Advanced Algorithms: Notes

Author: Martin Grünbaum (martin@itsolveonline.net)

June 8, 2014

Contents

1 Max-flow: Disposition

1. Flow network G
2. Flow $|f|$ (Capacity constraint, flow conservation)
3. Residual network, augmenting paths, cuts.
4. Max-flow min-cut theorem
5. Ford-Fulkerson method
6. Edmonds-Karp

2 Max-flow: Notes

A flow network $G = (V, E)$ is a directed graph where each edge $(u, v) \in E$ has a non-negative capacity $c(u, v) \geq 0$. If there is an edge $(u, v) \in E$ then there is no edge $(v, u) \in E$. If $(u, v) \notin E$ then $c(u, v) = 0$ for convenience. When $(u, v) \notin E$, $f(u, v) = 0$.

Flow networks have a source s and a sink t. For each vertex $v \in V$, the flow network contains a path $s \rightsquigarrow v \rightsquigarrow t$. The graph is therefore connected, meaning $|E| \geq |V| - 1$.

A flow is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies two properties:

Capacity constraint: For all $u, v \in V$, $0 \leq f(u, v) \leq c(u, v)$

Flow conservation: For all $u \in V - \{s, t\}$, $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$.

The value of a flow, $|f|$, is defined as:

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

In the **maximum-flow** problem, we are given a flow network G and we wish to find a maximum flow.

Edges are anti-parallel if there is both an edge (u, v) and an edge (v, u). This is not allowed, and to get around this we instead introduce a new edge x and re-structure the edges as follows: $(u, x), (x, v), (v, u)$. The capacity of the new edges involving x is the same as the capacity from (u, v). See page 711 in the book for an example.

2.1 Multiple sources and sinks

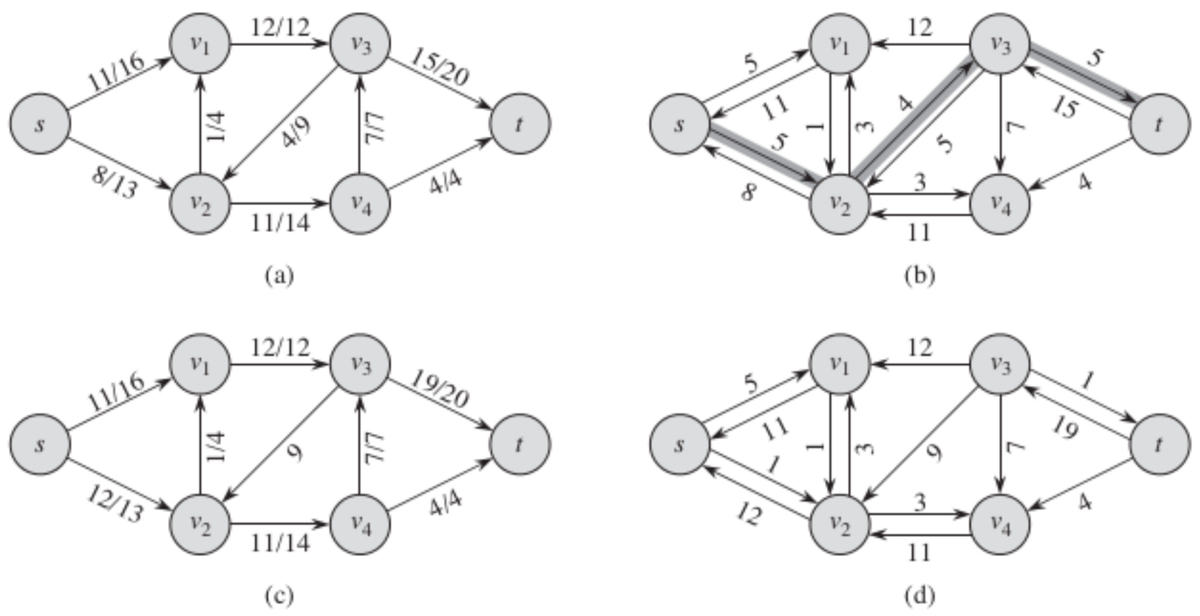
This can be accounted for by introducing a **supersink** and **supersource** with infinite flow and capacity out to all of the sources and from all of the sinks to the supersink. See page 713.

