Notes on Maximum Flow Algorithms

Pramook Khungurn

December 11, 2019

1 Networks, Flows, and Cuts

- A *network* is composed of
  - a set of vertices V;
  - a vertex s called the *source*;
  - a vertex t called the *sink*;
  - a function $c : V^2 \rightarrow \mathbb{R}^+ \cup \{0\}$ called the *capacity*.
- We can draw a network as a directed graph by drawing a directed edge from vertex u to v if $c(u, v) > 0$.
- A *flow* in a network G is a function $f : E \rightarrow \mathbb{R}$ with the following properties:
  - **Skew Symmetry:** for all $u, v \in V$, we have
  $$f(u, v) = -f(v, u).$$
  - **Conservation:** for all vertex u not equal to s or t, we have
  $$\sum_{v \in V} f(u, v) = 0.$$
  - **Capacity Constraint:** for all $u, v \in V$, we have
  $$f(u, v) \leq c(u, v).$$
- If $f(u, v) > 0$, we say that there is a flow out of u by the value $f(u, v)$ to v. if $f(u, v) < 0$, we say that there is a flow of value $f(u, v)$ from v to u.
- An *s,t-cut* of a network G is a partition of V into two sets A and B where $s \in A$ and $b \in B$.
- The *capacity* of the cut A, B is
$$c(A, B) = \sum_{u \in A, v \in B} c(u, v).$$
- Let f be a flow. The *value of the flow across the cut A, B* is given by
$$f(A, B) = \sum_{u \in A, v \in B} f(u, v).$$
- Obviously, $f(A, B) \leq c(A, B)$.
- The *value* of the flow f, denoted by $|f|$ is defined as $|f| = f(\{s\}, V - \{s\})$. In other words, it is the flow out of vertex s.

- We can prove by induction that $|f| = f(A, B)$ for any cut A, B.
- As a result, $|f| \leq c(A, B)$ for any cut A, B.
- Let G be a network and f is a flow. Define the *residual capacity* $r : V^2 \to \mathbb{R}^+ \cup \{0\}$ as follows: for all $u, v \in V$,
$$r(u, v) = c(u, v) - f(u, v).$$
We also define the *residual network* G_f, which is G with its capacity function replaced by r.
- The following lemma says that a flow in the residual network gives a new flow in the original network.

  Lemma 1.1. *Let G be a network, f a flow in G, and G_f the residual network.*

  (a) The function f' is a flow in G_f if and only if $f + f'$ is a flow in G.

  (b) The value function is additive: $|f + f'| = |f| + |f'|$ and $|f - f'| = |f| - |f'|$.

  Proof. Let f' be a flow in G_f. We have that f' satisfies skew symmetry and conservation if and only if $f + f'$ satisfies both. Now, we have that
$$f'(u, v) \leq c(u, v) - f(u, v) \iff f(u, v) + f'(u, v) \leq c(u, v)$$
  So, f' satisfies the capacity constraint if and only if $f + f'$ satisfies it too. We are done with (a).

  Now,
$$|f \pm f'| = \sum_{v \in V} (f \pm f')(s, v) = \sum_{v \in V} (f(u, v) \pm f'(u, v)) \sum_{v \in V} f(u, v) \pm \sum_{v \in V} f'(u, v) = |f| \pm |f'|.$$
  We are done with (b). □

- A flow of maximum value is called a *max flow.*
- An *augmenting path* is a path in G from s to t such that all edges on the path have positive capacity.
- We now have the famous *Max Flow-Min Cut* theorem.

  Theorem 1.2. *The following statements are equivalent:*

  (a) f is a max flow in G;

  (b) there is an s, t-cut A, B with $c(A, B) = |f|$;

  (c) there is no augmenting path in G_f.

2 Ford–Fulkerson Algorithm

- The Max Flow-Min Cut gives a simple algorithm for finding the maximum flow:
  1. Start with the zero flow $f(u, v) = 0$ for all $u, v \in V$.
  2. Find an augmenting path $(v_1, v_2), \ldots, (v_{k-1}, v_k)$ where $v_1 = s$ and $v_k = t$ in G_f.
  3. If there is no augmenting path, terminate.
  4. If there is an augmenting path, let d be the capacity of the edge in the path with the lowest capacity. (This capacity will now be referred to as the *bottleneck capacity.*) Define flow g as follows:
     - $g(v_i, v_{i+1}) = d$ for all i;

  - $g(v_{i+1}, v_i) = -d$ for all i;
  - $g(u, v) = 0$ for all other pairs of vertices $u, v \in V$.

5. Up date f to be $f + g$.
6. Go back to Step 2.

This algorithm is called the *Ford–Fulkerson* algorithm.

- The Ford–Fulkerson algorithm is, unfortunately, not polynomial time.

