Note: Your TA may not get to all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. The discussion worksheet is also a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

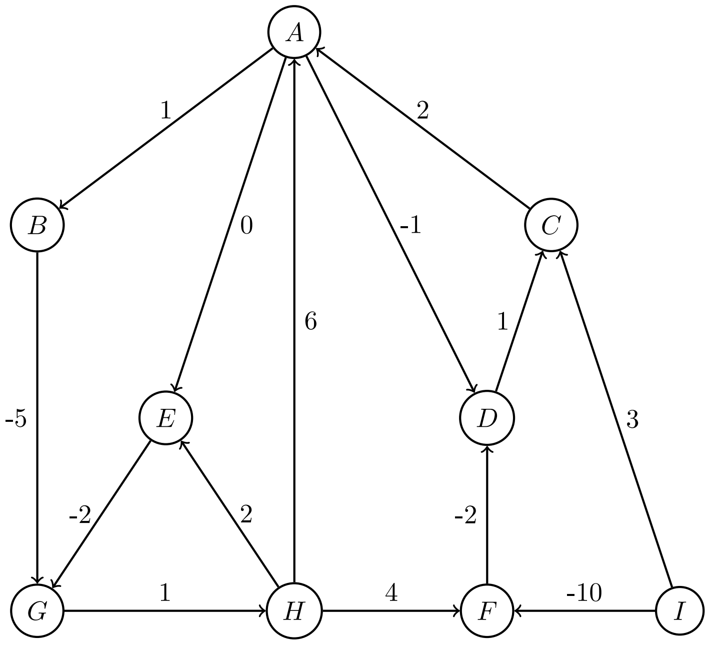
1 Shortest Paths with Negative Weights

(a) Dijkstra's algorithm doesn't work on graphs with negative edge weights. Here is one attempt to fix it:

  (a) Add a large number M to every edge so that there are no negative weights left.
  (b) Run Dijkstra's to find the shortest path in the new graph.
  (c) Return the path found by Dijkstra's, but with the old edge weights (i.e. subtract M from the weight of each edge).

  Show that this algorithm doesn't work by finding a graph for which it must give the wrong answer.

(b) Run the Bellman-Ford algorithm on the following graph, from source A. Process edges (u, v) in lexicographic order, sorting first by u then by v.

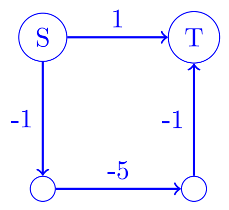


(c) What problem occurs when we change the weight of edge (H, A) to 1? How can we detect this problem when running Bellman-Ford? Why does this work?

(d) Let $G = (V, E)$ be a directed graph. Under what condition does the Bellman-Ford algorithm returns the same shortest path tree (from source $s \in V$) regardless of the ordering on edges? (Hint: Think about how there could be multiple shortest path trees).

Solution:

(a) The above algorithm doesn't work when the actual shortest path has more edges than other potential shortest paths. In this case, the paths with more edges have their weights increased more than the paths with fewer edges. We can see this in the following counterexample:

