Implementation and Comparison of Algorithms to Solve the Max Flow Problem

Naveen Suresh
Department of CSE,
PES University
Bangalore, India
naveen.213@gmail.com

Shruthi Gopinath
Department of CSE,
PES University
Bangalore, India
shruthigopinath2@gmail.com

Yash R Patil
Department of CSE,
PES University
Bangalore, India
yash.r.patil98@gmail.com@

***Abstract*—This purpose of this study is to implement various algorithms that solve the maximum flow network problem. Further, the implementations of these algorithms are compared, the results of which are documented and justified.**

I. Introduction

A flow network is a model of a system where some material is produced at its source, travels through the system, and is consumed at its end. Flow networks can model any of the following:

- Liquids flowing through pipes
- Parts through assembly lines
- Current through electricity networks
- Information through communication networks etc.

Let N = (V, E) be a flow network with s, t ∈ V being the source and the sink of N respectively.

- The ***capacity*** of and edge is a mapping $c : E \to R^+$, denoted by c_{uv} or $c(u, v)$. It represents the maximum amount of flow that can pass through an edge.
- A ***flow*** is a mapping of $f : E \to R^+$, denoted by f_{uv} or $f(u, v)$, subject to the following constraints:
  1) $f_{uv} \leq c_{uv}$, for each $(u, v) \in E$ (capacity constraint: the flow of an edge cannot exceed its capacity);
  2) $$\sum_{u:(u,v)\in E} f_{uv} = \sum_{u:(v,u)\in E} f_{vu},$$ for each $v \in \{s, t\}$. (Conservation of flows: The sum of flows entering a node must equal the sum of the flows exiting a node, except for the source and the sink nodes).
  3) The ***valueofflow*** is defined by $$|f| = \sum_{v:(s,v)\in E} f_{sv},$$ where s is the source of N. It represents the amount of flow passing from the source to the sink.

The maximum flow problem is to maximize $|f|$, that is, to route as much flow as possible from s to t.

Solutions to the max flow problem lie majorly in three different paradigms of approaches:

1) Finding an augmenting path and updating flow
2) Initializing preflow and computing actual flow

In this report, we discuss two algorithms from each paradigm, namely Edmonds-Karp and Dinic from paradigm 1 and Push-Relabel and Relabel-to-Front from paradigm 2.

II. Algorithms

For our project, we have implemented and compared the following algorithms:

A. Ford-Fulkerson

The Ford-Fulkerson algorithm seeks a increasing path in the residual graph. It saturates this path if it exists, otherwise it returns the maximum flow. Specifically, the algorithm establishes a minimal cut to check the optimality criterion. It is recalled that the flow is less than or equal to the cut.

Algorithm 1 Ford-Fulkerson Algorithm

Input: Given a Network G=(V,E) with flow capacity c, a source node s, and a sink node t
Output: Compute a flow f from s to t of maximum value

```
f(u, v) ← 0 for all edges
while there is a path p from s to t in G_f, such that c_f(u, v) > 0 for all edges (u, v) ∈ p: do
    Find c_f(p) = min{c_f(u, v) : (u, v) ∈ p}
    for each edge (u, v) ∈ p do
        f(u, v) ← + c_f(p) (send flow along path)
        f(v, u) ← − c_f(p) (the flow might be returned later)
    end for
end while
```

By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. However, there is no certainty that this situation will ever be reached, so the best that can be guaranteed is that the answer will be correct if the algorithm terminates. In the case that the algorithm runs forever, the flow might not even converge towards the maximum flow. However, this situation only occurs with irrational flow values. When the capacities are integers, the runtime of FordFulkerson is bounded by $O(Ef)$, where E is the number of edges and f is the value of the maximum flow in the graph. A variation of

the FordFulkerson algorithm with guaranteed termination and a runtime independent of the maximum flow value is the EdmondsKarp algorithm. When the augmenting path is found by applying breadth-first search (BFS), the implementation of Ford-Fulkerson is known to be Edmonds-Karp algorithm. Edmonds-Karp algorithm runs in $O(EV^3)$ when implemented using adjacency matrix.

B. Dinic's Blocking Flow Algorithm

Dinic's algorithm of solving the maximum flow problem aim at leveraging the concepts of blocking flow and level graphs.

1) Definition: Let $G = ((V, E), c, s, t)$ be a network with $c(u, v)$ and $f(u, v)$ as the capacity and the flow of the edge (u, v) respectively.

The residual capacity is a mapping $c_f : VxV \rightarrow R^+$ defined as,

- if $(u, v) \in E$, $c_f(u, v) = c(u, v) - f(u, v)$ $c_f(v, u) = f(u, v)$
- $c_f(u, v) = 0$ otherwise.

