Lecture Notes on Graphs, Optimization, and Approximation

Farhad Shahrokhi

1 Preface

These notes grew out of a graph theory course (CSCI5370) that I taught at UNT about 3 years ago. Since then, they have been updated and revised, and were used to teach many graduate and seminar classes such as approximation algorithms (CSCI6330), combinatorial optimization(CSCI6330), graph theory (CSCI5370), and design of algorithms (CSCI5450).

The current version combines and includes many topics from graph theory, design of VLSI, combinatorial optimization, and approximation algorithms. These notes can be used to teach advanced graduate classes in these areas. They can also be used to teach the applications of network flows in a first graduate course on the design of algorithms.

The notes are written for a student with a mature mathematical background, and contain many very recent research results. It is recommended that the instructor also uses a book, or a collection of research papers, in addition to using these notes,

I would like to thank many of my students who helped me in preparing these notes. I especially thank R. Chalapalli, C. Chao, Z. Tsao, S. Sambi, O. Yildiz, and O. Koyunko.

2 Network flow

Let $G = (V, E)$ be directed graph, with $|V| = n, |E| = m$. Let $C : E \to R^+$, where C denotes capacity function and R^+ denotes non-negative reals. For s and t being 2 distinct vertices in V, an st flow in G is a function $f : E \to R^+$ so that

- For any edge $ab \in E$, $f(a, b) \leq C(a, b)$.
- For any $x \in V - \{s, t\}$, $\sum_{xa \in E} f(x, a) = \sum_{ax \in E} f(a, x)$.

For any flow f, let $val(f)$ be the net flow out of source which is

$$\sum_{sa \in E} f(s, a) - \sum_{as \in E} f(a, s).$$

A maximum flow is one of the largest value.
Ford and Fulkerson's algorithm was the first algorithm for computing a Maxflow and has time complexity $O(mv^*)$ time where v^* denotes value of Maxflow. This algorithm is not suitable (in theory) in the presence of a large capacity. There are many efficient algorithms for computing a maximum flow. Dinic's algorithm has time complexity $O(n^3)$, for any n vertex graph.

Let $G = (V,E)$ and let $(A, \overline{A})$ be partition of V, so that $s \in A$ and $t \in \overline{A}$; $(A, \overline{A})$ is called an st cut. Observe that for any flow f and any st cut $(A, \overline{A})$ val$(f) \leq \sum_{\substack{ab \in E \\ a \in A, b \in \overline{A}}} C(a, b)$. Let

$$C(A, \overline{A}) = \sum_{\substack{ab \in E \\ a \in A, b \in \overline{A}}} C(a, b).$$

Let f be a flow from s to t in G. An augmenting path p is a sequence of vertices

$$a_1 a_2 a_3 ..... a_k,$$

so that

- $s = a_1$ and $t = a_k$.
- either $a_i a_{i+1} \in E$, or $a_{i+1} a_i \in E$, $1 \leq i \leq k - 1$.
- If $a_i a_{i+1} \in E$, then $f(a_i, a_{i+1}) < C(a_i, a_{i+1})$, $1 \leq i \leq k - 1$.
- If $a_{i+1} a_i \in E$, then $f(a_{i+1}, a_i) > 0$, $1 \leq i \leq k - 1$.

Let p be an augmenting path from s to t. If $e = a_i a_{i+1} \in E$, then e is called a forward edge in p. Similarly, if $e = a_{i+1} a_i \in E$ then e is called a backward edge on p. For any forward edge $e = a_i a_{i+1}$ define $\delta_e = C(a_i, a_{i+1}) - f(a_i, a_{i+1})$. Similarly, for any backward edge $e = a_{i+1} a_i$ on p, define $\delta_e = f_(a_{i+1}, a_i)$. Note that $\delta_e > 0$, regardless of e being forward or backward.
Define

$$\delta_p = min_{e \in p} \delta_e.$$

Let p be an augmenting flow from s to t and consider a new flow f_p constructed as follows:

- $f_p(a, b) = f(a, b)$, for any $ab \in E$ so that ab is not an edge on p.
- $f_p(a, b) = f(a, b) + \delta_p$, for any $ab \in E$ so that ab is a forward edge in p.
- $f_p(a, b) = f(a, b) - \delta_p$, for any $ab \in E$ so that ab is a backward edge in p.