  The problem is that it does not specify how to find an augmenting path.

  As such, there are graphs where if the augmenting path is found in a specified way, the algorithm runs forever.

- One tool used to analyze network flow algorithms is the following *flow decomposition lemma.*

  Lemma 2.1. *Any flow in G can be expressed as a sum of at most m* path flows *and a zero-valued flow.*

  By a *path flow*, we mean the flow that we generate from an augmenting path as seen in the description of the Ford–Fulkerson algorithm.

  Proof. Let f be a flow with $|f| > 0$. Consider the new capacity function $c'(u, v) = \max\{f(u, v), 0\}$, and a new network G' with this capacity function.

  Since the value of max flow in G is $|f| > 0$, there must be an augmenting path. Let p be the path flow of this augmenting path. We have that $f - p$ is a flow in G'. Consider the network G'' generated by the capacity function $c''(u, v) = \max\{f(u, v) - p(u, v), 0\}$. We have that G'' has one fewer edges than G' because the edge with the bottleneck capacity of p gets its capacity reduced to zero.

  We can repeat the same process to G''. This process can continue at most m times because one edge disappear each time. Also, each time, we take out a path flow from the original flow. Hence, the flow can be decomposed into at most m path flows. □

3 Heuristics of Edmonds and Karp

- Edmonds and Karp gave two heuristics for choosing augmenting paths.

  1. **Use the path with maximum bottleneck capacity first.** If the capacity function takes on integer values and f^* is a maximum flow, then this heuristics will result in at most $O(m \log |f*|)$ augmenting steps, where m is the number of edges.

     We can find the path with maximum bottleneck capacity with a modified Dijkstra's algorithm. Hence, the heuristics gives an $O(m^2 \log n \log |f^*|)$ time, where n is the number of vertices.
  2. **Use the path with the minimum hops first.** This results in an $O(m^2 n)$ algorithm for maximum flow. We will discuss this heuristic in details.

- The reason why the second heuristic works is that *the shortest augumenting path never gets shorter.* In fact, we have the following lemma.

  Lemma 3.1. *Let p be a shortest augmenting path in G. Let G' be the residual network obtained by using the path flow along p. Let q be a shortest augmenting path in G'. Then, q is not shorter than p.*

  To prove the lemma, we need the following definition.

  Definition 3.2. *A* level graph *L_G of network G is a graph formed by:*

1. *doing a BFS starting at s, thereby partitioning the graph into layers, and*
2. *removing any edges pointing backwards or pointing to vertices in the same layer.*

Proof. (Lemma 3.1) Let p be the shortest augmenting path. Note that all edges of p are present in the level graph. Consider the residual graph after augmenting along p. We have that one edge of the level graph will not appear in the residual graphs. Moreover, there may be at most new $|p|$ edges appearing in the residual graph, but they all point backwards in the level graph. Hence, these new edges cannot make the sink closer to the source. Therefore, the shortest path from sink to source cannot be shorter than the one in the original graph. □

- We now show that the distance between s and t must increase after augmenting along the shortest augmenting paths m times.

  Lemma 3.3. *We can augment along the shortest augmenting paths with the same length at most m times.*

  Proof. Consider the level graph. A shortest path must consists of only edges in the level graph. Since an edge in the level graph disappears each time we augment, we can augment at most m times. □

- Since the shortest distance from s to t can be at most $n-1$. There are at most mn augmentation. Since finding the shortest augmenting path takes $O(m)$ time, the second heuristics of Edmonds and Karp gives an $O(m^2n)$ algorithm for finding the maximum flow.

4 Blocking Flows and Dinic's Algorithm

- An edge in a network G is called *admissible* if it is in the level graph L_G.
- An *admissible path* is an s,t-path consist entirely of admissible edges.
- A *blocking flow* is a flow where at least on edge in every admissible path is saturated.
- We have the following lemma.

  Lemma 4.1. *Let G be a network where s and t are d hops apart. Let f be a blocking flow. Then, s and t are at least $d+1$ hops apart.*

  Proof. Since s and t are d hops apart in G, and augmenting a flow only add backward edges to G, we must have that s and t are at least d hops apart in G_f as well.

  For s and t to be exactly d hops apart, there must be an admissible path from s to t. However, by the definition of blocking flow, any s,t-path in G_f must contain an edge that is not admissible. Therefore, s and t must be at least $d+1$ hops apart. □

- We now have a framework for solving the max flow problem.

  1. Find a blocking flow f in G
  2. Push f through the network.
  3. Update $G \leftarrow G_f$.
  4. Go back to Step 1 until we cannot find more blocking flows.

  Note that We have to keep doing this at most n times because s and t can only be at most $n-1$ hops apart.

