Ho Chi Minh City University of Technology
— Faculty of Computer Science and Engineering —

Assignment Report

Discrete Structure (CO1007) - Semester 211

Group 4:
Hồ Vũ Khánh Vy - 2052329
Hoàng Đỗ Phương Nguyên - 1952360
Trần Đinh Anh Hùng - 2053066
Tôn Huỳnh Long - 2052153
Trần Cao Duy Trường - 2052299

Ho Chi Minh City, 04 December 2021

Contents

1 Preface

As part of the Discrete Structure for Computing course, students must complete a project with a group. This report is the culmination of our group's work on solving the problems presented in the assignment file.

The problems are as follows:

Task a: k^{th} shortest path problem

a1: Simple, with-loop condition - Covered in subsection 2.2

a2: Loopless condition - Covered in subsection 2.1

a3: Mandatory edge condition - Covered in subsection 2.3

Task b: Max flow problem

b1: Two algorithms for solving max flow
- Covered in subsection 3.2 and subsection 3.3

b2: Practical traffic problem - Covered in subsection 3.5

Our report will cover the basic steps to the algorithms used, as well as the methodology and data used in the practical traffic problem. A more extensive explanation of the code is written in the form of comments in the respective code file.

Our group's work was divided as follows:

Name	**Work**	**% point**
Khánh Vy	- Task a3: Algorithm and coding (main role) Graph example - Task a1, a2: Algorithm and coding (support)	20%
Phương Nguyên	- Task a2: Algorithm and coding (main role) - Task a1, a3: Algorithm and coding (support) - Summary report slides	20%
Anh Hùng	- Task a1: Algorithm and coding (main role) Graph example - Task a2, a3: Algorithm and coding (support)	20%
Huỳnh Long	- Task b: Data Structure design Task b1: Push-Relabel algorithm - Task b2: Traffic data collection and organization - LaTeX report organization and typesetting	20%
Duy Trường	- Task b1: Ford - Fulkerson algorithm - Task b2: Traffic data validation and testing - Task a1, a2, a3: Coding support	20%

2 Task a: The k^{th} shortest path problem

2.1 Loopless requirement

2.1.1 Algorithms

Dijkstra's Algorithm

First of all, all nodes are initialized with distance "infinite", and at the same time, a source node 0 is also initialized. After that, the distance of the starting node is marked as permanent, all other distances as temporarily. Next, the source node is set actively. Then, calculate the temporary distances of all neighbour nodes of the active node by summing up its distance with the weights of the edges. If such a calculated distance of a node is smaller than the current one, update the distance and set the current node as ancestor. This step is also called update and is Dijkstra's central idea. Then, set the node with the minimal temporary distance as active and mark its distance as permanent. Finally, repeat from the fourth to the end until there are not any nodes left with a permanent distance, which neighbours still have temporary distances.
Here is the pseudo-code for the algorithm:

```
Procedure Dijkstra(G, a)
//Initialization Step
forall vertices v
Label[v] : =
Prev[v] : = -1
endfor
Label(a) : = 0 //a is the source node
S : = empty

//Iteration step
while z  S
u : = a vertex not in S with minimal Label
S : = S{ u }
forall vertices v not in S
if (Label[u] + Wt(u, v)) < Label(v)
    then begin
    Label[v] : = Label[u] + Wt(u, v)
    Pred[v] : = u
    end
    endwhile
```

Yen's Algorithms

- ***Terminology and notations:***

  N: the size of the graph, for example: total nodes of a graph

  i: the i^{th} node of the graph, i_1 is the source node and i_N is the sink node of the graph

  d_{ij} : the cost (distance) from i to j, with $i \neq j$ and $d_{ij} > 0$.

  A^k : the k^{th} shortest path from 1 to N

  A_i^k : the deviation path from A^{k-1} at node i^k , where i ranges from 1 to Q_k (Q_k is the maximum value of i)

  R_i^k: the root path of A_i^k that follows $A^k - 1$ until the i^{th} node of $A^k - 1$

  S_i^k : the spur path of A_i^k that starts at the i^{th} node of A_i^k and ends at the sink node.

- ***Description:***

  The algorithm is divided into two steps. First step is determining the first k-shortest path A^k and the second step is determining all other k-shortest paths. It is assumed that A will contain the k-shortest path while B will hold the potential k-shortest path. To determine A^1 , the shortest path from the source to the sink node, Dijkstra algorithm will be used.

