An Experimental Evaluation of Variations of Ford-Fulkerson Algorithm for Maximum Network Flow

Rifayat Samee
University of Western Ontario
UWO ID: msamee
Student Number: 250994164
Email: msamee@uwo.ca

Introduction

Network flow is one of the extensively studied areas of graph theory. There are several efficient algorithms to find maximum flow in a flow network. The Ford-Fulkerson Algorithm is a greedy algorithm that computes maximum flow in a flow network. It was first proposed by L. R. Ford, Jr. and D. R. Fulkerson in 1956[1]. It is often called Ford-Fulkerson method instead of algorithm because the approach of finding the augmenting path to push more flow was not perfectly specified. The performance of Ford-Fulkerson is highly dependent on the process of selecting augmenting paths. In this project, we have implemented three variations of Ford-Fulkerson Algorithm for finding maximum flow and tried to evaluate their performance on randomly generated graph instances.

Flow Network

A flow network is a graph G(V, E), where V is the set of vertices and E is the set of edges. Every edge $(u, v) \in E$ has a capacity function c(u, v) associated with it. There are two special vertex s and t being the source and the sink.

A pseudo flow is a function $f\colon V \times V \to \mathbb{R}$ that satisfies the following two constraints:

1. Skew symmetry: f(u, v) = -f(u, v)
2. Capacity Constraints: $f(u, v) \le c(u, v)$

A feasible flow is a pseudo flow that satisfies following additional constraints for all vertices $v \in V \setminus \{s, t\}$: the net flow entering $v \in V \setminus \{s, t\}$, xf (v) = 0. This is called the flow conservation constraint. The value of a feasible flow |f| is the net flow into the sink t.

Maximum flow problem

The maximum flow problem is defined as follows:

> Given a flow network G(u, v), find a feasible flow f with maximum possible flow value |f|.

For simplicity, we will talk about integer maximum flow problem, where edge capacities are integers. Even if the edge capacities are rational numbers, we can apply scaling to make them integral.

Before describing the Ford-Fulkerson Algorithm to solve the maximum flow problem, we need to briefly discuss about residual network and augmenting path.

Residual network

Residual graph of a flow network consists of the same vertex set V. For every edge (u, v) ∈ E, if there is a feasible flow function f(u, v) then there will be a forward edge (u, v) with capacity $c_f(u, v) = c(u, v) – f(u, v)$ in the residual graph. And also there will be backward edge (v, u) in the residual graph with capacity $c_f(v, u) = f(u, v)$. These back edges are needed to redirect the flow by cancel out the previous flow along that edge.

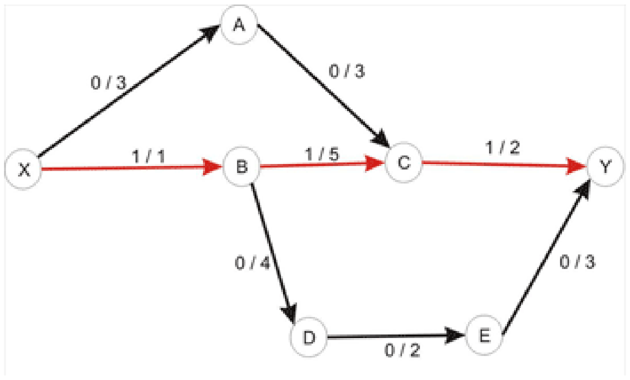

Figure 1: Flow network [2] (Red path shows a feasible flow in the network and X is the source and Y is the sink)

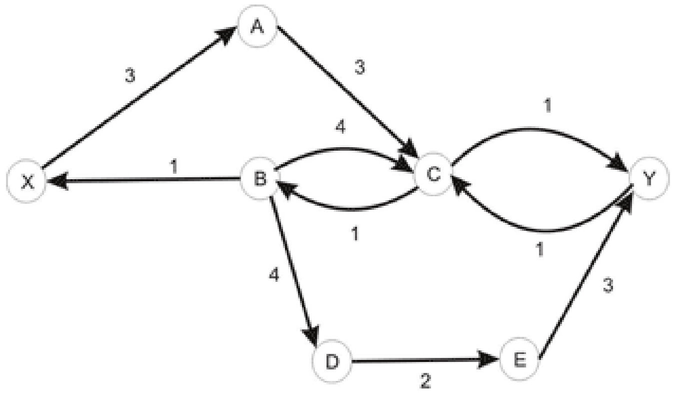

Figure 2: residual network of fig. 1 network [2]

