Solutions to Homework 4: Flows and NP-Completeness

Solution 1: We show how to modify Dijkstra's algorithm in order to solve the single-source maximum capacity path problem. We will focus only on the final capacities, and not on how to compute the actual path (but this is an easy extension).

For any two vertices $u, v \in V$, let $\mu(u, v)$ denote the capacity of the maximum capacity path from u to v. (If v is not reachable from u, then $\mu(u, v) = 0$.) To compute the max-capacity path from s to t, we solve the single-source max-capacity problem.

We modify Dijkstra's algorithm as follows. For each $v \in V$, define $m[v]$ to be the current estimate on the max-capacity path from s to v. Initially, $m[s] = +\infty$, and for all other vertices v, $m[v] = 0$. Because we are maximizing, rather than minimizing, the priority queue returns the maximum, rather than the minimum, element.

We also modify the relax operator $\mathsf{relax}(u, v)$ as follows. Given that $m[u]$ is the max-capacity of getting from s to u, we know that there is a path from s to v of capacity $\min(m[u], c(u, v))$. If this is larger than the current estimate, $m[v]$, we should take it. Thus, the relaxation rule for edge (u, v) is changed as follows (see Fig. 1(a)):

$$\text{relax}(u, v) : m[v] \leftarrow \max \begin{pmatrix} m[v], \\ \min(m[u], c(u, v)) \end{pmatrix}$$

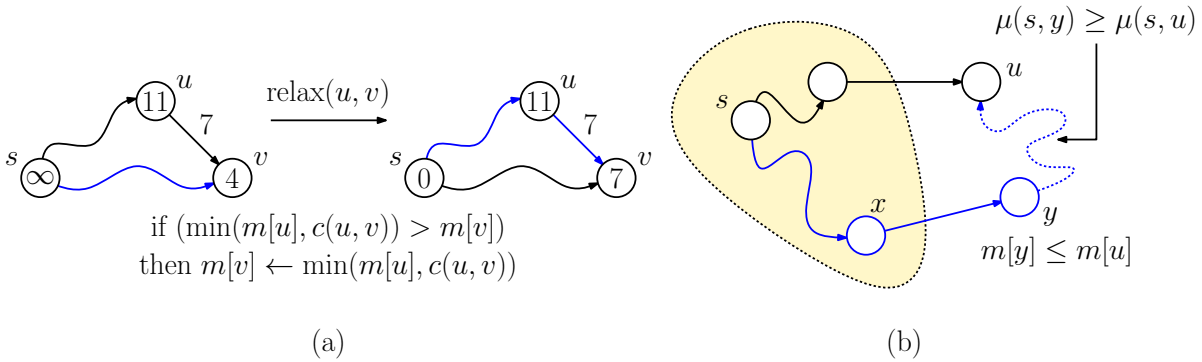


Figure 1: Adapting Dijkstra to compute the max-capacity path.

The final change to Dijkstra's algorithm is that the queue is reverse-ordered, returning the unprocessed vertex with the highest m-value. The running time is clearly the same as Dijkstra's algorithm. To establish the algorithm's correctness, we adapt the correctness proof of Dijkstra's algorithm.

Claim: Whenever the algorithm processes a vertex u, $m[u]$ contains its correct capacity value, that is, $m[u] = \mu(s, u)$.

Proof: Suppose to the contrary that this fails to be true for some vertex, and let u be the first such instance. Since $m[u]$ is based on evidence of an actual path, we have $m[u] < \mu(s, u)$.

Since this is incorrect, consider the true max-capacity path from s to u. This path must first cross an edge (x, y) where x is among the set of processed vertices and y is not (see Fig. 1(b)). (Possibly $x = s$ and/or $y = u$.)

Since x was processed earlier (and u is the first mistake), we know that $m[x] = \mu(s, x)$. Since (x, y) is an edge along the max-capacity path, we have $\mu(s, y) = \min(\mu(s, x), c(x, y))$. Since we performed $\mathsf{relax}(x, y)$, we have correctly propagated this information along the edge. This implies that

$$m[y] \;=\; m[x] + c(x, y) \;=\; \mu(s, x) + c(x, y) \;=\; \mu(s, y).$$

Since u was chosen before y to be processed, we know that $m[u] \geq m[y]$. Furthermore, any edges along the remaining max-capacity path from y to u can only decrease its capacity, implying that $\mu(s, u) \leq \mu(s, y)$. Combining these observations, we have

$$\mu(s, u) \;>\; m[u] \;\geq\; m[y] \;=\; \mu(s, y) \;\geq\; \mu(s, u),$$

which is a clear contradiction.

