COMS 4995-2: Advanced Algorithms (Spring'20) Feb 11, 2020

Lecture 7: Polynomial-time algorithms for max-flow (cont.)

Instructor: *Alex Andoni* Scribes: *Asif Mallik, Stoyan Atanassov*

1 Reminder: Max-Bottleneck Algorithm

The Max-Bottleneck algorithm is a variation of Ford-Fulkerson, in which each iteration applies an augmenting path with the highest possible capacity, i.e. a path P in the residual graph G_f, that has the maximum possible value (over all paths in G^f) for $\delta(P) = \min_{e \in P}(c_f(e))$.

We proved that, when running Max-Bottleneck:

1. The number of iterations (number of augmenting paths that are applied) for an original graph G with n nodes, m edges of integer capacity, and maximum edge capacity U, is $O(m \log nU)$;
2. The time per iteration (to find an augmenting path with the highest capacity) is $O(m \log U)$.

Thus, the total time for Max-Bottleneck is $O(m^2 \log U \log nU)$. **Note:** in the scribes for the previous lecture, we used C in lieu of U.

We also mentioned two approaches that could be used to find the highest capacity augmenting path during each iteration:

- **Binary search** for the maximum edge capacity for which an $s \to t$ path exists in G_f that includes only edges of greater than or equal such capacity. This can be illustrated by calling the following recursive routine with $min = 1$ and $max = U$:

  find-max-bottleneck(min, max)

  1. If $min = max$ then find and return any $s \to t$ path in G_f that contains only edges with capacities $\geq min$, return $\emptyset$ if no such path exists
  2. Let $mid = \lceil \frac{max+min}{2} \rceil$
  3. If an $s \to t$ path exists in G_f that contains only edges with capacities $\geq mid$, then return **find-max-bottleneck**(mid, max)
  4. Else, return **find-max-bottleneck**$(min, mid - 1)$

  When using breadth-first search to find $s \to t$ paths in G_f above, this approach runs in time $O(m \log U)$.

- **Dynamic programming**, using a variation of Dykstra's algorithm, that works in $O(m \log n)$ time. (A summary can be found in Sections 14.1-3 of *http://cs.cmu.edu/˜avrim/451f11/lectures/lect1013.pdf*).

2 Scaling Algorithm

This algorithm results in similar time as Max-Bottleneck, but adopts a more general approach – instead of finding and applying an augmenting path repeatedly, it attempts to solve the flow problem by using a series of graphs with increasingly accurate approximations of the edge capacities in the original graph.

The general idea is to proceed in a number of "scaling stages", where we begin by pushing as much flow as possible along the bigger capacity edges in the original graph, and then, as we fill those up, to refine the flow by including smaller capacity edges and additional capacity along the already included edges.

We define the total number of scaling stages that the algorithm employs as:

$$b = \lceil \log_2 U \rceil$$

Note: the number of scaling stages is in essence the number of bits sufficient to represent U.

For each scaling stage i, we also define an associated graph:

$$G^i : \text{with capacities } c^i(e) = \left\lfloor \frac{c(e)}{2^{b-i}} \right\rfloor$$

Note: G^i is basically the original graph G, but with capacities $c^i(e)$ equal to only the i most significant bits of their respective original values $c(e)$.

To illustrate how we build the graphs in each scaling stage, suppose in our original graph G the edge capacities, represented in binary, are:

$$\begin{aligned} c(e_1) &= 100.....1 \\ c(e_2) &= 001.....1 \\ c(e_3) &= 010.....0 \\ &..... \\ c(e_m) &= 010.....0 \end{aligned}$$

In stage 1, we construct graph G^1 with capacities equal to just the most significant bit of the original capacities above: $c^1(e_1) = 1, c^1(e_2) = 0, c^1(e_3) = 0, ...$ G^1, thus, includes only edges that have original capacity of at least $\frac{U}{2}$: think of this as a very rough approximation of the edge capacities. We then proceed to find the maximum flow for G^1 using these coarse capacity approximations (note that, if we multiply back that flow by 2^{b-i}, it will still be a valid flow in the original graph, by virtue or how we approximated the capacity in G^1).

In stage 2, we build graph G^2, by expanding the capacity length by one more bit, allowing us now to consider the two most significant bits of the original capacities in G, resulting in $c^2(e_1) = 2, c^2(e_2) = 0, c^2(e_3) = 1, ...$ (better approximations of the original capacities). We then feed the maximum flow from stage 1 as the starting flow to now find the maximum flow for G^2.

