CS 170 HW 10

Due on -

1 (Dynamic Programming) Greedy Cards

Ning and Evan are playing a game, where there are n cards in a line. The cards are all face-up (so they can both see all cards in the line) and numbered 2–9. Ning and Evan take turns. Whoever's turn it is can take one card from either the right end or the left end of the line. The goal for each player is to maximize the sum of the cards they've collected.

(a) Ning decides to use a greedy strategy: "on my turn, I will take the larger of the two cards available to me". Show a small counterexample ($n \leq 5$) where Ning will lose if he plays this greedy strategy, assuming Ning goes first and Evan plays optimally, but he could have won if he had played optimally.

(b) Evan decides to use dynamic programming to find an algorithm to maximize his score, assuming he is playing against Ning and Ning is using the greedy strategy from part (a). Help Evan develop the dynamic programming solution by providing an algorithm with its runtime and space complexity.

Solution:

(a) One possible arrangement is: $[2, 2, 9, 3]$. Ning first greedily takes the 3 from the right end, and then Evan snatches the 9, so Evan gets 11 and Ning gets a miserly 5. If Ning had started by craftily taking the 2 from the left end, he'd guarantee that he would get 11 and poor Evan would be stuck with 5.

There are many other counterexamples. They're all of length at least 4.

(b) Let $A[1..n]$ denote the n cards in the line. Evan defines $v(i, j)$ to be the highest score he can achieve if it's his turn and the line contains cards $A[i..j]$.

Evan suggests you simplify your expression by expressing $v(i, j)$ as a function of $\ell(i, j)$ and $r(i, j)$, where $\ell(i, j)$ is defined as the highest score Evan can achieve if it's his turn and the line contains cards $A[i..j]$, if he takes $A[i]$; also, $r(i, j)$ is defined to be the highest score Evan can achieve if it's his turn and the line contains cards $A[i..j]$, if he takes $A[j]$. Then, we have,

$v(i, j) = \max(\ell(i, j), r(i, j))$

where

$$\ell(i, j) = \begin{cases} A[i] + v(i + 1, j - 1) & \text{if } A[j] > A[i + 1] \\ A[i] + v(i + 2, j) & \text{otherwise.} \end{cases}$$

$$r(i, j) = \begin{cases} A[j] + v(i + 1, j - 1) & \text{if } A[i] \geq A[j - 1] \\ A[j] + v(i, j - 2) & \text{otherwise.} \end{cases}$$

(The formula above assumes that if there is a tie, Ning takes the card on the left end.)

There are $n(n + 1)/2$ subproblems and each one can be solved in $\Theta(1)$ time (that's the time to evaluate the recursive formula in part (b) for a single value of i, j).

2 (Greedy Algorithms) Doctors and Patients

A doctor's office has n customers, labeled $1, 2, \ldots, n$, waiting to be seen. They are all present right now and will wait until the doctor can see them. The doctor can see one customer at a time, and we can predict exactly how much time each customer will need with the doctor: customer i will take $t(i)$ minutes.

(a) We want to minimize the average waiting time (the average of the amount of time each customer waits before they are seen, not counting the time they spend with the doctor). What order should we use? You do not need to justify your answer for this part. (Hint: sort the customers by ____)

(b) Let $x_1, x_2, \ldots, x_n$ denote an ordering of the customers (so we see customer x_1 first, then customer x_2, and so on). Prove that the following modification, if applied to any order, will never increase the average waiting time:

- If $i < j$ and $t(x_i) \geq t(x_j)$, swap customer i with customer j.

(For example, if the order of customers is $3, 1, 4, 2$ and $t(3) \geq t(4)$, then applying this rule with $i = 1$ and $j = 3$ gives us the new order $4, 1, 3, 2$.)

(c) Let u be the ordering of customers you selected in part (a), and x be any other ordering. Prove that the average waiting time of u is no larger than the average waiting time of x—and therefore your answer in part (a) is optimal.

Hint: Let i be the smallest index such that $u_i \neq x_i$. Use what you learned in part (b). Then, use proof by induction (maybe backwards, in the order $i = n, n-1, n-2, \ldots, 1$, or in some other way).

Solution:

(a) Sort the customers by $t(i)$, starting with the smallest $t(i)$.