Solution 2: We do this via reduction to circulations with vertex demands and lower and upper flow capacities on each edge.

We generate a network $G = (V, E)$ as follows. We create a source vertex s and a sink vertex t. We create two sets of vertices, one for the drone stations, denoted $\{d_1, \ldots, d_m\}$, and one for the customers, denoted $\{c_1, \ldots, c_m\}$. We create the following edges (see Fig. 2(a)):

- For $i \in [1, m]$, and $j \in [1, n]$, edge (d_i, c_j) of capacity range $[0, 2]$ if $\text{dist}(d_i, c_j) \leq 10$.
- For $i \in [1, m]$, edge (s, d_i) of capacity range $[0, 5]$.
- For $j \in [1, n]$, edge (c_j, t) of capacity range $[\max(1, o_j - 2), o_j]$.
- Edge (t, s) of capacity range $[0, \infty]$.

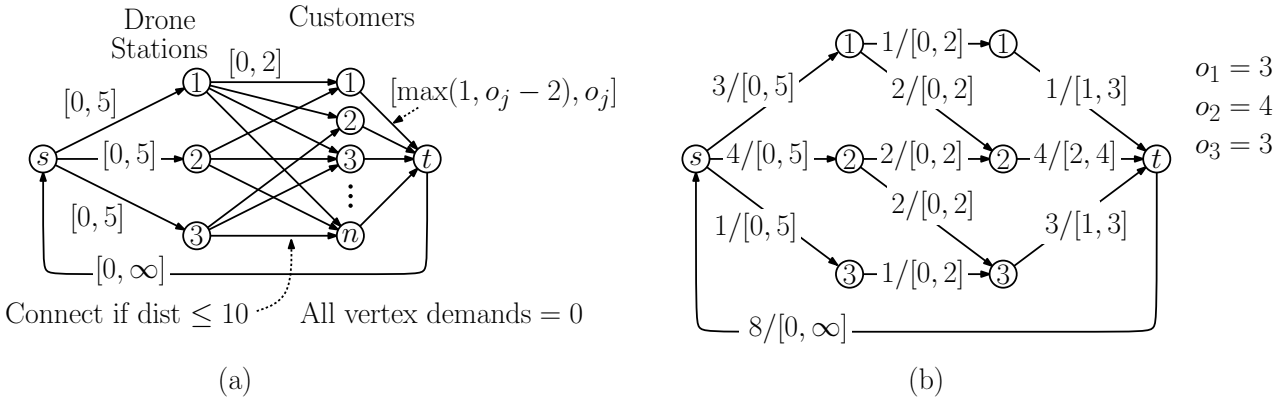


Figure 2: Solving the drone delivery problem by network flow.

All the vertex demands are set to 0, meaning that flow is conserved at every vertex. We then invoke the circulation algorithm to generate an integer-valued circulation f. If no circulation exists,

we declare that there is no valid delivery schedule. Otherwise, for each edge (d_i, c_j), we ship $f(d_i, c_j)$ deliveries from station d_i to customer c_j. (For example, in Fig. 2(b), $f(1, 2) = 2$, implying that we would ship two deliveries from d_i to c_j.) The following claim establishes the correctness of this reduction.

Claim: There exists a valid delivery schedule if and only if G has a feasible circulation.

Proof: ($\Rightarrow$) Suppose that there is a valid delivery schedule, where station d_i makes $g(d_i, c_j)$ deliveries to customer c_j. We create a circulation f in G as follows. First, for each edge (d_i, c_j), we assign it a flow of $g(d_i, c_j)$. Next, define the flow on edge (s, d_i) to be the total number of deliveries made by station d_i, and define the flow on edge (c_j, t) to be the total number of deliveries made to customer c_j. Finally, define the flow from t to s to be the total number of deliveries to all customers.