Observe that

$$val(f_p) = val(f) + \delta_p.$$

f_p is called the augmented flow; the process of constructing f_p is called flow augmentation.
Ford and Fulkerson's algorithm is the following:

- $f \leftarrow 0$
- Search for an augmenting path from s to t, if non exists stop, other wise go to step 3.
- Let p be an augmenting path; compute the augmented flow f_p, and update f to be $f + \delta_p$, and go to step 2.

Theorem 2.1 *Ford and Fulkerson's algorithm terminates and computes a maximum flow, provided that the capacities are integers.*

Proof Each flow augmentation must increase the value of the flow by at least 1. Thus, the algorithm must terminate. Consider the algorithm when it terminates. Let A be the set of all vertices which can be reached from s by an augmenting path. We include s in A. Note that $t \in \bar{A}$. We study the cut $(A, \bar{A})$. For any edge $ab \in E, a \in A, b \in \bar{A})$, it holds $f(a, b) = C(a, b)$. Similarly, for any for any $ab \in E, a \in bar A, b \in A$, we must have $f(a, b) = 0$. Consequently,

$$val(f) = C(A, \bar{A})$$

which implies f is a maximum flow and $(A, \bar{A})$ is a minimum cut. $\Box$

The above theorem gives rise an important result, namely, the Max-flow Min-cut theorem: The value of Max flow from s to t is equals to the capacity of a minimum st cut. The the theorem and the algorithm also give rise to integrality result: The value of flow on any edge is an integer. Time complexity of FF Algorithm depends on the number of iterations of, or number of flow augmentations. However, the time complexity also depends on on how we compute the augmenting paths. We define the *residual graph* $\hat{G} = (V, \hat{E})$ and residual capacity $\hat{c}$ as follows. For any $ab \in E$, let $\hat{c}(a,b) = c(a,b) - f(a,b)$ and $\hat{c}(b,a) = f(a,b)$. Now for any $ab \in E$, place $ab \in \hat{E}$, if $\hat{c}(a,b) > 0$; Similarly, place ba in $\hat{E}$, if $f(b,a) > 0$. The following Theorem is easy to prove.

Theorem 2.2 *There is an augmenting $st-$path in G if and only if there is an $st-$path in $\hat{G}$. Consequently, we can find an augmenting $st-$path in $O(m+n)$, if one exists*

We can now state the following.

Theorem 2.3 *Time Complexity of FF Algorithm is $O(Val(\hat{f})(m+n))$, where $\hat{f}$ is a maximum flow.*

Proof. Since at each iteration the value of flow is in creased by 1, the number of iterations is at most $Val(\hat{f})$. To finish the proof, we use the previous theorem. $\Box$.

2.1 Computing Edge Connectivity in a Directed Graph

The edge-connectivity of a (directed or undirected) graph denoted by $k(G)$, is the minimum number of edges whose removal from G results in a disconnected or trivial graph. The vertex-connectivity of a (directed or undirected) graph G, denoted by $k(G)$, is the minimum number of vertices whose removal from G results in a disconnected or trivial graph.

The st edge connectivity of a directed graph, denoted by $k_{st}(G)$, is the minimum number of edges whose removal makes s disconnected from t. Thus $k_{st}(G)$ is the capacity of minimum st cut, if we set capacities to be 1. Thus, computing st edge connectivity for a directed graph is equivalent to one Max flow with 0-1 capacity, and can be done in $O(nm)$ time. A Naive algorithm for edge connectivity of digraphs uses $O(n^2)$ Maxflow problems. A better algorithm that solves only $2(n-1)$ $0-1$ network flow problems is given next.

