MAXIMUM FLOW PROBLEM

Hardik Aggarwal[1], Dhananjay Goel[2]

[1]2021CSB1173
[2]2021CSB1165

ABSTRACT

The problem of finding the maximum flow in a graph has numerous applications in domains such as network optimization, transportation planning, and resource allocation. We have discussed in detail the different algorithms and optimizations used to solve the maximum flow problem efficiently. We discuss classical algorithms like Ford-Fulkerson and Edmonds-Karp and advanced approaches such as Dinic's algorithm and the Capacity Scaling heuristic. The strengths, weaknesses, and computational complexities of these algorithms are thoroughly analyzed to understand their trade-offs and choose the most suitable approach for their specific requirements.

KEYWORDS

Network- It is a directed graph G made up of vertices V and edges E along with a function c that gives each edge e ∈ E a non-negative integer value that corresponds to the capacity of e. Further naming two vertices, one as source and one as sink, the network is known as a flow network.

Flow network - In a flow network, we have a directed graph where each edge has a capacity indicating the maximum amount of flow that can pass through it. The flow represents the quantity of a resource (such as water, data, or traffic) flowing through the network.

Flow – A flow in a flow network is function *f*, which assigns each edge *e* a non-negative integer value, namely the flow. A flow in an edge can be understood as analogous to water flowing through a pipe per second. The function must fulfill the following two conditions:

1. The flow of an edge cannot exceed the capacity.

$$f(e) \leq c(e)$$

2. And the sum of the incoming flow of a vertex u must be equal to the sum of the outgoing flow of u except in the source and sink vertices

$$\sum_{(v,u)\in E} f\left((v,u)\right) = \sum_{(u,v)\in E} f\left((u,v)\right)$$

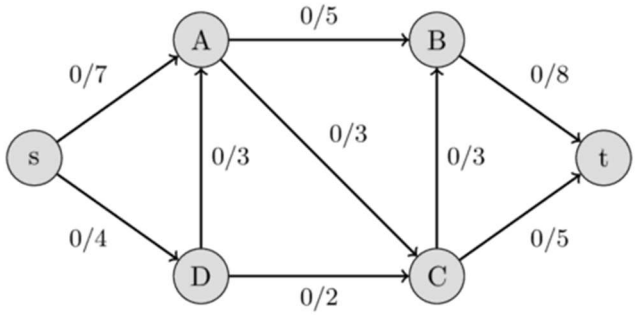
The source vertex s only has an outgoing flow, and the sink vertex t has only an incoming flow.

$$\sum_{(s,u)\in E} f\left((s,u)\right) = \sum_{(u,t)\in E} f\left((u,t)\right).$$

The Maximum Flow Problem –

The following illustration is an example of a flow network: When we think of edges as water pipes, their capacity, and flow are the maximum and current amounts of water that can pass through the pipe, respectively, per second. Overflowing a pipe's capacity will result in the water not flowing through it. When water exits certain pipes, the vertices serve as junctions and subsequently disperse the water in some manner to other pipes. This serves as the second flow condition's purpose as well. Each junction's other pipes must receive all the incoming water, which must be distributed to them. The source *s* is origin of all the water, and the water can only drain in the sink *t*.

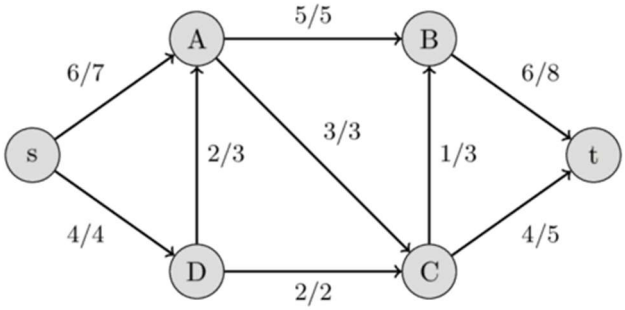
The following image shows a flow network. The first value of each edge represents the flow, which is initially 0, and the second value represents the capacity.



