MF1, MF4: Ford-Fulkerson Algorithm, Edmonds-Karp Algorithm

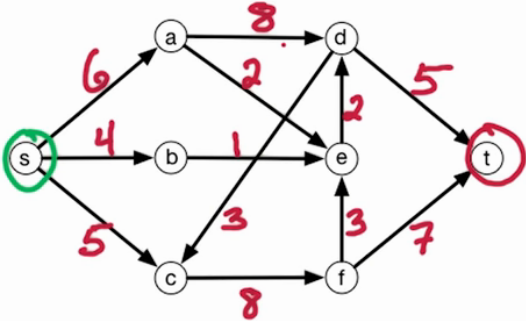
Notes for CS-8803-GA: Introduction to Graduate Algorithms

Georgia Tech (Dr. Eric Vigoda), Fall 2017

as recorded by Brent Wagenseller

**Max-flow problem**

In short, the max-flow problem figures out the maximum flow from vertex s to vertex t in a directed graph; each edge has a weight that represents the **supply** (units, can be internet traffic, gallons of oil, product, etc) capacity. We want to maximize this supply without exceeding edge capacities.



**Flow network: Max-Flow Problem**

* Inputs to a Flow network
  + directed graph G=(V, E)
  + s, t ∈ V
    - s is a source vertex and t is a sink vertex; both must exist
  + capacities ce > 0 for every edge e ∈ E
* Goal: Find flows fe for e ∈ E
  + It seems there is an fe for every edge e
  + Our goal is to maximize the flow from sto t; Our goal is to specify the flow fe along every edge so we maximize the flow from s to t without violating the constraints.
* Capacity Constraints
  + **Capacity Constraint**: For all e ∈ E 0 <= fe <= ce
    - fe is nonnegative but does not exceed its c
  + **Conservation of Flow**: For all v ∈ V – {s ∪ t}
    - The flow originates at s and ends at t, but for all *other* vertices, the flow must be conserved; that is to say, the flow in MUST equal the flow out
    - The flow into v is

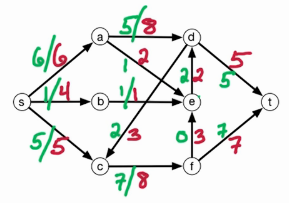


* + - The sum over edges w to v of the flow of the edge fwv
    - The flow out of v is



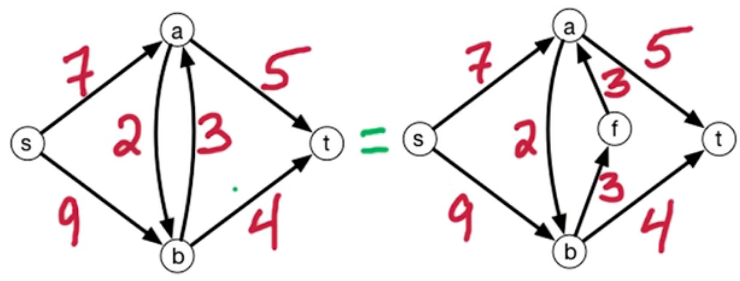
* + - The sum over edges v to z of the flow of the edge fvz
  + A valid flow satisfies capacity constraints and the conservation of flow!
    - Finding a valid flow of maximum size IS THE GOAL OF MAX-FLOW
    - The **size of flow** size(f) is the total flow sent out of s and into t (either is fine, it should be the same)

Example of Max-Flow



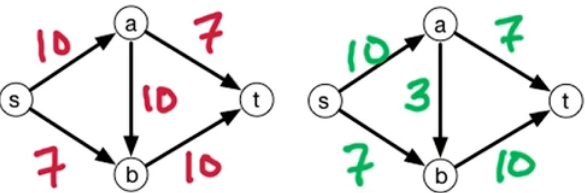
* The above is an example of a max-flow
  + The red numbers are the max possible
  + The green are figured out – THESE are the fe discussed previously
* Notice that, for the green numbers, the flow-in number equals the flow-out
* The total flow is this is
  + Size(f) = 12
    - This can be capped early on, because the sum of the two edges into t is 12
      * NOTE: It may actually be less than 12
* Note that there is a cycle – this is ok!
  + That said, the full cycle is not used
  + Part of the cycle can be used, but its difficult to tell which part will be used beforehand