- How to find a blocking flow? We present the following algorithm by Dinic.

  1. Compute the layer graph L_G.
  2. Start at vertex $v = s$ and keep carrying out the following operations in L_G:
     - **Advance:** If v has an outgoing edge (v, w) with $c(v, w) > 0$, go to w.
     - **Augment:** If $v = t$, then we have found a path from s to t. Augment the path with the bottleneck capacity, and remove all the saturated edges from the level graph. Then start at s again.
     - **Retreat:** If v does not contain any outgoing edges, we remove v and all its incident edges from the graph, and backtrack to the vertex that lead to v.

     We continue doing these operations until s does not have any out going edges.

- The above algorithm runs in $O(mn)$ time. The analysis goes as follows.

  Observe that each time we perform Augment, at least an edge in the level graph disappear. Therefore, there can be at most m Augment. So all the Augment takes $O(mn)$ time.

  We can Retreat at most n time, and it takes $O(m)$ time to remove all the edges. So all the Retreat takes $O(m + n)$ time.

  An Advance that leads to the release can be charged with the Retreat. A series of Advances that lead to an Augment can also be charged with the Augment. So, there can be no more than $O(mn)$ advances.

- Since finding a blocking flow takes $O(mn)$ time, we have a max flow algorithm that runs in $O(mn^2)$ time.

5 Dinic's Algorithm in Unit Capacity Network

- A *unit capacity network* is a network whose edges have capacity 0 or 1.
- Dinic's algorithm runs faster in a unit capacity network.

  Lemma 5.1. *In a unit capacity network, Dinic's algorithm to find a blocking flow takes $O(m)$ time.*

  Proof. An Augment in a unit capacity graph makes all edges in the augmenting path disappear. Therefore, all Augments take at most $O(m)$ time. □

- Moreover, fewer blocking flows are needed to complete the max flow.

  Lemma 5.2. *Let d be a positive integer. In a unit capacity network, after pushing d blocking flows, pushing only m/d more blocking flows completes the max flow.*

  Proof. After pushing d blocking flows, the distance between s and t is at least d. Consider the level graph of the residual graph. Let V_i be the set of vertices in the ith layer.

  Define $A_i = V_0 \cup V_1 \cup \cdots \cup V_i$, and $B_i = V - A_i$. We have that (A_0, B_0), (A_1, B_1), (A_2, B_2), and (A_{d-1}, B_{d-1}) are cuts. The capacity of (A_i, B_i) is equal to the number of edges going from Layer i to Layer $i+1$. Consider the the cut among these d cuts with the lowest capacity. We must have that this cut's capacity is at most m/d. Therefore, the max flow in the residual graph has value at most m/d. Since each blocking flow increases the value of the overall flow by 1, only m/d more blocking flows are needed. □

- In conclusion, Dinic's algorithm is very fast in unit capacity networks.

Theorem 5.3. *The running time of Dinic's algorithm in a unit capacity network is $O(m^{3/2})$.*

Proof. The last lemma tells us that Dinic's algorithm runs in $O(m(d + m/d))$ for any integer m. We can minimize the expression $d + m/d$ by choosing $d = \sqrt{m}$. □

6 Scaling Algorithm

- We can also find max flow faster than the general case when we assume that the capacity are integers.
- Let U be an integer upper bound on all the capacity of all edges.
- We decompose the network G into $k = \log U$ networks: G_0, G_1, …, G_{k-1}. Here, G_i is the network whose capacity is given by
$$c_i(u,v) = \left\lfloor \frac{c(u,v)}{2^i} \right\rfloor.$$
In other words, $c_i(u,v)$ is obtained by $c(u,v)$ truncating i least significant digits off.
- Now, we start by finding a maximum flow f_0 in G_0. Since this is a unit capacity network, it can be done in $O(m^2)$ time. (We know that it's actually $O(m^{3/2})$, but let's say it's $O(m^2)$ for now.)

  Next, we will find the maximum flow f_1 in G_1. We do so by observing that the flow $2f_0$ is a flow in G_1. We push the $2f_0$ to the network first, and then try to find the max flow in the residual graph.

  The nice thing is that $|f_1| \leq 2|f_0| + m$. The reason is that there is a cut in G_0 which has capacity $|f_0|$. Looking at the same cut in G_1, the value of each edge is doubled and may be incremented by 1. So the capacity of that cut does not exceed $2|f_0| + m$.

  This means that we only need at most m augmentations. Thus, finding $|f_1|$ can be accomplished in $O(m^2)$.

  Of course, this line of reasoning genaralizes to all G_i. So, we can use the same trick to find the max flow in G_2, G_3, and so on until we reach G_{k-1}, which is G itself. Hence, we have a max flow algorithm which runs in $O(m^2 \log U)$.