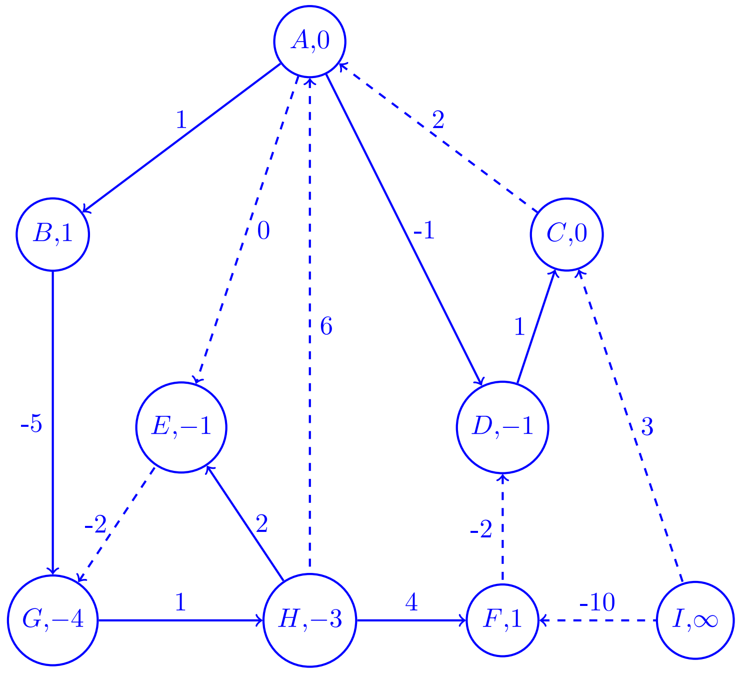


The shortest path is “down-right-up” (weight -7). After adding $M = 5$ to each edge, we increase the actual shortest path by 15. The path “right” only increases by 5 and so the algorithm returns this path as the shortest path.

(b) The following table summarizes the updates to shortest path lengths (we skip edges that don’t cause an update) in the first iteration:

Edge	A	B	C	D	E	F	G	H	I
(A, B)	0	1	∞	∞	∞	∞	∞	∞	∞
(A, D)	0	1	∞	-1	∞	∞	∞	∞	∞
(A, E)	0	1	∞	-1	0	∞	∞	∞	∞
(B, G)	0	1	∞	-1	0	∞	-4	∞	∞
(D, C)	0	1	0	-1	0	∞	-4	∞	∞
(G, H)	0	1	0	-1	0	∞	-4	-3	∞
(H, E)	0	1	0	-1	-1	∞	-4	-3	∞
(H, F)	0	1	0	-1	-1	1	-4	-3	∞

In the second iteration, we fail to improve any shortest path lengths and the algorithm terminates. The resulting shortest path tree (dashed edges are non-tree edges):



Remember that you can terminate the Bellman-Ford algorithm as soon as one iteration step does not change any distances.

(c) Changing the weight of (H, A) to 1 introduces a negative cycle. We can detect this by checking whether, after making $|V| - 1 = 9$ passes over the edges (relaxation steps), we can still update the distances. Try it out! The reason it works is that after k passes, we have found the length of all shortest paths from s with at most k edges. Since a simple path can only have at most $|V|-1$ edges, if we can update the distances on the $|V|$-th pass, the new shortest path contains a cycle. A cycle can only be part of a shortest path if it is of negative weight.

(d) If and only if the shortest path tree rooted at s is unique, i.e. if for each $v \in V$ there exists a unique shortest path from s to v. Note that this is not the case in the example: there are two paths of length -1 from A to D.

2 Service scheduling

A server has n customers waiting to be served. Customer i requires t_i minutes to be served. If, for example, the customers were served in the order $t_1, t_2, t_3, \ldots, t_n$, then the i-th customer would wait for $t_1 + t_2 + \cdots + t_{i-1}$ minutes. For simplicity, assume the t_i are distinct.

We want to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i)\,.$$

For example, if there are three customers with waiting times $3, 5, 4$ minutes and they are served in that order, the total waiting time is 11 minutes: the first customer waits 0 minutes, the second waits 3 minutes, and the third waits 8 minutes.

(a) Given the list of the t_i's, give an efficient algorithm for computing the optimal order in which to serve the customers (no runtime/correctness needed).

(b) Consider any solution besides the one you found in part (a). Show how to swap two people in that solution's order to improve its cost.

(c) Conclude from the previous part that your solution is optimal.

Solution:

(a) We use a greedy strategy, by sorting the customers in increasing order of service times and serving them in this order. (The running time is $O(n \log n)$).

(b) Let $s(j)$ denote the j-th customer in the ordering. Then

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{s(j)} = \sum_{i=1}^{n} (n-i) t_{s(i)}\,.$$

For any other ordering, there is some i, j such that $t_{s(i)} > t_{s(j)}$ but $i < j$. From the above equation, we see that swapping the positions of the two customers gives a better ordering.

(c) Since any order except the ordering we found in part (a) can be improved, the only choice for the optimal ordering is the one we found in part (a).

