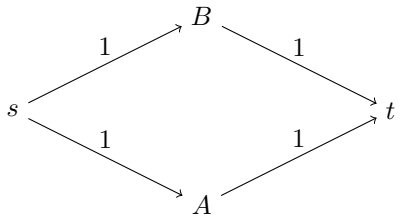
6.854 Advanced Algorithms

Problem Set 5

John Wang
Collaborators: Jason Hoch

Problem 1-a: An edge is upward critical if replacing its capacity c by any $c' > c$ increases the maximum flow value. Does every network have an upward-critical edge? Give an algorithm to identify all upward-critical edges in a network. Its running time should be substantially better than that of solving m maximum-flow problems.

Solution: Not every network has an upward-critical edge. Consider the graph below:



We see that the capacities on all of the edges is 1. However, increasing the capacity of any edge will not increase the flow, since the max flow will always stay at 2 (one unit of flow coming from the s, A, t path and the other coming from the s, B, t path. Therefore, we see that not all networks have an upward-critical edge.

To find an algorithm for detecting upward critical edges, we will first note that if we increase the capacity of an edge, and afterwards, the residual graph of a max flow now has an augmenting path from s to t, then the max-flow can be increased. Assuming integer capacities, increasing the capacity by 1 suffices for creating a new augmenting path. Therefore, we can create an algorithm to find these new augmenting paths.

First, we perform a BFS to determine the connected components in the residual graph of a max flow. This is done by starting at the source s and finding all nodes that s can navigate to (assuming that 0 weight edges are non-existent). Then, we will go to t and reverse all of the edges in the residual graph. We will perform a BFS to find all the backward connected components of t. This backward BFS will find all nodes v which have a path to t. Now, we can label the first set of components S and the second set T, and it is sufficient to find an edge that connects S to T when its capacity is increased.

In order to find such an edge, we can iterate over all edges $e = (u, v)$ in the residual graph, and see if $u \in S$ and $v \in T$ or vice versa. If this is the case, then we can increase the edge's capacity and create an augmenting path from s to t, thus increasing the max flow. Note that all such possible augmenting paths will be found by the algorithm.

The total time of this algorithm requires $O(m+n)$ for the two BFS searches for the connected components and $O(m)$ for iterating over the edges and checking if the edges are in sets S and T. Thus, the total time of the algorithm is $O(m + n) = O(m)$. □

Problem 1-b: An edge is downward critical if replacing its capacity c by any $c' < c$ decreases the maximum flow value. Is the set of upward-critical edges the same as the set of downward critical-edges? Describe an algorithm for identifying all downward critical edges, and analyze your algorithm's worst-case complexity. Its running time should be substantially better than that of solving m maximum-flows.

Solution: The set of downward-critical edges is not the same as the set of upward-critical edges. Consider the figure that was presented in part a, with all edges of capacity 1. If we decrease the capacity of any of the edges by 1, we will decrease the maximum flow by 1 as well, since we will remove the flow path of either s, A, t or s, B, t with such a decrease in capacity. This means that every edge is in the set of downward-critical edges. Since we already showed that the set of upward-critical edges for this graph is empty, we see that these two sets are not always the same.

Now notice that if any edge $e = (u, v)$ can be removed and the maximum flow does not change, then that edge e is not downward-critical. If the maximum flow does change, then e must be downward critical because

we can removed $c' = w(e)$ from the current capacity $c = w(e)$ of the edge and decrease the maximum flow value. Thus, we need to find all edges, which when removed, no longer have the sames flows from s to t. We can do this by finding looking at each edge $e = (u, v)$ and finding out if u and v are connected in the residual graph with a path of flow equal to the flow that used to go over $e = (u, v)$.

We can use BFS to examine each edge $e = (u, v)$ and examine whether there is a path from u to v in the residual graph after removing e whose flow is greater than or equal to the flow over e. If there does not exist such a path, report e as downwards critical. This requires m iterations of BFS, so the total runtime is $O(m(m + n)) = O(m^2)$. $\square$

6.854 Advanced Algorithms

Problem Set 5

John Wang
Collaborators: Jason Hoch

Problem 2-a: Show how to solve the minimum flow problem by using two applications of any maximum flow algorithm that applies to problems with zero lower bounds on edge flows (e.g. the algorithms described in lecture). Your algorithm should detect if there is a feasible flow and, if there is one, return a minimum flow.