2.2 Ford-Fulkerson method

Three basic principles: **residual networks**, **augmenting paths** and **cuts**. Essential for **max-flow min-cut** theorem (Theorem 26.6).

Intuition is as follows: We have a flow network G. We iteratively alter the flow of G, by finding an augmenting path in an associated residual network G_f. Once we know the edges that belong to an augmenting path, we can identify specific edges in G to increase or decrease the flow of. Each iteration increases overall flow, but it may do so by decreasing the flow along certain edges. This is repeated until the residual network G_f has no more augmenting paths.

max-flow min-cut shows that upon termination, this yields a maximum flow.



2.2.1 Residual network

Given a network $G = (V, E)$ with a flow f, the **residual network** of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}.$$

Residual capacity $c_f(u, v)$ is defined by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E, \\ 0 & \text{otherwise} \end{cases}$$

Note: that $(u, v) \in E$ implies $(v, u) \notin E$, so there is always only one of the three above cases that applies.

Because the edges in E_f are either edges from E or an edge in the opposite direction, $|E_f| \leq 2|E|$.

Intuition: A residual network G_f consists of edges with capacities that represent how we can alter the flow on edges of G. G can admit an additional amount of flow along an edge, equal to the capacity minus the current flow. If the edge can admit more flow, that edge is placed into G_f with a value of $c_f(u, v) = c(u, v) - f(u, v)$. The residual network may also contain edges that are not in G: In order to represent a possible decrease of a flow $f(u, v)$ on an edge in G, we place an edge (v, u) into G_f with residual capacity $c_f(v, u) = f(u, v)$. In other words, an edge that can admit flow in the opposite direction, at most cancelling out flow entirely. See Figure **??** for an example.

Flows in a residual network satisfy the definition of a flow, but with respect to capacities c_f in the network G_f. If f is a flow in G and f' is a flow in the corresponding residual network G_f, we define $f \uparrow f'$, the **augmentation flow** of f by f', as a function from $V \times V$ to $\mathbb{R}$ defined by

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Intuition: Increase the flow $(f(u, v))$ by $f'(u, v)$, but decrease it by the flow in the opposite direction $(f'(v, u))$. Pushing flow in the reverse direction is also called **cancellation**.

2.2.2 Augmenting path

An augmenting path p is a simple path from s to t in the residual network G_f. By the definition of a residual network, we may increase the flow of an edge (u, v) by up to $c_f(u, v)$ without violating the capacity constraint on whichever of (u, v) and (v, u) is in the original flow network G.

The maximum amount by which we can increase flow on each edge of an augmenting path p is the **residual capacity** of p, given by $c_f(p) = min\{c_f(u, v) : (u, v) \text{ is on p}\}$. More specifically, if p is an augmenting path in G_f, we define a function $f_p : V \times V \to \mathbb{R}$ as

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{is on } p, \\ 0 & \text{otherwise.} \end{cases}$$

Then f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$. See Lemma 26.2, page 720. It remains to be shown that augmenting f by f_p produces a different flow in G whose value is closer to the maximum. Corollary 26.3 on page 720 shows this by immediate proof, using Lemma 26.1 and 26.2.

2.2.3 Cuts of a network

We know, based on the above, that we can augment flows in G and that doing so can produce a new flow closer to the maximum. But how do we know that when it terminates, the algorithm has in fact found a maximum flow? Max-flow min- cut tells us that a flow is maximum only if its residual network contains no augmenting paths.

A **cut** (S, T) of a flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow then the **net flow** $f(S, T)$ across the cut (S, T) is defined to be

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

The **capacity** of the cut (S, T) is

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

Intuitively, the capacity of the cut is the capacity of all vertices going from S to T, while the flow is the flow of vertices going from S to T, minus the flow going from T to S. A **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network.

Theorem 26.6 (Max-flow min-cut theorem, p. 723/724) involves proving the equivalence of 3 different conditions:

1. f is a maximum flow of G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

1 => 2: Assume that f were a maximum flow in G and there **was** an augmenting path. This means, by the proof of augmenting paths, that we could create a new flow f' in G with a strictly larger flow value than f, i.e. that $|f'| > |f|$. This contradicts f being a maximum flow.

2 => 3: Suppose that there are no augmenting paths, that is there is no path from s to t in G_f.