Augmenting path

An augmenting path is a path from source to sink in the residual graph. For the residual graph of figure 2, X→A→C→B→D→E→Y is an augmenting path. The residual capacity of an augmenting path is the minimum capacity of the edges that belongs to the path, also known as bottleneck value for the path. For X→A→C→B→D→E→Y path the residual capacity is 1. This residual capacity means, by using the path described above we can push maximum 1 unit of flow to the sink. Note that if we use this path, we are actually redirecting the previous 1 unit of flow from the path X→B→C→Y to the path X→B→D→E→Y by cancelling out flow in the edge B→C.

Ford-Fulkerson algorithm

General idea of Ford-Fulkerson algorithm is to find an augmenting path in the residual graph and increase the flow along that path until there is no augmenting path in the residual graph.

```
FORD-FULKERSON-METHOD (G(V, E), s, t)
    for each edge (u, v) ∈ E
        f(u, v) = 0
    while there exists a path p from s to t in the residual graph $G_f$
        $c_f(p) = \min\{ c_f(u,v): (u, v) \text{ is in } p\}$
        for each edge (u, v) in p
            if (u, v) ∈ E
                $f(u, v) = f(u,v) + c_f(p)$
            else
                $f(v, u) = f(v, u) - c_f(p)$
```

As we can see, the efficiency of Ford-Fulkerson algorithm highly depends on how we are going to find augmenting paths in the residual graph G_f . If the value of the maximum flow is |f|, there can be at most |f| augmenting paths in the residual graph increasing the flow by 1 unit each time. So in the worst case, the while loop will run for |f| times. And the overall complexity of the algorithm will be O(E|f|).

Notice that, by choosing augmenting paths differently we can change the overall runtime of the algorithm. For example, if we consider augmenting paths with maximum residual capacity, the total number of augmenting paths (iteration of the while loop) should be reduced. The intuition behind this claim is as we are pushing larger amount of flow each time, we need less amount of iterations to reach maximum flow of the graph.

In this project, three variations of Ford-Fulkerson Algorithm have been implemented and their performance have been analyzed on randomly generated graphs.

The implemented variations are:

1. *Arbitrary path variation*: Choose an arbitrary augmenting path in each iteration.
2. *Fattest path variation*: Choose augmenting path with maximum residual capacity.
3. *Shortest path variation*: Choose shortest augmenting path (minimum number of edge path) also known as Edmonds–Karp Algorithm.

Body

Implementation details for the variations of Ford-Fulkerson

While implementing each of the variations of Ford-Fulkerson Algorithm, adjacency list representation of the graph was used. The space complexity of adjacency list is O(V+E) and we can insert an edge in constant time. To traverse all the nodes and edges it will take O(VE) time in adjacency list. All the algorithm were implemented in C++ language. The fattest path variation uses priority queue from STL (Standard Template Library). All the implementations of this project can be found in the project's GitHub repository [6]. Some implementation details for each of the Variations are given below:

Variation 1 (choose arbitrary augmenting path)

In this variation, to choose arbitrary augmenting paths in the residual graph, depth-first-search was used. Depth-first-search arbitrarily chooses a vertex to explore first and hence chooses an augmenting path that is random. Depth-first search was implemented recursively and implementation does not use explicit stack data structure.

Variation 2 (Choose augmenting path with maximum residual capacity (Fattest path))

In this variation, we need to choose augmenting path with maximum residual capacity. Recall that, residual capacity of a path is the minimum capacity edge of the path. So we need to find an augmenting path for which minimum capacity of any edge along the path is maximum. So this is a max-min problem. To find the max-min distance of the paths, a greedy approach similar to Dijkstra's algorithm was used. All the edge capacity is non negative hence there is no negative weighted cycle in the graph. So Dijkstra's algorithm will terminate. STL priority queue was used as Min heap while implementing the greedy approach.

Variation 3 (Choose shortest augmenting path)

In this variation, we need to find augmenting paths with fewer number of edges. The project used breadth first search to find the shortest path in the residual graph. Here shortness of a path refers to number of edges used by the path. For constant edge cost we can use breadth first search to find shortest path in a graph. STL queue was used to implement breadth first search.

Theoretical time complexity of the variations

Let's assume that n is the number of vertices and m is the number of edges in a flow network. C is the maximum possible capacity for a single edge.