To see that this yields a feasible circulation in G, observe that each edge (d_i, c_j) receiving flow must exist, because the fact that the delivery was made implies that $\text{dist}(d_i, c_j) \leq 10$. Next, to see that the capacity constraints are satisfied, observe that the validity of the schedule implies the following:

- No single drone station sends more than 2 deliveries to any customer (satisfying the capacity constraint on (d_i, c_j)).
- No drone station sends more than 5 deliveries (satisfying the capacity constraint on (s, d_i)).
- Each customer receives between $\max(1, o_j - 2)$ and o_j deliveries (satisfying the capacity constraint on (c_j, t)).

It is also easy to see that the node demands (all zero) are satisfied, since the flow on each edge incoming to d_i is balanced by the outgoing deliveries from this station, and the flow on each edge leaving c_j is balanced by the incoming deliveries to this customer. Finally, the flow coming into and out of s and t are easily seen to equal the total number of deliveries in the entire schedule. Therefore, this is a valid circulation in G.

($\Leftarrow$) Suppose that G has a feasible circulation, denoted by f. We may assume that this is an integer flow. We define a delivery schedule as follows. For each edge (d_i, c_j) where $f(d_i, c_j) > 0$, we make this number of deliveries from station d_i to customer c_j. We will show that this is a valid delivery schedule. First, observe that flow is only sent along existing station-customer edges, meaning that they are all within the 10-mile radius. By the capacity constraints, we know that:

- $f(d_i, c_j) \leq 2$, implying that no customer receives more than 2 deliveries from any station.
- By the upper capacity constraint of 5 on edge (s, d_i) and flow conservation, no station sends more than 5 deliveries
- By the capacity constraints of $[\max(1, o_j - 2), o_j]$ on edge (c_j, t) and flow conservation, each customer receives between $\max(1, o_j - 2)$ and o_j deliveries.

Therefore, this satisfies all the requirements of a valid schedule.

Solution 3: In all cases, the certificates are straightforward, but the verification procedure is sometimes subtle.

(a) The certificate consists of k sequences of vertices, each corresponding to one of the cycles. To verify, we check that:

- every vertex of G appears in exactly one of the k sequences, and
- each is a valid cycle by verify for every consecutive pair of vertices in these sequences (wrapping around at the end), there is an edge between them in G.

If all these tests pass, the verification process accepts, and otherwise it rejects. This is clearly correct, and it is easy to see that it involves a linear number of accesses to G.

(b) The certificate consists of the set of k edges, each corresponding to an element in the cycle cover. To verify, we remove these edges from the digraph, and test whether the resulting digraph is acyclic. (Recall that this can be done by running DFS and checking that there is no back-edge in the DFS tree.) If so, the verification process accepts, and otherwise it rejects. The edges can be removed in time linear in the size of G (e.g., zeroing out entries in G's adjacency matrix and DFS runs in polynomial time.)

To see that this is correct, consider any cycle cover E' of size k. Clearly, when the edges of E' are removed from G, there can be no cycle remaining, since any remaining cycle is not covered by E' (see Fig. 3(a)). Conversely, if the removal of any set of k edges from G breaks all the cycles, then every cycle that is in G must go through at least one of these edges.

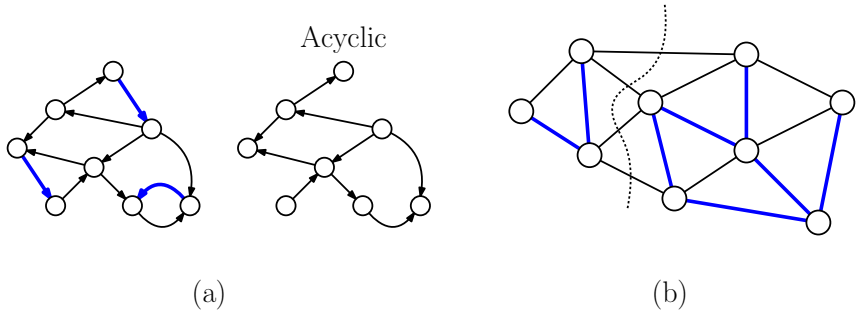


Figure 3: Verification procedures.

