CS 125 Section #5 **Graph Traversal and Linear Programs** October 6, 2016

1 Depth first search

1.1 The Algorithm

Besides breadth first search, which we saw in class in relation to Dijkstra's algorithm, there is one other fundamental algorithm for searching a graph: depth first search. To better understand the need for these procedures, let us imagine the computer's view of a graph that has been input into it, in the adjacency list representation. The computer's view is fundamentally *local* to a specific vertex: it can examine each of the edges adjacent to a vertex in turn, by traversing its adjacency list; it can also mark vertices as visited. One way to think of these operations is to imagine exploring a dark maze with a flashlight and a piece of chalk. You are allowed to illuminate any corridor of the maze emanating from your current position, and you are also allowed to use the chalk to mark your current location in the maze as having been visited. The question is how to find your way around the maze.

We now show how the depth first search allows the computer to find its way around the input graph using just these primitives.

Depth first search uses a **stack** as the basic data structure. We start by defining a recursive procedure search (the stack is implicit in the recursive calls of search): search is invoked on a vertex v, and explores all previously unexplored vertices reachable from v.

```
Procedure search($v$)
    vertex $v$
    explored($v$) := 1
    previsit($v$)
    for $(v, w) \in E$
        if explored($w$) = 0 then search($w$)
    rof
    postvisit($v$)
end search
```

```
Procedure DFS ($G(V, E)$)
    graph $G(V, E)$
    for each $v \in V$ do
        explored($v$) := 0
    rof
    for each $v \in V$ do
        if explored($v$) = 0 then search($v$)
    rof
end DFS
```

By modifying the procedures previsit and postvisit, we can use DFS to solve a number of important problems, as we shall see. It is easy to see that depth first search takes $O(|V| + |E|)$ steps (assuming previsit and postvisit take $O(1)$ time), since it explores from each vertex once, and the exploration involves a constant number of steps per outgoing edge.

The procedure search defines a tree (** well, actually a *forest*, but let's not worry about that distinction right now **) in a natural way: each time that search discovers a new vertex, say w, we can incorporate w into the tree by connecting w to the vertex v it was discovered from via the edge (v, w). The remaining edges of the graph can be classified into three types:

- Forward edges - these go from a vertex to a descendant (other than child) in the DFS tree.
- Back edges - these go from a vertex to an ancestor in the DFS tree.
- Cross edges - these go from "right to left"– there is no ancestral relation.

Remember there are four types of edges; the fourth is the "tree edges", which were edges that led to a new vertex in the search.

Question: Explain why if the graph is undirected, there can be no cross edges.

One natural use of the previsit and postvisit procedures is that they could each keep a counter that is increased each time one of these routines is accessed; this corresponds naturally to a notion of time. Each routine could assign to each vertex a preorder number (time) and a postorder number (time) based on the counter. If we think of depth first search as using an explicit stack, then the previsit number is assigned when the vertex is first placed on the stack, and the postvisit number is assigned when the vertex is removed from the stack. Note that this implies that the intervals $[preorder(u), postorder(u)]$ and $[preorder(v), postorder(v)]$ are either disjoint, or one contains the other.

An important property of depth-first search is that the contents of the stack at any time yield a path from the root to some vertex in the depth first search tree. (Why?) This allows us to prove the following property of the postorder numbering:

Claim 1. *If $(u, v) \in E$ then $postorder(u) < postorder(v) \iff (u, v)$ is a back edge.*

Proof. If $postorder(u) < postorder(v)$ then v must be pushed on the stack before u. Otherwise, the existence of edge (u, v) ensures that v must be pushed onto the stack before u can be popped, resulting in $postorder(v) < postorder(u)$ — contradiction. Furthermore, since v cannot be popped before u, it must still be on the stack when u is pushed on to it. It follows that v is on the path from the root to u in the depth first search tree, and therefore (u, v) is a back edge.

The other direction is trivial. □

Exercise. *What conditions do the preorder and postorder numbers have to satisfy for (u, v) to be a forward edge? A cross edge?*

Claim 2. *$G(V, E)$ has a cycle iff the DFS of $G(V, E)$ yields a back edge.*

Proof. If (u, v) is a back edge, then (u, v) together with the path from v to u in the depth first tree form a cycle.

