Homework 4: Flows and NP-Completeness

Handed out Thu, Apr 10. **Due at the start of class (11am), Tue, Apr 29**. (electronic submission through Gradescope.)

Problem 1. (10 points) You are given an s-t network $G = (V, E)$, where each edge $(u, v) \in E$ stores a nonnegative capacity $c(u, v)$. Given any path in G, define its *capacity* to be the minimum capacity of any edge on the path. (The trivial path, consisting of no edges has a capacity of $+\infty$.)

Present an efficient algorithm which, given G, computes the path from s to t that has maximum capacity. (If there are multiple such paths of equal capacity, return any of them.) What is your algorithm's running time? Present a formal proof that your algorithm is correct.

(Hint: This can be solved by a straightforward adaptation of Dijkstra's algorithm, particularly altering the meaning $d[v]$ for each vertex v, and how the relax operator works. You may quote bounds on the running time of Dijkstra's algorithm without proof. The main objective of this problem is for you to explicitly show how to adapt the proof of correctness from Dijkstra's algorithm to this problem.)

Problem 2. (10 points) Your friend has a new drone delivery startup, and he has asked you to help him by designing software to assist with scheduling this day's deliveries.

- There are m drone stations throughout the city. For $1 \leq i \leq m$, let d_i denote the (x, y) coordinates of the ith drone station (see Fig. 1(a)).
- There are n customers expecting to receive deliveries this day. For $1 \leq j \leq n$, let c_j denote the (x, y) coordinates of the jth customer.
- Customer j has ordered $o_j \geq 1$ deliveries. Each delivery requires a separate flight, which may come from any of the drone stations that is within 10 miles of the customer.
- FAA requirements state that each drone station can launch at total of at most 5 deliveries per day, and at most 2 flights can go from any one drone station to any one customer. (Note, however, that a customer can receive deliveries from different drone stations, but at most two per station.)
- Customers understand that drone deliveries are unreliable, and so a customer will be satisfied if at least $\max(1, o_j - 2)$ packages arrive.

A *delivery schedule* is a multiset of station-customer pairs (d_i, c_j), called *deliveries*. Such a schedule is *valid* if the following constraints are observed:

(a) Drone station d_i can deliver to customer c_j only if $\text{dist}(d_i, c_j) \leq 10$.

(b) Each drone station makes at most 2 deliveries to any single customer and a total of at most 5 deliveries overall.

(c) Customer c_j receives a total of at least $\max(1, o_j - 2)$ deliveries and at most o_j deliveries.

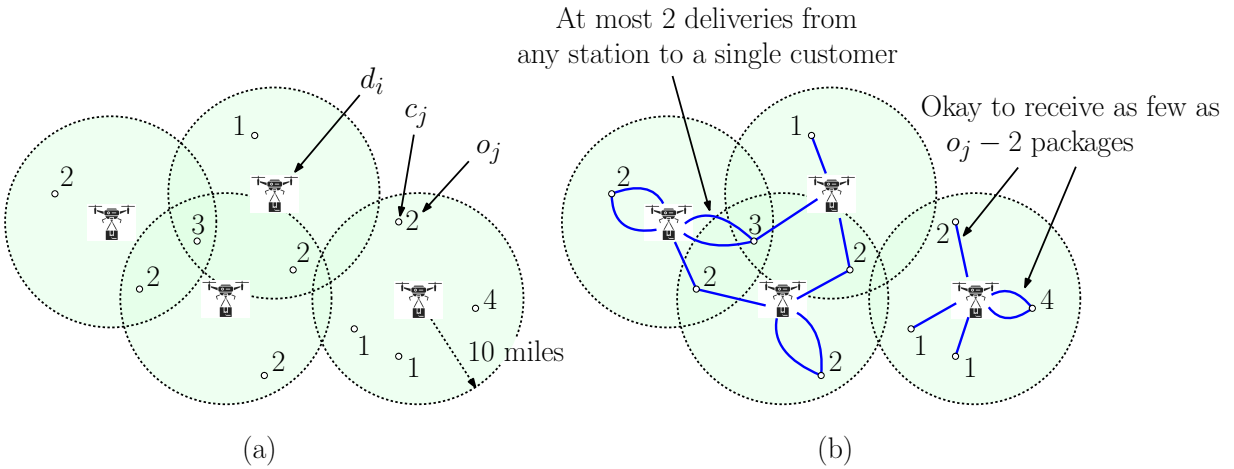

Figure 1: Drone delivery service. Black points are drone stations and hollow points are customers with package counts o_j: (a) Input and (b) Possible solution.