  To find A^k where k ranges from 2 to k, the algorithm assumes that all paths from A^1 to A^{k-1} have previously been found. The k iteration can be divided into two steps, finding all the deviation A_i^k and opting a minimum length path to become A^k. It is noted that in this iteration, i ranges from 1 to Q_k^k

  The first process can be further subdivided into three operations, choosing the R_i^k , finding all the deviation S_i^k and then adding A_i^k to the container B. The root path R_i^k is chosen by finding a sub-path in A^{k-1} that follows the first i nodes of A^j, where j ranges from 1 to $k-1$. Then, if a path is found, the cost of edge $d_{i(i+1)}$ of A^j is set to infinity. Next, the spur path S_i^k is found by computing the shortest path from the spur node I to the sink node. The removal of the previously used edges from i to $i+1$ ensures that the spur path is different. The addition $A_i^k = R_i^k + S_i^k$ is added to B. Next, the edges that were removed are set to their initial values.

  The second process determines a suitable path for A^k by finding the path in the container B with the lowest cost (distance). This path will

be eliminated from the container B and then inserted into the container A and the algorithm continues to the next iteration.

- ***Pseudo-code:***

```
function KShortestPath_Loopless(Graph, source, sink, K) :
    // Determine the shortest path from the source to the
sink.
    A[0] = Dijkstra(Graph, source, sink);
// Initialize the set to store the potential kth shortest
path.
B = [];

for k from 1 to K :
// The spur node ranges from the first node to the next to
last node in the previous k-shortest path.
for i from 0 to size(A[k - 1]) - 2 :

    // Spur node is retrieved from the previous k-shortest
path, k - 1.
    spurNode = A[k - 1].node(i);
// The sequence of nodes from the source to the spur node of
the previous k-shortest path.
rootPath = A[k - 1].nodes(0, i);

for each path p in A:
if rootPath == p.nodes(0, i) :
    // Remove the links that are part of the previous shortest
paths which share the same root path.
    remove p.edge(i, i + 1) from Graph;

for each node rootPathNode in rootPath except spurNode :
remove rootPathNode from Graph;

// Calculate the spur path from the spur node to the sink.
// Consider also checking if any spurPath found
spurPath = Dijkstra(Graph, spurNode, sink);

// Entire path is made up of the root path and spur path.
totalPath = rootPath + spurPath;
// Add the potential k-shortest path to the heap.
if (totalPath not in B) :
   B.append(totalPath);

// Add back the edges and nodes that were removed from the
graph.
restore edges to Graph;
restore nodes in rootPath to Graph;
```

```
if B is empty :
// This handles the case of there being no spur paths, or no
spur paths left.
// This could happen if the spur paths have already been
exhausted (added to A),
// or there are no spur paths at all - such as when both the
source and sink vertices
// lie along a "dead end".
break;
// Sort the potential k-shortest paths by cost.
B.sort();
// Add the lowest cost path becomes the k-shortest path.
A[k] = B[0];
// In fact we should rather use shift since we are removing
the first element
B.pop();

return A;
```

2.1.2 Process

We will use this graph to test the algorithm:

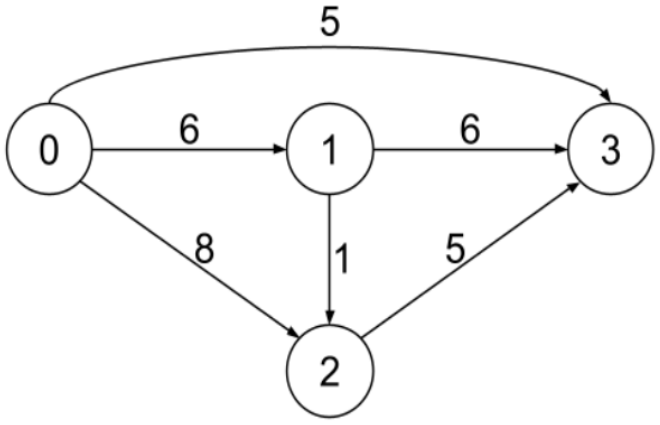

Figure 1: Sample graph to test the algorithm.