The residual graph is graph $G_f = ((V, E_f), c_f|_{E_f}, s, t)$, where $E_f = \{(u, v) \in VxV : c_(u, v) > 0$

An augmenting path is an s-t path in the residual graph G_f is the graph $G_L = ((V, E_L), c_L|_{E_L}, s, t)$, where $E_L = \{(u, v) \in E_f : dist(v) = dist(u) + 1\}$.

A blocking flow is an s-t flow f such that the graph $G' = (V, E'_L, s, t)$ with $E'_L = \{(u, v) : f(u, v) < c_f|_{E_L}(u, v)\}$ contains so s-t path.

Algorithm 2 Dinic' Algorithm

Input: Given a Network G=((V,E), c, s,t)
Output: Compute a flow f from s to t of maximum value

1: Set $f(e) = 0$ for each e $\in$ E
2: Construct G_L from G_f of G. If dist(t) = ∞, stop and output f
3: Find a blocking flow f' in G_L
4: Augment flow f by f' and go back to step 2.

It can be shown that the number of layers in each blocking flow increases by at least 1 each time and thus there are at most $|V|$-1 blocking flows in the algorithm. For each of them:

- the level graph G_L can be constructed by breadth-first search in $O(E)$ time.
- a blocking flow in the level graph G_L can be found in $O(VE)$ time

with total running time O(E+VE)=O(VE) for each layer. As a consequence, the running time of Dinic's algorithm is $O(V^2E)$

C. Push Relabel

In this method, instead of maintaining a feasible flow, and improving it until it becomes optimal, we maintain a preflow, which is an assignment of flows to edges that allows vertices to have more incoming flow than outgoing flow (but not vice versa) and that satisfies the capacity constraints.

An assignment of a non-negative flow $f(u, v)$ to each edge (u, v) of a network (G = (V, E), c, s, t) is a preflow if:

- for each edge (u, v), $f(u, v) \leq c(u, v)$
- for each vertex v $\in$ V - {t},

$$\sum_u f(u, v) - \sum_w f(v, w) \geq 0$$

we define the excess flow at u as,

$$e_f(v) := \sum_u f(u, v) - \sum_w f(v, w)$$

At each step, we pick a vertex v with positive excess flow, and select an outgoing edge (v, w) with positive residual capacity, and push flow out of v and into w, provided that $h(v) > h(w)$. If there is no vertex for which we can do such a push operation, we take any vertex with positive excess flow and increase it's height (relabel operation).

Algorithm 3 Push-Relabel Algorithm

Input: Given a Network G=((V,E), c, s,t)
Output: Compute a flow f from s to t of maximum value

1: h[s]:=—V—
2: for each $v \in V - \{s\}$ do h[v]:=0
3: for each (s, v) $\in$ E do $f(s, v) := c(s, v)$
4: **while** f is not a feasible flow **do**
5: Let c'(u, v) = c(u, v) + f(u, v) - f(v, u) be the capacities of the residual network.
6: **if** there is a vertex v $\in$ V - $\{s, t\}$ and a vertex w $\in$ V such that $e_f(v) > 0$, $h(v) > h(w)$, and $c'(v, w) > 0$ **then**
7: push min$\{c'(v, w), e_f(v)\}$ units of flow on the edge (v, w)
8: **else**
9: let v be a vertex such that $e_f(v) > 0$, and set h[v]:= h[v] + 1
10: **end if**
11: **end while**
12: **return** f

The algorithm executes $O(V^2)$ relabels, $O(VE)$ saturating pushes and $O(V^2E)$ nonsaturating pushes. Data structures can be designed to pick and execute an applicable operation in $O(1)$ time. Therefore, the time complexity of the algorithm is $O(V^2E)$

D. Relabel to Front Algorithm

In the relabel to front algorithm, we maintain a linked list L consisting of all vertices in V-{s,t}. A key property is that the vertices in L are topologically sorted according to the admissible network. Further, for any node with excess flow, it is discharged entirely by relabeling that node to be able to rid itself of all it's excess flow.

This algorithm has a $O(V^3)$ time complexity.

