COL351: Analysis and Design of Algorithms **Fall 2024**

Tutorial Sheet 12

Announced on: Oct 27 (Sun)

- This tutorial sheet contains problems in network flow. When presenting your solutions, please draw/describe the flow network clearly. The correctness and running time arguments, if required, can be brief (1-2 sentences).
- Problems marked with ($\star$) will not be asked in the tutorial quiz.

Note: For some of the problems, we've only provided the main ideas and not the full solutions. We expect you to fill the gaps by yourself. If there are any doubts, you can contact your tutorial TA, head TA or the instructor.

---

1. In a standard $s-t$ maximum flow problem, we assume edges have capacities, and there is no limit on how much flow is allowed to flow through a node. In this problem, we consider a variant of the maximum flow problem with node capacities (and no edge capacities). Let $G = (V, E)$ be a directed graph, with source s, sink t, and nonnegative node capacities u_v for each $v \in V$. Given a flow f in this graph, the flow through a node v is defined as

$$\sum_{e \in \delta^-(v)} f_e,$$

where $\delta^-(v)$ denotes the edges coming into v. We say that a flow is feasible if it satisfies the usual flow-conservation constraints and the node-capacity constraint: the flow through a node v cannot exceed u_v. Design a polynomial-time algorithm to find a $s-t$ maximum flow in such a node-capacitated network.

One can reduce this problem to the standard edge capacity instance on a new graph H, which is defined as follows:

- For each vertex $v \in V,\ v \neq s, t$, split v into 2 vertices v_{in} and v_{out}, and add an edge of capacity u_v from v_{in} to v_{out}.
- For an edge $(s, v) \in E$ ($(v, t) \in E$) , add an edge (s, v_{in}) ((v_{out}, t)) of ∞ capacity in H.
- For any edge $(u, v) \in E$ such that $u, v \neq s, t$, add an edge of ∞ capacity from u_{out} to v_{in}

It can be seen that the $s-t$ max flow in the edge capacitated network H corresponds

to $s - t$ max-flow in the node capaciated network G. (**Try it yourself**).

Time Complexity: One needs to find the max-flow in H, which has $O(n)$ vertices and $O(m)$ edges and hence, max-flow in H can be found in polynomial time using Edmond-Karp algorithm.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Claiming that one can reduce the node capacity instance to an edge capacity instance - 0.25 points
  - Forming the correct edge capacity instance from a given node capacity instance - 1.5 points
  - Brief justification of why the max-flow in node capacity instances corresponds to the max-flow in the edge capacity instance - 1 point
  - Brief justification of Polynomial Time complexity - 0.25 points

2. Let M be an $n \times n$ matrix with each entry equal to either 0 or 1. Let m_{ij} denote the entry in row i and column j. A diagonal entry is one of the form m_{ii} for some i. *Swapping* rows i and j of the matrix M denotes the following action: We swap the values m_{ik} and m_{jk} for every $k \in \{1, 2, \ldots, n\}$. Swapping two columns is defined analogously. We say that M is *re-arrangeable* if it is possible to swap some of the pairs of rows and some of the pairs of columns (in any sequence) so that after the swapping, all the diagonal entries of M are equal to 1.

   a) Give an example of a matrix M which is not re-arrangeable but for which at least one entry in each row and each column is equal to 1.

   Consider M to be the below 3×3 matrix.
   $$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

   b) Give a polynomial-time algorithm that determines whether a matrix M with $0 - 1$ entries, is re-arrangeable.

   Reduction: Consider each row of M as a man and each column as a woman. Each 1 in M represents a man-woman compatible couple. The problem is now reduced to finding the perfect matching amongst the compatible couples.
   Draw a bi-partite graph G where we have n vertices on each side (men vertices on the left side and women vertices on the right). We have an edge (i, j) iff $M[i, j] = 1$.

Claim: M is *re-arrangeable* iff G has a perfect matching.

Brief justification:
Making the $M[i,i] = 1$ (after multiple row/column swapping) implies matching the i^{th} man with i^{th} woman (in this specific ordering). Each row/column swapping is equivalent to swapping the position of man/woman in the corresponding row/column ordering. Making $M[i,i] = 1 \; \forall \; i \in [0, n-1]$ implies finding the perfect matching in G.
If edge $(i,j) \in$ maximum matching of G, then $\exists$ a series of row/column swapping which causes the matching of i^{th} man with the j^{th} woman in the original ordering.