(c) The certificate consists of a subset E' of edges (u, v). To verify, we first check that each is indeed an edge of G and compute their sum of weights. If this sum is more than z, we reject immediately. Otherwise, we build the subgraph $G(E')$ consisting of only the edges of E'. We then check that the resulting subgraph spans G (that is, it is connected and contains all the vertices of G). If so, the verification accepts and otherwise it rejects. We can easily compute the total weight in linear time, and we can check that $G(E')$ spans G in linear time by running DFS in this subgraph.

To see that this is correct, suppose that G has a cut cover E' of weight at most z. The certificate consisting of these edges must pass the above checks. In particular, if $G(E')$ does not span G, then we could partition G's vertex set as $V = X \cup Y$, such that no edge connects

X and Y, implying that this cut is not covered (see Fig. 3(b)). On the other hand, there is a subset of edges E' of total weight at most z that spans G, these edges form a cut cover because for any partition $V = X \cup Y$, there must be at least one edge of E' that connects a vertex in x to a vertex in y, implying that every cut is covered.

(If it is not connected, then there is a cut involving the connected components that is not covered by this edge set.)

By the way, this problem is not NP-complete. This is an equivalent way to formulate the minimum-spanning tree decision problem! Observe that any spanning subgraph covers every cut, and hence the minimum spanning tree, the lowest weight spanning subgraph, is the cut cover of minimum cost.

Solution 4: We will do parts (a) and (b) together. The size k^* of the smallest vertex cover is a value between 1 and $n = |V|$. We can determine this value by performing binary search with the oracle, which involves $O(\log n)$ calls to the VC oracle. If it returns "no", we increase the size parameter and if it return "yes", we decrease it.

Once we know k^*, computing the vertices of the vertex cover is complicated by the fact that G may have multiple vertex covers. A natural idea is to remove vertices one by one and check whether there is still a vertex cover of size k^*. If there is still one, then the vertex we tested was not needed, and so we remove it. Unfortunately, this might fail if the graph has multiple vertex covers. (Suppose, for example, that G is a complete bipartite graph such as $K_{3,3}$. There are two disjoint vertex covers of size $k^* = 3$. After the removal of any vertex, a vertex cover of size 3 remains. If we are not careful, we might delete all the vertices without putting any in the vertex cover!)

The trick is handle the removal differently. For any vertex u, define $G \setminus \{u\}$ to be the graph resulting by removing u and its incident edges from G. A key observation is that whenever we remove a vertex u, there are two possibilities:

- If u is a member of *any* vertex cover of size k, then $G \setminus \{u\}$ has a vertex cover of size $k-1$ (namely, the original vertex cover minus u)
- If u is not a member of *every* vertex cover of size k, then $G \setminus \{u\}$ still has a vertex cover of size k (namely, any original vertex cover that did not include u)

This suggests the following strategy. For each vertex u, remove it and check whether $G \setminus \{u\}$ has a vertex cover of size $k-1$. If so, we add u to our vertex cover, we set $G \leftarrow G \setminus \{u\}$, and recursively check whether G has a vertex cover of size $k-1$. The code is presented below and an example is shown in Fig. 4.

Claim: If G has a vertex cover of size k, then `get-VC`(G, k) returns such a cover.

Proof: Observe that if G has a vertex cover of size k, then whenever we process a vertex u that is in any vertex cover of size k, the VC oracle will return "yes", and we will add u to V' and recursively find a vertex cover of size $k-1$ in $G \setminus \{u\}$. Since G has a vertex cover of size k, this test must succeed at least k times, after which $k = 0$ and the algorithm terminates. When the last vertex is removed, that is, $k = 1$, the oracle tells us that $G \setminus \{u\}$ has a vertex cover of size $k - 1 = 0$, which means that all the edges of G have been covered, implying that V' is indeed a vertex cover.

```
    get-VC(G, k) {                                      // find VC of size k in G
        if (VC(G, k) = "no") Error - k is wrong!
        V' = empty                                      // V' holds the VC
        for each (u in V) and while (k > 0)
            G' = G with vertex u and its incident edges removed
            if (VC(G', k-1) == "yes")                   // u is in some VC
                V' = V' + {u}                           // add u to the VC
                G = G'                                  // remove u
                k = k-1
    return V'                                           // final vertex cover
}
```