Similarly, in stage 3, we build G^3 where we take the three most significant capacity bits, resulting in capacities $c^3(e_1) = 4, c^3(e_2) = 1, c^3(e_3) = 2, ...$ and we use the maximum flow from stage 2 as the starting flow to find the maximum flow in G^3. And so on.

2.1 The Algorithm

1. Set $b = \log_2 U, f^0 = 0$
2. For stage i in 1 .. b:
   - Find f^i = max flow in G^i, starting with flow $2 \cdot f^{i-1}$, where f^{i-1} is the resulting flow from the previous stage
3. Report flow f^b

Note:

- For stage 1 (graph G^1), the flows per edge will be either 0 or 1 (since all edge capacities in G^1 are equal to only the most significant bit of the edge capacities in G). When we go to stage 2 (graph G^2), we multiply those G^1 flows by 2, but that is still a valid starting flow for G^2 (since the capacities in G^2 are now the first two bits of the capacities in G), etc;
- We use Ford-Fulkerson to find the maximum flow for any G^i graph above, but with a starting flow, based on the maximum flow found in the preceding stage in the loop.

Claim 1. *$2 \cdot f^{i-1}$ is a valid flow in graph G^i.*

Proof. The proof is by induction on i:

$$\begin{aligned} \forall e \in E: \quad & f^{i-1}(e) \le c^{i-1}(e) \\ & 2 \cdot f^{i-1}(e) \le 2 \cdot c^{i-1}(e) \\ \text{but, since we at least double capacities in each stage}: \quad & c^{i-1}(e) \le c^i(e) \\ \text{therefore}: \quad & 2 \cdot f^{i-1}(e) \le 2 \cdot c^{i-1}(e) \le c^i(e) \end{aligned}$$

So, doubling the flow from the previous iteration satisfies capacity constraints (note that, obviously, it is also positive).

It also satisfies flow conservation, since equality of flows from previous iteration would still hold if all flows are multiplied by 2. □

Corollary 2. *The Scaling Algorithm is correct.*

By construction:

- Each flow that we find is valid
- Flow f^b that we compute in the last iteration is the maximum flow in the original graph G, because it is computed using original capacities (all capacity bits are included in iteration b).

2.2 Runtime Analysis

As mentioned, the number of scaling stages that the algorithm utilizes is:

$$b = O(\log U) \tag{1}$$

For each of these stages, the runtime is basically the runtime of Ford-Fulkerson which, in stage i is bounded by the number of augmenting paths in G^i, which is itself bounded as follows:

$$\text{Number of augmenting paths in } G^i \leq |f^i| - |2 \cdot f^{i-1}| \tag{2}$$

This is because, in stage i, we do a "warm" start with flow $|2 \cdot f^{i-1}|$ from stage $i-1$ and then with each augmenting path, we increase the value of the flow by at least 1 until we reach the maximum possible flow $|f^i|$ (this was already mentioned when we first analyzed Ford-Fulkerson).

To place an upper bound on the difference in equation 2, consider graph G^{i-1}. By the Min-Flow/Max-Cut Theorem, it must be that, in graph G^{i-1}:

$$\exists\ s-t\ cut\ S,\ s.t.\ c^{i-1}(S) = |f^{i-1}| \tag{3}$$

By how much can the value of this cut increase from stage $i-1$ to stage i? For each outgoing edge e in the $s-t$ cut S above, the capacity increases from $c^i(e)$ to $c^{i+1}(e)$, where:

$$c^i(e) = \begin{cases} 2c^{i-1}(e) & \textit{if i-th most significant bit of } c(e) = 0 \\ 2c^{i-1}(e) + 1 & \textit{if i-th most significant bit of } c(e) = 1 \end{cases}$$

This means that the cut capacity of the cut S in graph G^i is:

$$c^i(S) \in [2 \cdot c^{i-1}(S), 2 \cdot c^{i-1}(S) + m] \tag{4}$$

The lower bound above corresponding to the case when for all outgoing edges in S the bit that we included in stage i is 0, and the upper bound corresponding to the case when for all outgoing edges that bit was 1 (note that we have at most m edges).

