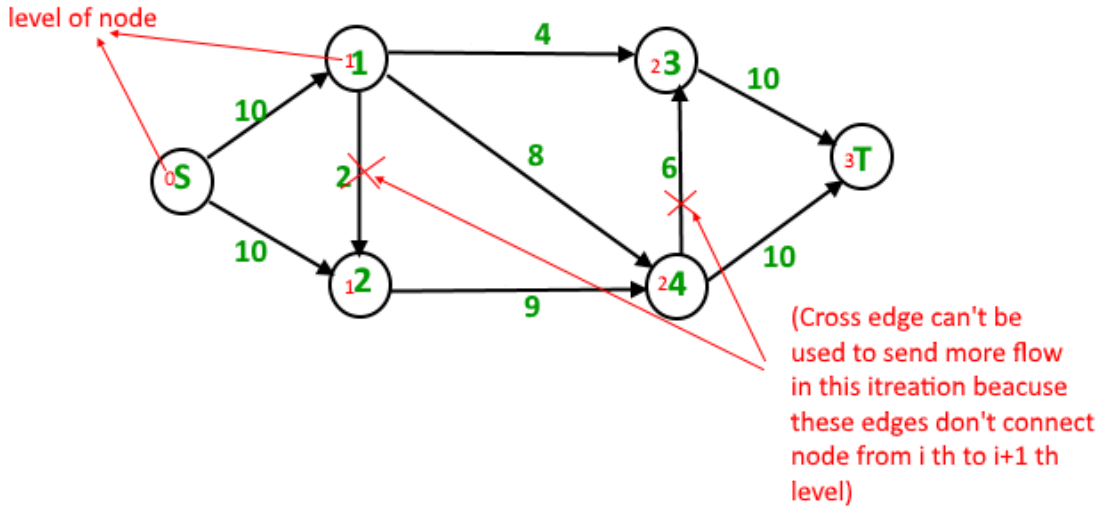

Figure 8: Level graph of first iteration from example in Fig. 5

From Fig. 8, we receive the flow path with block level (means every flow path should have levels as 0, 1, 2, 3)

- 4 units of flow on path s – 1 – 3 – t.
- 6 units of flow on path s – 1 – 4 – t.
- 4 units of flow on path s – 2 – 4 – t.

We continue iteration until there is no more flow.

3.2.3 Code Input

We input graph with adjacency list with addEge(s,d,w) function with s and d is an edge of the graph and w is the initial capacity of the edge.

```
#include "Dinic.h"
int main()
{

   //EXAMPLE DINIC ALGORITHM

   Graph g(6);
   g.addEdge(0, 1, 10);
   g.addEdge(0, 2, 10);
   g.addEdge(1, 2, 2);
   g.addEdge(1, 3, 4);
   g.addEdge(1, 4, 8);
   g.addEdge(2, 4, 9);
   g.addEdge(3, 5, 10);
   g.addEdge(4, 3, 6);
   g.addEdge(4, 5, 10);
   int d_source = 0;
   int d_des = 5;
   cout << "Maximum flow of Dinic's Algorithm from " << d_source << " to " << d_des << " is " << g.DinicMaxflow(0,
5) << endl;
}
```

Figure 9: Maximum Flow With Dinic Algorithm

3.2.4 Application

3.2.4.a Scenario

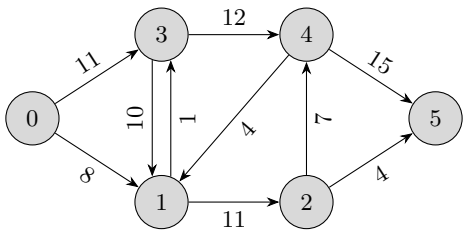

Figure 10: Maximum number of vehicles that can be transported from s to t at a time

From the Fig. 10, we assume that s is Ben Thanh Market and t is Bach Khoa University. The remaining nodes are the intersections of the roads. We assume that the capacity of the edges is the number of vehicles that can be transported from s to t at a time.
By using the Ford-Fulkerson or Dinic's algorithm, we can find the maximum vehicle that can be transported from s to t at a time without causing the congestion.

3.2.4.b Solving Problem

```cpp
#include "Dinic.h"
// Driver Code
int main()
{

   Graph g(6);
   g.addEdge(0, 1, 8);
   g.addEdge(0, 3, 11);
   g.addEdge(1, 2, 11);
   g.addEdge(1, 3, 1);
   g.addEdge(2, 4, 7);
   g.addEdge(2, 5, 4);
   g.addEdge(3, 1, 10);
   g.addEdge(3, 4, 12);
   g.addEdge(4, 1, 4);
   g.addEdge(4, 5, 15);
   int d_source = 0;
   int d_des = 5;
   cout << "Maximum flow of Dinic's Algorithm from " << d_source << " to " << d_des << " is " << g.DinicMaxflow(0,
5) << endl;
}
```

Figure 11: Solving Application With Dinic Algorithm

By using Dinic's algorithm, we find that the maximum number of vehicles that can be transported from s to t at a time is 19 without causing the congestion.