Better Than Naive Algorithm for Edge Connectivity

- Fix $s \in V$.
- For any $t \in V - \{s\}$ compute an st, and a ts minimum cut and then select the best of the 2 cuts.
- Select the best cut among $2(n-1)$ cuts calculated above.

2.2 Computing vertex connectivity in a digraph

The st vertex-connectivity of a directed graph G, denoted by $k^1_{s,t}(G)$, is the minimum number of vertices whose removal from G disconnects s from t. We transform this problem to st edge connectivity, in a new graph $G^* = (V^*, E^*)$ that has the vertex set $V^* = \{x_1, x_2 | x \in V\}$. There is a directed edge $x_1x_2 \in E^*$ for any $x \in V$ (of capacity 1). Also, for any $xy \in E$ there is an edge x_2y_1 of capacity ∞ in E^*.

Definition 2.1 *An st* **disconnecting vertex set** *is a set of vertices whose removal makes s and t disconnected. A disconnecting edge set is defined similarly.*

Observation 2.1 *sl Let A be a disconnecting vertex set in G then $E_A = \{a_1a_2 | a \in A\}$ is a disconnecting edge set between s_2 and t_1 in G^*. Conversely: Let $A \subset V$. If $E_A = \{a_1a_2 | a_1, a_2 \in A\}$, is a disconnecting edge set between s_2 and t_1 in G^*, then A disconnects s from t in G.*

- It follows from observation that a minimum cut which disconnects s_2 from t_1 in G^* is a minimum vertex set disconnecting s from t in G, and thus computing $K^1_{st}(G)$ is reduced to solving one max. flow problem in G^*.
- Computing vertex connectivity, or $k^1(G)$, of a directed graph G is the same as computing the edge connectivity of G^*, or $k(G^*)$, and can be done using Algorithm 1. Note that we will need to solve $O(n)$ many max flow problems.

2.3 Computing vertex and edge connectivity of undirected graphs

Let $G = (V, E)$ be a graph with $s, t \in V$, $s \neq t$. An st cut is a partition $(A, \overline{A})$ of V, so that $s \in A, t \in \overline{A}$.

Capacity of $(A, \overline{A})$ is denoted by

$$C(A, \overline{A}) = \sum_{\substack{e=ab \in E \\ a \in A \\ b \in \overline{A}}} C(e).$$

Let $|(A, \overline{A})|$ denote the number of edges in $(A, \overline{A})$. Now, follow the transformation that is given below:

- Replace any $ab \in E$ with a pair of directed edges ab ba , to obtain a digraph in $\hat{G}$.
- Define $Cap(a, b) = Cap(b, a) = C(a, b)$

Theorem 2.4 *Let $(A, \overline{A})$ be a minimum st cut in $\hat{G}$, then $(A, \overline{A})$ is a minimum st cut in G.*

Proof: To prove the claim observe that the capacity of any st cut in G equals to the capacity of the same cut in $\hat{G}$. □

• Using the theorem, computing a minimum st edge disconnecting set (a minimum st cut, or st edge connectivity) reduces to solving one max flow problem in $\hat{G}$. Therefore computing edge connectivity of a graph G, or $k(G)$ reduces to solving $(n-1)$ max- flow problems, using Algorithm 1. The next theorem tell us how to compute vertex connectivity of a graph.

Theorem 2.5 *sl Let A be a minimum disconnecting vertex st set in $\hat{G}$, then A is a minimum disconnecting vertex set in G.* □

• Observe that, using the theorem computation of $k^1_{st}(G)$ is reduced to computing $k^1_{st}(\hat{G})$ which requires solving 1 maximum flow problem.

• Also, observe that computing the vertex connectivity of G, or $k^1(G)$, reduces to computing the vertex connectivity of $\hat{G}$, and can be done by solving $O(n)$ max-flow problems.