Time complexity: Any max-flow algorithm can be used to find the Maximum Bipartite Matching (MBM). We could use the Ford-Fulkerson algorithm which is polynomial time algorithm.

- Written "I do not know how to approach this problem" - 0.6 points
- – Claiming that one can reduce the given problem to an instance of finding perfect matching in bipartite graphs - 0.25 points
  – Forming the correct bipartite graph G from a given matrix M - 1.5 points
  – Brief justification of why M is re-arrangeable iff G has a perfect matching - 1 point
  – Brief justification of Polynomial Time complexity - 0.25 points

3. You have a collection of n software applications, $\{1, \ldots, n\}$, running on an old system, and now you would like to port some of these to the new system. If you move application i to the new system, you expect a net (monetary) benefit of $b_i \geqslant 0$. The different software applications interact with one another; if applications i and j have extensive interaction, then you will incur an expense if you move one of i or j to the new system but not both – let's denote this expense by $x_{ij} \geqslant 0$. So, if the situation were really this simple, you would port all n applications, achieving a total benefit of $\sum_i b_i$.

   Unfortunately, there's a problem. Due to small but fundamental incompatibilities between the two systems, there's no way to port the application 1 to the new system; it will have to remain on the old system. Nevertheless, it might still pay to port some of the other applications, accruing the associated benefit and incurring the expense of the interaction between applications on different systems. So, here's the question: Which of the remaining applications, if any, should be moved? Give a polynomial-time algorithm to find a set $S \subseteq \{2, \ldots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

Solution: This can be solved by the min-cut algorithm. We first create an undirected graph as follows: It has a vertex for every software application – call these $v_1, \ldots, v_n$, where v_i corresponds to application i. We also have a special vertex t. If applications i and j have an associated value x_{ij}, then we have an edge between v_i and v_j of capacity x_{ij}. For every vertex v_i, $i \neq 1$, we have an edge (v_i, t) of capacity b_i.

Now, we show that a v_1-t min-cut corresponds to the desired solution. Let X be a $v_1 - t$ cut. Then, the capacity of X is

$$\sum_{i,j:v_i \in X, v_j \notin X} x_{ij} - \sum_{i:v_i \notin X} b_i + \sum_v b_v.$$

Note that it is exactly the expense minus benefit of moving the applications which are not in the set X (plus a term which is fixed). Thus, finding a min-cut is same as finding the set of applications for which expense minus benefit is minimized, or benefit minus expense is maximized.
Time Complexity: Using any polynomial-time max-flow algorithm, the given problem can be solved in polynomial time.

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Noting that maximising the benefit minus expense is equivalent to minimising the expense minus benefit - 0.25 points
  - Forming the correct min-cut instance with correct capacities - 1.5 points
  - Brief justification of why the min-cut instance corresponds to minimising the expense minus benefit - 1 point
  - Brief justification of Polynomial Time complexity - 0.25 points

4. Extend the proof of Hall's theorem to show that, for every bipartite graph $G = (V \cup W, E)$ with $|V| \leqslant |W|$,

$$\text{maximum cardinality of a matching in } G = \min_{S \subseteq V}(|V| - (|S| - |N(S)|)).$$

It is easy to see that for any $S \subseteq V$:

$$\text{maximum cardinality of a matching in } G \leqslant (|V| - (|S| - |N(S)|))$$

To complete the proof, we now show that there exists a set $S^* \subseteq V$ such that

$$\text{maximum cardinality of a matching in } G = (|V| - (|S^*| - |N(S^*)|))$$

Proof. Recall from the Lectures that the maximum bipartite matching between can be found as maximum flow in the following directed network H:

- Add a source s, adding edges of unit capacity from s to every vertex of V. Similarly, add a sink t and add edges of unit capacities from every vertex of W to t.
- Direct the edges of the bipartite graph from V to W and give them $+\infty$ capacity.

Consider any $s-t$ min-cut (X, Y) of H. Note that size of (X, Y) is equal to the size of maximum matching in G.