Present an efficient algorithm which is given arrays of drone station coordinates $d[1..m]$, customer coordinates $c[1..n]$, and package order counts $o[1..n]$. The algorithm determines whether there is a valid delivery. (Hint: Reduce to circulations. Provide a *detailed proof* of your reduction's correctness. See the Survey-Design application in Lecture 14 for an example of what we are looking for.)

Problem 3. (10 points) In this problem we will explore some examples of efficient verification algorithms. For each of the decision problems listed below, show that the problem is in NP by presenting a polynomial time verification algorithm. In each case, explain what the certificate is, and present a short description of your verification algorithm. (If the algorithm is not too technical, this description can be given in English.) Recall that you only need to verify instances where the answer to the decision problem is "yes."

(Hint: Be careful. The efficient verification algorithm is not obvious in all cases. Note that these problems are not necessarily NP-complete.)

(a) k-Hamiltonian Cycle (k-HC): You are given an undirected graph $G = (V, E)$ and an integer k. Do there exist k vertex-disjoint simple cycles in G that include all the vertices of G? (See Fig. 2(a).) For this problem, assume that a cycle must traverse at least three distinct vertices and cannot repeat any vertex, except the first and last.

(b) k-Edge Cycle Cover (k-ECC): Given a directed graph $G = (V, E)$ and a positive integer k, does there exist a subset $E' \subseteq E$ of size k such that every cycle in G passes through at least one edge of E'? (See Fig. 2(b).)

(c) Cut Cover: Consider a positively edge-weighted undirected graph $G = (V, E)$, where $w(u, v)$ denotes the weight on edge (u, v). A *cut* in such a graph is a partition of the vertex set $V = X \cup Y$, such that both X and Y are nonempty. An edge of E is said to *cross* the cut if one endpoint is in X and the other is in Y.

The Cut Cover problem is as follows: Given such a graph and a positive integer z, does there exist a subset $E' \subseteq E$ of total weight at most z such that for every cut in G at

least one edge of E' crosses the cut? (See Fig. 2(c).)

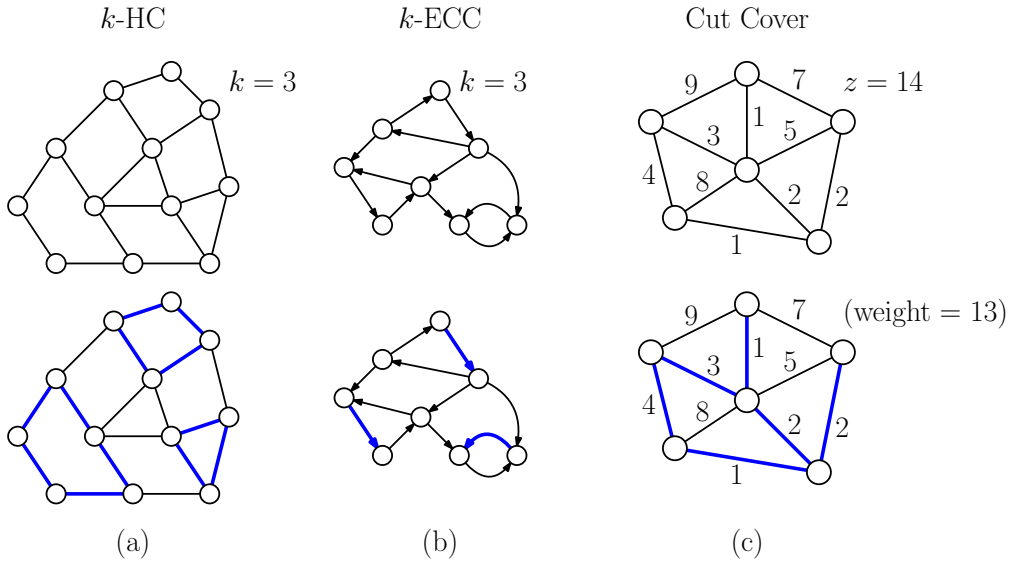

Figure 2: Verification algorithms.

Problem 4. (10 points) The *Vertex-Cover* decision problem (VC) is: given a graph $G = (V, E)$ and an integer k, where $1 \leq k \leq |V|$, does there exist a subset $V' \subseteq V$ of size k such that every edge of G has at least one endpoint in V'. (For example, the graph in Fig. 3 has a vertex cover of size 4.) This topic will be discussed in Lecture 18, but this question can be answered without knowing the material from that lecture.