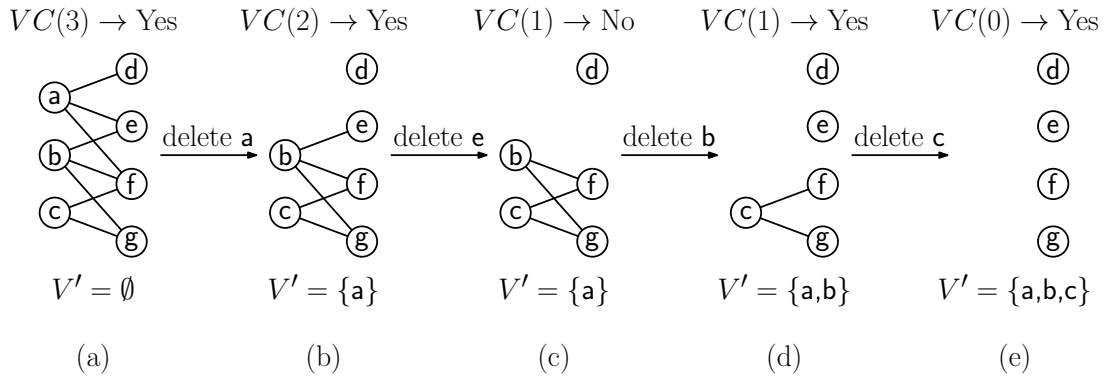


Figure 4: Finding an optimal vertex cover. The original graph has an optimal vertex cover $V' = \{a, b, c\}$ of size 3. Whenever we remove one of these vertices and invoke the oracle on $k - 1$, the test succeeds.

Finding the optimum vertex cover size takes $O(\log n)$ calls to the VC oracle, and `get-VC` makes $O(n)$ oracle calls, for a total of $O(n + \log n) = O(n)$ oracle calls. The running time of each iteration (which involves just deleting a vertex from G) is clearly polynomial in the size of G.

By the way, there is an alternative solution that does not involve vertex removal, but requires more oracle calls. We start with G and repeatedly attempt to add edges to G, as long as the addition of the edge does not alter the size of the vertex cover. The key observation is that when the algorithm terminates, the graph will have a particular structure. It will consist of k^* vertices, denoted V', that are connected to all the vertices of the graph (that is, they have degree $n - 1$). The remaining $n - k^*$ vertices are connected to just vertices of V' (that is, they have degree k^*). It is easy to see that V' is a vertex cover of size k^* in the final graph, and further, it is not possible to add an edge to this graph without increasing the size of the vertex cover. We can identify the vertices of V' by computing the degrees of all the vertices. We output V' as the vertices of the vertex cover. This algorithm makes $O(n^2)$ calls to the oracle.

Solution 5:

(a) The reduction is quite simple, since both problems involve covering a set of elements by sets. Given an instance $(G = (V, E), k)$ of VC, we generate an instance $(\Sigma = (X, S), k')$ of SC as follows. Let $X = E$ be the set of edges, and for each vertex $u \in V$, we create a set s_u

consisting of the edges that are incident to u. Clearly, this can be done in polynomial time (in fact linear time) in the size of G. Finally, we set $k' \leftarrow k$.

To establish correctness, we prove the following claim.

Claim: G has vertex cover of size k if and only if Σ has a set cover of size k'.

Proof:

($\Rightarrow$) Suppose that G has a vertex cover V' of size k. This means that for every edge $e = (u, v) \in E$, either u or v or both are in V'. From our construction, this implies that either of the sets s_u or s_v contains the edge e. Since $X = E$, it follows that the collection of sets $\{s_u\}_{u \in V'}$ constitutes a set cover for Σ of size $k = k'$.

