Maximum (Max) Flow

1. Introduction

Maximum (Max) Flow is one of the problems in the family of problems involving flow in networks.

In Max Flow problem, we aim to find the maximum flow from a particular source vertex **s** to a particular sink vertex **t** in a directed weighted graph **G**.

There are several algorithms for finding the maximum flow including Ford-Fulkerson method, Edmonds-Karp algorithm, and Dinic's algorithm (there are a few others, but they are not included in this visualization yet).

The dual problem of Max Flow is Min Cut, i.e., by finding the max **s-t** flow of **G**, we also simultaneously find the min **s-t** cut of **G**, i.e., the set of edges with minimum weight that have to be removed from **G** so that there is no path from **s** to **t** in **G**.

1-1. Motivation-Applications

Max-Flow (or Min-Cut) problems arise in various applications, e.g.,

1. Transportation-related problems (what is the best way to send goods/material from **s** (perhaps a factory) to **t** (perhaps a super-sink of all end-users)
2. Network attacks problems (sabotage/destroy some edges to disconnect two important points **s** and **t**)
3. (Bipartite) Matching and Assignment problems (that also has specialized algorithms, see Graph Matching visualization
4. Sport teams prospects
5. Image segmentation, etc...

2. Visualization

This visualization page will show the execution of a chosen Max Flow algorithm running on a flow (residual) graph.

To make the visualization of these flow graphs consistent, we enforce a graph drawing rule for this page whereby the source vertex **s**/sink vertex **t** is always vertex 0/**V**-1 and is always drawn on the leftmost/rightmost side of the visualization, respectively. Another visualization-specific constraint is that the edge capacities are integers between [1..99].

These visualization-specific constraints do **not** exist in the standard max flow problems.

2-1. Input

The input for a Max Flow algorithm is a flow graph (a **directed weighted** graph **G** = **(V, E)** where edge weight of edge **e** represent the capacity **c(e)** (the unit is problem-dependent, e.g., liters/second, person/hour, etc) of flow that can go through that edge) with two distinguished vertices: The source vertex **s** (with in-degree 0) and the sink/target/destination vertex **t** (with out-degree 0). The flow graph is usually **s-t** connected, i.e., there is at least one path from **s** to **t** (otherwise the max flow is trivially 0).

In this visualization, these two additional inputs of **s** (usually vertex 0) and **t** (usually vertex **V**-1) are asked before the execution of the chosen Max Flow algorithm and can be customized by the user.

2-2. Output

The output for a Max Flow algorithm is the max flow value and an assignment of flow **f** to each edge that satisfies two important constraints:

1. **Capacity constraints** (flow on each edge (**f(e)**) is between 0 and its (unit) capacity (**c(e)**), i.e., $0 \leq \mathbf{f(e)} \leq \mathbf{c(e)}$ — not negative and not more than the capacity), and
2. **Equilibrium constraints** (for every vertex except **s** and **t**, flow-in = flow-out)

so that the value of the flow ($\mathbf{value(f)} = \sum_{v:\,(s,\,v) \in E} \mathbf{f(s,v)}$) is maximum.

In this visualization, we focus on showing the final max flow value and the final ST-min cut components at the end of each max flow algorithm execution, instead of the precise assignment of flow **f** to each edge, i.e., **f(e)** must be computed manually from the initial capacity **c(e)** (first frame of the animation) minus the final residual capacity of that edge **e** (last frame of the animation). This missing feature will likely be added in the next iteration of this visualization page.

Discussion: Is there other ways to compute the value of the flow **value(f)**?

2-3. The Answer

[This is a hidden slide]

2-4. Residual Graph

At the start of the three Max Flow algorithms discussed in this visualization (Ford-Fulkerson method, Edmonds-Karp algorithm, and Dinic's algorithm), the initial flow graph is converted into residual graph (with potential addition of back flow edges with initial capacity of zeroes).

The edges in the residual graph store the *remaining* capacities of those edges that can be used by future flow(s). At the beginning, these remaining capacities equal to the original capacities as specified in the input flow graph.

A Max Flow algorithm will send flows to use some (or all) of these available capacities, iteratively.

Once the remaining capacity of an edge reaches 0, that edge can no longer admit any more flow. In the near future, we will update this visualization so that any edge in the residual graph that has capacity 0 (including the initial zeroes of the back flow edges) is **not** shown in the visualization.

3. Specifying Input Flow Graph

There are three different sources for specifying an input flow graph:

1. **Draw Graph**: You can draw **any** directed weighted (weight ∈ [1..99]) graph as the input flow graph with vertex 0 as the default source vertex (the left side of the screen) and vertex **V**-1 as the default sink vertex (the right side of the screen),
2. **Modeling**: Several graph problems can be reduced into a Max Flow problem. In this visualization, we have the modeling examples for the famous Maximum Cardinality Bipartite Matching (MCBM) problem, Rook Attack problem (currently disabled), and Baseball Elimination problem (currently disabled),
3. **Example Graphs**: You can select from the list of our selected example flow graphs to get you started.

4. Max Flow Algorithms

There are three different max flow algorithms in this visualization:

1. The slow O(**mf** × **E**) **Ford-Fulkerson** method,
2. The O(**V** × **E^2**) **Edmonds-Karp** algorithm, or
3. The O(**V^2** × **E**) **Dinic's** algorithm.

There are a few other max flow algorithms out there, but they are not available in this visualization yet.