Then:

$$\begin{aligned} \text{By Min-Flow/Max-Cut Theorem}: &\quad |f^i| \leq c^i(S) \\ \text{Adding the upper bound from (4)}: &\quad |f^i| \leq c^i(S) \leq 2 \cdot c^{i-1}(S) + m \\ \text{Which is by (3), equivalent to}: &\quad |f^i| \leq c^i(S) \leq 2 \cdot |f^{i-1}| + m \end{aligned}$$

So, the maximum flow in G^i is at most twice the maximum flow in G^{i-1} plus m. Thus, the boundary for the number of augmenting paths in stage i from equation 2 becomes:

$$\text{Number of augmenting paths in } G^i \leq |f^i| - |2 \cdot f^{i-1}| \leq m \tag{5}$$

Note: how, by not starting from zero, but using the flow from graph G^{i-1}, we effectively reduced significantly how much the flow can further go up in graph G^i.

Combining the facts that we run $O(\log U)$ stages (equation 1), each with at most m iterations of Ford-Fulkerson (equation 2), and that it takes $O(m)$ time to find an augmenting path (say, by using breadth-first search), for the running time of the Scaling Algorithm, we have:

$$\text{Total Time}: O(m \cdot m \cdot \log U)$$

This is close to the running time of Max-Bottleneck, only better by a log factor. But this algorithm

illustrates a general scaling approach that can be applicable to other problems:

- We begin by solving a problem very "coarsely";
- Subsequently, we refine it iteratively;
- In each iteration, we use the results from the previous one.

3 Shortest Augmenting Path Algorithm

In this section, we look at a strongly polynomial time algorithm which depends only on m and n. It does not depend on U, the maximum capacity of the graph. Therefore, the time complexity of this algorithm is $(m \cdot n)^{O(1)}$,

The advantage for using such algorithm is when U is very large as is the case when the capacities are real numbers. In this case instead of working with integers, we works with the real word model which assumes the following:

- numbers/registers/words are real numbers
- reasonable operations on these registers are possible in constant time such as min, max, addition, subtraction and multiplication

This class of algorithms are often known as combinatorial algorithms as it does not take into account the capacities and instead runs on the structure of the graph itself.

3.1 The Algorithm

- Run the Ford-Fulkerson algorithm except instead of finding the augmenting path with the maximum flow, find the shortest augmenting path at each step of the algorithm
- The length of the augmenting path is defined as the number of hops that are required to get from s to t in a given path

Correctness: Follows immediately from the correctness of Ford-Fulkerson algorithm because here we are essentially replacing any positive capacities with 1.

3.2 Runtime Analysis

Definition 3. *Let $d_{G_f}(s, v)$ be the minimum distance from s to v in the graph G_f where G_f is the residual graph for the flow f*

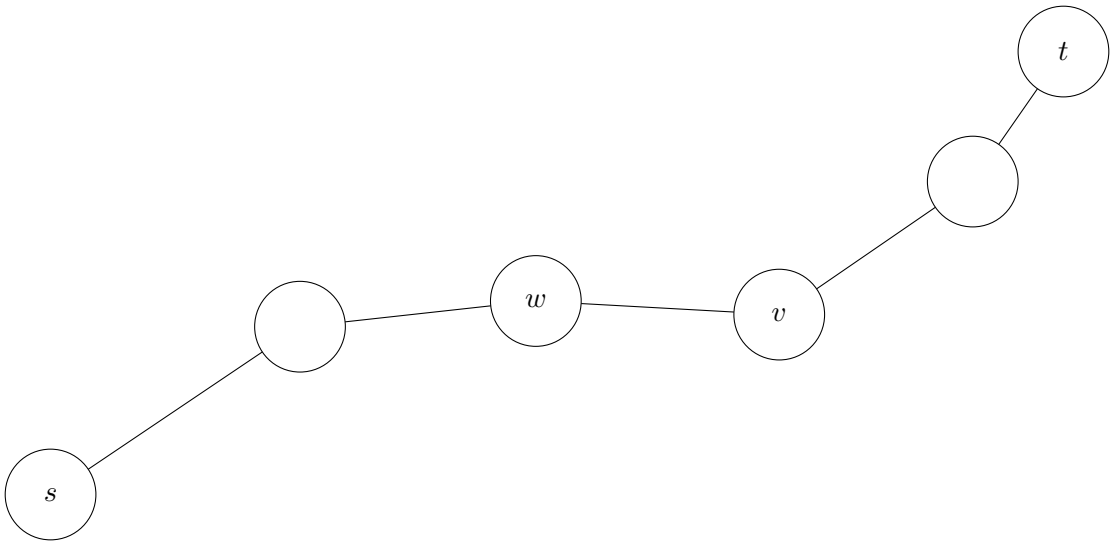
Claim 4. *Fix a flow f. Let P be the shortest augmenting path in G_f. Let f' be the flow after augmenting P. Then*

$$d_{G_f}(s, v) \leq d_{G_{f'}}(s, v)$$

Proof. The proof is by contradiction. We use the shorthands d' for $d_{G_{f'}}$ and d for d_{G_f}. Suppose,

$$A = \{v : d'(s, v) < d(s, v)\} \neq \emptyset$$

In that case, there must exist a minimizer $v \in A$ of $d'(s, v)$. Consider the shortest path from s to v in $G_{f'}$ so that $len(P') = d'(s, v)$.