Conversely, for any cycle in $G(V, E)$, consider the vertex assigned the smallest postorder number. Then the edge leaving this vertex in the cycle must be a back edge by Claim 1, since it goes from a lower postorder number to a higher postorder number. □

1.2 Application of depth first search: Topological sorting

Consider the following problem: the vertices of a graph represent tasks, and the edges represent precedence constraints: a directed edge from u to v says that task u must be completed before v can be started. An important problem in this context is scheduling: in what order should the tasks be scheduled so that all the precedence constraints are satisfied?

We now suggest an algorithm for this scheduling problem. Given a directed graph $G(V, E)$, whose vertices $V = \{v_1, \dots v_n\}$ represent tasks, and whose edges represent precedence constraints: a directed edge from u to v says that task u must be completed before v can be started. The problem of topological sorting asks: in what order should the tasks be scheduled so that all the precedence constraints are satisfied.

Note: The graph must be acyclic for this to be possible. (Why?) Directed acyclic graphs appear so frequently they are commonly referred to as DAGs.

Exercise. *If the tasks are scheduled by decreasing postorder number, then all precedence constraints are satisfied.*

Solution.
If G is acyclic then the DFS of G produces no back edges by Claim 2. Therefore by Claim 1, $(u, v) \in G$ implies $postorder(u) > postorder(v)$. So, if we process the tasks in decreasing order by postorder number, when task v is processed, all tasks with precedence constraints into v (and therefore higher postorder numbers) must already have been processed.

There's another way to think about topologically sorting a DAG. Each DAG has a *source*, which is a vertex with no incoming edges. Similarly, each DAG has a *sink*, which is a vertex with no outgoing edges. (Proving this is an exercise.) Another way to topologically order the vertices of a DAG is to repeatedly output a source, remove it from the graph, and repeat until the graph is empty. Why does this work? Similarly, once could repeatedly output sinks, and this gives the reverse of a valid topological order. Again, why?

Exercise. *Give a linear-time algorithm that takes as input DAG G and two vertices s, t and returns the number of simple paths from s to t in G.*

Solution.
A sketch of the solution: Use topological sort to sort the vertices. Consider the list of vertices S lying between s and t, inclusive, in the topological sort. Here's the key: any path from a to b must use only vertices which lie between a and b in the topological sort. Now we only have to think about the subgraph consisting of vertices S and edges between vertices in S. Beginning with the last vertex in S (which is t) and moving backwards to the first vertex in S (which is s), for each vertex v in S, calculate the number of paths from v to t:

$$npaths(v) = \sum_{v<x \text{ in } S} [\exists(v, x)] \cdot npaths(x)$$

All edges between vertices in S are considered exactly once and for a constant time, so this algorithm is linear in the size of the original graph.

2 Breadth-first search review

- While DFS uses a stack, BFS uses a **queue**, and that's what makes the two algorithms fundamentally different.
- BFS is also a $O(|E| + |V|)$ search, often done from just one node in a graph.
- BFS gives you a list of nodes in increasing path-length from source.

Exercise. *Suppose we defined forward, back, and cross edges analogously to DFS. Show that:*

1. *in a BFS of an unweighted directed graph, there are no forward edges.*
2. *in a BFS of an undirected graph, there are no forward or back edges. Moreover, for any cross edge (v, u), either $dist[v] = dist[u]$ or $dist[v] = dist[u] \pm 1$.*

Solution.
Write $d(w, v)$ for the distance from vertex w to vertex v in a graph (be it directed or not).