Let $S^* = V \cap X$. It can be seen that there are no edges from S^* to $W \cap Y$, otherwise the cut X, Y would have ∞ capacity. Hence, $N(S^*) = W \cap X$. So, the capacity of (X, Y) is equal to the capacity of edges outgoing edges of X which is equal to $|V| - |S^*| - |N(S^*)|$.
Hence, Proved.

□

5. Recall that a vertex cover is a subset S of vertices such that every edge is incident to some vertex in S. Prove that the problem of computing a minimum-cardinality vertex cover in bipartite graphs can be solved in polynomial time.

Consider the bipartite graph $G = (V, E)$ where $V = A \cup B$. We will prove the following theorem, whose proof will implicitly give us an algorithm to find minimum vertex cover in bipartite graphs.

Konig's Theorem: If G is bipartite, then the maximum cardinality of a matching equals the number of vertices in a minimum vertex cover of G.

Proof. Consider any vertex cover $U \subseteq V$ and any maximum cardinality matching $M \subseteq E$. Since U covers all edges of G, it necessarily covers all edges of M. Since M is a matching, no two edges of M share an endpoint. So, we can infer that $|M| \leqslant |U|$.

To complete the proof, we now show that there exists a set $U^* \subseteq V$ such that $|M| = |U^*|$.
We create a directed flow network H to find the maximum cardinality matching, as described in the previous question (see Figure 1 for ease of understanding).

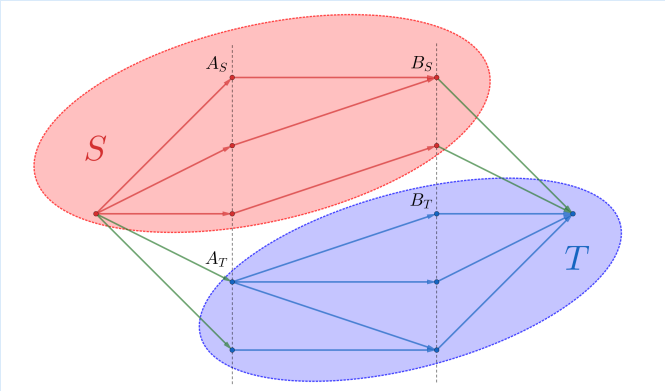


Figure 1: Minimum cut (S, T) in a flow network induced by a bipartite graph

Consider the $s - t$ min-cut (S, T) of H. Let $A_S = A \cap S$, similarly define A_T, B_S, B_T. As argued in the previous question, there is no edge from A_S to B_T and hence, $A_T \cup B_S$ is a valid vertex cover for G. Also, note that:

$$|A_T \cup B_S| = |A_T| + |B_S| = capacity(S, T) = |M|$$

Hence, $U^* = A_T \cup B_S$ is the minimum vertex cover for G.

□

Putting it all together, the minimum vertex cover is $A_T \cup B_S$, and it can be easily found from the minimum cut:

a) Find a minimum cut (S, T) in the flow network of the maximum matching on bipartite graph with parts A and B.

b) A minimum vertex cover is comprised of the sets $A_T = (A \cap T)$ and $B_S = (B \cap S)$.

Time Complexity: Since min-cut can be computed in polynomial time, it can be easily shown that the minimum vertex cover of bipartite graphs can also be computed in polynomial time.

6. ($\star$) In sociology, one often studies a graph G in which nodes represent people and edges represent those who are friends with each other. Let's assume that friendship is symmetric for this question, so we can consider an undirected graph. Now, suppose we want to study this graph G, looking for a close-knit group of people. One way to formalize this notion would be as follows. For a subset S of nodes, let $e(S)$ denote the number of edges in S, i.e., the number of edges with both ends in S. We define the cohesiveness of S as $e(S)/|S|$. A natural thing to search for would be the set S of people achieving the maximum cohesiveness. Give a polynomial-time algorithm that takes a rational number α and determines whether there exists a set S with cohesiveness at least α. Give a polynomial-time algorithm to find a set S of nodes with maximum cohesiveness.

Let $G = (V, E)$ denote the given graph. Construct a directed graph H, which has one vertex x_v for every vertex $v \in V$, one vertex y_e for every edge $e \in E$, and two special vertices s and t. We have edges (s, y_e) of capacity 1, and edges (x_v, t) of capacity α. Further, if $e = (u, v)$, is an edge in G, then we have directed edges (y_e, x_u) and (y_e, x_v) of infinite capacity.

