
# Informatics Institute of Technology
# Department of Computing

Module: 5SENG001W Object Oriented Programming

Degree Program: BEng Software Engineering

Module Leader: Mr. Sudharshana Welihindha

**Algorithm Coursework**

Student ID: **2018470**

UoW Number: **w1761930**

Tutorial Group: **D**

Student First Name: **Mohamed**

Student Surname: **Rashad**

"I confirm that I understand what plagiarism / collusion / contract cheating is and have read and understood the section on Assessment Offences in the Essential Information for Students. The work that I have submitted is entirely my own. Any work from other authors is duly referenced and acknowledged"

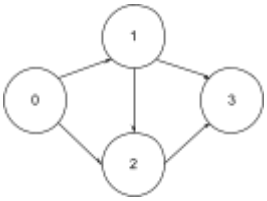
**A) Graph Representation**

There are two main approaches to represent a graph, a) Adjacency Matrix b) Adjacency List
This implementation uses Adjacency List.

**Adjacency List**

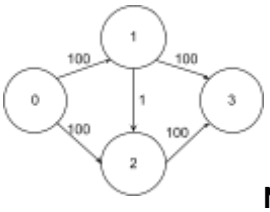
V - Vertices, E - Edges, e - edges of a given vertex

In an adjacency list the e are represented in a 2d array where the number of vertices is the length of the parent array and number of e is the length of the child array. Therefore the indices of the parent array represents a vertex and the child array at that index contains all the edges of that vertex. Therefore it has a space complexity of $O(V+E)$.




The adjacency list representation of this graph would be,
`graph = [[1,2], [2,3], [3], []]`

Compared to adjacency matrix this allows to find all the edges that go from a given node in $O(n)$ time which is way efficient than $O(n^2)$, since it only has to iterate over the edges a given vertex contains. Two main techniques that can be used to represent the capacity are maintaining an adjacency matrix or a dictionary(similarly to adjacency list). Here dictionary is chosen over the adjacency matrix since it has a better space complexity which is again $O(V+E)$.




This can be represented as follows,
`graph = [[1,2], [2,3], [3], []]`
edgeCap = [{ 1: 100, 2: 100 }, {2: 1, 3:100}, {3:100}, {}]

**NOTE:** A fully object oriented approach to this, can be taken as well in order to represent the graph with vertex, this implementation has taken a hybrid approach, where each vertex represents Node Class in which the adjacency nodes and their capacities are stored in an ArrayList and and a HashMap.

## Algorithm - Edmand Karp

Edmand Karp is a specific implementation of Ford Fulkerson method which uses breadth first search(BFS) instead of depth first search to find augmenting paths. BFS finds a shortest path from a vertex to another. In this algorithm, it maintains a residual graph which has reverse edges along with the normal edges where in reverse edges the capacity is zero and it decreases (becomes minus) when flow is added to the corresponding normal edges. Therefore it allows undo operations when finding max flow.
In order to find an augmenting path it uses the residual graph. An augmenting path is a path from sink to source where the residual capacity is positive (>0) in all edges. The residual capacity simply means capacity minus the flow. Therefore when the flow is minus in residual graph flow can be sent through them. Once a path is found it takes the minimum residual

capacity and adds that amount of flow across all the edges in that path. This process happens until there are no more augmenting paths to be found. By getting the total flow that got added, max flow can be found. This Algorithm has a time complexity of $O(VE^2)$.

**B) Algorithm on bridge_1.txt (smallest benchmark**

```
C - Capacity, F - Current Flow, R - Residual Capacity
Edge - <from> F/C---->R <to>

Augmenting Path - 1 | Min Residual Capacity - 1
1  1/4---->3  5
0  1/1---->0  1

Augmenting Path - 2 | Min Residual Capacity - 1
4  1/1---->0  5
0  1/4---->3  4

+---------------+---------------+---------------+---------------+---------------+
|  File name    |    Vertices   |     Edges     |    Max Flow   |  Duration(ms) |
+---------------+---------------+---------------+---------------+---------------+
|  bridge_1.txt |             6 |             9 |             2 |          9.94 |
+---------------+---------------+---------------+---------------+---------------+
```

## C) Performance Analysis

In order to understand performance of the algorithm, BFS should be considered. In BFS it goes through all the edges by iterating over vertices and then iterating over the edges the vertex for that iteration contains. Therefore it does a constant amount of work (removing the vertex from the queue and etc) for each iteration over vertex and constant amount work at each iteration through edges (for the vertex of that iteration). Therefore it takes O(V+E) time complexity to complete. But since all the V has at least one incident edge(an edge pointing towards the V), n <= 2m can be derived (consider residual graph for 2m). Therefore the time complexity can be simplified to O(E). Something to note is that to traverse the edges of a vertex, it takes only O(n) time since an adjacency list is used as described above.

With that in mind let's consider the time complexity of Edmand Karp. It uses BFS to find an augmenting path, and then increases the flow of that path. In order to get the growth function the time it takes to increase the flow can be ignored since BFS takes much longer. In Edmond Karp when a path is augmented at least one edge gets saturated ( the edges with the minimum residual capacity). And if assumed that it never comes back the time complexity would be $E*E$. But that's not the case, since in the residual graph the reverse edges can be used to add flow and as a result the saturated edges can come back. Hence if the maximum number of times that an edge gets saturated can be found, the Big O can be evaluated.
Consider running the algorithm on a network with flow f ($G_f$ ), if the shortest distance between s to v is $d_f(u,v)$. This path will always increase in the running time of the algorithm. for example if,
$d_f(u,v) = d_f(s,u) + 1$ and if (u,v) gets saturated once, in order for that edge to come back BFS should find (v,u) the reverse edge as an augmenting path. Therefore when it comes back the path must have increased. If the new path for flow $f\sim$ is $d_{f\sim}(u,v)$,
$d_{f\sim}(u,v) > d_f(s,u) + 1$. Hence if it increases by one each time it can only increase V times (it's proved that it increases by 2 hence upto V/2 times). From this it can be concluded that the worst case time complexity of this algorithm is is $O(E*E*V) = O(VE^2)$.