1. Suppose (v, u) is a forward edge with respect to the BFS tree rooted at w. Then it must be the case that $d(w, u) > d(w, v) + 1$, because (v, u) is not an edge in the tree, and thus cannot connect v to any of its children in the tree. On the other hand, since (v, u) is an edge, it must be that $d(w, u) \leq d(w, v) + 1$, contradiction.
2. Forward edges and back edges are the same thing for undirected graphs, so by the reasoning from previous part neither type of edges exist. If (v, u) is a cross edge for the BFS tree rooted at w, we have $d(w, v) \leq d(w, u) + 1$ and $d(w, u) \leq d(w, v) + 1$; this means that $d(w, v) = d(w, u)$ or $d(w, v) = d(w, u) \pm 1$. But $d(w, v)$ is exactly $\texttt{dist}[v]$ by the properties of BFS, and similarly for $d(w, u)$, so we're done.

3 Shortest Paths

3.1 Review

In class, we saw several shortest paths algorithms:

- **Djikstra's Algorithm** (single-source): modified BFS
- **Bellman-Ford** (single-source): for $|V|$ times, go through the edges and, upon looking at (u, v) update the shortest path to v if going to u and then via (u, v) would give a better answer than what we have so far
- **Floyd-Warshall** (all pairs): dynamic programming; the subproblems are $D(i, j, k)$, the shortest path from i to j using only intermediate vertices in $\{1, \ldots, k\}$

Question: What are the constraints on the graph for each of the above algorithms to work?

Question: What are the runtimes for each of these algorithms?

Exercise. *Consider the shortest paths problem in the special case where all edge costs are non-negative integers. Describe a modification of Dijkstra's algorithm that works in time $O(|E| + |V|m)$, where m is the* *maximum cost of any edge in the graph.*

Solution.
The key observation is that the `deletemin` operations we perform are on vertices with nondecreasing priority (and this goes back to the analogy with the explorers from class): formally, this is because each time we insert a new value $\texttt{dist}[v] + \texttt{length}(v, w)$ into the heap, it is greater than the minimum value $\texttt{dist}[v]$ we delete at the beginning of the step in which we consider the vertex v.

Consider then the following implementation of the priority heap: we keep an array H of size $|V|m$, the entries of which are subsets of V. Then inserting $(w, \texttt{dist}[w])$ works by adding w to $H[\texttt{dist}[w]]$, and `deletemin` works by keeping a counter into H that starts at 0 and on each call to `deletemin` increments itself repeatedly until it points to a nonempty entry $H[i]$, and returns the first element of $H[i]$.

Since the values in `dist` are always the lengths of paths of at most $|V| - 1$ edges in G and no such path is longer than $|V|m$, H won't overflow; and by the observation in the beginning, the `deletemin` operation works correctly.

Finally, for the runtime, each `insert` can be done in time $O(1)$, say by using linked lists to implement the sets $H[i]$; and the total time for all `deletemin` operations is at most $O(m|V| + |V|)$, because we traverse each cell of H at most once, and each vertex in V at most once.

3.2 Johnson's Algorithm

One of the difficulties that we encounter with Djikstra's algorithm is that of negative edge weights. The first way we might try to fix this problem is by adding some large number to the weight of every edge, in order to get rid of all the negative weights; however, this doesn't preserve the shortest paths. (Why?)

Instead, we will give a slightly different way to do this type of edge weight modification, known as Johnson's Algorithm.

Exercise. *Suppose that we assign some weight $w(v)$ to each vertex v, and that for each edge (u, v), we add $w(u) - w(v)$ to its weight. Prove that the shortest path between any pair of vertices is the same in the updated graph as in the original (albeit with a different weight).*

Solution.
Suppose we had any path from u to v. If its total weight in the original graph was e, then its weight in the new graph is $e + w(u) - w(v)$; however, note that this changes every path from u to v by the same amount (and doesn't depend on the intermediate vertices in the path), so the shortest path is preserved.

Exercise. *Here is a method for determining the edge weights $w(v)$: add a new vertex v_0, and add an edge (v_0, v) with weight 0 for each vertex v in the original graph. Then, we set $w(v)$ to be the length of the shortest path from v_0 to v.*

Solution.
Suppose we have some edge $e(u, v)$. Then, we want that $e(u, v) + w(u) - w(v) \geq 0$. However, this is the same as saying that $e(u, v) + w(u) \geq w(v)$; we see that this property is true because the path from v_0 to

v which takes the shortest path to u and then follows the edge (u, v) has weight $e(u, v) + w(u)$, so it must be at least as large as $w(v)$, the shortest path from v_0 to v.

