CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 3 — January 24, 2023

Prof. Jelani Nelson *Scribes: Alejandro Sanchez and Sohom Paul*

1 Overview

In the last lecture we finished discussing the BNW algorithm for the single-source shortest paths problem. We ended the lecture by talking about how scaling can be used to speed up max flow algorithms like Ford-Fulkerson. We now continue our discussion of max flow algorithms by looking at a different approach, namely, blocking flows. We will use blocking flows to motivate link-cut trees, which we will discuss next lecture.

2 Preliminaries

Recall the statement of max $s - t$ flow:

Definition 2.1 (Capacitated Graph)**.** A *capacitated graph* is an unweighted directed graph $G = (V, E)$ equipped with a *capacity function* $u : E \to \mathbb{R}$. Typically, we will have u map into a finite set of positive integers $\{1, \ldots, U\}$.

Definition 2.2 (Max Flow)**.** Given as input a capacitated graph $G = (V, E, u)$ and a pair of vertices $s, t \in V$, the *maximum flow problem* is to compute some flow $f : E \to \mathbb{R}$ that maximizes the *value* of the flow, given by $f^* = \sum_{e=(s,\cdot)} f_e$, subject to the following constraints:

- **Positivity:** $\forall e \in E, f_e \geq 0$
- **Capacity:** $\forall e \in E, f_e \leq u_e$
- **Conservation of Flow:** $\forall v \in V \setminus \{s, t\}, \sum_{e=(v,\cdot)} f_e = \sum_{e=(\cdot,v)} f_e$

A flow f that satisfies the constraints but is not necessarily optimal is called a *feasible* flow.

Note that the optimization problem defining max flow has a linear objective and linear constraints, so it can be solved by a general-purpose linear program solver. However, faster solutions exist where we exploit the underlying graph structure of the problem.

In undergraduate algorithms, we have seen the Ford-Fulkerson algorithm for max flow, which works by repeatedly augmenting the flow in the residual graph until no more flow can be routed. We recall the definitions of these objects.

Definition 2.3 (Residual Graph)**.** Given a flow f on capacitated graph $G = (V, E, u)$, we define the *residual graph* G_f as a new capacitated graph $G_f = (V, E \cup \text{rev}(E), u')$ (where $\text{rev}(E)$ denotes reversing all the edges). To compute u', we initialize $u'_e = f_e$ for $e \in E$ and 0 otherwise. Then, for each $e \in E$, we apply $u'_e \leftarrow u'_e - f_e$ and $u'_{\text{rev}(e)} \leftarrow u'_{\text{rev}(e)} + f_e$.

Definition 2.4 (Augmentation)**.** Given a flow f_1 in G and a flow f_2 in the residual graph G_{f_1}, we *augment* the first flow to give a new feasible flow on G given by $f_1 + f_2$.

3 Historical Development of Max Flow Algorithms

Broadly speaking, there are three classes of max-flow algorithms.

- *pseudopolynomial* algorithms run in $\text{poly}(m, n, U)$ time[1];
- *weakly polynomial* algorithms run in $\text{poly}(m, n, \log U)$ time
- *strongly polynomial* algorithms run in $\text{poly}(m, n)$ time[2]

Note that naive Ford-Fulkerson takes $O(mf^*) = O(mnU)$ time and is thus pseudopolynomial, while Ford-Fulkerson with scaling achieves $O(m^2 \log U)$ time and is thus weakly polynomial. Finally, Edmonds-Karp (which is variant of Ford-Fulkerson where we always search for $s - t$ paths in the residual graph using breadth-first search) takes only $O(m^2 n)$ time and is thus strongly polynomial.

We have achieved the following runtimes for the max flow problem:

- strongly polynomial algorithms:
  - [Orl13] runs in $O(mn)$ time
- weakly polynomial algorithms:
  - [GR98] runs in $O(m \min\{\sqrt{m}, n^{2/3}\} \log(n^2/m) \log U)$ time
  - [LS14] runs in $\tilde{O}(m\sqrt{n} \log U)$ time
  - [CKL+22] runs in $m^{1+o(1)} \log U$ time

In particular, the approaches of [LS14] and [CKL+22] make use of heavy continuous optimization machinery.

4 Blocking Flows

4.1 Intuition

Recall that in Ford-Fulkerson, we iterate the process of computing the residual graph with respect to the current flow, finding a flow within the residual graph, and augmenting to get a new flow. However, in Ford-Fulkerson, we always augment using only a path, which leads to our algorithm taking a large number of iterations in the worst case. Today, we will be able to speed things up by augmenting according to a *blocking flow.*

Definition 4.1 (Level Graph)**.** Given a residual graph G_f, the *level graph* L is the subgraph given by edges (v, w) such that $\text{dist}_{G_f}(s, w) = \text{dist}_{G_f}(s, v) + 1$, where dist denotes the shortest path distance. We call the set of edges appearing in the level graph *admissible.* An *admissible path* is a path that only uses admissible edges.