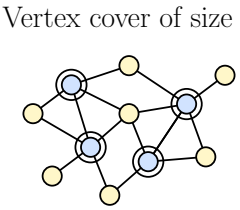

Figure 3: Vertex cover decision problem (VC).

Suppose that you had an oracle that can answer VC queries in polynomial time. (This is highly unlikely, since VC is NP-complete.) That is, given a pair (G, k), this oracle runs in polynomial time and returns "yes" or "no" depending on whether G has a vertex cover of size k. We want to convert this decision algorithm into an optimization algorithm.

(a) Explain how to use this oracle to determine the size k^* of the smallest vertex cover in G and further to identify which vertices are in the vertex cover. If there are multiple vertex covers of size k^*, your algorithm can return any of them. The number of oracle calls should be polynomial in the size of the graph.

(b) Provide a clear justification of why your procedure is correct and show that its running time is polynomial. (Hint: First apply the oracle to determine the optimum size k^*.

Next, repeatedly modify G by adding or removing edges and/or vertices and using the oracle to determine whether the size of vertex cover has changed. Take care to consider the fact that the graph may have multiple, possibly overlapping, vertex covers of the same optimal size.)

Problem 5. (10 points) Recall the *Set Cover* (SC) problem from Lecture 7. We are given a set system $\Sigma = (X, S)$, where $X = \{x_1, \ldots, x_n\}$ is a finite set of objects, called the *universe*, and $S = \{s_1, \ldots, s_m\}$ is a collection of subsets of X. We assume that every element of X belongs to at least one set of S. The Set-Cover decision problem (SC) is: given a set system $\Sigma = (X, S)$ and an integer k, where $1 \leq k \leq |S|$, does Σ have a set cover consisting of k sets?

Recall the VC problem from Problem 4.

(a) Prove that VC $\leq_P$ SC. That is, show that there is a polynomial-time procedure which given an instance (G, k) of VC, generates an instance (Σ, k') of SC, such that G has a vertex cover of size k if and only if Σ has a set cover of size k'. Your solution should include the following elements:

- Present a procedure that maps (G, k) to (Σ, k'). (In particular, as a function of the vertices and edges of G, what are the elements of X and what are the sets S? What is the value of k'?)
- Explain briefly why your procedure runs in polynomial time in the size of G.
- Present a proof of the claim that G has a vertex cover of size k if and only if Σ has a set cover of size k'.

(b) In class, we showed that the greedy set-cover approximation produces a cover whose size is larger than the optimum by a factor of $\ln n$, where $n = |X|$. Given the result from (a), what does this imply about the approximability of the vertex cover problem.

In particular, describe this vertex cover approximation algorithm, in terms of G alone (without making reference to your reduction) and describe the size of the resulting vertex cover with respect to the size of the optimal vertex cover. Briefly justify your answer. (Note that this does not yield the best approximation bound for vertex cover. But here we just want you do indicate what is implied by the reduction from (a).)

(Note: Challenge problems count for extra credit points. These additional points are factored in only *after* the final cutoffs have been set and can only increase your final grade.)

Challenge Problem. In the drone problem, you were asked to determine whether there is a valid delivery schedule. But, if there are multiple valid schedules, we would naturally prefer one that maximizes the number of packages delivered. In this problem, we'll see how to achieve this.

(a) In Lecture 14 we showed that the problem of computing a circulation in a network G could be reduced to the max-flow problem in an s-t network G'. We showed that if G has a valid circulation, then a maximum flow f' in G' could be transformed into such a circulation f. Unfortunately, this circulation need not be the one carrying the most flow.

Define the *total value* of a circulation to be the sum of flow values on all the edges. (Not just those crossing a cut.) Apply the circulation-to-flow reduction from Lecture 14 to the network shown in Fig. 4 and show the resulting circulation. Show that the result is *not* the circulation of maximum value by exhibiting one of greater value. (Hint: I believe that there is a circulation of total value 26.)

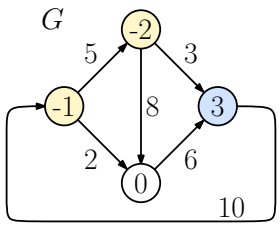

Figure 4: Circulations maximizing total flow.

(b) Returning to the drone delivery application in Problem 2, explain how modify and/or augment your construction from Problem 2 to maximize the total number of deliveries made. (Hint: This can be done by one or more invocations of a max-flow algorithm applied to an appropriate network. For full credit, see if you can do this with just a constant number of calls to max-flow. For partial credit, you can make $O(\log n)$ calls to max-flow.)