Exercise. *We can put together the facts above to obtain an algorithm: run Bellman-Ford to figure out the weights to assign to the vertices, augment the edge weights of hte graph, and then run Djikstra's Algorithm from each vertex.*

What is the runtime? How does it compare to Floyd-Warshall?

Solution.
The runtime is $O(EV)+O(E)+VO(E\log V) = O(EVlogV)$ or $O(EV)+O(E)+VO(V^2) = O(V^3)$, which are no better than Floyd-Warshall's $O(V^3)$.

However, it turns out there is an $O(E + V \log V)$ implementation of Djikstra's algorithm, which allows us to get a better bound of $O(EV + V^2 \log V)$, which is better than Floyd-Warshall on sparse graphs.

4 Linear programs: an example reduction

Exercise. *Express the following problem as a linear programming problem: A gold processor has two sources of gold ore, source A and source B. In order to kep his plant running, at least three tons of ore must be processed each day. Ore from source A costs $20 per ton to process, and ore from source B costs $10 per ton to process. Costs must be kept to less than $80 per day. Moreover, Federal Regulations require that the amount of ore from source B cannot exceed twice the amount of ore from source A. If ore from source A yields 2 oz. of gold per ton, and ore from source B yields 3 oz. of gold per ton, how many tons of ore from both sources must be processed each day to maximize the amount of gold extracted subject to the above constraints?*

Solution.
Let x and y be the tons of ore from source A and B, respectively. Then our constraints are:

$$\begin{aligned} x + y &\geq 3 \\ 20x + 10y &\leq 80 \\ x &\leq 2y \end{aligned}$$

and we wish to maximize $2x + 3y$.

Exercise. *How would you encode TSP (the Traveling Salesman Problem) as an integer linear program? Recall that given a weighted graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}_{\geq 0}$, the TSP asks for a cycle in G that covers each vertex exactly once and has minimal weight.*

Solution.
Make a variable a_e for each edge, for the number of times which we visit that edge. We want to minimize

$$\sum_e w(e) a_e$$

under the constraints $0 \le a_e \le 1$, and in addition,

$$\forall v \in V : \sum_{e \sim v} a_e = 2.$$

At this point, if the a_e end up being integers, we've made sure that we have a subgraph of G in which every vertex is of degree two, that has minimal weight among all such subgraphs. But such a graph could be a disjoint union of several cycles.

To fix that, we of course add more variables. The key point is that we can encode connectedness to a given vertex. So fix a vertex v_0, and let $f_{e,v}$ be new variables, for each edge $e \in E(G - v_0)$ and each vertex incident to e. We require $f_{e,v} \ge 0$, and additionally

$$\forall e = (v,u) \in E(G - v_0) : f_{e,v} + f_{e,u} = 2|V| a_e \quad \clubsuit$$

to encode the 'flow' over an edge e, and

$$\forall v \in V(G - v_0) : \sum_{e \sim v} f_{e,v} < 2|V|$$

to encode that all vertices should be in the same cycle as v_0, so that they have to get less than the average flow in a cycle where v_0 is not present. Unfortunately, we can't have non-strict inequalities in the linear program! So we modify the last one a little bit:

$$\forall v \in V(G - v_0) : \sum_{e \sim v} f_{e,v} \le 2|V| - 2 \quad \spadesuit$$