Suppose w is vertex immediately preceding v in P'. So, it immediately follows that

$$d'(s, v) = d'(s, w) + 1$$

By minimality of $v \in A$, $w \notin A$. Therefore

$$d'(s, w) \geq d(s, w)$$

$$\implies d(s, v) > d'(s, v) = d'(s, w) + 1 \geq d(s, w) + 1$$

However, this implies that there exists no edge from w to v in G_f as if there was it would mean

$$d(s, v) \leq d(s, w) + 1$$

which clearly is not true from the previous inequality. Thus, (v, w) must have been in the shortest augmenting path P. Thus,

$$d(s, w) = d(s, v) + 1 > d'(s, v) + 1 = d'(s, w) + 2 \geq d(s, w) + 2$$

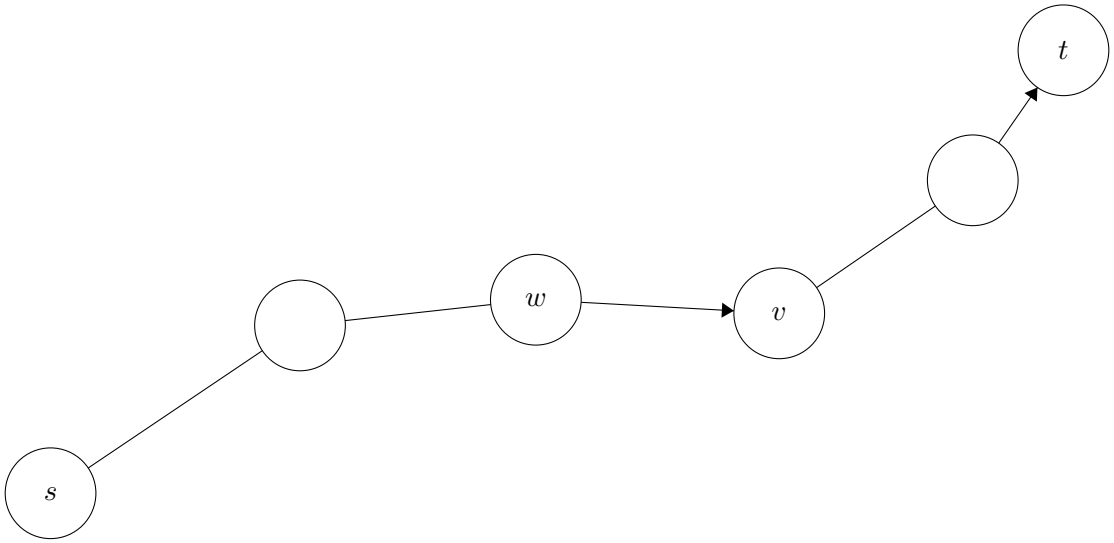
which is clearly a contradiction. □

Observation 5. $\forall v, d_{G_f}(s, v) \leq n$

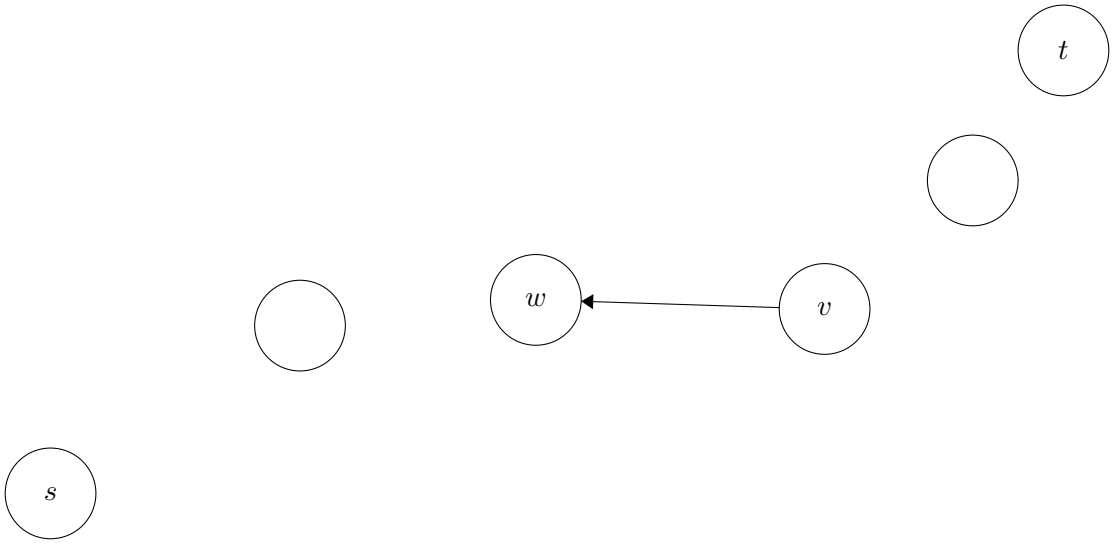
Claim 6. *For all edge (v, w) it is saturated at most $\frac{n}{2}$ times.*

Proof. Let P be the that saturates $v \to w$. In which case:

$$d(s, w) = d(s, v) + 1$$



after augmenting P, the vertex $v \to w$ no longer exists, but the vertex $w \to w$ does.



Next time we saturate the vertex (v, w) we do so in the direction $w \to v$, as such:

$$d'(s, v) = d'(s, w) + 1 \geq d(s, w) + 1 = d(s, v) + 2$$

due to the non-decreasing feature of distances with each augmentation. Therefore, each time this vertex is augmented, the distance to v increases by 2. Obviously, the distance cannot exceed n, therefore, the vertex can be only augmented at most $\frac{n}{2}$ times. □

Therefore,

$$\textit{Number of paths augmented} \leq \textit{Number of edges} \cdot \textit{Max number of times each can be augmented} = \frac{m \cdot n}{2}$$

Since the running time for finding the shortest augmented path at each iteration is $O(m)$ due to breadth-first search and the number of iteration is bounded by $\frac{mn}{2}$, the total time of the algorithm is $O(m^2 n)$

3.3 Some Closing Remarks

