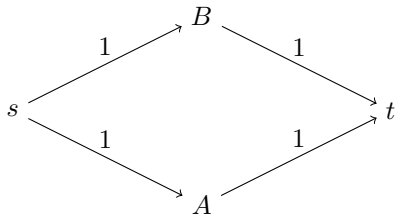
6.854 Advanced Algorithms

Problem Set 5

John Wang
Collaborators: Jason Hoch, Ryan Liu

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, Ryan Liu

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: We will first find out if there exists a feasible flow, then return the feasible flow if it exists. We will do this by creating two super nodes A and B which will act as a source and sink, respectively, and by creating a new graph G' based on the original graph G. We shall take all of the edge capacities on G and use the edge capacities $u'_{ij} = u_{ij} - l_{ij}$ in our new graph G'.

Each edge (i, j) in G' will will now have capacity u'_{ij} based on the augmented edge capacity given above. In addition, we will add new edges from A and B to each edge. For each edge (i, j) (where there is a directed edge from node i to j), we will create a directed edge (A, j) of capacity l_{ij} and a directed edge (i, B) also of capacity l_{ij}. Therefore, we will have an edge (A, j) going from the new source to the end node of every pre-existing edge (i, j), as well as a new edge (i, B) going from the beginning node of every pre-existing edge (i, j) to the new sink. Finally, we will add an edge (t, s) of infinite capacity going from the original sink to the original source.

Now, we will run a max-flow algorithm through this graph G' using A as the source and B as the sink. We then take the flow going along each edge (i, j) in G' which also exists in G and send flow $l_{ij} + f_{G'}(i, j)$ over that edge. Therefore, we set $f_G(i, j) = l_{ij} + f_{G'}(i, j)$. If each of the edges (A, i) and (j, B) are saturated for all $i, j \in V(G)$, then there exists a feasible flow, otherwise there does not exist a feasible flow. Now we shall prove that f_G is a feasible and valid flow, and that if there does exist a feasible and valid flow, then this algorithm will find it.

In order to show the first assertion, we must note that f_G will always be greater than l_{ij} for any edge (i, j) since $f_{G'}(i, j) \geq 0$. Thus, we only have to prove that f_G is a valid flow, i.e. that it satisfies flow conservation and the edge capacities. We know that $f_G(i, j) < u_{ij}$ because we know that $f_G(i, j) = l_{ij} + f_{G'}(i, j) \leq l_{ij} + (u_{ij} - l_{ij}) = u_{ij}$. Therefore, we can be sure that the newly constructed flow will not be larger than the capacity on any edge. Thus, we only need to show that flow conservation holds, so that $\sum_i f_G(i, u) = \sum_j f_G(u, j)$. We know that all of the edges (A, i) are saturated so that any flow $f_{G'}(i, j)$ will saturate the edges into (j, B) into the sink B. Since $f_{G'}(i, j)$ is a valid flow in G', it is clear that the flow will saturate the edges (i, j) originally in G so that $l_{ij} + f_{G'}(i, j)$ will become a valid flow.

Therefore, we have found a feasible flow if all of the edges (A, i) are saturated. Now, we must find a minimum flow. This can be done by creating another graph G^* which consists of the all of the edges $(i, j) \in G$ reversed (so that the edges go from t to s). The capacities will be set at $f_G(j, i) - l_{ji}$. Basically, we are looking at how much flow this edge can lose while still being greater than the minimum flow l_{ji}. Then we can find a max flow in G^* from t to s.

This new max flow in G^* will tell us how much flow we can subtract away from f_G while still following the fact that $f_G(i, j) \geq l_{ij}$. Thus, once we set $f(i, j) = f_G(i, j) - f_{G^*}(i, j)$, we will have a valid flow (since both f_G and f_{G^*} are valid flows), and we know it must be minimum because it is the maximum amount of flow that can be removed while still satisfying the lower bounds on each edge. □

Problem 2-b: Prove the following min-flow max-cut theorem. Define the lower capacity bound of an s-t cut $(S, T = V \setminus S)$ as $\sum_{(i,j)\in S\times} l_{ij} - \sum_{(i,j)\in T\times S} u_{ij}$. Show that the minimum value of all feasible flows from node s to node t is equal to the maximum lower capacity bound over all $s - t$ cuts.

Solution: We need to show that for any flow f and for any s-t cut (S, T), we must have $f \geq \sum_{(i,j)\in S\times} l_{ij} - \sum_{(i,j)\in T\times S} u_{ij}$. We know that any flow ust satisfy the minimum capacities, so we know that the flow from S to T is at least $\sum_{(i,j)\in S\times T} l_{ij}$. Moreover, we know that the most flow that can possibly be going from T to S is given by the sum of the maximum capacities. This is $\sum_{(i,j)\in T\times S} u_{ij}$. This means

that each flow f must be greater than $\sum_{(i,j)\in S\times} l_{ij} - \sum_{(i,j)\in T\times S} u_{ij}$. Now, we must show that there exists a flow f which equals the lower bound on the cut capacity. Suppose that f is now a minimum flow and we have $f > \sum_{(i,j)\in S\times} l_{ij} - \sum_{(i,j)\in T\times S} u_{ij}$. This means that either $f > \sum_{(i,j)\in S\times T} l_{ij}$ or that $\sum_{(i,j)\in T\times S} u_{ij}$ is not maximal. This means that the residual graph with edges of capacity $u_{ij} - l_{ij}$ will have extra capacity from t to s on the reverse edges. Since there is capacity available across any cut along this path, we can find an augmenting path in this graph, which means we can decrease the flow. However, this means that f is not a minimum flow, which is what we assumed. This is a contradiction, so $f = \sum_{(i,j)\in S\times} l_{ij} - \sum_{(i,j)\in T\times S} u_{ij}$ if f is a minimum flow. □