Anti-Parallel Edges



* **Anti-Parallel Edges** are edges that create a continuous loop between exactly two vertices
* This is not desired; the algorithm will have trouble with this
* To break this up we will need to take one of the edges and break it in half
  + In the example above, we added vertex f between a and b
  + Note the max-flow stays the same; so the flow into and out of f is still 3, which was the flow from b to a

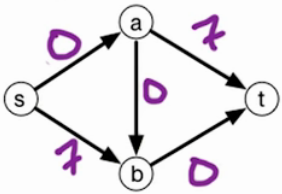
Toy Example We Will Use



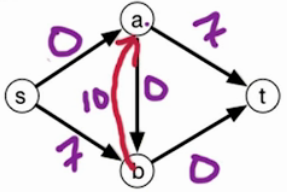
* The above (left) is the flow diagram and its max-flow components fe (right)
  + Note that fe changes while the graph is figured out
* Residual capacity is the amount of ‘space’ left on that edge that can be used by future augmented paths.
* An **Augmented path** is a path from S to T that ‘augments’ (increases usage, decreases available units) along all edges fe in the path
  + This is the maximum it can possibly be across all sub-paths; so if we picked S to A and A to T, fe for SA would be 7 (as that is the max that can go across AT)
    - Dr. Vigoda labels this process as st-path ρ and denotes this process as:



* + - Note this is basically saying ‘find the min capacity on any subpath and use that as the capacity for all subpaths for this iteration’
  + If no more augmenting paths exist, the algorithm is done!
  + The algorithm will pick a random path from S to T; in the example, Dr. Vigoda picked S to A to B to T
    - When we do this, we change fe to accommodate the new flow
    - In our example, this means the deck fe became 10 for all paths mentioned
  + Notice that we picked a path that blocked another path; here are the available capacities:



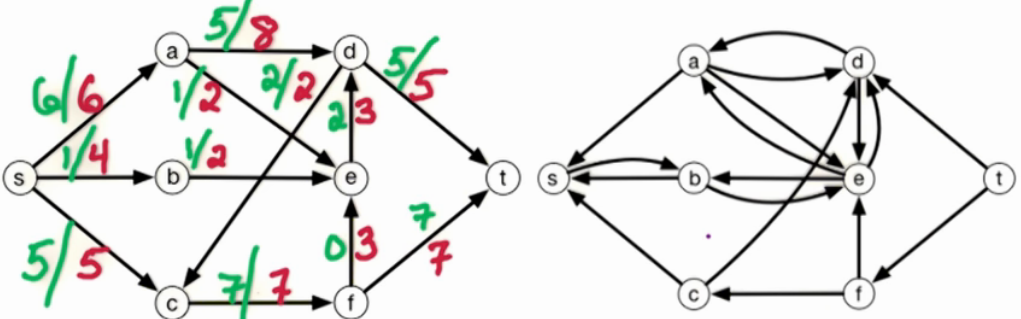
* + - If you notice, we can actually ‘push back’ on 7 units going from A to B and re-route them to edge AT
      * This can be thought of as a ‘back-edge’ that doesn’t really exist, but we can still use it anyway
      * Dr. Vigoda drew it as such:



* + - * Dr. Vigoda called this number a residual

More on Residual Networks

* A residual network captures the ‘snapshot’ of the current available edges
  + Residual networks are re-built EVERY time a new path is considered
  + It ALSO introduces back-edges which did NOT exist before; these allow future paths to ‘push back’ on flow going in the opposite direction of that edge
    - Think of it as a valve shutting off water, but we can only do it if that water has an alternate path it can take without using that edge
    - These back-edges typically equal the flow in the opposite direction; so if 10 units are on AB (that is to say, the fe of AB is 10) and its capacity is 11, when we re-build the residual network AB will have free capacity of 1, fe = 11, and the back-edge BA will have capacity 10.
* A **residual network** still use G=(V, E) with ce for e ∈ E, and flow fe for e ∈ E
  + Actually, since the residual is a function of the current flow, we will use Gf = (V, Ef)
* How Residual Edges are Determined
  + IF the original edge AB exists and the current flow is less than capacity, we will add edge AB with capacity cvw – fvw
    - These are the original edges we are adding
  + IF there is some flow on AB at all, we add edge BA, and its capacity is the current fe
    - You can actually end up adding BOTH AB and BA edges
  + Mathematically, the above two statements can be stated as:
    - if \overrightarrow{vw}\in Eand f_{vw}<c_{vw}, then add \overrightarrow{vw}to G^fwith capacity c_{vw}-f_{vw}
    - if \overrightarrow{vw}\in Eand f_{vw}>0, then add \overrightarrow{wv}to G^fwith capacity f_{vw}
* Residual paths / back edges work because, ultimately, they cannot be used for the last edge to T; this last edge MUST be a part of the original graph. This means that we can have all of the theoretical back edges we want, but if that last leg to T in the original graph does not have capacity, there is no further path to T.
* Example of the original graph on the left (with red numbers = capacity and green numbers = f) vs the residual graph:



* + Note that if an original path has no current flow at all (i.e. edge FE) then it does NOT have a back-edge
  + Note that if the flow is maxed on an original edge, it is NOT a part of the residual path
  + Since T has no incoming edges, this is a solved max-flow graph
* The residual network is a collection of original edges plus select introduced back-edges that is used to find the max-flow of the graph

**Ford-Fulkerson algorithm**

This works as before, but instead of using the graph of available capacity, we will use the residual network. The algorithm:

1. set f_e=0for all e\in E
2. build the residual network G^ffor the current for f

#initially, this is the same as the input flow network

1. check for a s-t path Pin G^f, if there is no such path, return fas the output

#for this we use either DFS or BFS

#if there is no path, output(max flow)

1. let c(P)be the minimum capacity along Pin G^f

#again, minimum here means use the capacity of the smallest subpath used; if the flow needs to travel through A of capacity 10, B of capacity 6, and C of capacity 8, overall only 6 units can flow through

1. augment falong Pby c(p):
   * for a forward edge e\in P, increases f_eby c(P)
   * for a backward edge \overrightarrow{vw}\in P, decrease f_{wv}by c(P)
2. repeat from step 2.

**Runtime**

If capacities are integers, then the flow increases by \ge 1unit each iteration. So if C= size of max-flow, it requires \le Citerations.

Each iteration involves a DFS or BFS, so takes O(|V|+|E|)=O(|E|)time (assuming Gis connected). Therefore the overall runtime is O(|E|C). This is *pseudo*-polynomial since it depends on the numbers in the input.  We want the running time to be independent of the capacities and the size of the flow.

Better algorithm: [Edmonds-Karp ’72] takes the shortest augmenting path in G^f, which results in O(|V||E|)rounds, O(|V||E|^2)total time.

**General max-Flow Points**

* In general, the ford-fulterson algorithm solves the **max-flow** problem: what is the maximum flow that can leave S and enter T?
* The output of a max-flow algorithm is flow f\* of max size
  + That is to say, the final residual of the edges going to T
    - This will ALSO be the flow OUT of S
    - In other words, Size(f) = fout (s) = fin (t)
* When did the Ford-Fulkerson algorithm stop?
  + When there was no augmenting pathin residual Gf\*
    - In other words, when the capacity is maxed on all paths from S to T in the residual graph Gf\*
    - Note: An **augmenting path** is a path from S to T that still can carry capacity from S to T
    - Also note: It appears that f\* is the ‘current’ flow
    - It takes O(V + E) time to check if f is a max flow (the time to run DFS); note this is NOT the time for the Ford-Fulkerson Algorithm, its simply for the check to see if the flow is a max flow!
* A limitation of Ford-Fulkerson is the capacity MUST be an integer; also, its slower than some alternatives out there! (Edmonds-Karp is better, see next lesson!)