OBJECTIVE

The value of the flow of a network is the sum of all the flows that get produced in the source *s* or equivalently to the sum of all the flows that are consumed by the sink *t*. A **maximal flow** is a flow with the maximal possible value. The objective is to determine the maximum flow that can be achieved while respecting the capacity constraints of the edges.

The following image shows the maximal flow in the flow network.



MAXIMAL FLOW ALGORITHMS:

A. Ford-Fulkerson Algorithm -

Residual Capacity: A residual capacity of a directed edge is the capacity minus the flow. If there is a flow along some directed edge (*u, v*), then the reversed edge has capacity 0 and flow as $f((v,u)) = -f((u,v))$. This also defines the residual capacity for all the reversed edges. We can create a residual network from all these edges, which is just a network with the same vertices and edges, but we use the residual capacities as capacities.

Augmenting path: An augmenting path is a simple path in the residual graph, i.e., along the edges whose residual capacity is positive.

Algorithm

First, we set the flow of each edge to zero. Then we look for an augmenting path from s to t. If such a path is found, then we can increase the flow along these edges by updating $f((u,v))$ += C and $f((v,u))$ - = C for every edge (*u, v*) in the path where C is the smallest residual capacity in the path or the bottleneck value. We keep on searching for augmenting paths and increasing the flow. Once such an augmenting path no longer exists, the flow obtained is maximal.

Time Complexity Analysis

We can find the augmenting path using DFS or BFS, both of which works in *O(V+E)* that is roughly O(E) in graphs. If all the capacities of the network are integers, then for each augmenting path the flow of the network increases by at least one in each iteration. Therefore, the complexity of Ford-Fulkerson is O(E*F) , where F is the maximal flow of the network.

Limitations

1. **No guarantee of termination:** One of the main drawbacks of the Ford-Fulkerson algorithm is that it does not always terminate. Some graphs will cause the algorithm to get stuck into an infinite loop which might cause serious computing problems.

2. **Inefficient in dense graphs**: The Ford-Fulkerson algorithm is not very efficient in dense graphs, where the number of edges is very large in comparison to sparser graphs. In these cases, the algorithm can be very slow because of the time complexity of O(E*F).

B. Edmonds-Karp Algorithm -

Edmonds-Karp algorithm is just a modified implementation of the Ford-Fulkerson method that uses BFS for finding augmenting paths instead of DFS.

Algorithm

The intuition is, that every time we find an augmenting path, we find the shortest augmenting path using BFS from source to sink. We repeatedly find shortest augmented paths in the residual graphs and obtain the maximal flow.

Time Complexity

One of the edges becomes saturated while augmenting, and the distance from the edge to the source will be longer if it appears later again in an augmenting path. Let the distance to reach a vertex v from source S is d. If we introduce back edges along the path, they do not decrease the distance as they go back up a level. Any path that would use one of the back edges must use two more than the augmented path with distance d. Therefore, the shortest found augmented path increases monotonically and bounds the length of one iteration to O(|E|).

When we use an augmented path as described in the theorem, we delete the edge from the level graph which got saturated. These edges won't come back in the level graphs. As each augmentation deletes one edge from the level graph, after |E| iterations the level graph would be empty.

So, each time we build a longer path than the previous one to augment further. As there are only |V| possible shortest path lengths, for each possible path, we have up to |E| iterations. So the total number of iterations are at most |E||V| and each iteration performs BFS in O(|E|).

Thus, the complexity comes out to be $O(|V|\cdot|E^2|)$.

Advantage over Ford- Fulkerson –

1. Using a DFS can lead to zig zagging through the flow graph to find the sink which can cause a longer augmented path. Longer augmented paths are generally undesirable because the longer the path, the higher the chance for a small bottleneck value which results in a longer runtime.
2. Using the shortest path as an augmenting path is a great approach to avoid the DFS worse case
3. It also gives the time complexity in polynomial time instead of being dependent on the constant F (maximum flow) in case of the use of DFS.

C. Dinic's Algorithm -

Blocking flow: A blocking flow of some network is such a flow that every path from source to sink contains at least one edge which is saturated by this flow. Note that a blocking flow is not necessarily maximal.