Define $S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$. That is, the set S contains all those vertices for which there could be pushed more flow along, but which perhaps have not because a later capacity limits that possibility. Define $T = V - S$. A partition (S, T) is a cut, where $s \in S$ and $t \notin S$ (since there is no path from s to t, or we would not have a maximum flow).

Consider two vertices (u, v) where $u \in S$ and $v \in T$:

If $(u, v) \in E$, we must have that $f(u, v) = c(u, v)$. If this were not the case we would have $(u, v) \in E_f$, since we would be able to push more flow out until at capacity. Then, by the definition of S we would have that $v \in S$. This is a contradiction.

If $(v, u) \in E$, we must have that $f(v, u) = 0$. If this were not the case we would have $(v, u) \in E_f$, since the residual capacity $c_f(u, v) = f(v, u)$ would be positive. This means $(u, v) \in E_f$, and we would have that $v \in S$. This is a contradiction.

Therefore:

$$\begin{aligned} f(S,T) =& \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \\ =& \sum_{u\in S}\sum_{v\in T} c(u,v) - \sum_{u\in S}\sum_{v\in T} 0 \\ =& c(S,T) \end{aligned}$$

3 => 1: The value of **any** flow f in a flow network G is bounded from above by the capacity of **any** cut of G. Proof:

$$\begin{aligned} |f| =& f(S,T) \\ =& \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \\ \leq& \sum_{u\in S}\sum_{v\in T} f(u,v) \\ \leq& \sum_{u\in S}\sum_{v\in T} c(u,v) \\ =& c(S,T) \end{aligned}$$

Because of this, $|f| \leq c(S, T)$ for **all** cuts (S, T). Therefore, $|f| = c(S, T)$ implies that f is a maximum flow.

Algorithm 1 Ford-Fulkerson Method

```
1: procedure FORD-FULKERSON METHOD(G,s,t)
2:     Initialize flow $f$ to 0
3:     while there exists an augmenting path $p$ in residual network $G_f$ do
4:         Augment flow $f$ along $p$
5:     end while
6: end procedure
```

Running time of the simple algorithm depends on how we find the augmenting path p. If we assume an appropriate data structure where we can represent the directed graph, the time to find an appropriate path can be linear in the number of edges, if using breadth-first or depth-first search.

This gives us $O(E)$ work per iteration of the while loop, and at most the same number of iterations as the value of the maximum flow (since we increase by at least one unit per iteration). The total running time is therefore $O(|f*|E)$, where $|f*|$ is the maximum flow.

2.3 Edmonds-Karp

Edmonds-Karp works by finding the shortest augmenting path each time. We choose the augmenting path as a shortest path from s to t in the residual network, where each edge has unit distance (weight). Edmonds-Karp runs in $O(VE^2)$ time.

Algorithm 2 Ford-Fulkerson Basic Algorithm

```
procedure FordFulkersonSimple(G,s,t)
    for each edge $(u, v) \in G.E$ do
        $(u, v).f = 0$
    end for
    while there exists a path $p$ from $s$ to $t$ in residual network $G_f$ do
        $c_f(p) = \min\{c_f(u, v) : (u, v)\text{is in } p\}$
        for each edge $(u, v)$ in $p$ do
            if $(u, v) \in G.E$ then
                $(u, v).f = (u, v).f + c_f(p)$
            else
                $(v, u).f = (v, u).f - c_f(p)$
            end if
        end for
    end while
end procedure
```

(Lemma 26.7) If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$ the shortest-path distance $delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

(Theorem 26.8) If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE^2)$

Each iteration of Edmonds-Karp, at least one edge along the augmenting path is a critical edge (an edge that bottlenecks the path, i.e. an edge that becomes saturated). The main idea behind the theorem is to show that each edge can become critical at most $|V|/2$ times. And since there are $O(E)$ pairs of vertices that can have an edge between them in a residual network, the total number of critical edges during the entire execution of Edmonds-Karp is $O(VE)$.

Using breadth-first search to find an augmenting path in $O(E)$ time, this gives a total running time of $O(VE^2)$.

3 Fibonacci heaps: Disposition

1. Mergeable heaps
2. Structure
3. Operations
    - Make-Heap
    - Insert
    - ExtractMin
    - Union / Merge
    - DecreaseKey
    - Delete

4 Fibonacci heaps: Notes

4.1 Mergeable heaps

A mergeable heap is one which supports the following operations:

Make-Heap() Creates and returns a new heap containing no elements.