Now we claim that there is a set S of cohesiveness at least α if and only if the minimum $s - t$ cut in H has capacity at most $|E|$. To prove the 'if' direction, consider any minimum $s - t$ cut A in the directed graph H. We are given that the capacity of A is at most $|E|$. Let A_e denote the vertices of type y_e in A and A_v denote the vertices of type x_v in A. Note that if $y_e \in A$, where $e = (u, v)$, then both $x_u, x_v \in A$. (If not, then the capacity of the cut will be infinite.) Thus, $e(A_v) \geqslant |A_e|$. Now, the capacity of this cut is $|E| - |A_e| + |A_v| \cdot \alpha \geqslant |E| - e(A_v) + |A_v| \cdot \alpha$. Thus, if the capacity of the min-cut is at most $|E|$, then $|e(A_v)|/|A_v| \geqslant \alpha$, and so, A_v is the desired set.

Next, to show the 'only if' direction, let A_v be a set of vertices of cohesiveness at least α. Consider the cut A consisting of A_v and y_e where e has both endpoints in A_v. The capacity of this cut is $|E| - e(A_v) + |A_v| \cdot \alpha \leqslant |E|$, and so, the min-cut has capacity at most $|E|$.

Finally, we can find the set of maximum cohesiveness by binary search on α. Note that the cohesiveness of any set is a fraction of the form $\beta/n!$, where β lies between 0 and $n \cdot n!$. Thus, we can perform a binary search on β; the time taken will be $\log(n \cdot n!)$, which is $\mathcal{O}(n \log n)$.

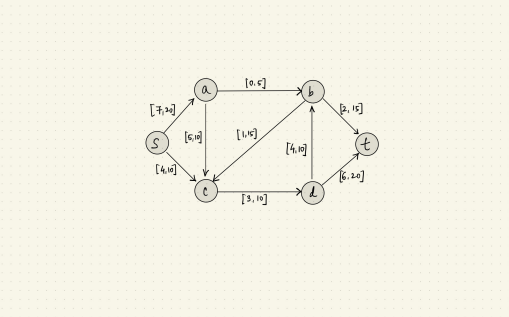
7. ($\star$) Consider the following generalization of the maximum flow problem. The input consists of (a) a directed graph $G = (V, E)$, (b) a source vertex $s \in V$ and a destination vertex $t \in V$, and (c) for each edge $e \in E$, there is an *upper* bound u_e and a *lower* bound ℓ_e on the flow through that edge. A flow $f := \{f_e\}_{e \in E}$ is said to satisfy capacity constraints if, for every edge e, it holds that $\ell_e \leqslant f_e \leqslant u_e$. The conservation constraints for each vertex are defined as before. Assume that u_e and ℓ_e are nonnegative integers. The goal, as before, is to compute a feasible flow with maximum value in the given graph.

   Show that the maximum flow problem with both upper and lower bounds on capacities can be reduced to the maximum flow problem with only upper bounds on capacities. That is, given an instance of the generalized maximum flow problem (with both upper and lower bounds), show how to construct an instance of the standard maximum flow problem (with only upper bounds) such that given a maximum flow in the instance of the standard problem, you can recover a maximum flow in the instance of the generalized problem.

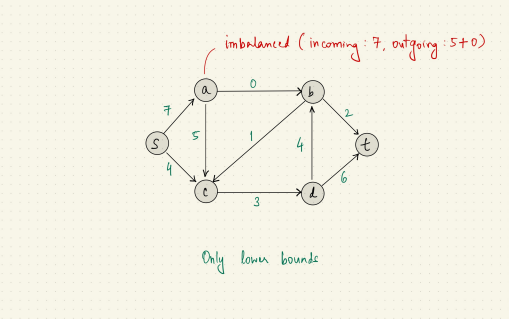
   The main idea is to transform the given instance $\mathcal{I} = \langle G = (V, E), \{\ell_e\}_{e \in E}, \{u_e\}_{e \in E} \rangle$ with both lower and upper bounds on capacities into an instance $\mathcal{I}' = \langle G' =$

$(V', E'), \{u_e\}_{e \in E'}\rangle$ with only upper bounds on capacities, and show that $\mathcal{I}$ admits a *feasible* flow if and only if $\mathcal{I}'$ admits a *saturating* flow.
We will illustrate the construction through an example and leave the details for general graphs as an exercise.