(b) First observe that swapping x_i and x_j does not affect the waiting time customers $x_1, x_2, \ldots, x_i$ or customers $x_{j+1}, x_{j+1}, \ldots, x_n$ (i.e., for customers x_k where $k \leq i$ or $k > j$). Therefore we only have to deal with customers $x_{i+1}, \ldots, x_j$, i.e., for customer k, where $i < k \leq j$. For customer x_k, the waiting time before the swap is

$$T_k = \sum_{1 \leq l < k} t(x_l),$$

and the waiting time after the swap is

$$T'_k = \sum_{1 \leq l < i} t(x_l) + t(x_j) + \sum_{i < l < k} t(x_l) = T_k - t(x_i) + t(x_j).$$

Since $t(x_i) \geq t(x_j)$, $T'_k \leq T_k$, so the waiting time is never increased for customers $x_{i+1}, \ldots, x_j$, hence the average waiting time for all the customers will not increase after the swap.

(c) Let u be the ordering in part (a), and x be any other ordering. Let i be the smallest index such that $u_i \neq x_i$. Let j be the index of x_i in u, i.e, $x_i = u_j$ and k be the index of u_i in x. It's easy to see that $j > i$. By the construction of x, we have $T(x_i) = T(u_j) \geq T(u_i) = T(x_k)$, therefore by swapping x_i and x_k, we will not increase the average waiting time. If we keep doing this, eventually we will transform x into u. Since we never increase the average waiting time throughout the process, u is the optimal ordering.

3 (Greedy) Tree Perfect Matching

A *perfect matching* in an undirected graph $G = (V, E)$ is a set of edges $E' \subseteq E$ such that for every vertex $v \in V$, there is exactly one edge in E' which is incident to v.

Give an algorithm which finds a perfect matching *in a tree*, or reports that no such matching exists. Describe your algorithm, prove that it is correct and analyse its running time.

Solution: Let v be a leaf vertex in T. Since v has only one incident edge $e = (u, v)$, e must be in every perfect matching in T. Add e to E' and remove u, v from T. Repeat until there are no edges remaining. If there are no vertices remaining, return E', otherwise output 'no matching'.

One point to note: removing u, v might cause the graph to become disconnected. In that case we just run the algorithm on each connected component.

The running time of this algorithm is $O(|V|)$. We visit every node at most twice: once when we search into its subtree, and once when we remove it from T after it's been matched.

4 (Linear Programming) Minimum Spanning Trees

Consider the minimum spanning tree problem, where we are given an undirected graph G with edge weights $w_{u,v}$ for every pair of vertices u, v.

An *integer* linear program that solves the minimum spanning tree problem is as follows:

$$\begin{aligned}
&\text{Minimize} \quad \sum_{(u,v)\in E} w_{u,v} x_{u,v} \\
&\text{subject to} \quad \sum_{\{u,v\}\in E: u\in S, v\in V\setminus S} x_{u,v} \geq 1 \quad \text{for all } S \subseteq V \text{ with } 0 < |S| < |V| \\
&\qquad \sum_{\{u,v\}\in E} x_{u,v} \leq |V| - 1 \\
&\qquad x_{u,v} \in \{0, 1\}, \quad \forall (u, v) \in E
\end{aligned}$$

(a) Show how to obtain a minimum spanning tree T of G from an optimum solution of the ILP, and prove that T is indeed an MST. Why do we need the constraint $x_{u,v} \in \{0, 1\}$?

(b) How many constraints does the program have?

(c) Suppose that we *replaced* the binary constraint on each of the decision variables $x_{u,v}$ with the pair of constraints:

$$0 \leq x_{u,v} \leq 1, \quad \forall (u,v) \in E$$

How does this affect the optimum value of the program? Give an example of a graph where the optimum value of the relaxed linear program differs from the optimum value of the integer linear program.

Solution:

(a) $T = \{(u,v) \in E : x_{u,v} = 1\}$. The first constraint ensures that T is connected (there is at least one edge crossing every cut). The second constraint ensures that T is a tree. Moreover, every spanning tree T is a feasible solution of the ILP. The objective is the weight of T, and so the optimum is the MST. We need $x_{u,v} \in \{0,1\}$ because it's not clear what you'd do with a fractional edge.

(b) There are $2^{|V|} + |E| - 1 = \Theta(2^{|V|})$ constraints.

(c) $v_{LP} \leq v_{ILP}$. The new linear program solution's objective value v_{LP} is at most integer linear program's objective value v_{ILP}, because every feasible solution of the ILP is a feasible solution of the LP.

One example is a cycle with 3 nodes, $w_{u,v} = 1, \quad \forall u, v \in E$. The optimal ILP formulation picks any two of the edges for a total objective cost of 2. The optimal LP formulation picks $x_{u,v} = \frac{1}{2}$ for all edges, for a total objective cost of $\frac{3}{2}$.