4-1. Similar But Not The Same

For the three Max Flow algorithms discussed in this visualization, successive flows are sent from the source vertex **s** to the sink vertex **t** via available **augmenting paths** (augmenting path is a path from **s** to **t** that goes through edges with positive weight residual capacity (**c(e)-f(e)**) left).

The three Max Flow algorithms in this visualization have different behavior on how they find augmenting paths.

However, all three Max Flow algorithms in this visualization stop when there is no more augmenting path possible and report the max flow value (and the assignment of flow on each edge in the flow graph).

Later we will discuss that this max flow value is also the min cut value of the flow graph (that famous Max-Flow/Min-Cut Theorem).

5. Ford-Fulkerson Method

```
start with 0 flow
while there exists an augmenting path: // iterative algorithm
 find an augmenting path (for now, 'any' graph traversal will do)
 compute bottleneck capacity
 increase flow on the path by the bottleneck capacity
```

5-1. Max-Flow/Min-Cut Theorem

This famous theorem states that in a flow network, the **maximum flow** from **s** to **t** is equal to the total weight of the edges in a **minimum cut**, i.e., the smallest total weight of the edges that have to be removed to disconnect **s** from **t**.

In a typical Computer Science classes, the lecturer will usually spend some time to properly explain this theorem (explaining what is an st-cut, capacity of an st-cut, net flow across an st-cut equals to current flow f assignment that will never exceed the capacity of the cut, and finally that Max-Flow/Min-Cut Theorem). For this visualization, we just take this statement as it is.

Discussion: For live class in NUS, we will actually discuss these theorem.

5-2. Cuts and Flows, Definitions

[This is a hidden slide]

5-3. Cuts and Flows, Equal

[This is a hidden slide]

5-4. Cuts and Flows, Weak Duality

[This is a hidden slide]

5-5. Max-Flow/Min-Cut Theorem, Formally

[This is a hidden slide]

5-6. Augmenting Path Theorem

Using the Max-Flow/Min-Cut Theorem, we can then prove that flow **f** is a maximum flow if and only if there is no (more) augmenting path remaining in the residual graph.

As this is what Ford-Fulkerson Method is doing, we can conclude the correctness of this Ford-Fulkerson Method, i.e., if Ford-Fulkerson Method terminates, then there is no augmenting path left and thus the resulting flow is maximum (and we can also construct the equivalent Min-Cut, next slide).

5-7. Finding Edges in the Min-Cut

We can constructively identify the edges in the Min-Cut as follows:

1. Run Ford-Fulkerson (or any other Max Flow) algorithm until it terminates.
2. Let **S** be the set of vertices that are still reachable from the source **s**.
   We can run DFS (or BFS) in the residual graph from the source vertex **s**.
   All the vertices that are still reachable are in **S**.
   Let **T** be the remaining vertices, i.e., **T = V \ S**.
3. For every edge in **S**, enumerate outgoing edges:
   If edge exits **S** (and into **T**), add to min-cut.
   If both ends of edge are in **S**, then continue.

That's it, **(S,T)** is an st-cut, edges from (**S** → **T**) are the minimum cut, and the flow that goes through this minimum cut **(S,T)** is the maximum possible.

5-8. The Proofs

[This is a hidden slide]

5-9. Analysis of Ford-Fulkerson Method

Ford-Fulkerson method always terminates if the capacities are integers.

This is because every iteration of Ford-Fulkerson method always finds a new augmenting path and each augmenting path must has bottleneck capacity at least 1 (due to that integer constraint). Therefore, each iteration increases the flow of at least one edge by at least 1, edging the Ford-Fulkerson closer to termination.

As the number of edges is finite (as well as the finite max capacity per edge), this guarantees the eventual termination of Ford-Fulkerson method when the max flow **mf** is reached and there is no more augmenting path left.

In the worst case, Ford-Fulkerson method runs for **mf** iterations, and each time it uses O(**E**) DFS. The rough overall runtime is thus O(**mf** × **E**) — this is actually not desirable especially if the value of **mf** is a huge number.

Discussion: What if the capacities are rational numbers? What if the capacities are floating-point numbers?

5-10. Non-Integer Capacities

[This is a hidden slide]

6. Shortest Augmenting Paths First

Idea: What if we don't consider **any** augmenting paths but consider augmenting paths with the smallest number of edges involved first (so we don't put flow on more edges than necessary).

6-1. Idea 1: Edmonds-Karp

Implementation: We first ignore capacity of the edges first (assume all edges in the residual graph have weight 1), and we run O(E) BFS to find the shortest (in terms of # of edges used) augmenting path. Everything else is the same as the basic Ford-Fulkerson Method outlined earlier.

It can be proven that Edmonds-Karp will use at most O(VE) iterations thus it runs in at most in O(VE * E) = O(VE^2) time.

6-2. The Proofs

[This is a hidden slide]

6-3. Idea 2: Dinic's

Dinic's algorithm also uses similar strategy of finding shortest augmenting paths first.

But Dinic's algorithm runs in a faster time of O(V^2 × E) due to the more efficient usage of BFS shortest path information.

This slide will be expanded.

7. Efficient Max Flow Algorithm Implementation

When you are presented with a Max Flow (or a Min Cut)-related problem, we do not have to reinvent the wheel every time.

You are allowed to use/modify/adapt/enhance our implementation code for Max Flow Algorithms (Edmonds-Karp/Dinic's): maxflow.cpp | py | java | ml.