1. Input the graph. Note that input the graph via 2 sub-steps: Firstly, input the cost of each edge using adjacency matrix and then, input the direction of the graph using the function addEdge()
2. Calculate all the paths from the source node to the end node, and their total length and store them to an array.
3. Input the number k and it will calculate the k^{th} shortest length then show its path.

2.1.3 Results

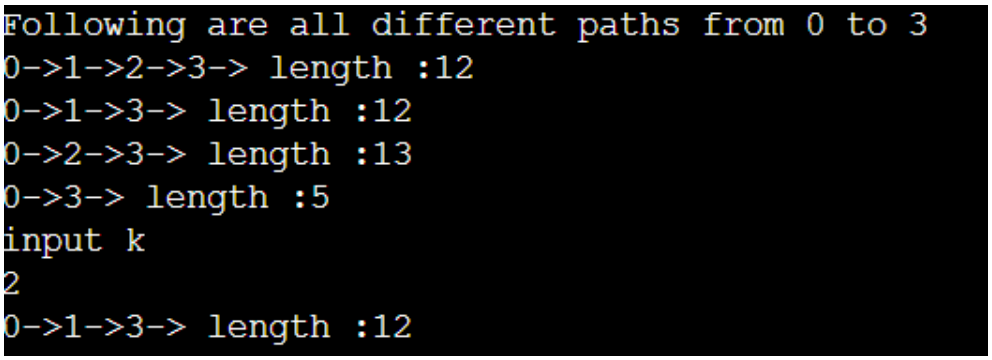

Figure 2: Output of the program, with input $k = 2$.

2.2 With loop requirement

2.2.1 Solution

Following the algorithm of the last section:

- For K = 1, we apply Dijkstra's algorithm to the graph with k = 1 and we are allowed to remove all the loops. So the solution for k = 1 for the graph would be: $0 \to 1 \to 2 \to 3$ with the length: 11
- For K = 2, we need to transform the graph for this part: A loop will now be considered as a new node. For example, we have a loop at node 0, so from node 0 we create a new node name 1. Node 1 will have the same edges as node 0. The same will apply for other nodes.
  The new graph will have the following pairs replacing loops: node 0 and 1 replacing the loop at node 0, node 2 and 3 replacing the loop at node 1, node 4 and 5 replacing the loop at node 2. We then apply the Yen's algorithm for the new graph to figure out the solution.

2.2.2 Process

1. We input the graph performed by matrix and add the edge.
2. We calculate all the paths from the source node to the end node, and their total length and store them to an array.
3. We input the number k and it will calculate the kth shortest length then show its path.

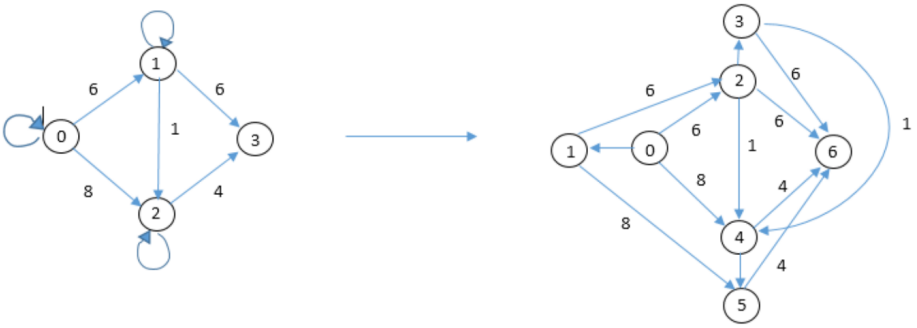

Figure 3: The graph before and after transforming. To make it easy to visualize, we consider that there is no length for the loop.

2.2.3 Results

```
Following are all different paths from 0 to 6
0->1->2->3->4->5->6 length :11
0->1->2->3->4->6 length :11
0->1->2->3->6 length :12
0->1->2->4->5->6 length :11
0->1->2->4->6 length :11
0->1->2->6 length :12
0->1->4->5->6 length :12
0->1->4->6 length :12
0->2->3->4->5->6 length :11
0->2->3->4->6 length :11
0->2->3->6 length :12
0->2->4->5->6 length :11
0->2->4->6 length :11
0->2->6 length :12
0->4->5->6 length :12
0->4->6 length :12
input k
3
0->2->3->4->6 length :11
```

Figure 4: Output of the program, with input $k = 3$.