5 (Max-Flow LP) Min Cost Flow

In the max flow problem, we just wanted to see how much flow we could send between a source and a sink. But in general, we would like to model the fact that shipping flow takes money. More precisely, we are given a directed graph G with source s, sink t, costs l_e, capacities c_e, and a flow value F. We want to find a nonnegative flow f with minimum cost, that is $\sum_e l_e f_e$, that respects the capacities and ships F units of flow from s to t.

(a) Show that the minimum cost flow problem can be solved in polynomial time.

(b) Show that the shortest path problem can be solved using the minimum cost flow problem

(c) Show that the maximum flow problem can be solved using the minimum cost flow problem.

Solution:

(a) Write min-cost flow as a linear program. One formulation is as follows:

$$\min \sum_e l_e f_e$$

$$\text{subject to } 0 \le f_e \le c_e \qquad \forall e \in E$$

$$\sum_{e \text{ incoming to } v} f_e = \sum_{e \text{ outgoing from } v} f_e \qquad \forall v \in V, v \neq s, t$$

$$F + \sum_{e \text{ incoming to } s} f_e = \sum_{e \text{ outgoing from } s} f_e$$

(b) Consider a shortest path instance on a graph G with start s, end t, and edge weights l_e. Let $F = 1$ and let $c_e = 1$ for all edges e. We will now show that in the min cost flow f, the subgraph H of G consisting of edges with nonzero flow, all directed paths from s to t have length equal to the shortest path from s to t. Therefore, to find a shortest path from s to t, it is enough to use a DFS or BFS to find any path between s to t in H.

Suppose that each $l_e > 0$ (we can contract any edge with $l_e = 0$). First, we show that H is a DAG. If H is not a DAG, it must contain a cycle. Let $\delta > 0$ be the minimum flow along any edge of this cycle. Subtracting δ from the flow of all edges in this cycle results in a flow that still ships 1 unit of flow from s to t. This flow, though, has smaller cost than before, a contradiction to the fact that f is a min-cost flow. Therefore, H is a DAG.

Consider any path P between s and t in H. Let δ be the minimum flow on any edge of this path. One can write the flow f as a linear combination of paths including P by writing

$$f = \delta p + \delta_1 q_1 + \ldots + \delta_k q_k$$

where p and q_i are the unit flows along the paths P and Q_i respectively, δ_i is the minimum amount of flow on some arbitrary path Q_i after subtracting flow from $P, Q_1, Q_2, \ldots, Q_{i-1}$. Since subtracting δ_i units of flow decreased the flow from s to t by δ_i units, $\delta + \sum_i \delta_i = 1$. Since the objective function for min-cost flow is linear, we can write

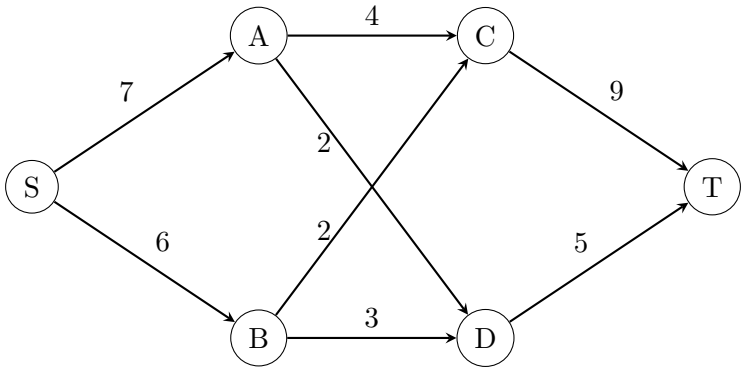
$$\sum_e l_e f_e = \delta \sum_e l_e p_e + \sum_i \delta_i \sum_e l_e q_{ie} = \delta \sum_{e \in P} l_e + \sum_i \delta_i \sum_{e \in P_i} l_e$$

In particular, the cost of the flow f is the average length of the paths $P, Q_1, \ldots, Q_k$ according to the weights $\delta, \delta_1, \ldots, \delta_k$. Therefore, if P or some Q_i is not a shortest path from s to t, one can decrease the cost of f by reassigning the flow along that path to a shortest path between s and t. This means that if f has minimum cost, P and all Q_is must be shortest paths between s and t. Since P was chosen to be an arbitrary path from s to t in H, all paths in H must be shortest paths between s and t.