2.4 Edge-Path Form for Flows

Next, we study a new formulation of flows. Let $G = (V, E)$ be a directed graph with $s, t \in V$, $s \neq t$. For any $ab \in E$, let P_{ab} be the set of all paths from s to t that contain ab Let $C : E \to R^+$ be the usual capacity function on E.

A Flow (IN EDGE PATH FORM) is a function $f : P \to R^+$, where P is the set of all paths from s to t, so that $\sum_{p \in P_{ab}} f(p) \leq C(a, b)$, $\forall ab \in E$. The value of f denoted by $val(f)$ and is defined to be $\sum_{p \in P} f(p)$.

Observation 2.2 *Let f be an edge path (EP) flow. Then there is flow $\hat{f}$ so that*

$$val(\hat{f}) = val(f).$$

□

Proof: Define $\hat{f}(a,b) = \sum_{p \in P_{ab}} f(P), \forall ab \in E$, then $\hat{f}$ is a flow so that $val(f) = val(\hat{f})$.

Theorem 2.6 *Let f be an st flow in G. Then there is an Edge path flow from s to t, so that $val(f) \geq val(\hat{f})$.*

Proof: In order to prove this, we decompose the flow on st paths. Take a path and find the least amount of flow on an edge in it and subtract that flow from all edges in that path. Delete any edge with zero flow. Continue doing this, until s and t are disconnected.

Let v^*=$val(f)$. Formally we repeat the following 3 steps, until s and t are disconnected:

- Perform a bfs from s and find a directed path p from s to t.
- Define $\hat{f}(p) = min_{ab \in p} f(a,b)$.
- Update the value of f on any $xy \in p$ to be $f(x,y) - f(a,b)$ and remove xy if the flow on xy is 0.

Observe that, at any iteration

$$val(\hat{f}) + val(f) \geq v^*$$

so, after at most m (number of edges) iterations

$$val(\hat{f}) + 0 \geq v^*.$$

□

Note that we have proved the following:

Theorem 2.7 *Let $G = (V, E)$ be a digraph with $s, t \in V$, then the value of st EP(Edge Path) Maxflow equals to the value of st flow.*

Observation 2.3 *If all capacities are one, then each path (in the EP flow model) hosts one unit of flow, and therefore these paths are edge disjoint. It follows that in the EP form, the number of paths hosting flows is the maximum number of edge disjoint st paths. This gives a simple proof to the following well known result.*

Theorem 2.8 (Menger's Theorem-(Directed graph) Edge version) *Maximum number edge disjoint st paths equals to the minimum number of edges where removal discontinues t from s.*

Proof: Note by Maxflow Min-cut theorem, the minimum number of edges whose removal discontinues t from s equals to the value of maximum st flow. By previous observation, the value of a maximum st flow equals to the maximum number of edge disjoint paths. □

Theorem 2.9 (Menger's Theorem-(Undirected graph) Edge version)

Proof: Replace any edge in G by a pair of back to back edges. Use the previous result, which is the directed graph edge version. □

Theorem 2.10 Menger's Theorem-(Directed graph) Vertex version) *The maximum number of vertex disjoint path from s to t equals to the minimum number of vertices whose removal discontinues t from s.*

Proof: Apply the standard transformation. That is, stretch vertices to construct $\hat{G}$. Now observe that in $\hat{G}$, the maximum number of edge disjoint paths equals to the minimum capacity of edges which removal discontinues t from s. To complete the proof, observe that maximum number of edge disjoint st paths in $\hat{G}$ equals to the maximum number of vertex disjoint st path in G. Thus the minimum capacity cut in $\hat{G}$ equals to the minimum number of vertices whose removal disjoin t from s in G. □

Theorem 2.11 (Menger's Theorem-(Undirected graph) Vertex version) *The maximum number of vertex disjoint paths from s to t equals to the minimum number of vertices whose removals disconnect t from s.*

Proof: Replace any edge in G by a pair of back to back edges. Use the previous result, which is the directed vertex disjoint version.