2.3 Mandatory edge requirement

2.3.1 Solution and Process

The idea is to find all paths that are possible to go from s to d. If the path contains all mandatory paths we store it and its length in an array for later comparison. Assume each node has one loop. then the loop will be extended into another node. There are maximum $2n$ nodes (each node has one loop in theory) $\Rightarrow$ Graph $2n$ defined for n nodes.

Example if a no loop is 0-2 then extend the loop and the path is 0-1-2 (not for this exercise).
Following the below map and much like in the previous problem, at first the 0 node has one loop, then the loop will be extended into node 0 to node 1, and node 1 will be transformed into node 2 and so on. At the end, node 3 has no loop so there are 7 nodes at the end and node 3 was turned to be node 6.

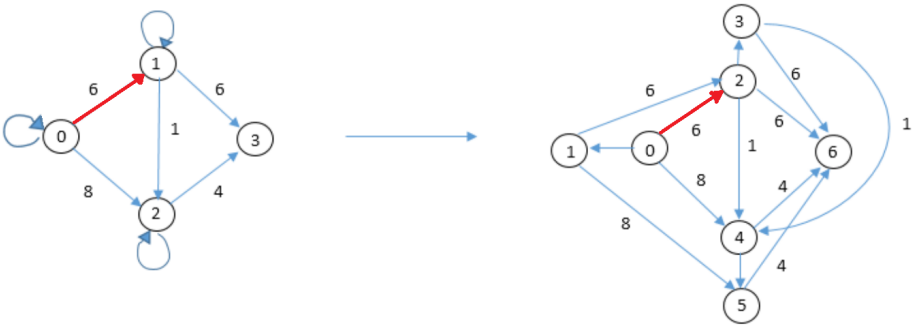

Figure 5: The graph before and after transforming.

After inputting the map, we will use another matrix with values 0 and 1, 1 (or logical TRUE) is defined as a mandatory edge and 0 (or logical FALSE) is defined as not. In this case the mandatory edge is the edge from 0 to 2. Shortest path is the path without any loops. In case there are loops on the shortest path and the loop's length is 0 then the second shortest path is the path with the loop but remains the same length. Comparison will base on this rule to find the suitable path.

1. Find all possible paths as A1
2. After finding out one possible path, check whether the path contains mandatory edges or not.
3. If the path meets the requirement: contains mandatory edges then store the path and its length.
4. After finding all paths, input k.

2.3.2 Results

```
Following are all different paths from 0 to 6
0->2->3->4->5->6-> length :11
0->2->3->4->6-> length :11
0->2->3->6-> length :12
0->2->4->5->6-> length :11
0->2->4->6-> length :11
0->2->6-> length :12
input k
3
0->2->3->4->6-> length :11
```

Figure 6: Output of the program, with input $k = 3$.

As we can see, the output is very similar to that in the last requirement. The only difference is that every path found contains $0 \to 2$, which is our mandatory edge.

3 Task b: The maximum flow problem

3.1 Common code for this problem

For this task, we have decided to use a common data structure for both algorithm solutions.
The classes are structured as follows:

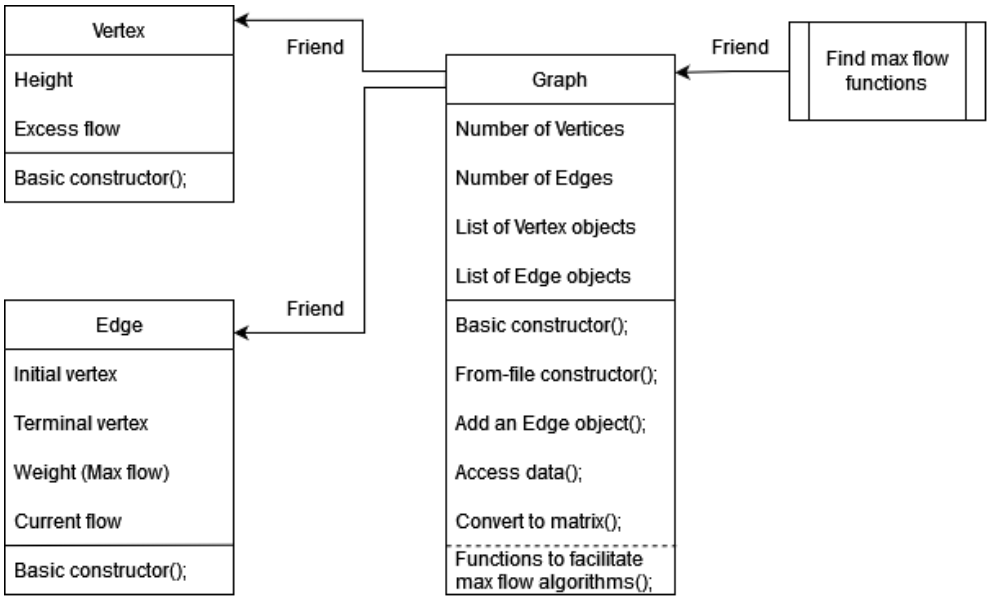

Figure 7: Diagram of our classes.

- **class Edge**
  The data of this class are integers u, v denoted the initial and the terminal of the edge, respectively; the integer containing the weight of the edge, for max flow problems it means the capacity of the edge; and an integer indicating the number of flow the edge currently has, this is used in the Push - Relabel algorithm.
  This class is friend with the Graph class, for ease of access when we write our algorithm.
  The constructor for the class only needs 3 parameters, because the flow at the beginning is usually 0.

- **class Vertex**
  This class only contains data for the Push-Relabel algorithm: integer h (height) and excess (excess flow)
  This class is also friend with the Graph class, for ease of access when we write our algorithm.
  The constructor for this class can be left empty, as its data is only needed for one specific algorithm.

- **class Graph**
  This class contains data on the number of vertices (V), the number of edges (E), as well as a vector of Vertex objects and a vector of Edge objects.
  One particular constructor for this class can also read a .txt file and convert it to the corresponding Graph object.
  There are methods to add an edge to the Edge object list, get V and E, or convert the graph into a matrix.

- **The maxFlow functions** are not class members and take a copy of the graph instead. And as such, they are friends with the Graph class and can access its private data. They return the max flow from the inputted source to the inputted sink.

3.2 The Ford - Fulkerson Algorithm

3.2.1 Pseudocode for the algorithm

Input: graph G with flow capacity c, a source node s, and a sink node t
Output: a maximum flow f from s to t
$k = 0$; $G^{(0)} = G$;
$c^{(0)}(u, v) = c(u, v)$, $c^{(0)}(v, u) = 0$, $\forall (u, v) \in G^{(0)}$;
While $\exists$ a path $\Pi^{(k)}(s, t)$ in $G^{(k)}$ such that $c^{(k)}(u, v) > 0$, $\forall (u, v) \in \Pi^{(k)}$ do
  Find $f(\Pi^{(k)}) = \min\{c^{(k)}(u, v) | (u, v) \in \Pi^{(k)}\}$;
  For each edge $(u, v) \in \Pi^{(k)}$ do
    If $(u, v) \in G$ then
      $c^{(k+1)}(u, v) = c^{(k)}(u, v) - f(\Pi^{(k)})$;
      $c^{(k+1)}(v, u) = c^{(k)}(v, u) + f(\Pi^{(k)})$;
    Else
      $c^{(k+1)}(u, v) = c^{(k)}(u, v) + f(\Pi^{(k)})$;
      $c^{(k+1)}(v, u) = c^{(k)}(v, u) - f(\Pi^{(k)})$;
  $k++$;

3.2.2 C++ code

For the purpose of differentiating the two algorithms, functions ending in "_FF" will be for the Ford - Fulkerson algorithm.
Only 1 extra function is designed for this implementation of the algorithm - it performs a Breath-First Check to see if a path from source to sink exists or not.

3.3 The Push - Relabel Algorithm

3.3.1 Intuition and steps of the algorithm

The intuition of the Push-Relabel algorithm is that we visualize the edges as pipes carrying a kind of fluid and the vertices as points with a certain height.
Naturally, if the point has fluid that can flow out, the fluid will flow from larger height to lower height. If there's no way for the point to **push** this excess away, we **relabel** it to a larger height and try again.
The excess fluid at a vertex v is the difference between the fluid flow in and the fluid flow out.

$$Excess_v = Inflow_v - Outflow_v$$

For the maximum flow problem, we would need the excess flow of each vertex except for the sink and the source to be 0 by the end of routine. This means that no flow is trapped at any vertex and thus, we obtain the maximum amount that can flow from source to sink.