(c) We can use binary search to find the true max flow value. Consider a max flow instance on a graph G with capacities c_e where we wish to ship flow from s to t. Create a min-cost flow instance as follows. Set the capacities to be equal to c_e and let the lengths be arbitrary (say $l_e = 1$ for all edges). We know that the max flow value $F_{max} \leq \sum_e c_e$. If the capacities are integers, F_{max} is also an integer. Therefore, we can binary search to find its true value. For an arbitrary F, we can find out if there is a flow that ships more than F units of flow by querying our min-cost flow instance with value at least F. If there is a flow with value at least F, it will return a flow with finite cost. Otherwise, the program is infeasible.

6 (Max Flows) Bottleneck Edges

Consider the following network (the numbers are edge capacities):



(a) Find the following:

- A maximum flow f, specified as a list of $s-t$ paths and the amount of flow being pushed through each.
- A minimum cut, specified as a list of edges that are part of the cut.

(b) Draw the residual graph G_f (along with its edge capacities). In this residual network, mark the vertices reachable from S and the vertices from which T is reachable.

(c) An edge of a network is called a *bottleneck edge* if increasing its capacity results in an increase in the maximum flow. List all bottleneck edges in the above network.

(d) Give a very simple example (containing at most four nodes) of a network which has no bottleneck edges.

(e) Give an efficient algorithm to identify all bottleneck edges in a network. (Hint: Start by running the usual network flow algorithm, and then examine the residual graph.)

Solution:

(a) The maximum flow is given by the following sequence of updates:

- Route 4 units of flow along $S - A - C - T$
- Route 2 units along $S - A - D - T$
- Route 2 units along $S - B - C - T$
- Route 3 units along $S - B - D - T$

The resulting flow is feasible and has value 11. It produces the mincut $(\{S, A, B\}, \{C, D, T\})$ (the edges across the cut are (A, C), (A, D), (B, C), (B, D)) of capacity 11, certifying the optimality of the flow.

Notice that $(\{S, A, B, D\}, \{C, T\})$ (edges (A, C), (B, C), (D, T)) is also a mincut.

(b) The residual graph is shown in the figure. Vertices S,A and B are reachable from S. T can be reached from vertices C and T.

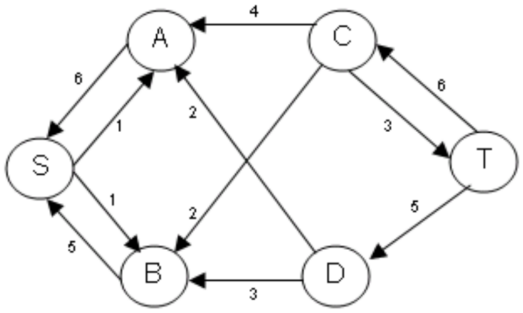


Figure 1: The residual graph

(c) (A, C) and (B, C) are bottleneck edges. The other edges belonging to a mincut are not bottleneck, as increasing their capacity does not increase the capacity of the minimum (s, t)-cut.

(d) The following figure shows an example with no bottleneck edges. The optimal flow saturates all edges, but augmenting the capacity of any of them does not increase the capacity of the minimum cut, i.e. does not open up any new path for flow to run from source to sink.

(e) Run the usual network flow algorithm and consider the final residual graph. Let S be the set of vertices reachable from s and T the set of vertices from which t is reachable in this graph. By the optimality of the flow S and T must be disjoint, as they are separated by a saturated cut. Suppose now that $e = (u, v)$ is a bottleneck edge. As we increase e's capacity we are able to route flow from s to u, through e and from v to t in the residual graph. This implies that $u \in S$ and $v \in T$. Moreover, if an edge $e = (u, v)$ has $u \in S$ and $v \in T$ (notice that e must then be in a minimum cut), increasing its capacity allows us to route flow from s to u (as $u \in S$), through the new capacity of e and to t (as $v \in T$). Hence, the set of bottleneck edges is the set $E(S, T)$, i.e. the set of edges which originate in S and end in T.

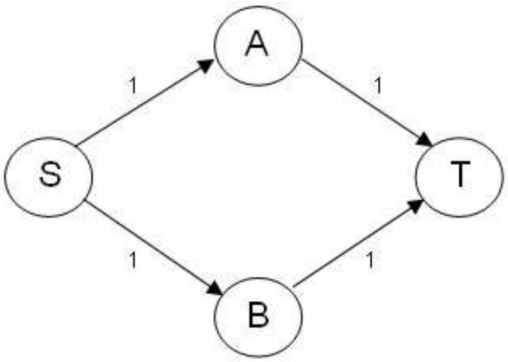


Figure 2: An example with no bottleneck edges

The running time is dominated by computing the max flow. Using the Ford-Fulkerson algorithm from the lecture and textbook, this is $O(|E| \cdot f)$, where f is the size of the max flow.