Solution: □

6.854 Advanced Algorithms

Problem Set 5

John Wang
Collaborators: Jason Hoch

Problem 3-a: Suppose that the numbers of faculty and students are equal, each student wants to meet exactly d faculty, and each faculty member is on the request list of d students. Conclude that one can schedule a single slot in which every student is meeting someone.

Solution: We will construct a bipartite graph where the left side of the graph is composed of the n students, and the right side is composed of the n faculty members. An edge connecting a student u to a faculty member v in the graph implies that the student wants to meet faculty member v. This implies that there are exactly d outgoing edges from any u in the set of students S and exactly d incoming edges to any v in the set of faculty members F.

We can construct a flow graph by creating a source s and a sink t. From the source, there are edges (s, u) for all $u \in S$ with capacity 1. For the sink, there are edges (v, t) for all $t \in F$ with capacity 1. For the edges (u, v) where $u \in S$ and $v \in F$, we will set the capacity to 1. To show that we can schedule a single slot in which every student is meeting someone, we need to find a perfect matching between S and F.

In order to do that, consider the following flow:

$$f(u,v) = \begin{cases} 1 & \text{if } u = s \\ 1/d & \text{if } u \in S, v \in F \\ 1 & \text{if } v = t \end{cases} \tag{1}$$

We see that this is a valid flow through the graph since at each node $u \in S$, there are exactly d outgoing edges of flow $1/d$, and a single incoming edge of flow 1. For each node $v \in S$, there are d incoming edges of flow $1/d$ and a single outgoing edge of flow 1. Moreover, we see that all of these flows are within their capacity limits by inspection. This ensures that this is a valid flow. We see that the value of this flow is n, so that its cut is of capacity n as well.

By the integrality theorem, there exists some max flow for which all flow values are integers. This implies that there exists some max flow, of value greater than or equal to n, which has a min-cut of capacity n. This means that there must be n edges crossing the min-cut (since all edges have capacity at most 1), which further implies that there exists n paths from s to t of capacity 1. This shows that each student can be matched up to a faculty member. □

Problem 3-b: Conclude that it is possible to schedule all the meetings to take place in d time slots.

Solution: We will prove that the capacitated graph we constructed above has d perfect matchings where no matching has an edge in common (d different max flows without shared edges in the min-cut). If this is the case, then it is sufficient to set these perfect matchings in arbitrary order in order to fill up all of the meetings in d time slots. We will call a graph where each student has exactly d faculty requests, and each faculty member has exactly d students on their request list, a d-request graph.

We will prove the following lemma: if G is a d-request graph, then G has d perfect matchings, no pair of which have any edge in common. We show this by induction on d. For $d = 1$, we see that a perfect matching is trivially satisfied by following all of the edges, so that each fauclty member is paired with the only student on their list. Now suppose $d = k$ and we have proven this fact for all d through $k - 1$. Now let there be subsets $A \subset S$ and $B \subset F$.

Let us define $N_G(A) = \{u \in V(G) | u \text{ is adjacent to v} \in A\}$ as the neighborhood of A. Now suppose by contradiction that $|N_G(A)| < |A|$. We know that $\sum_{u \in A} \sum_{v \in F} e(u, v) = d|A|$, by the fact that A is part of a d-request graph. This implies by the pigeonhole principle that there exists a vertex in $N_G(A)$ whose degree is greater than d. This contradicts the fact that G is a d-request graph (and that there are exactly d requests for all nodes in S). Thus, we find that $|N_G(A)| \geq |A|$.

Since we know that $|N_G(A)| \geq |S|$ for all $S \in A$, Hall's Marriage Theorem (Theorem 26.3-4 in CLRS) shows that G has a perfect matching. Now we can remove every edge e which was in this perfect matching, and create a new $k - 1$-request graph. This is true because each edge removes 1 from the incoming edges

for each node $v \in F$ and the outgoing edges for each node $u \in S$. We know that the $k-1$-request graph has $k-1$ perfect matchings by our inductive hypothesis. This implies that there are $k-1+1=k$ perfect matchings for a k request graph. This completes the proof. □

Problem 3-c: Consider an arbitrary set of desired meetings. Obviously one needs at least as many slots as there are faculty to meet a given student, and students to meet a given faculty. Prove that one can arrange all meetings with no more slots than this number s.

Solution: Let $s = \max\{d_f, d_s\}$ where d_f is maximum number of faculty on the list of any student and d_s is the maximum number of students who any faculty is required to meet. □

Problem 3-d: Show that the schedule can be computed in $O(s^2n^{3/2})$ time, where s is defined in (c) and n is the total numbmer of students and faculty members.