The point is that we can make the flow be noticeably far from 2 in a spanning cycle; that is, if $a_e \in \{0,1\}$ and they define a spanning cycle, we can set the variables $f_{e,v}$ in a way that makes $\spadesuit$ hold. To see how that works, consider the path $v_1, v_2, \ldots, v_{|V|-1}$ consisting of $|V|-1$ vertices and $|V|-2$ edges obtained by removing v_0 and edges adjacent to it from the spanning cycle. Assign $f_{(v_1,v_2),v_1} = 2|V|-2$, and then require $f_{(v_{i-1},v_i),v_i} + f_{(v_i,v_{i+1}),v_i} = 2|V|-2$ for $i = 2, \ldots, |V|-2$. This forces the values $f_{(v_1,v_2),v_2} = 2, f_{(v_2,v_3),v_2} = 2|V|-4, f_{(v_2,v_3),v_3} = 4, f_{(v_3,v_4),v_3} = 2|V|-6$ and so on, until we have $f_{(v_{|V|-2},v_{|V|-1}),v_{|V|-2}} = 2|V| - 2 \times (|V|-2) = 4$. At this point we must set $f_{(v_{|V|-2},v_{|V|-1}),v_{|V|-1}} = 2|V|-4 < 2|V|-2$, and all constraints are satisfied.

Now assume all the variables ended being integers. Then there cannot be a cycle in our solution that is disjoint from the one containing v_0, for some vertex v in it we'll have $\sum_{e \sim v} f_{e,v} \ge 2$ by pigeonhole. Conversely, no spanning cycle will be excluded from consideration, because in that case by the above argument there is an assignment of the $f_{e,v}$ that makes $\spadesuit$ and $\clubsuit$ hold.

4.1 (More) Shortest Paths

Exercise. *Suppose we have a graph $G = (V,E)$, and want to find the shortest path between some two vertices S and T. Write a linear programming formulation of this problem (you may assume that there are no negative cycles).*

Solution.
For each edge (u, v), we have a variable $x_{(u,v)}$, which we want to be 1 if the edge is in our shortest path, and 0 otherwise. Let $w_{(u,v)}$ be the weight of that edges.

Then, we have the objective function

$$\min \sum_{(u,v)\in E} w_{(u,v)} x_{(u,v)}$$

and constraints

$$\begin{aligned}\sum_{(s,v)} x_{(s,v)} - \sum_{(u,s)} x_{(u,s)} &= 1\\ \sum_{(t,v)} x_{(t,v)} - \sum_{(u,t)} x_{(u,t)} &= -1\\ \sum_{(w,v)} x_{(w,v)} - \sum_{(u,w)} x_{(u,w)} &= 0 (w \neq s, t).\end{aligned}$$

As it turns out, it actually isn't necessary to make this an integer linear programming problem, nor is it necessary to explicitly restrict the $x_{(u,v)}$ to be in $[0, 1]$.

5 Max flow / min cut

1. **Maximum flow is equal to minimum cut**:
   (a) Maximum flow $\leq$ minimum cut is clear.
   (b) Maximum flow $\geq$ minimum cut comes from the augmenting paths algorithm: when there are no more augmenting paths to be found, all the vertices reachable from S in the residual network form a cut. All edges crossing this cut must have zero residual capacity, which means the flow is at least the sum of their original capacities, which in turn is at least the flow of the minimum cut.

   In particular, if e is an edge in the original graph with capacity $c(e) = c$ (and with $\bar{e}$ not in the graph), and we ship flow $f \leq c$ across e, then e has residual capacity $c - f$. The reverse edge $\bar{e}$ then has residual capacity $0 - (-f) = f$.

2. **Ford-Fulkerson algorithm** To find the maximum flow of a graph with integer capacities, we repeatedly use DFS to find an **augmenting path** from start to end node in the residual network (note that you can go backwards across edges with negative residual capacity), and then add that path to the flows we have found. We stop when DFS is no longer able to find such a path.
   (a) **Residual flow**: To form the residual network, we consider all edges $e = (u, v)$ in the graph as well as the reverse edges $\bar{e} = (v, u)$.
      i. Any edge that is not part of the original graph is given capacity 0.
      ii. If $f(e)$ denotes the flow going across e, we set $f(\bar{e}) = -f(e)$.
      iii. The **residual capacity** of an edge is $c(e) - f(e)$.
   (b) Correctness:

This algorithm might be a little confusing. Some clarifying points:

i. We think of the residual network as of a networking without any flow whose capacities can however change during the course of the algorithm

ii. after each new augmenting path has been added, we have a flow on our original network, and we change the residual capacities accordingly.