[1]Note that U takes $\log U$ bits to represent, so a $\text{poly}(U)$ factor is actually exponential in the input length.

[2]We work in the *word RAM* model, which means given an input of B bits we can perform memory access and arithmetic on words of size $O(\log B)$ in constant time. This means that we do not have to incur a $\log U$ runtime penalty just because the input values take $\log U$ bits to represent.

Definition 4.2 (Blocking flow)**.** A *blocking flow* $\tilde{f}$ in residual graph G_f is one that only contains admissible edges and is such that every admissible $s-t$ path has at least one edge fully saturated by $\tilde{f}$.

We show an example of a blocking flow in Fig. 1. Note that blocking flows need not be max flows.

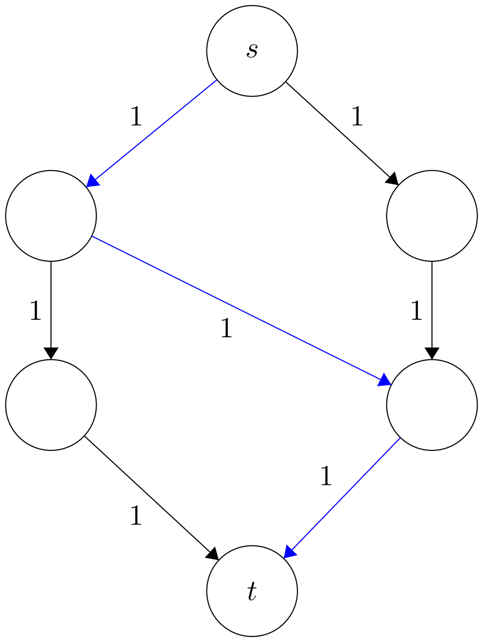

Figure 1: Vertices above are horizontally aligned according to their level. A blocking flow is highlighted in blue. Note that the blocking flow is not a max flow for this graph.

4.2 Monotonically Increasing $s-t$ Distance

This is the main lemma we will need in the analysis of our blocking flow algorithms, as it will give us an upper bound to the number of iterations our algorithm will take.

Lemma 4.3. *Let G_f be some residual graph and L be the associated level graph. Suppose we augment f along some blocking flow. Then the new level graph L' will be such that* $\text{dist}_{L'}(s,t) > \text{dist}_L(s,t)$.

Proof. Define level i to be the set of vertices at distance i from s in the *original* graph G_f. Bear in mind that after we augment along our blocking flow, we will have a new residual graph G'_f, but we will still be referring to level sets according to their distance in G_f. Observe that G_f contains some edges that go up exactly 1 level[3] and some edges that decrease or keep the level the same[4]. Edges that go up more than one level do not exist. Finally, note that s is in level 0 and t is in level $D := \text{dist}_L(s,t)$.

[3]By definition, these are the edges that comprise L.

[4]Naturally, these are the edges in $G_f \setminus L$.

When we augment along an admissible flow, we can potentially saturate edges in L and introduce new reverse edges that go down from the current level, but we certainly cannot introduce new edges that go up levels as this would mean we pushed flow along an edge in $G_f \setminus L$, contradicting admissibility. It follows that $\text{dist}_{L'}(s,t) \geq D$, as we need to follow at least D edges in L' in order to go from level 0 to level D. Indeed, we also can conclude that equality cannot hold, because that would mean that every edge we followed has increased the level, implying we only used edges in L. This contradicts the assumption that we augmented along a blocking flow, as we could have simply pushed more flow along this path. □

Note that the maximum finite value $\text{dist}_L(s,t)$ can be for a level graph associated with any flow is $n-1$, as there cannot be a path with more than n vertices in a graph. This means that we can increase $\text{dist}_L(s,t)$ at most $O(n)$ times before we disconnect s and t. Hence, all we need is to find an algorithm to compute a single blocking flow, and iterating it $O(n)$ times on successive residual graphs will give an algorithm to solve the max flow problem.

4.3 Finding a Blocking Flow, Unit Capacity Case

For now, let us assume that our input graph has unit capacities (i.e. $u_e = 1$ for all e). We run an algorithm that is essentially a modified depth-first search on the level graph L looking for admissable $s-t$ paths. When we arrive at vertex v, we take one of three operations:

1. `advance`: If there is a neighbor w of v such that $(v,w) \in L$, move to w. Store $\text{p}(w) \leftarrow v$ to maintain the current path.
2. `retreat`: If there are no admissible edges from v, go back to $\text{pred}(v)$ and remove the edge $(\text{pred}(v), v)$ from L.
3. `augment`: If we have reached t, then add 1 unit of flow to each edge in the $s-t$ path found, remove the path from L, and go back to s.