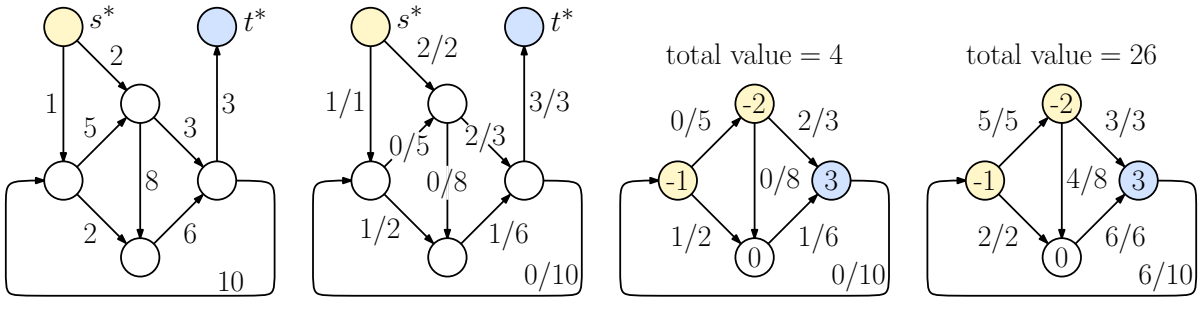
($\Leftarrow$) Suppose that Σ has a set cover S' of size k'. By construction, each set of S' is generated as the edges incident to some vertex of G. Let V' denote the k' vertices that generated these sets. The elements of the universe X are just the edges of G. Since S' is a set cover, each element $e = (u, v) \in X$, is in some set of S'. (These sets are either s_u or s_v, since these are the only sets containing this edge.) Therefore, the vertices V' cover all the edges of G, implying that G has a vertex cover of size $k' = k$, as desired.

(b) We can adapt the greedy set cover algorithm as follows. We select the vertex of G having the *highest degree*. We add this vertex to our vertex cover V', remove this vertex and all of its incident edges, and recompute the vertex degrees. We repeat until no more vertices remain.

This is a faithful simulation of the greedy set-cover algorithm in the vertex-cover context. It follows from (a) and the analysis of the greedy set-cover heuristic from class that the resulting vertex cover is larger than the optimal vertex cover by a factor of at most $\ln m$, where $m = |X| = |E|$.

Solution to the Challenge Problem:

(a) The circulation to max-flow reduction is shown in Fig. 5(a). A max-value flow is shown in Fig. 5(b) and the associated circulation is shown in Fig. 5(c), which has total value 4. (This can be increased marginally to 5 by routing flow through the edge of capacity 8.) There is, however a circulation of much larger value by saturating both of the edges coming out of the vertex of demand -1 (see Fig. 5(d)).



(a) (b) (c) (d)

Figure 5: Counterexample to Lecture 14 reduction.

(b) To obtain maximum number of deliveries for the drone problem we start with the flow solution f provided by the Lecture 14 reduction for the circulation network given in Problem 2. As shown in part (a), this need not yield the flow with the maximum value. To augment the circulation, we first convert it to a standard s-t network by removing the edge (t, s), and we then compute the residual network. For each edge (u, v), let $f(u, v)$ be the flow on this edge and let $[\ell(u, v), c(u, v)]$ denote the capacity limits for this edge. We create a new forward edge (u, v) with capacity $c(u, v) - f(u, v)$ and a backwards edge (v, u) with capacity $f(u, v) - \ell(u, v)$. We then compute the max-flow in this graph, which we denote by f'. We take the sum $f + f'$ to be the final flow. We then map this to a delivery schedule, by the same procedure described in the solution to the delivery problem.

By the same analysis we gave for the Ford-Fulkerson algorithm, the summed flow is a valid flow in updated network. We can turn this into a valid circulation by assigning the edge (t, s) a flow equal to the total flow out of s (which equals the total flow into t). We assert that this yields the maximum number of deliveries. The reason is that the sum of flows out of s is equal to the total number of deliveries, and singe f' is the maximum flow in the residual network, we have effectively maximized the total flow out of s.

There is, by the way, a solution that does not involve creating the residual network. Instead, we just modify the lower capacity bound on the edge (t, s). By setting its value to k, the existence of a circulation implies that the total number of deliveries is at least k. This means, we can perform a binary search to determine the optimum number of deliveries. Since there are $O(n)$ possible delivery values, so the binary search can be performed with $O(\log n)$ calls to a circulation oracle.