Here's the argument for maximum flow $\geq$ minimum cut, which also establishes correctness. The set of reachable nodes after the algorithm terminates defines a cut of the graph. In particular, it must be true that min cut $\leq$ cut found $=$ flow found $\leq$ max flow, hence min cut $=$ max flow, and in fact all inequalities are equalities, and the flow found is a maximum flow.

I find the following part to be the trickiest in this argument: why is the flow found equal to the cut found? The reason we can make this argument is because we have the residual network. Suppose we have a source s and a target t. Consider the cut determined by the algorithm: let the nodes reachable from s form the set S, and the nodes unreachable from s form the set T. We know that all edges crossing from S to T are at full capacity: otherwise we would've been able to reach something in T from S. But we also know that there is no flow coming from T to S, because then the residual edge going in the opposite direction would be able to carry some flow, again contradicting the un-reachability of T. So, the total flow flowing into T is equal to the size of the cut between S and T; and on the other hand, by conservation of flow, it is equal to the total flow coming into T.

(c) The runtime is $O((m+n)(\text{max flow}))$.

(d) **Non-integer variant** If we use BFS instead of DFS above, this method also works for non-integral capacities and runs in time $O(nm^2)$.

3. **Matchings**: The maximum size matching in a bipartite graph can be solved by taking the integer maximum flow from one side to the other.

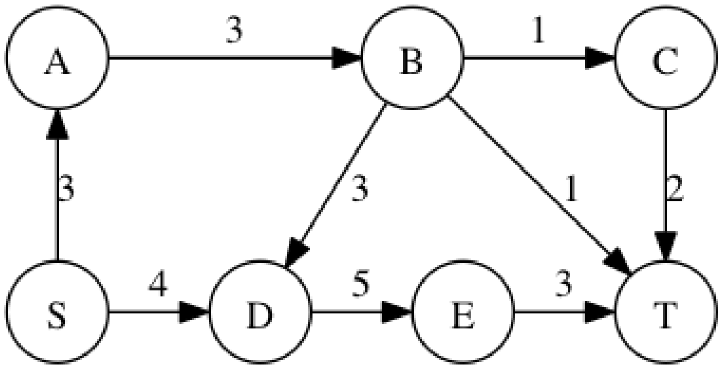
4. **Karger's algorithm** (We didn't cover this in class, but it's pretty cool.) To try to find a minimum cut on a graph where all edges have weight 1, randomly contract edges until there are only two vertices left, and then output the weight of the cut which separates the two vertices.

   (a) To *contract* an edge (u, v) means replacing (u, v) by a new node w, and then replacing u or v with w in all edges of the graph where they occur.

   (b) With probability at least $1/\binom{n}{2}$, Karger's algorithm will output a particular minimum cut.

Exercise. *Perform the augmenting paths algorithm on this example, showing the residual graph at each step:*

$$\begin{tikzcd}
A \arrow[r, "3"] & B \arrow[r, "1"] \arrow[dl, "3"] \arrow[drr, "1"] & C \arrow[d, "2"] & \\
S \arrow[u, "3"] \arrow[r, "4"] & D \arrow[r, "5"] & E \arrow[r, "3"] & T
\end{tikzcd}$$



5.1 Power Grid

Exercise. *Suppose you are the owner of a power plant with some customers, you have a power grid in place where each power line (edge) has some maximum energy it can carry, and each customer has a maximum amount of energy they require. What is the most energy that you can dispense with the current configuration so that no customer gets more than they request?*

Solution.
This is the same as the max flow algorithm: we set the power plant as the source, and add a sink node with an edge from each customer, with capacity equal to the maximum amount of energy that the customer requires.

Exercise. *Now, suppose that each of the transformers along the way (e.g. each vertex of the graph) also has a maximum flow which can pass through it. What is the most energy that you can dispense with the current configuration so that no customer gets more than they request?*

Solution.
We only need to make one adjustment to the previous solution: for each transformer t, add a second vertex t' with an edge $t \to t'$ with the capacity of that transformer, and make all the edges out of t come out of t' instead.