Algorithm 4 Relabel-to-Front Algorithm

Input: Given a Network G=((V,E), c, s,t)
Output: Compute a flow f from s to t of maximum value

```
INITIALIZE-PREFLOW(G,s)
L = G.V - {s, t}, in any order
for each vertex u ∈ G.V − {s, t} do
    u.current = u.N.head
end for
u = L.head
while u ≠ NIL do
    old-height = u.h
    DISCHARGE(u)
    if u.h > old-height then
        move u to the front of list L
    end if
    u = u.next
end while
return f
```

III. Results

A. Dataset

The input to all the above-mentioned algorithms must be a single-source, single-sink directed graph. Hence, the generation of a new dataset must ensure the following conditions:

1) There must be no incoming edges to the source vertex
2) There must be no outgoing edges from the sink vertex
3) There must be no self loops

By ensuring that these criteria were met, 14 different datasets were created. They were of the following dimensions:

TABLE I
Datasets Created

V	*E*	*No. of Instances*
100	2500	3
100	5000	3
500	5000	3
500	20000	2
1000	20000	1
2000	50000	1
2000	100000	1
2002	23936	1
2002	69800	1

B. Observations

As the Edmonds-Karp algorithm runs in $O(VE^2$ time, while Dinic's Algorithm runs in $O(V^2E)$ time, it is evident that asymptotically, Dinic's algorithm is expected to perform better. From the comparison between the results of the two algorithms, it is observed that increase in time required for the execution of Dinic's algorithm is $\sqrt{increase in time for Edmonds - Karp's algorithm}$.

Further, it can be noticed that the discharge operation on the topologically sorted active node list (in relabel-to-front) is much more efficient than the iterative push operation (in generic push relabel). The execution times as well as the change in times with the changes in graph dimensions is significantly lesser for relabel-to-front as compared to push-relabel.

TABLE II
Average Running Time of Implemented Algorithms (seconds)

V	*E*	*E-K*	*Dinic*	*P-R*	*R-T-F*
100	2500	0.002993	0.00711	0.3848	0.2151
100	5000	0.003874	0.01406	0.67398	0.54136
500	5000	0.0212	0.01047	0.07526	1.4801
500	20000	0.07735	0.02385	0.35137	0.077549
1000	20000	0.14059	0.02288	0.6340	0.0626
2000	50000	0.6114	0.085750	2.992	0.2562
2000	100000	0.8991	0.2376	5.0342	0.5943
2002	23936	192.709	0.035959	9.0269	0.145523
2002	69800	521.019	0.072344	52.814	0.189265

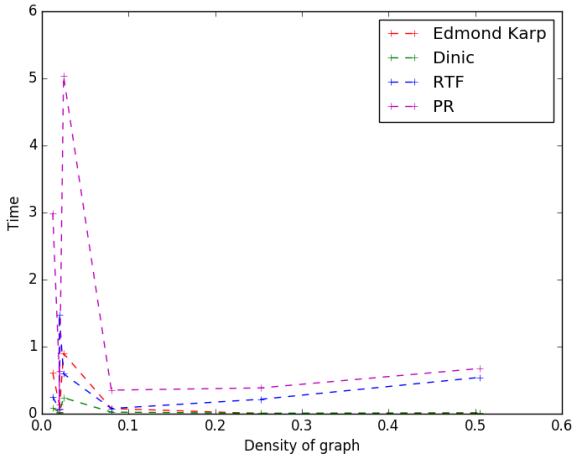

Fig. 1. Density of Graph vs. Execution Time

The execution times for push relabel and relabel to front are higher than Edmonds-Karp and Dinic's at smaller graph sizes due to the overhead of initializing neighbor lists and preflows, which do not exist in the augmenting-path algorithms.

The execution times may not match with the expected time complexities due to various factor. This is because the graphs were randomly generated, and some factors are variable and hard to account for. These parameters might be

- density of the graphs
- number of augmenting paths that exist from source to destination
- Number of disjoint points from the connected graph
- Data structures used

Hence, although the results follow the general trend as expected, the exact change in execution times with respect to the time complexities may not be observed.

From fig.1, it is evident that as the density of the graph increases, there is a steady increase in execution time. The spike in the graph can be explained due to the high execution time for the less dense, but enormous 100000-edge graph.

IV. Conclusion

By implementing these four algorithms, we have realized that in general, preflow initializing algorithms perform better than augmenting path algorithms, aside from small cases. Further, various other methods of choosing from the active node list exist for Push-Relabel, aside from Relabel-to-Front. These would further enhance the performance of the preflow algorithms. However, Dinic's algorithm proves to be useful without the memory overhead, and is a viable option in most cases.

Further algorithms which have better time complexities such as Binary blocking flow algorithm ($O(Emin(V^{2/3}, sqrt(E)) \log V^2/E \log U)$), the James B Orlin's + KRT algorithm ($O(VE)$). However, their implementations are still under research, and for most practical applications, the explored algorithm perform reasonably well.