Layered Network: A layered network of a network G is a network built in the following way- First, for each vertex *v* we calculate *level[v]* - the shortest path (unweighted) from s to this vertex using only edges with positive capacity. Then we keep only those edges (v, u) for which *level[u]* = *level[v]* + *1*. This formed network is acyclic.

Residual Network: A residual network G^R of network G is a network that contains two edges for each edge $(v, u) \in G$:

- (v, u) with capacity $c_{vu}^R = c_{vu} - f_{vu}$
- (u, v) with capacity $c_{uv}^R = 0 - (-f_{vu}) = f_{vu}$

The algorithm is based on the concept of level graphs, subgraphs of the original graph that retain the same structure but include only edges whose residual capacities are positive. These level graphs help in finding blocking flows efficiently. On each phase we construct the layered network of the residual network of G. Then we find an arbitrary blocking flow in the layered network and add it to the current flow.

Algorithm

We first construct the level graph using BFS. From the source node, nodes are assigned levels based on their distance from the source in terms of the number of edges. Only consider edges with positive residual capacity in the BFS traversal. This step ensures that the level graph represents a valid flow.

While there exists a path from the source to the sink in the level graph, perform the following steps:

1. Use DFS to find a blocking flow in the level graph. The DFS starts from the source node and explores the graph along edges with positive residual capacity, updating the flow values and residual capacities as it progresses. When a dead end is reached or the sink is encountered, backtrack to find an alternative path. This process effectively finds a path from the source to the sink in the level graph with the maximum possible flow.

2. Update the level graph by removing edges with zero residual capacity. This step ensures that the level graph maintains only relevant edges with positive residual capacity for subsequent iterations.

Calculate the maximum flow as the sum of flows leaving the source node.

Proof of correctness

Let's show that if the algorithm terminates, it finds the maximum flow.

If the algorithm terminated, it couldn't find a blocking flow in the layered network. It means the layered network has no path from source to sink. It means the residual network has no path from source to sink. It means that the flow is maximum.

Number of phases

The algorithm terminates in less than *V* phases. To prove this, we need to show that the level of nodes strictly increases in each phase until it reaches a maximum level of V, i.e., $\text{level}\,[u]_{i+1} > \text{level}\,[u]_i$ for u ∈ *V*. Since the node level cannot exceed V, the algorithm will terminate after at most V phases.

Let's assume that the level of nodes does not increase after a phase. Then the same path will remain as the augmenting path which was taken in the previous phase which is not possible since one of the edges of that path must have been saturated and hence the augmenting path must not remain the same. Hence it contradicts our assumption which concludes that after each phase the level of nodes strictly increases reaching to the maximum level V in at most V phases.

Finding blocking flow

In order to find the blocking flow on each iteration, we may simply try pushing flow with DFS from *s* to *t* in the layered network while it can be pushed. In order to do it more quickly, we must remove the edges which cannot be used to push anymore. To do this we can keep a pointer in each vertex which points to the next edge which can be used.

Time Complexity

A single DFS run takes $O(k + V)$ time, where *k* is the number of pointer advances on this run. Summed up over all runs, number of pointer advances cannot exceed *E.* On the other hand, total number of runs won't exceed *E,* as every run saturates at least one edge. In this way, total running time of finding a blocking flow is *O(VE)*.

Since there are less than V phases, time complexity of Dinic's algorithm is $O(V^2E)$

D. Capacity Scaling Heuristic

In capacity scaling, the idea is to start with an initial zero flow and gradually increase the flow until reaching the maximum flow. The algorithm iteratively augments the flow along the network paths with available capacity, effectively pushing more flow through the network.

The key insight of capacity scaling is that only edges with capacities above a certain threshold contribute to the maximum flow. Initially, this threshold can be set to the largest capacity in the network. As the algorithm progresses, the threshold is halved at each iteration until it becomes smaller than any capacity in the network. This process helps focus the algorithm's attention on the critical edges that can significantly increase the flow.