Then, the Push-Relabel algorithm will function as follows:

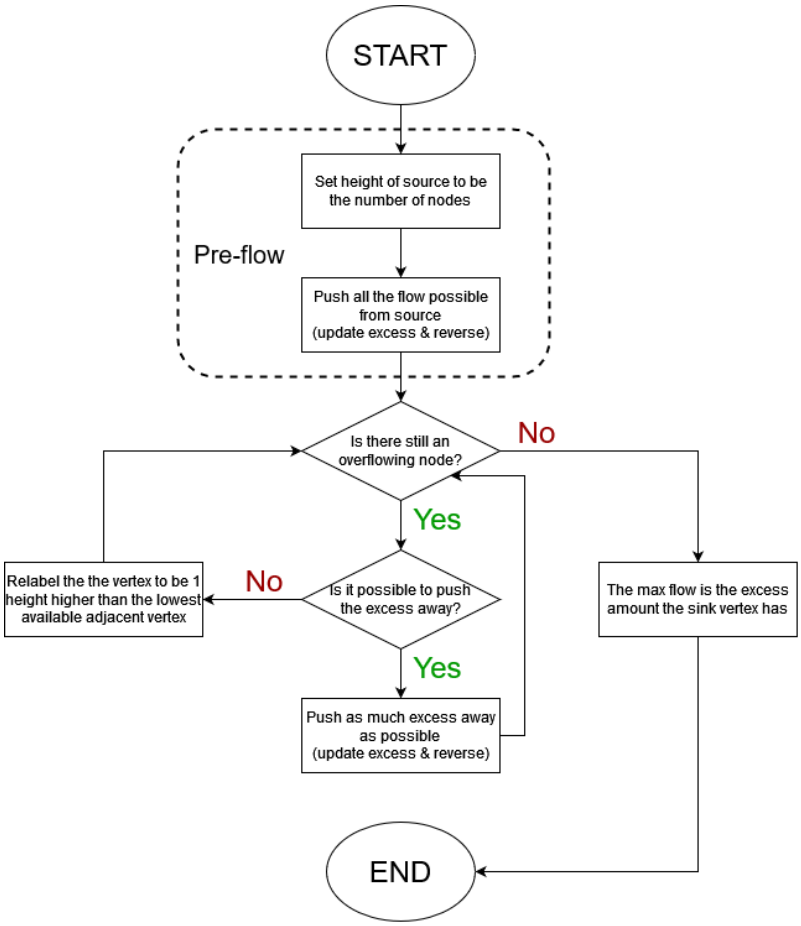

Figure 8: A flow chart of the steps of the Push-Relabel algorithm.

1. Set the height of the source vertex to be equal to the number of vertices of the graph and all other vertices to be 0. **Push** all the flows possible from the source. As it is the source, we don't care about its excess flow. This step is called **Pre-flow**.

2. The preflow process will cause the vertices that are terminals of edges from the source to have large amounts of excess. While there are vertices with excess in the graph (bar the source and sink), we will try to **push** the excess flow to another vertex; if that is not possible, then we will **relabel** that vertex so that its height is 1 more than the lowest adjacent vertex and try again. Whenever we **push** flow, a reverse edge is updated or added in the residual graph. This reverse edge allows

us to **push** flow back and "undo" some excess flow. The flow on this reverse edge is not added to excess because it counts as a "return" of flow.

3. Finally, the max flow of the graph is the total inflow of the sink vertex. Or, in other words, the excess flow of that vertex.

3.3.2 C++ code

For the purpose of differentiating the two algorithms, functions ending in "_PuRe" will be for the Push-Relabel algorithm.
Based on the flow chart in the previous section, the code for our algorithm is as below:

```
int maxFlow_PuRe(Graph G, const int source, const int sink)
{
    if (source < 0 || sink < 0 || source >= G.getV() || sink >= G.getV()) return -1; //check validity

    G.preFlow_PuRe(source);
    while (G.overflowVertex_PuRe(source, sink) != -1)
    {
        int u = G.overflowVertex_PuRe(source, sink);
        if (!G.push_PuRe(u))
            G.relabel_PuRe(u);
    }

    return G.getExcess(sink);
}
```

The functions "preFlow_Pure()" and "relabel_PuRe()" are void functions and only work on the data of the Graph object.
The functions "overflowVertex_PuRe()" will find the first node (excluding source and sink) that is overflowing and return its index. If the returned value is -1 then there's no overflow.
The function "push_PuRe()" returns a boolean value, FALSE if no push is possible and TRUE is it's possible and the function has already done it.
The specific workings of these functions are explained within the comments of the code.