Problem 2-c: A group of students wants to minimize their lecture attendance by sending only one of the group to each of n lectures. Lecture i begins at time a_i and ends at time b_i. It requires r_{ij} time to commute from lecture i to lecture j. Do not assume these times are integers. Develop a min-flow based algorithm for identifying the minimum number of students needed to cover all the lectures.

Solution: We will create a graph G for which we will find the min-flow. There will be a source s and a sink t. Each lecture will have an edge (a_i, b_i) on the graph G. The capacity of (a_i, b_i) will be infinite, but the minimum flow will be $l_i = 1$. We will connect the source to each lecture with an edge (s, a_i) with infinite capacity and minimum flow $l_{sa_i} = 0$. We will also connect each lecture with the sink using an edge (b_i, t) with infinite capacity and minimum flow $l_{b_i t} = 0$. Finally, we will create edges (b_i, a_j) if $b_i + r_{ij} \leq a_j$. In other words, we will connect the end of lecture i with the beginning of lecture j if it is possible to complete lecture i and also attend j. The capacity of this edge (b_i, a_j) will be infinite with minimum capacity of $l_{b_i a_j} = 0$.

Finding the minimum flow over this graph will give the minimum number of students needed to cover all the lectures. This is because each lecture needs to be attended by at least one student, given by the requirement $l_{a_i b_i} = 1$ (the minimum cost of any edge (a_i, b_i) is 1). Here, units of flow represent students. Since each lecture requires one unit of flow (a single student), and students can move from one lecture i to another j only if $b_i + r_{ij} \leq a_j$, we can be sure that the min-flow algorithm will find the minimum number of students required to attend all the lectures. This is because the only way students can attend a lecture are either by coming from another lecture i where feasible (i.e. that $b_i + r_{ij} \leq a_j$) or by going to their first lecture. Both of these mechanisms are capture by the flow network. □

6.854 Advanced Algorithms

Problem Set 5

John Wang
Collaborators: Jason Hoch, Ryan Liu

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: Consider the following algorithm. For each faculty member in the bipartite graph G whose outdegree is less than s, we will create imaginary edges e' to the student with the lowest number of incoming edges (if there are multiple students with this number, choose the student arbitrarily). Keep increasing the number of outgoing edges of each faculty member until the degree equals s, each time choosing the student with the lowest current number of incoming edges (after taking into account the new imaginary edges that have been added).

It is clear that the number of edges emanating from each faculty member will now be s. However, we can now see that the number of incoming edges into each student is exactly s as well. This is because we will increase the number of edges by some number $E = \sum_{i=1}^{n} s - deg(v_i)$ where $v_i \in F$. We know that there are exactly E edges from faculty to students, which means the sum of incoming edges for all student nodes is $\sum_{i=1}^{n} deg(v_i)$. This means that increasing the number of total edges by E increases the number of incoming edges for students to $\sum_{i=1}^{n} s - deg(v_i) + deg(v_i) = \sum_{i=1}^{n} s$. Since we always pick the lowest in-degree student node, each student will contain $\frac{1}{n}\sum_{i=1}^{n} s = s$ in-degrees.

Therefore, we have created a new graph with n faculty members and n students, each with exactly s edges. This means we can apply part b from above, and we can schedule all the meetings in this graph in s time slots. However, we have some extra edges which correspond to fake meetings that have been just created in this graph. Instead of having these meetings, however, we will just disregard them. This does not change the fact that all of these meetings can be held in s time. However, by disregarding these meetings, we are able to schedule all meetings in the original graph in s time slots, which solves our problem. □

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, Ryan Liu

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, Ryan Liu

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. There will be two intermediate nodes K and N, where K represents all the new results that are available and N representss all of the previous results. We will have an edge from s to K with capacity n, and an edge from s to N with capacity $n-k$. These intermediate nodes will assure that once we find the max flow through bipartite flow network, we will obtain a maximum of $n-k$ old results, which means we will obtain at least $n-(n-k)=k$ new results in the merged list.

Each item $r_j \in R$ will be connected to N or K with an edge (N, r_j) or (K, r_j) of capacity 1, depending on whether $r_j \in N$ or $r_j \in K$ respectively. 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:

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

Where S is chosen so that $-v_i(u) + p_{ij} + S$ for any $u \in N_c$ is greater than $-v_i(u) + p_{ij}$ for any $u \in N_u$. This ensures that the min-cost flow will choose the already clicked on items first (i.e. all of the items $u \in N_c$). This means that all previously clicked items will have precedence over any unclicked items in the set of nodes connected to the node N.

When we find the min-cost max-flow, we will obtain a flow of size n whose cost is minimized. 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 at least 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. □