Solution: Notice that our bipartite graph has n vertices and $m = sn$ edges. This is because there are at most s outgoing edges on any vertex $u \subset S$ and s incoming edges on any vertex $v \subset F$. We have also shown in class that maximum flow can be achieved in a bipartite graph in $O(m\sqrt{n}) = O(sn^{3/2})$ by using a reduction to a unit graph and performing blocking flows on the unit graph.

We have already shown in part (c) that we can arrange all meetings with no more than s time slots. This implies that we can find s maximum flows iteratively. On the ith iteration, we will find a max flow of the graph G_i, schedule student-faculty member meetings according to the matchings provided by the max-flow (edges leading from students to faculty member nodes will each be a meeting), then remove all the (u, v) edges where $u \in S$ and $v \in F$ from the max-flow. This will leave a graph G_{i-1} with all previous max-flow matchings removed. Performing s iterations of this will result in a complete schedule over s time slots.

The algorithm requires $O(sn^{3/2})$ work during each iteration. Since there are s iterations, the total cost is $O(s^2n^{3/2})$, which is what we wanted. □

6.854 Advanced Algorithms

Problem Set 5

John Wang
Collaborators: Jason Hoch

Problem 4: At a certain point in the season, each team i in the American League has won a certain number w_i of games, and there remain q_{ij} games to be played between teams i and j. Assuming no ties or cancelled games, develop an efficient, flow-based algorithm for deciding if the Red Sox can still win the league pennant, i.e. whether a particular team can still win more games than any other team after all games have been played.

Solution: First, we will notice that we can reduce the problem to where the Red Sox win all of their remaining games. Let w_r be the number of games that the Red Sox have won so far. If it is possible for the Red Sox to win the league pennant with $w < w_r^*$ wins, where $w_r^* = w_r + \sum_{j=1}^{n} q_{rj}$ represents the number of wins if the Red Sox win every remaining game, then it is surely possible that the Red Sox will win the pennant with $w = w_r^*$, the maximum possible number of wins. This follows because the Red Sox winning a game monotonically increases the Red Sox's record while monotonically decreasing the records of other teams. This implies that we only need to check to see if the Red Sox can win the pennant if the Red Sox win all of their remaining games.

We will now create a bipartite graph which can be converted into a flow network which will solve our problem. First, we create a set of nodes T where each $t_i \in T$ represents the ith team in the American League. Next, we will create a set of nodes M where each node $m_{ij} \in M$ represents the remaining matches to be played between teams i and j. There will be source node s and a sink node t. Each team will be connected to s with an edge (s, t_i) with capacity x_i. Here $x_i = w_r^* - w_i - 1$ is the maximum number of games that a team can win before having more wins than the Red Sox. Each match will be connected to the sink with edges (m_{ij}, t) with capacity q_{ij}. Finally, each team i will be connected to all matches m_{ij} that it participates in (i.e. team 1 will be connected to all m_{ij} where either $i = 1$ or $j = 1$) with an edge (t_i, m_{ij}) of capacity q_{ij}. Notice that there are a total of $\binom{n}{2} = n(n-1)/2$ matches, each with 2 incoming edges corresponding to the two teams that play in each match.

Now, we shall remove all edges that connect to matches that the Red Sox participated in. Thus, we remove any edges with m_{rj} or m_{ir} as ending nodes for all $j, i \in n$. We have created a graph where each match m_{ij} has two incoming edges (t_i, m_{ij}) and (t_j, m_{ij}). These edges have capacity q_{ij} and correspond the to the number of games that team t_i or t_j wins, respectively out of the matches that t_i and t_j play against each other. Since the only outgoing edge of m_{ij} is the edge (m_{ij}, t) of capacity q_{ij}, there can be at most a total flow of q_{ij} going into m_{ij} by the flow conservation property. This corresponds to the condition that if t_i wins some number l of its games against t_j, then t_j must win the other $q_{ij} - l$ games.

Now, we will find the max flow on this graph, which takes $O(n^4)$ time using Olin's algorithm, since there are $n + n(n-1)/2 = O(n^2)$ nodes and $O(n^2)$ edges, and Olin's algorithm for finding flow takes $O(EV)$ time. Because of the capacity constraints of x_i on each edge (s, t_i), we can be sure that no team will win more games than the Red Sox with a maximum flow. However, we must check to make sure that each team plays the requisite number of games. To check this, we must have $f(m_{ij}, t) = q_{ij}$ for all i, j. This condition ensures that for each match m_{ij}, the total number of possible games q_{ij} are played. If such a flow is found by the max-flow algorithm, then it is possible for the Red Sox to win the pennant by winning all their games, and having team i win a total of $\sum_{j=1}^{n} f(t_i, m_{ij})$ games each.