3.4 Testing the algorithms

To test the algorithms we have written, let's make them solve this max flow problem in this graph from the supplemental slides.
Using our intuition, it's quite easy to see that the path along the middle is saturated at 5, the path along the bottom is saturated at 6. This leaves the top path being bottlenecked by the edge (B, D) and can only flow 4. We can try to divert some flow from the top path, but the nodes we can flow them to don't have any free capacity. So in total, the max flow is $5 + 6 + 4 = 15$.

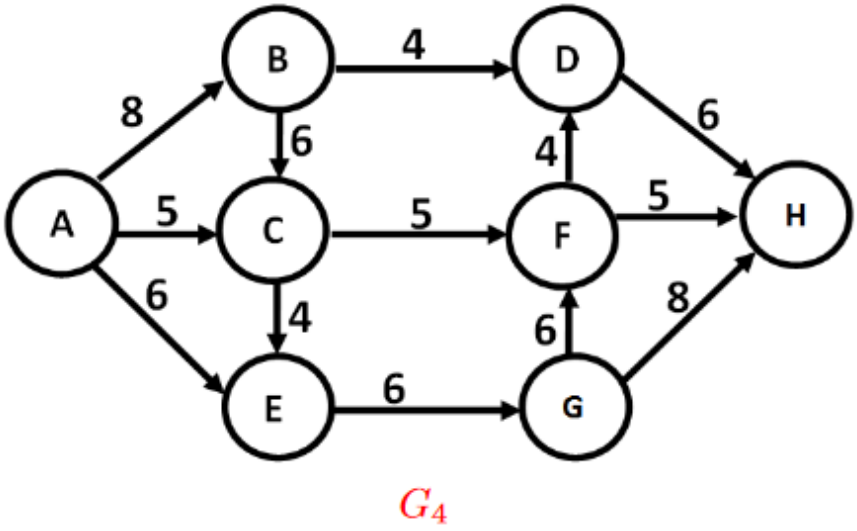

Figure 9: The graph G_5.

We will denode vertex A as 0, B as 1 and so on. The data for this graph is in the *test.txt* file. Then, running the program, we get:

```
D:/Graph/busData.txt
The matrix representing this graph is:
     0     8     5     0     6     0     0     0
     0     0     6     4     0     0     0     0
     0     0     0     0     4     5     0     0
     0     0     0     0     0     0     0     6
     0     0     0     0     0     0     6     0
     0     0     0     4     0     0     0     5
     0     0     0     0     0     6     0     8
     0     0     0     0     0     0     0     0
-----------------------
According to the Ford-Fulkerson algorithm, max flow is: 15
According to the Push-Relabel algorithm, max flow is: 15
-----------------------
```

Figure 10: The output of the program.

As we can see, both algorithms produce the result of 15.

3.5 Finding maximum flow in a practical situation

3.5.1 Problem

Suppose that there's an event at the Ho Chi Minh University of Technology, and students must travel from the first campus in District 10 to the second campus in Thu Duc District. Some students want to make the journey by bus, and we assume they don't care much about the ticket price or the time it takes. The time right now is 4:30pm, what is the maximum number of students that can travel between the two campuses?

3.5.2 Solution

First, we will pick some points as our vertices. Bus stops within walking distance of each other or with the point itself will count as one node for the purpose of solving this problem.
The nodes are:

0. The source node is the first campus, we will count the stops on both Lý Thường Kiệt street and Tô Hiến Thành street to be in this same vertex.
1. This node is the "Bảo Tàng Miền Đông" bus stop near Hoàng Văn Thụ Park.
2. This node is the "54 Phạm Văn Đồng" bus stop.
3. This node is the "Trụ Chiếu Sáng C185 Phạm Văn Đồng" bus stop.
4. This node is the "15/3 Kha Vạn Cân" bus stop.
5. This node is the "Công An Quận 9 xa lộ Hà Nội" bus stop.
6. This node is the "334 - 336 Xô Viết Nghệ Tĩnh" bus stop.
7. This node is the "794 Xa Lộ Hà Nội" bus stop, not too far from the Landmark 81.
8. This node is the "359 Lý Thái Tổ" bus stop
9. This node is the "59 - 61 Phạm Viết Chánh" bus stop
10. And of course, the sink node is at the second campus and the stops nearby it.