There are faster max-flow algorithms than the ones we covered:

- The fastest algorithm (from the 1990's) is combinatorial and runs in time $O(m^{1.5} \log n \log U)$. The main idea it employs is to "do more work" per iteration, which can be illustrated, in the context of Shortest Augmenting Path, roughly as follows:
  - Originally, we took an augmenting path and we incremented along it;
  - But, to find that augmenting path, we explored a large portion of the graph, thus finding many more shortest paths;
  - Maybe we can augment along all those shortest paths at the same time.
- Allowing for restrictions on capacity, there are more recent algorithms which utilize both combinatorics and continuous optimization. These algorithms achieve time of $\tilde{O}(m^{\frac{10}{7}})$ (Mądry 2011), when $U = O(1)$ (capacities are constant), and $\tilde{O}(m\sqrt{n} \log U)$ (Lee, Sidford 2012).

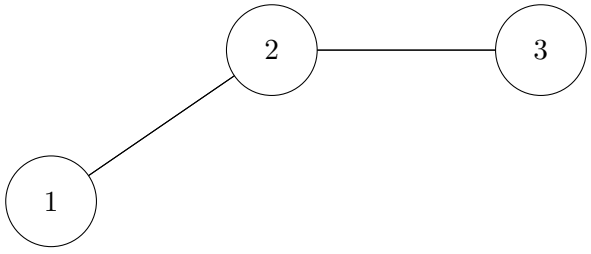
4 Spectral Graph Theory

Definition 7. *Let G be an undirected unweighted graph. A_G is then called the adjacency matrix of G where*

$$(A_G)_{ij} = \begin{cases} 1 & \textit{if there exists edge } (i,j) \\ 0 & \textit{otherwise} \end{cases}$$

Observation 8. *A_G is a symmetric matrix with 0 diagonal*

Example 9. *The following graph G*



has the adjacency matrix

$$A_G = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Definition 10. *D_G is a diagonal matrix if*

$$(D_G)_{ij} = \begin{cases} \textit{degree of } i & i = j \\ 0 & \textit{otherwise} \end{cases}$$

Example 11. *The following is an example of a diagonal matrix*

$$D_G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Motivations for Spectral Graph Theory:

- Spectral analysis on adjacency matrices of graphs to find theoretical properties of such matrices
- Applications such as random walks and diffusion on graphs