Variation 1

This variation, chooses augmenting paths in random order. So the worst case run time of this variation is O(m|f|) where m the number of edges and |f| is the values of maximum flow. In worst case, this algorithm will explore |f| number of augmenting paths with residual path capacity of 1.

Variation 2

In this variation, the algorithm needs to consider total O(mlog(nC)) augmenting paths in worst case[3]. C is the maximum capacity value of a single edge. Normal implementation of Dijkstra's algorithm runs in $O(n^2)$ time. So time complexity of the algorithm will be $O(n^2 m\log(nC))$. The project uses min heap data structure to implement Dijkstra's algorithm. As insertion and deletion in a min heap can be done in logarithmic time, this implementation of the algorithm runs in $O(m^2\log(n)\log(nC))$ time.

Variation 3

This variation considers shortest augmenting path first. In 1972, Edmonds and Karp [4] and, in 1970, Dinic[5] independently proved that if each augmenting path is shortest one, the algorithm will perform O(nm) augmentation steps. When implementing this variation, breadth first search (BFS) was used to find the shortest augmenting path. Run time of BFS is O(n+m). So overall runtime of the algorithm is $O(nm(n + m)) \sim O(m^2 n)$ using breadth first search.

Test set generation for evaluating performance

We have generated random graphs for evaluating the performance of the variations. For each graph, number of vertices $2 \leq n \leq 100$, number of edge $m \leq n^2$ and the maximum capacity of a single edge $C \leq 1000$. There were three different test files with 150, 100, 50 test cases respectively. For each test file there were two different versions, one consists of sparse graphs only and other consists of dense graphs only. We used random generator (i.e. *rand()*) function from stdlib library and changed the seed of the random number generator with respect to system time using *srand()* function.

Performance analysis

The run time and number of iteration tables are given in the appendix B. As expected from the theoretical time complexity, arbitrary path variation considered most number of augmenting paths while finding maximum flow. Number of iterations performed by fattest path variation is the least amount among all the three variations of the Ford-Fulkerson algorithm. Even for the dense graph datasets, the comparison remains same. Figure 5 and 6 shows the number of iterations performed by the three variations for sparse graph and dense graph dataset respectively. Figure 3 and Figure 4 shows the runtimes of the variations for sparse graph and dense graph datasets. For all the data sets, arbitrary path variation has the highest runtime which correlates to the number of iterations performed by the algorithm. Even though fattest variation considered much less augmenting paths than the shortest path variation, it takes more time to execute compared to the shortest path variation. This increased runtime for the fattest path variation is caused by the overhead related to Dijkstra's algorithm which uses a min heap data structure. If we analyze the run time complexity of the two algorithms we can see that for $n \leq 100$, $m \leq n^2$ and $C \leq 1000$, $O(m^2\log(n)\log(nC)) > O(m^2 n)$. Thus the implementation of the fattest path variation takes more time than the shortest path variation.

Conclusion

This project has successfully implemented three major variations of the Ford-Fulkerson algorithm. To improve the run time of the fattest path variation we can consider more complex data structures. For example, we can use Fibonacci heap data structure instead of STL priority queue. Fibonacci heap's *find-minimum()* operation takes constant O(1) amortized time. Which will reduce run time for fattest path variation considerably. Other than Fibonacci heap, there are some other data structure like splay tree as well as binomial heap that can be used. Moreover, instead of using a variation of Dijkstra's algorithm we can use Jarnik's (Prim's) minimum spanning tree algorithm to implement fattest path variation. The overall time complexity of Jarnik's minimum spanning tree algorithm is O(E + VlogV) with Fibonacci heap.

This project tested the variations on randomly generated graphs. Some obvious worst cases can be designed manually for the variations. The performance evaluation found by this project on random graphs matches with the theoretical complexities of the variations. Beside Ford-fulkerson algorithm, there are lots of different algorithms to solve maximum flow problem efficiently. Some even have better run time than Ford-Fulkerson (See Appendix B).