If the max-flow has some match m_{ij} whose flow is less than q_{ij}, then it is impossible for the Red Sox to win the pennant. The algorithm requires $O(n^4)$ running time to compute the max-flow, and $O(n^2)$ time to perform the checks on the edges. This gives a total running time of $O(n^4)$. $\square$

6.854 Advanced Algorithms

Problem Set 5

John Wang
Collaborators: Jason Hoch

Problem 5: The Re:Search engine starts from the assumption that each old and each new result had some (predicted) value v_i to the user which would be accrued if the result was in the merged list at position i. It also used experimental data to specify a change penalty p_{ij} for moving a result that was at position i in the old list to a different position j in the merged list. The system also enforces that any old item the user actually clicked on must appear somewhere in the merged result set, and also that at least k new results must appear to provide an up to date sense of the results. Given old and new result lists with all the values described above as input, give a polynomial-time algorithm for building a best merged result list of n items. In particular, formulate the problem as a min-cost max-flow problem.

Solution: We will define three sets of items: K is the set of k new results that must appear in the new merged list, N_c is the set of old clicked results of size n_c, and N_u is the set of old unclicked results of size n_u. The problem is to create a list of items of size n, using all k elements of K, all n_c elements of N_c, and $n-k-n_c$ items from N_u, to create a merged list of the best value.

We will use a bipartite graph and create a min-cost max-flow problem. The left half of the bipartite graph will be R, the set of possible results that a user obtains, and the right half of the graph will be P, where each node p_i represents the ith position in the search result. A flow from a node r_j in R to a node p_i in P will correspond to result r_j showing up in position p_i in the search results. Note that $R = \{K \cup N_c \cup N_u\}$ is the union of new results, old clicked results, and old unclicked results.

To create the bipartite flow network, we will first create two nodes s and t which will serve as source and sink nodes respectively. Each item $r_j \in R$ will be connected to s with an edge (s, r_j) of capacity 1. Each node $p_i \in P$ will be connected to t with an edge (p_i, t) of capacity 1 as well. Finally, each node r_j in R will be connected to every node in P with edges (r_j, p_i) of capacity 1 for all j and i. Notice that all edges in this graph will have capacity 1, which corresponds to the fact that only a single result should be listed in any position p_i. In order to make sure this list of results is the best, we will have to incorporate costs.

Let $c(u, p_i)$ be the cost function for edge (u, p_i) where $u \in \{K \cup N_c \cup N_u\}$ and j is the index of the previous position of item u if it was in the list. Additionally, let $v_i(u)$ be the value to the user if result u was in the merged list at position i. We will use the following definition of the cost function:

$$c(u, p_i) = \begin{cases} -v_i(u) - S & \text{if } u \in K \\ -v_i(u) + p_{ij} - S & \text{if } u \in N_c \\ -v_i(u) + p_{ij} & \text{if } u \in N_u \end{cases} \tag{2}$$

Where S is chosen so that $c(u_1, p_i) < c(u_2, p_i)$ for all $u_1 \in K \cup N_c$ and $u_2 \in N_u$. We can achieve this by setting S such that the following inequality holds:

$$\max_{i,j} \left(-v_i(u_1) + p_{ij} - S\right) < \min_{i,j} \left(-v_i(u_2) + p_{ij}\right) \tag{3}$$

This ensures that any edge (u_1, p_i) where $u_1 \in N_c, K$ will have a lower cost than any edge (u_2, p_i) where $u_2 \in N_u$. This means that all k new results and n_c old clicked on results will have a higher precedence than any of the old unclicked results. When we find the min-cost max-flow, we will obtain a flow of size n whose cost is minimized. This means that all edges (u, p_i) with $u \in K$ or $u \in n_c$ will have precedence over any edges where $u \in N_u$. Since we have assumed that the number of slots in the merged list exceeds $k + n_c$, we know that all new results and old clicked results will be in the solution of the min-cost max-flow. Moreover, we know that the new results and the old clicked on results will be placed in positions that minimized the cost function (which therefore maximize the value which is the negative of the cost function).

Therefore, a min-cost max-flow will provide n results in the best merged list which have the highest value to users, incorporates k new resulsts, and retains all of the n_c old clicked results. Min-cost max-flow problems can be solved in polynomial time based on algorithms given in class. Therefore, we have found a polynomial time algorithm for building a best merged result list of n items. □