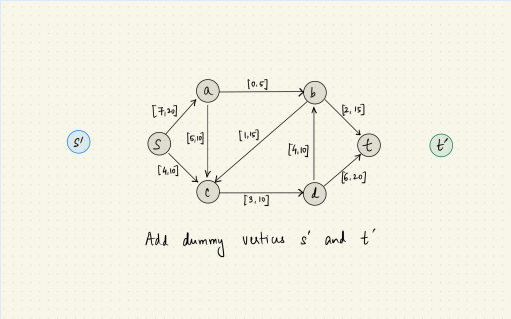
a) Consider an instance with lower and upper bounds as shown below. The notation "$[\ell_e, u_e]$" denotes a lower bound of ℓ_e and an upper bound of u_e on the capacity.



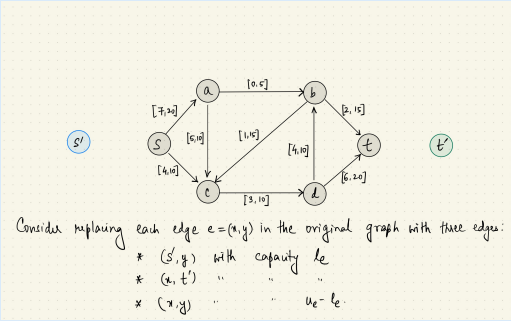
b) Observe that if we only consider the flow pertaining to the lower bounds, the instance may not admit a feasible flow. (This contrasts with the usual upper-bound-only instance where a zero-valued flow is always feasible.) Thus, in a network with lower bounds on capacities, even determining the existence of a feasible flow is an interesting problem.



c) We will create the instance $\mathcal{I}'$ by adding dummy vertices s' and t', which will serve as the source and sink of the new instance.

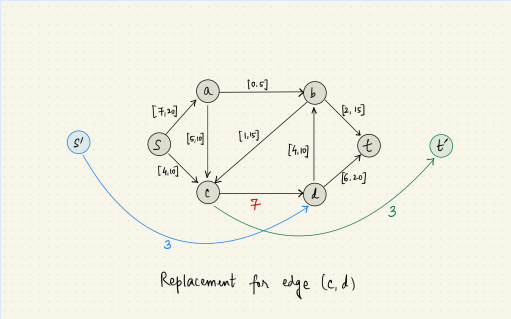


d) Next, for any edge $e = (x, y) \in E$ in the original instance $\mathcal{I}$, we will have three edges in the modified instance $\mathcal{I}'$: An edge (s', y) with upper bound ℓ_e, an edge (x, t') with upper bound ℓ_e, and an edge (x, y) with upper bound $u_e - \ell_e$.



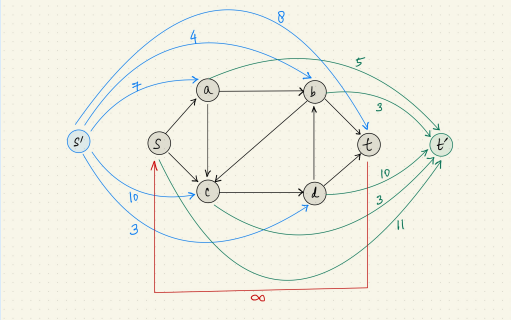
The idea behind this construction is that in a saturating flow in the new instance (if one exists), the source s' injects ℓ_e units of flow into y (simulating the lower bound for y) and the sink t' extracts ℓ_e units of flow out of x (simulating the lower bound for x). Additionally, the edge (x, y) is only responsible for determining the flow corresponding to the additional capacity of $u_e - \ell_e$.

e) The construction for a particular edge (c, d) in the network is given by:



f) We carry out these changes for every edge in the original instance. After this step, if there are multiple edges in the same direction between two vertices, we merge

such edges into a single edge by adding up their capacities. Finally, we create an edge from the original sink t to the original source s of infinite capacity. (This last step is done because the original source and sink are no longer the source and sink in the new graph, and therefore flow conservation constraints need to be satisfied for s and t.)



As an exercise, you should now check that $\mathcal{I}$ has a feasible flow if and only if $\mathcal{I}'$ has a saturating flow (i.e., the edges going out of the source s' are used to their full capacity.)