In order to estimate the capacity of students carried by bus between 2 given nodes, we will calculate the expected number of buses that will go on the path, then multiply it with the number of seats of the bus used by that line. Because HCMUT students are not the only ones using the bus, and crowded buses are not very pleasant to ride for a long distance, let us be conservative with our estimate and only take 60% of the theoretical maximum passengers as the capacity.
For example: Line 50 goes through the path $0-1-2-3-4-10$. This line operates until 6:30pm and the average time between buses is 15 minutes, so we can expect $120/15 = 8$ buses. The type of bus on this line carries at max 80 passengers. So that is a total capacity of $8 * 80 * 0.6 = 384$

Similarly, we can calculate the capacity of all edges as such:

Bus line (1)	**Path** (2)	**Closes at** (3)	**Time left** (minutes) (4) = (3) - 4:30pm	**Average time between buses** (minutes) (5)	**Buses expected** (6) = (4)/(5)	**Bus capacity** (7)	**Total capacity** (60% estimate) (8) = (6)*(7)*0.6	**Edges added to our graph** (u - v - weight) (9)
50	0 - 1 - 2 - 3 - 4 - 10	6:30pm	120	15	8	80	384	0 - 1 - 384 1 - 2 - 384 2 - 3 - 384 3 - 4 - 384 4 - 10 - 384
08	0 - 6 - 5 - 10	8:30pm	240	6	40	68	1632	0 - 6 - 1632 6 - 5 - 1632 5 - 10 - 1632
38	0 - 8	7:00pm	150	16	9	47	254	0 - 8 - 254
69	0 - 8	8:00pm	210	12	18	51	551	0 - 8 - 551
10	8 - 6 - 7 - 5 - 10	6:45pm	135	12	11	55	363	8 - 6 - 363 6 - 7 - 363 7 - 5 - 363 5 - 10 - 363
70-3	0 - 9	6:00pm	90	60	2	60	72	0 - 9 - 72
53	9 - 7 - 5 - 4 - 10	7:30pm	180	12	15	55	495	9 - 7 - 495 7 - 5 - 495 5 - 4 - 495 4 - 10 - 495

One crucial problem with our estimates, is that the time differences can change the actual number to possible flow. For example, a student catching bus 69 at vertex 0 at around 7:50pm may or may not be able to catch bus 10 at vertex 8 for it is too late. However, the total capacity of the path of bus 10 is quite small and comparatively only slightly higher than that bus 38, this counterbalances the flaw somewhat as there are not that many students catching the bus 69 who can then catch the bus 10 - meaning only early bus 69 taker can flow through.

Then, with our estimates for the buses completed, we can add them to a .txt file (specifically the file *busData.txt*) and run our program. And we can see that, as expected, both algorithms yield the same result of 2267.

```
The matrix representing this graph is:
     0   384     0     0     0     0  1448     0   805    72     0
     0     0   384     0     0     0     0     0     0     0     0
     0     0     0   384     0     0     0     0     0     0     0
     0     0     0     0   384     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0   879
     0     0     0     0   495     0     0     0     0     0  1811
     0     0     0     0     0  1448     0   363     0     0     0
     0     0     0     0     0   858     0     0     0     0     0
     0     0     0     0     0     0   363     0     0     0     0
     0     0     0     0     0     0     0   495     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
-----------------------
According to the Ford-Fulkerson algorithm, max flow is: 2267
According to the Push-Relabel algorithm, max flow is: 2267
-----------------------
```

Figure 11: The result of the program.

So, we can make the estimation that over 2000 students can complete the journey entirely by bus.

4 Conclusion

In closing, we have presented our solutions to the tasks given. For a clearer examination of our implementations, please check the included code files. We would like to thank you for reading through our report and we look forward to receiving ways to improve upon this work.