Reference

[1] Ford, L. R.; Fulkerson, D. R. (1956). "Maximal flow through a network". Canadian Journal of Mathematics. 8: 399–404. Doi:10.4153/CJM-1956-045-5

[2] Topcoder.com

[3] https://www.topcoder.com/community/data-science/data-science-tutorials/maximum-flow-augmenting-path-algorithms-comparison/

[4] Edmonds, Jack; Karp, Richard M. (1972). "Theoretical improvements in algorithmic efficiency for network flow problems". Journal of the ACM. Association for Computing Machinery. 19 (2): 248–264. doi:10.1145/321694.321699

[5] Dinic, E. A. (1970). "Algorithm for solution of a problem of maximum flow in a network with power estimation". Soviet Math. Doklady. Doklady. 11: 1277–1280

[6] https://github.com/Sanzee/Ford-Fulkerson-variation

Appendix A

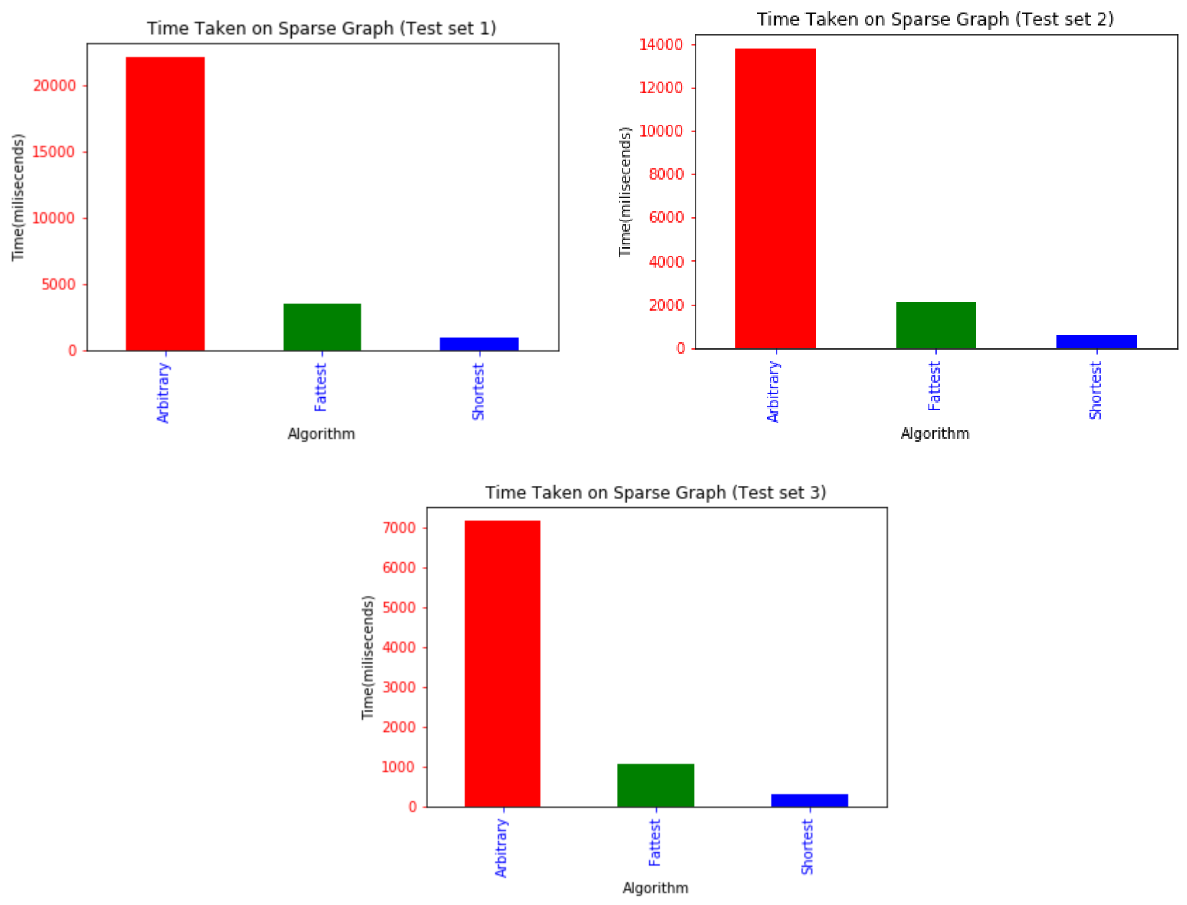

Figure 3: Running Time on sparse graph dataset

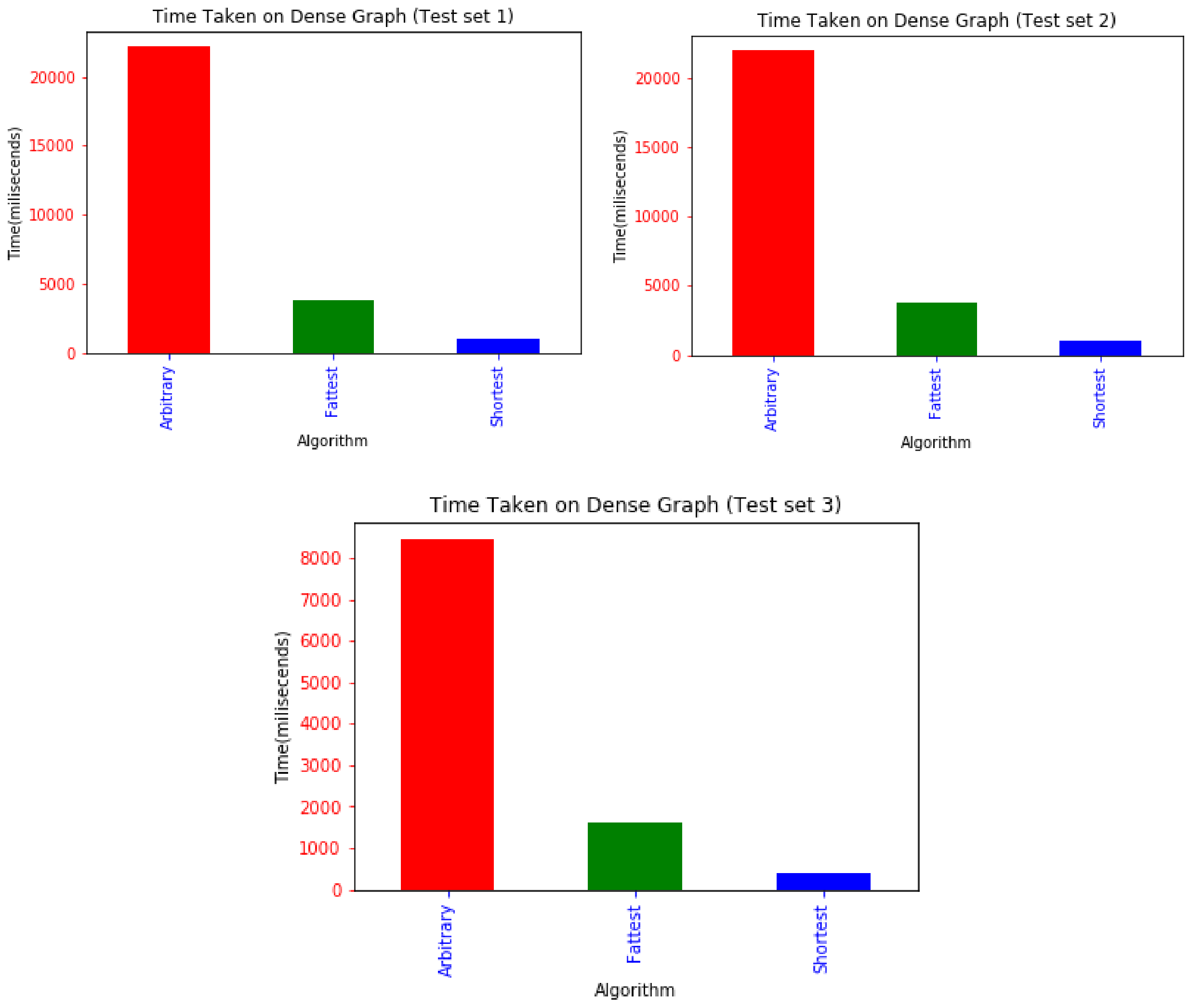

Figure 4: Running Time on Dense graph dataset

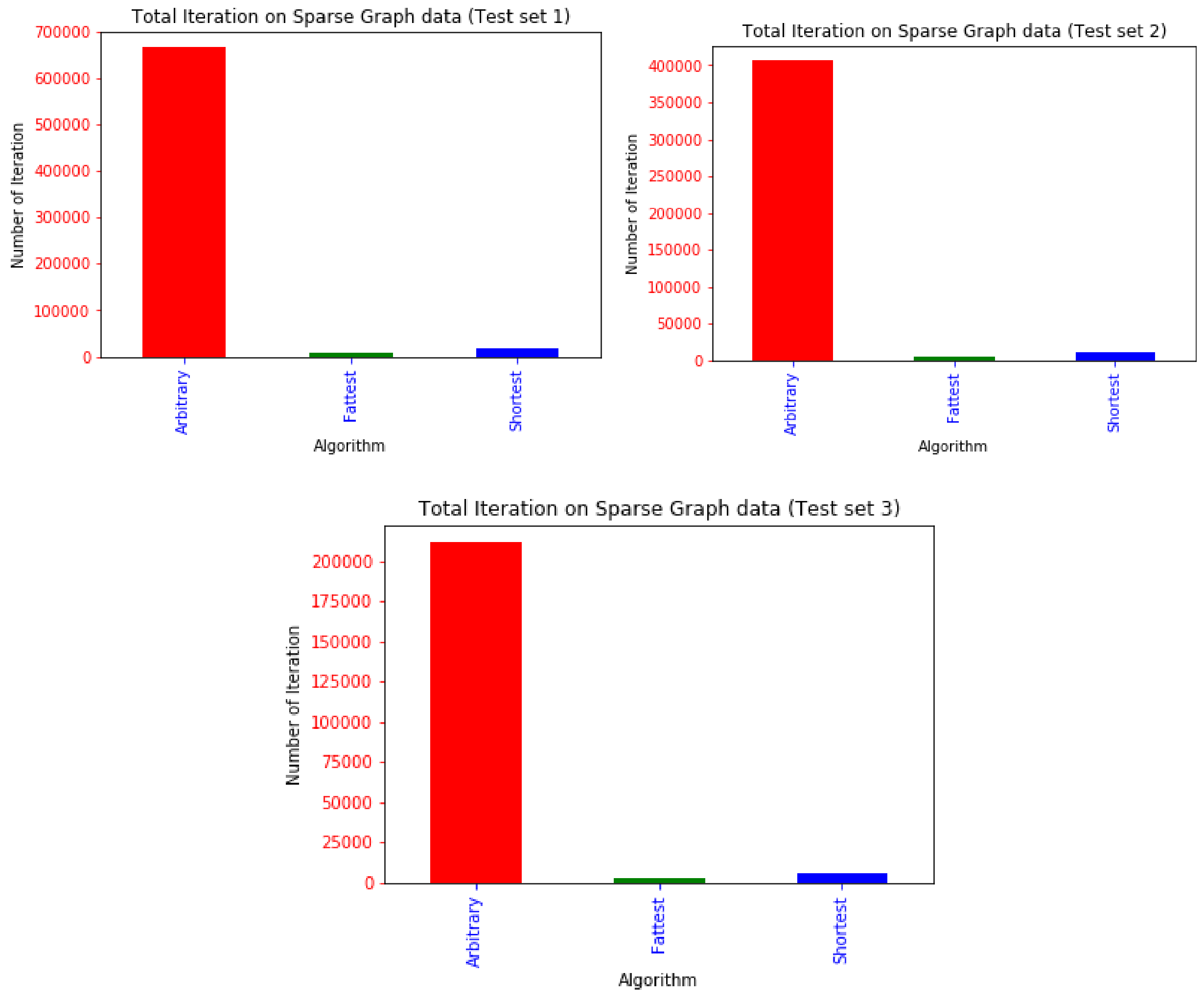

Figure 5: Number of iterations on sparse graph dataset

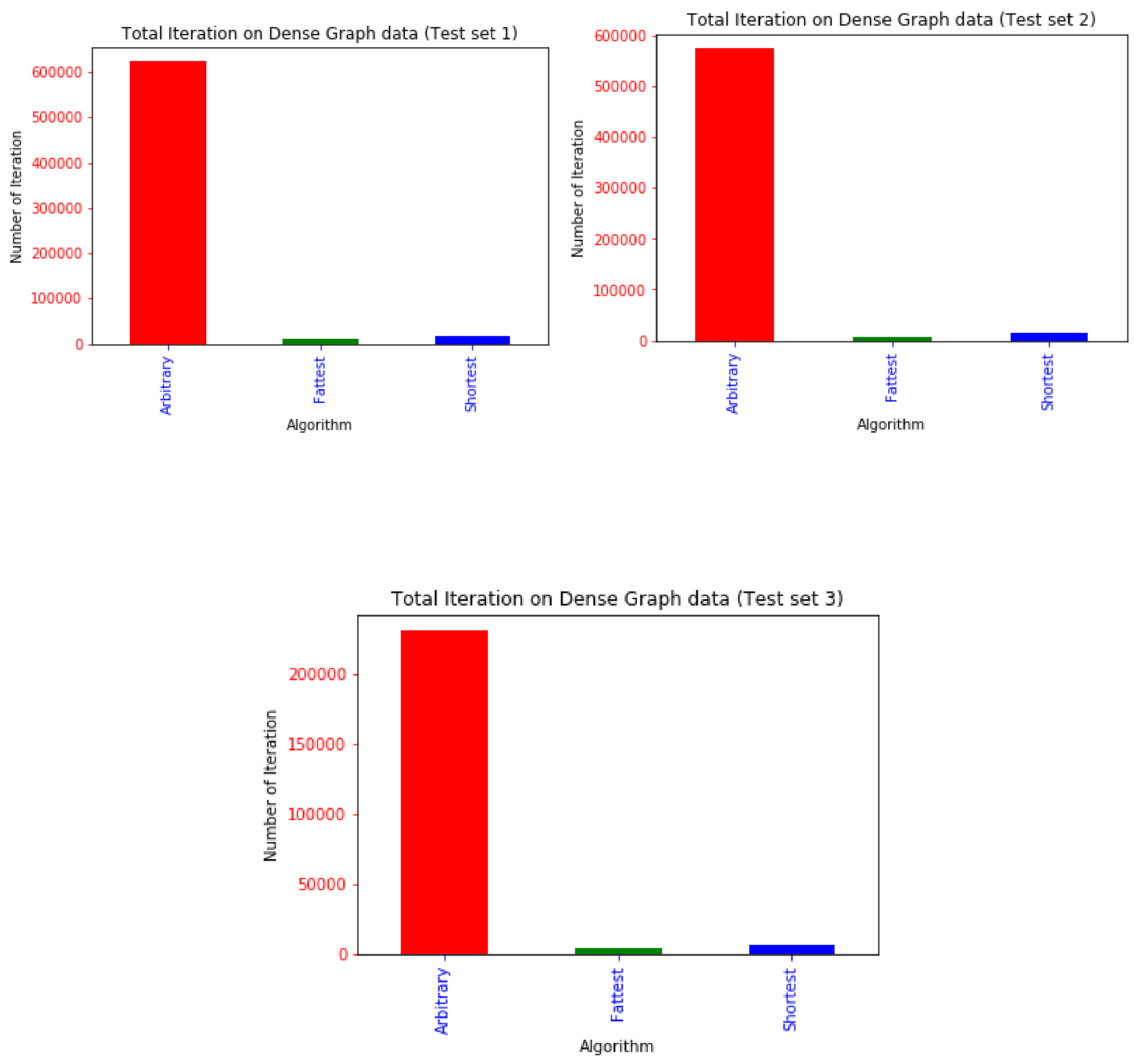

Figure 6: Number of iterations on Dense Graph Dataset

Appendix B