Insert(H,x) Inserts element x, whose key has already been filled in, into heap H.

Minimum(H) Returns a pointer to the element in heap H whose key is minimum.

Extract-Min(H) Deletes the element from heap H whose key is minimum, returning a pointer to the element.

Union(H_1,H_2) Creates and returns a new heap that contains all the elements of heaps H_1 and H_2. The original heaps are destroyed by this operation.

In addition to the above operations, Fibonacci heaps also support the following two operations:

Decrease-Key(H,x,k) Assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.

Delete(H,x) Deletes element x from heap H.

Table 1: Running time of operations

Procedure	Binary heap(worst case)	Fibonacci heap (amortized)
Make-heap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(lg\ n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
Extract-Min	$\Theta(lg\ n)$	$\mathbb{O}(lg\ n)$
Union	$\Theta(n)$	$\Theta(1)$
Decrease-Key	$\Theta(lg\ n)$	$\Theta(1)$
Delete	$\Theta(lg\ n)$	$\mathbb{O}(lg\ n)$

Fibonacci heaps are theoretically desirable when the number of Extract-Min and Delete operations is small relative to the number of other operations performed. This can be the case for some algorithms in graph problems, that call decrease-key once per edge. Fast algorithms for computing minimum spanning trees and finding single-source shortest paths make essential use of Fibonacci heaps.

From a practical point of view, however, the constant factors and complexity of Fibonacci heaps make them less desirable than ordinary binary (or k-ary) heaps for most applications. As such, Fibonacci heaps are pre-dominantly of theoretical interest.

4.2 Structure of Fibonacci heaps

A Fibonacci heap is a collection of rooted trees that are min-heap ordered. Each tree obeys the min-heap property: the key of a node is greater than or equal to the key of its parent. Each node contains a pointer to its parent and a pointer to any one of its children. The children are linked together in a circular, doubly linked list. In a circular, doubly linked list we can insert a node into any location or remove a node from anywhere in the list in $O(1)$ time. We can also concatenate two such lists in $O(1)$ time.

Each node also contains the number of children (**x.degree**) and a boolean-valued attribute x.mark, to indicate whether it has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node becomes unmarked whenever it is made the child of another node. Root nodes are unmarked.

We access a Fibonacci heap H by a pointer $H.min$ to the root of a tree containing the minimum key; we call this node the minimum node of the Fibonacci heap.

The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap. Finally, $H.n$ refers to the number of nodes currently in the heap.

In the following, the potential function of $\Phi(H) = t(H) + 2m(H)$ is used, where $t(H)$ is the number of trees and $m(H)$ is the number of marked nodes. The potential function captures the state of a system at a given point, and we can then calculate an amortized time as the actual time plus a constant times the **change** in state 'energy'. This provides a valid upper bound on running time.

4.3 Analysis: Running time of methods

4.3.1 Make-Heap

Creates a fibonacci heap with no trees. This gives a potential energy of $\Phi(H) = 0$, since $t(H) = 0$ and $m(H) = 0$. The total running time is thus the actual cost of $O(1)$.

4.3.2 Insert

Insert simply inserts a node as a root-tree in the Fibonacci heap, increasing the number of trees by one. To determine the amortized cost, let H be the input Fibonacci Heap and H' be the resulting Fibonacci heap. Then, $t(H') = t(H) + 1$ and $m(H') = m(H)$. The increase in potential is thus: $((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1$.

Since the actual cost is $O(1)$, the amortized cost is $O(1) + 1 = O(1)$.

4.3.3 Find-Min

We store a pointer to the minimum node of a Fibonacci heap. As the potential is not changed (we alter no state in the tree), the amortized running time is the actual cost of $O(1)$.

4.3.4 Uniting two Fibonacci heaps

To unite two Fibonacci heaps, we concatenate the root lists of the two heaps H_1 and H_2 and then determine the new minimum node. The change in potential is:

$$\begin{aligned}
&\Phi(H) - (\Phi(H_1) + \Phi(H_2)) \\
&= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\
&= 0
\end{aligned}$$

Since $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortized time is thus equal to the actual running time of $O(1)$.