Exercise. *Upon revisiting your power allocation algorithm, you realize that customers will probably be happier if they receive excess energy than if they receive insufficient energy. Given the same configuration as above, but this time where each customer has a minimum amount of energy that s/he requires, how could you determine whether it is possible to meet all the customer needs?*

Solution.
In class, we saw the reduction of a max flow problem to a linear programming problem. Here, it suffices to take that same reduction, but with two changes:

- For the edges from the customers to the sink, setting the constraint $f_{edge} \geq capacity$, instead of $capacity \geq f_{edge}$.
- Take the min flow instead of the max flow.

5.2 Class Planning

Exercise. *Suppose you have N students and M classes. Each student signs up to take 4 classes, and each class has some maximum total enrollment. We want to assign each student to some subset of the classes they have chosen so that we maximize the total enrollment across all of the classes, and no class exceeds its maximum enrollment. How should we compute such a configuration?*

Solution.
This is the same as the following max flow problem:

- Add a source node, with an edge to each student node with capacity 4.
- For each student node, add an edge to every class node that s/he signed up for with capacity 1.
- Add a sink node, with an edge from each class node with capacity the maximum enrollment of that class.

5.3 Minimum Edge Flows

Exercise. *Suppose that each edge, in addition to having a maximum capacity, also has a minimum capacity. Reduce the problem of determining whether such a flow exists to a max flow problem.*

Solution.
This was shamelessly taken from `http://web.engr.illinois.edu/~jeffe/teaching/algorithms/2009/notes/18-maxflowext.pdf`

We reduce from a maximum flow problem. Given a network $G = (V, E)$ with capacities c and demands (lower capacities) d, consider the following new graph $G' = (V', E')$ given by adjoining two new nodes s', t' and capacities as follows:

1. for each vertex $v \in V$, $c'(s' \to v) = \sum_{u \in V} d(u \to v)$ and $c'(v \to t') = \sum_{w \in V} d(v \to w)$.
2. for each edge $u \to v$ in E, set $c'(u \to v) = c(u \to v) - d(u \to v)$.
3. set $c(s \to t) = \infty$.

If there are multiple edges connecting two vertices, replace them by a single one with the same total capacity.

Then one can show that G has a *feasible* (s, t) flow, i.e. a flow that respects the capacities and demands, iff G' has a *saturating* (s', t') flow, which can be detected by the max flow algorithm.

Indeed, suppose that f is a feasible flow on G with magnitude $|f|$. Then

$$f'(u \to v) = f(u \to v) - d(u \to v)$$
$$f'(s' \to v) = \sum_{u \in V} d(u \to v)$$
$$f'(v \to t') = \sum_{w \in V} d(v \to w)$$
$$f'(t \to s) = |f|$$

is a saturating (s', t') flow on G'. This f' diverts $d(u \to v)$ units of flow from u directly to the target t' and injects $d(u \to v)$ units of flow into v directly from the source s'.

Similarly, for any saturating (s', t') flow on G', the flow $f = f'|_E + d$ s a feasible flow on G.

Now, use this to solve the following problem from Salil's quals:

Exercise. *Suppose we have some m by n matrix of real numbers, such that the sum of every column and every row is an integer. Prove that there exists a way to round every entry up or down, such that all the row and column sums are preserved.*

Solution.
We can write this as a flow problem:

- We have a source node s, a sink node t, and nodes $r_1, \ldots, r_m$ and $c_1, \ldots, c_n$ for each row and column.
- Add an edge $s \to r_i$ with capacity equal to the sum of the row r_i for each row.
- Add an edge $c_j \to t$ with capacity equal to the sum of the row c_j for each column.
- For each entry a_{ij} in the matrix, add an edge $r_i \to c_j$ with minimum capacity $\lfloor a_{ij} \rfloor$ and maximum capacity $\lceil a_{ij} \rceil$.

Because the matrix exists, we know that there is a solution to this flow problem. However, because all the constraints are integers, we know that if a solution exists, then an integer one also exists; hence, there exists a valid rounding of the entries of the matrix.