At each iteration, the algorithm performs a modified version of the Ford-Fulkerson method, such as the Edmonds-Karp algorithm or the Dinic's algorithm, to find an augmenting path with available capacity. The augmenting path is a path from the source to the sink that allows increasing the flow without violating the capacity constraints of any edge.

By repeatedly finding and augmenting paths, the algorithm increases the flow until reaching a point where no augmenting paths exist. At this stage, the algorithm terminates, and the flow obtained is the maximum flow for the given network.

The capacity scaling technique improves the efficiency of solving maximum flow problems by reducing the number of iterations required. By focusing on edges with significant capacity, it avoids considering edges that have no impact on the maximum flow. This approach can lead to substantial speedups, especially in networks with large capacities where most edges have relatively small capacities.

Time Complexity

U is defined as the largest capacity of an edge in the initial network. Total iterations done will be log(U) as after each iteration, we are halving the value of U. Finding augmenting paths and increasing the flow for each path upto the blocking flow requires time of *O(VE)* as discussed in the above algorithms.

Hence the total time complexity will be O (|V||E| * log(U)).

CONCLUSIONS

Algorithm	Time Complexity
Ford Fulkerson (with DFS)	$O(EF)$, where F is the maximum flow
Edmonds-Karp	$O(VE^2)$
Dinic's algorithm	$O(V^2E)$
Capacity Scaling	$O(VE\ logU)$, where U is the maximum capacity of the network

The choice between which algorithm should be used depends on the characteristics of the flow network and the requirements of the problem at hand.

Dinic's Algorithm can be extended to handle more complex flow network variants such as multiple sources and sinks, whereas Edmonds-Karp is specifically designed for single-source, single-sink networks.

While applying the capacity scaling heuristic it is essential to consider the network properties and evaluate the trade-off between runtime and accuracy. As in situations where exact maximum flow is not necessary and approximation is sufficient, capacity scaling provides faster solution.

APPLICATIONS FOR MAXIMAL FLOW PROBLEM

1. **Transportation Networks** - optimizing the flow of traffic, passengers, or goods through the network, minimizing congestion and maximizing efficiency. By modeling the road network as a graph and assigning capacities to the edges representing the roads, the maximum flow algorithm can determine the optimal flow of vehicles through the network

2. **Disaster Management -** used to determine evacuation routes, resource allocation, and logistics planning during emergencies. Maximum flow algorithms can aid in optimizing the allocation of evacuees to shelters based on capacity, location, and distance considerations.

3. **Image Segmentation** - treating image as a graph, the algorithm finds the optimal boundary between different regions based on flow of pixels. The source node is connected to the pixels of the foreground region, and the sink node is connected to the pixels of the background region. The segmented regions are extracted based on the cut obtained from the maximum flow algorithm

4. **Water Management**- determining the optimal distribution of water resources in irrigation networks, hydroelectric power generation, or water supply networks. By modeling the network as a graph, where nodes represent water sources, treatment plants, storage tanks, and demand points, and edges represent pipes or channels, the algorithm determines the optimal flow of water through the network.

ACKNOWLEDGEMENT

We would like to express our sincere gratitude and appreciation to Dr. Sudarshan Iyengar and Teaching Assistant Poonam Adhikari, as well as all the other Teaching Assistants, for their invaluable support and guidance throughout the completion of this report.

REFERENCES

1. CP Algorithms - https://cp-algorithms.com/graph/edmonds_karp.html
2. Wikipedia - https://en.wikipedia.org/wiki/Maximum_flow_problem
3. https://cp-algorithms.com/graph/edmonds_karp.html#max-flow-min-cut-theorem
4. https://cseweb.ucsd.edu/classes/sp11/cse202-a/lecture8-final.pdf
5. https://www.topcoder.com/thrive/articles/edmonds-karp-and-dinics-algorithms-for-maximum-flow#:~:text=The%20Edmonds%2DKarp%20algorithm%20is,path%20from%20source%20to%20sink.
6. https://www.ripublication.com/irph/ijert20/ijertv13n7_06.pdf