4.4 ExtractMin

ExtractMin works by first making a root of each of the minimum nodes children and removing the minimum node from the root list. Then the root list is consolidated by linking roots of equal degree, until at most one root remains of each degree. To do this efficiently, an array is created that stores a current pointer to a root tree of a given degree.

We then consolidate the root list, to reduce the number of trees in the Fibonacci heap. The following steps are repeatedly executed until every root in the root list has a distinct degree:

1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.
2. **Link** y to x: remove y from the root list, and make y a child of x. Then we increment the attribute $x.degree$ and clear the mark on y.

The array of degrees -> root pointers is used to look up whether there is another tree of the same degree as we have just increased x to, and if there is they link. If not, the entry in the array is updated to point to x.

The root list is then emptied and re-created based on the pointers in the array of degrees -> root pointers. Finally, we decrement $H.n$ by 1 and return a pointer to the deleted node.

There are several things we must estimate the cost of: extracting the minimum node, the main for-loop through the root list and within that the while-loop that links together trees.

The amortized cost of extracting the minimum node of an n-node Fibonacci heap is $O(D(n))$. Let H denote the Fibonacci heap prior to the ExtractMin operation. The actual cost of extracting the minimum node contributes $O(D(n))$ work from processing at most $D(n)$ chilren of the minimum node.

Upon calling consolidate, the size of the root list is at most $D(n) + t(H) - 1$: the original $t(H)$ root-list nodes, minus the extracted root node, plus the children of the extracted node, at most $D(n)$. Every iteration of the inner while-loop, we know that one root is linked to another, and so the total number of iterations of the while loop over all iterations of the for loop is at most the number of roots in the root list. Hence, the total amount of work performed in the for loop is at most proportional to $D(n) + t(H)$. The total actual work in extracting the minimum node is therefore $O(D(n)+t(H))$. The potential before extracting the minimum node is $t(H)+2m(H)$. The potential afterwards is at most $(D(n)+1)+2m(H)$, since at most $D(n)+1$ roots remain and no nodes get marked during this operation. The amortized cost is thus at most:

$$\begin{aligned}
&O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\
=&O(D(n)) + O(t(H)) - O(t(H)) \\
=&O(D(n))
\end{aligned}$$

if we scale up the units of the potential to dominate the constant hidden in $t(H)$. We will see later that $D(n) = O(lg\ n)$, giving an amortized running time of $O(lg\ n)$ for extracting the minimum node.

4.4.1 DecreaseKey

DecreaseKey on a node x (with parent y) works as follows: Decrease the key of x. If x.key $<$ y.key, cut x and turn it into a root-node. Then perform a cascading cut on y upwards, which will cut any node that is already marked, terminating when it reaches a root node or an unmarked node. Reaching an unmarked node will mark it. Finally, we check whether $x.key < H.min.key$, and if so update x to be the minimum node of H.

This process means that a node can lose one child (causing it to be marked), but losing another will cause it to be cut as it was already marked from losing a child before.

DecreaseKey takes $O(1)$ time, plus the time to perform the cascading cuts. Each recursive call of cascading cut also takes $O(1)$ time, giving $O(c)$ for c cascading cuts. The actual time of DecreaseKey, including recursive calls, is thus $O(c)$.

The call to cut x creates a new tree rooted at x and clears x's mark bit (which may have been false already). Each call of cascading cut, except for the last one, cuts a marked node and clears its marked bit. Afterwards, the Fibonacci heap contains $t(H)+c$ trees (the original $t(H)$ trees, $c-1$ trees produced by cascading cuts and the tree rooted at x). It contains at most $m(H)-c+2$ marked nodes ($c-1$ nodes were unmarked by cascading cuts and the last call may have marked a node). The change in potential is therefore, at most:

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))=4-c$$

This leads to an amortized cost of at most:

$$O(c)+4-c=O(1)$$

since, again, we can scale up the units of the potential to dominate the constant hidden in $O(c)$.

4.4.2 Delete

Deleting a node works by first setting its key to minus infinity, and then extracting the minimum. We already know that decreasing a key takes $O(1)$ amortized time, and extracting minimum takes $O(D(n))$ amortized time. As we will see that $D(n)=O(lg\ n)$, deleting a node runs in amortized time of $O(lg\ n)$

4.5 Bounding the maximum degree of a Fibonacci heap

Pages 523 - 526 of Lemma's and corollaries, need to be written and explained!