Warning: The `augment` operation does *not* compute the new residual graph G'_f; we wait until we have completed finding the blocking flow in order to do so.

In the algorithm above, we `advance` and `retreat` successively until we find an opportunity to `augment`. We do this iteratively until we have no more admissible paths from s to t, which indicates that we have found a blocking flow for L. Once we have found a blocking flow for L, we reconstruct the new residual graph, G'_f. From G'_f, we construct its corresponding level graph L. This is summarized in Algorithm 1

4.3.1 Runtime analysis

For our first analysis, we reiterate the fact that we need to find at most $O(n)$ blocking flows before we end up disconnecting s and t. Therefore, the outer loop of Algorithm 1 runs at most $O(n)$ times. The overall runtime will depend on the time it takes to find a blocking flow.

Claim 4.4. Finding a blocking flow according to the algorithm specified in Subsection 4.3 takes $O(m)$ time.

Algorithm 1 Max Flow Blocking Algorithm – Unit Capacities

```
Require: $G$ – a capacitated graph.
1: $G_f \leftarrow G$
2: repeat
3:     $L \leftarrow LevelGraph(G_f)$
4:     repeat
5:         augment flow as much as possible in $L$
6:     until we find a blocking flow $\tilde{f}$
7:     $G_f \leftarrow Residual(G_f, \tilde{f})$
8: until there is no path from $s$ to $t$ in $G_f$
```

Proof. Note that we can only `advance` along a given edge once, as it will either trigger a future `retreat` or `augment`. Thus we take $O(m)$ time across all calls to `advance`. Similarly, each edge contributes only $O(1)$ work to the total time spent `retreat`ing or `augment`ing, so those operations take $O(m)$ time as well. We conclude that finding a blocking flow can be done in $O(m)$ time for unit capacity graphs. □

Combining this algorithm with our outer loop as described in Algorithm 1 means that we have a $O(mn)$ runtime. However, using a different analysis gives us a different bound on the number of iterations, and therefore a more accurate overall runtime.

Claim 4.5. The overall runtime of Algorithm 1 is $O(m^{3/2})$.

Proof. As the shortest path distance from s to t in G_f increases each iteration, we deduce that after d iterations the shortest $s-t$ path has at least d edges. However, as the graph is unit capacity, then any max flow from s to t in G_f can be written as a disjoint union of paths and cycles, as no edge can be routing flow for two different paths at once. Thus the total $s-t$ flow is upper bounded by the number of paths from s to t, which is in turn upper bounded by m/d as there are m total edges. As each iteration of our algorithm will add at least 1 unit of flow until termination, we conclude that there can be only m/d further iterations before the algorithm terminates. Thus the total number of iterations is upper bounded by $\min_d(d + m/d) = O(\sqrt{m})$. This gives us an overall runtime of $O(m^{3/2})$. □

4.4 Finding a Blocking Flow, General Case

For the general case with non-unit capacities, we consider the same algorithm as in Subsection 4.3, but where `augment` will route as much flow as the path has capacity for.

Claim 4.6. With general capacities, the algorithm described above takes $O(mn)$ time to find a blocking flow.

Proof. As before, we consider the total time spent performing each operation:

- `advance`: At some point in each sequence of `advance`s, we will either end up with a `retreat` or an `augment`. We can have at most $O(n)$ consecutive `advance`s (as there are no paths of length greater than n), and there are at most $O(m)$ `retreat`s and `augment`s (see below), so we spend at most $O(mn)$ time in `advance`.

- `retreat`: We can only `retreat` across each edge once, so we spend at most $O(m)$ time in `retreat`.

- `augment`: Every time we `augment`, at least one edge in that path gets fully saturated, so we remove that edge from the graph. Thus we can augment at most m times. Each augment takes at most $O(n)$ time (from the path length), so spend at most $O(mn)$ time in `augment`. $\square$

As before, we note that there are $O(n)$ iterations so computing a max flow takes $O(mn^2)$ time in a general capacity graph.

4.5 Link-Cut Trees Preview

Note that Algorithm 1 may encounter the same vertex across multiple `advance`s, so it stands to reason that a clever choice of data structure would be able to memoize the $v - t$ paths. This forms the basis for link-cut trees.

References

[CKL$^+$22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 612–623. IEEE, 2022.

[GR98] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.

[LS14] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in õ(vrank) iterations and faster algorithms for maximum flow. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 424–433. IEEE Computer Society, 2014.

[Orl13] James B. Orlin. Max flows in $o(nm)$ time, or better. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 765–774. ACM, 2013.