**Comparison of Ford-Fulkerson vs Edmonds-Karp**

* Ford-Fulkerson
  + Finds augmenting paths using DFS or BFS
    - Again, an augmenting path is one that can potentially alter flow paths (if the flow paths are maxed there are no more augmenting paths for the graph)
  + Runs in O(mC) time
    - C = size of max flow
  + Assumes integer capacities
* Edmonds-Karp
  + Find augmenting paths using BFS and takes the shortest path available to t
  + Runs in O(m2n) time
  + Can be floats (although still must be positive)

**Edmonds-Karp Algorithm**

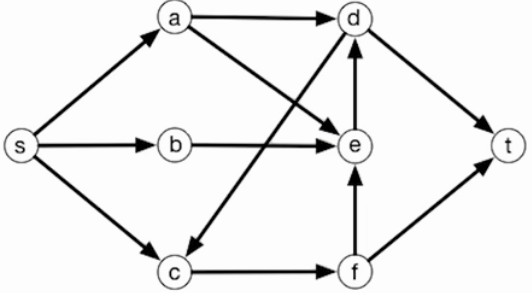
* The defining difference between Edmonds-Karp and Ford-Fulkerson is E-K requires the use of BFS and disallows DFS
* E-K does not rely on integer capacities
* Both F-F and K-E will require at least one full edge to reach capacity during one round of path searching
  + This is unimportant for F-F, but will be important for K-E

**Running Time of Edmonds-Karp Algorithm**

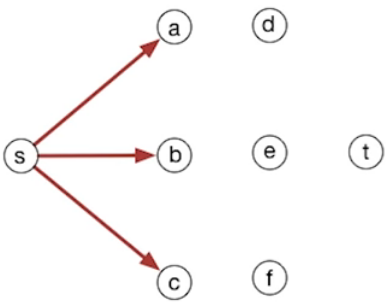
* The number of rounds will be mn
  + ‘m’ is edges
  + ‘n’ is vertices
  + Therefore, the max number of rounds is edges\*vertices
  + Since BFS runs in linear time based on the edges, so this is m
  + O((BFS time)\*edges\*vertices) = O(m2n)
* Each time we run the BFS to find the potential augmenting path, we are guaranteed to eliminate 1 edge
  + That said this edge may come into play on subsequent rounds
* Key Lemma (To Prove runtime of E-K): For every edge e, e is deleted and reinserted at most n/2 times

Using BFS in K-E

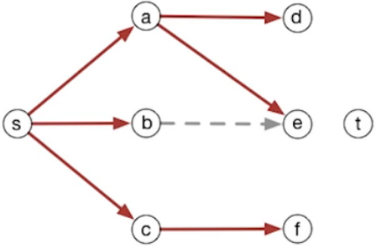
* BFS also specifies a start vertex S
* BFS finds the minimum distances from S to V
  + Note that BFS does NOT capture nor use edge weights
  + If we wanted to sue the weights we would have to use Dijkstra’s Algorithm
* BFS Example
  + Here is our example graph:



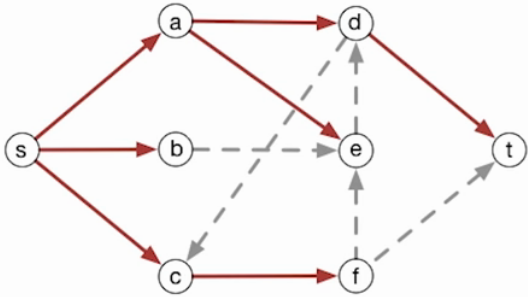
* + After one iteration of BFS, A, B, and C are found:



* + After the second iteration, D, E, and F are found:



* + - Note that the edge B to E exists, but BFS makes a tree and only takes the first edge it finds. It was also possible that it could have found BE first, in which case AE would have been the dotted line
  + The final iteration:



* + - Note, again, that the dotted lines could have ended up being the tree, but BFS only takes the first thing it finds
    - Also note this will iterate again until all augmentation paths are found
      * This means that the available edges to the BFS in the graph WILL change for each BFS search (and there are multiple BFS searches, which occur until all augmentation paths are found)
  + Note that for each iteration, BFS saves a counter that notes the distance from S to the current level; this means that, in this example,
    - Level 0 = S
    - Level 1 = A, B, C
    - Level 2 = D, E, F
    - Level 3 = T
  + For the example, the path – and its levels – are



* + - The level ALWAYS increases in the path by one each time; it never increases more than 1 at a time, nor can it stay the same or go down
      * This is critical for the proof of the lemma.
      * As it turns out, once a level increases it never decreases
  + Claim Proven: For every z ∈ V, level(z) does not decrease
    - For every vertex in the set of vertices, the level of any particular vertex, once increased, never decreases over subsequent runs of BFS
    - This hinges on the fact that if an edge is deleted from a graph its because that edge was traversed in full; if A goes to B, A will be level i and B will be level (i + 1); if that edge is deleted and we travel BA, B’s level stays the same and A’s increases
    - Basically, if we delete an edge and add it back in later, its level will increase by at least 2

**Edmonds-Karp algorithm (Proofs as provided by Dr. Vigoda)**

To analyze the running time of the Edmonds-Karp algorithm, we are going to prove the following two claims:

* a) in every round, at least one edge  is deleted from G^f
* b) an edge is added or deleted from G^fat most ntimes

Hence, since there are medges, the number of rounds will be no more than mn.

For **Claim a)**, at least one edge ein Pis fully occupied since bis the min residual capacity along P. And such ewill be deleted afterwards.

For **Claim b)**, let’s see what happened when an edge is added to or deleted from G^f:

* if \overrightarrow{vw}\in Pis a forward edge then
  + \overrightarrow{vw}is deleted if it is fully occupied (i.e. f_{vw} = c_{vw})
  + \overrightarrow{wv}is added if previously f_{vw}=0
* if \overrightarrow{zv}\in Pis a backward edge then
  + \overrightarrow{zv}is deleted if flow on it is removed (i.e. f_{vz} = 0)
  + \overrightarrow{vz}is added if previously f_{vz}=c_{vz}

In G^f, let level(v)=minimum number of edges in paths from sto v. Note that G^fis changing so level(v)may also changes.

**Claim:** level(v) does not decrease during the execution of the algorithm

**Proof:** Note that in step 3 we take the shortest path P.

Let P=v_0\to v_1\to \dots \to v_lwhere v_0=s, v_l=t. We know that level(v_0)=level(s)=0.

Consider the relation between level(v_i)and level(v_{i+1}):

* level(v_{i+1}) \leq level(v_i) + 1by definition of level(\cdot)
* level(v_{i+1}) \geq level(v_i) + 1since otherwise there exist a shorter path s\to\dots\to v_{i+1}\to\dots\to t

Thus we know level(v_{i+1}) = level(v_i)+1and level(v_i)=iin P.

For any v\in G, level(v)decreases iff some edge \overrightarrow{uv}is added where level(u) < level(v)-1so s\to\dots\to u \to vmakes vnearer to s. However, in order to add such \overrightarrow{uv} to G^f, \overrightarrow{vu} must be in the shortest path Pfirst. Thus we know level(u) = level(v)+1. \qquad\blacksquare

So level(v)must stay the same or increase during the execution of the algorithm.

Now consider what happens if an edge \overrightarrow{vw}is deleted from and later added into G^f:

* When it is deleted we know level(w)=level(v)+1
* When it is added later we know level(v)'=level(w)'+1

Since level(\cdot)never decrease we know level(v)'=level(w)'+1\geq level(w)+1=level(v)+2. Which means if an edge \overrightarrow{vw} is deleted and added later then level(v)at least increases by 2.

Thus an edge can be deleted and added later for no more than n/2times since level(\cdot)\leq n.

And remember that at least 1 edge is deleted each round we know the total number of rounds is no more than nm.

Thus the overall running time of Edmonds-Karp algorithm is O(nm^2).

The current best algorithm for the max-flow problem runs in O(nm)time by [[Orlin ’13]](http://dl.acm.org/citation.cfm?doid=2488608.2488705).