Algorithms for maximum flow

Algorithm	Direct implementation	Implementation with dynamic tree	Source
Blocking flow	$O(V^3)$	O(VElog(V))	[Dinits; Sleator and Tarjan]
Push-relabel (generic)	$O(V^2E)$	O(VElog(V))	[Goldberg and Tarjan]
Push-relabel (highest label)	$O(V^2\sqrt{E})$		[Cheriyan and Maheshwari; Tunçel]
Pseudo flow	$O(V^2E)$	O(VElog(V))	[Hochbaum]
Compact abundance graphs		O(VE)	[Orlin 2012]

Runtime table (Sparse Graph)

Variation	Test dataset 1	Test dataset 2	Test dataset 2
Arbitrary	22081ms	13784ms	7142ms
Fattest path	3446ms	2100ms	1063ms
Shortest path	953ms	594ms	297ms

Runtime table (Dense Graph)

Variation	Test dataset 1	Test dataset 2	Test dataset 2
Arbitrary	22145ms	2194ms	8439ms
Fattest path	3829ms	3766ms	1594ms
Shortest path	1047ms	985ms	391ms

Number of iteration table (Sparse Graph)

Variation	Test dataset 1	Test dataset 2	Test dataset 2
Arbitrary	665780	405925	211279
Fattest path	8328	5156	2585
Shortest path	16834	10450	5247

Number of iteration table (Dense Graph)

Variation	Test dataset 1	Test dataset 2	Test dataset 2
Arbitrary	6622889	574310	231199
Fattest path	9609	7935	3517
Shortest path	18025	15339	6694