3 Activity Selection

Assume there are n activities each with its own start time a_i and end time b_i such that $a_i < b_i$. All these activities share a common resource (think computers trying to use the same printer). A feasible schedule of the activities is one such that no two activities are using the common resource simultaneously. Mathematically, the time intervals are disjoint: $(a_i, b_i) \cap (a_j, b_j) = \emptyset$. The goal is to find a feasible schedule that maximizes the number of activities k.

Here are two potential greedy algorithms for the problem.

Algorithm A: Select the shortest-duration activity that doesn't conflict with those already selected until no more can be selected.

Algorithm B: Select the earliest-ending activity that doesn't conflict with those already selected until no more can be selected.

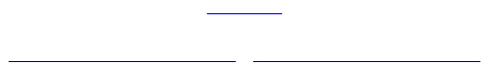
(a) Show that Algorithm A can fail to produce an optimal output.

(b) Show that Algorithm B will always produce an optimal output. (Hint: To prove correctness, show how to take any other solution S and repeatedly swap one of the activities used by Algorithm B into S while maintaining that S has no overlaps)

(c) **Challenge Problem:** Show that Algorithm A will always produce an output at least half as large as the optimal output.

Solution:

(a) There are many examples that work, but here is a simple one, easiest explained pictorally (each line represents an activity):



The optimal strategy is to pick the two longer activities, but strategy A picks the shorter activity, and then cannot pick another.

(b) Consider any optimal schedule S, and suppose the first i activities in S are the same as the first i activities chosen by Algorithm B (i might be zero). If both schedules have i activities, Algorithm B's solution is the same as S, so the proof is done. Otherwise, both Algorithm B's solution and the optimal solution must have a $(i+1)$st activity: If Algorithm B's solution is a subset of S, it would have added one of the remaining activities in S, and if Algorithm B's solution has more than i activities but S only has i activites, S is not optimal. In either case, we have a contradiction.

Then, consider swapping the $(i+1)$st activity in the optimal solution with the $(i+1)$st activity in our solution. By definition of Algorithm B, its $(i+1)$st activity ends at least as early as the $(i+1)$st activity in S. This means it can't overlap with later activities in S, so the feasibility and size of S are maintained by this swap. This means that given any optimal schedule where the first i activities are the same as in Algorithm B's solution, we can find an optimal schedule where the first $i+1$ activities are the same as in Algorithm B's solution. Starting from any optimal schedule and applying this repeatedly, we find an optimal schedule which is exactly the same as in Algorithm B's solution.

(c) Let I_{opt} be any optimal set of activities and I be the activities chosen by the shortest-duration greedy strategy.

We show the following:

(a) Every activity in I_{opt} overlaps at least 1 activity in I.

Suppose there exists activities in I_{opt} that does not overlap with any activity in I. Let $x \in I_{opt}$ be such an activity of shortest duration. If x had shorter duration than any other activities in I, it would have been added before the others were; contradiction. Otherwise, x would have been added just after all those in I are added; contradiction.

(b) Any activity in I can overlap at most 2 activities in I_{opt}.

Let I_k be the set of activities added by the greedy strategy just after the kth iteration. We show by induction that any activity in I_k overlaps with at most 2 activities in I_{opt}.

- Base case: $I_0 = \emptyset$; vacuously true.
- Inductive case: Assume true for I_k and let x be the activity added during the $k+1$th iteration. If $x \in I_{opt}$, we are done. Otherwise, suppose for the purpose of contradiction that x overlapped with strictly more than two activities in I_{opt}. Then it must be that one such activity begins and ends while x is still active; contradiction.

The original claim follows immediately as a corollary.

The two claims above imply $|I_{opt}| \leq 2|I|$.

4 Finding Counterexamples

In this problem, we give example greedy algorithms for various problems, and your goal is to find a counterexample where they do not find the best solution.

(a) In the travelling salesman problem, we have a weighted undirected graph $G(V,E)$ with all possible edges. Our goal is to find the cycle that visits all the vertices exactly once with minimum length.

One greedy algorithm is: Build the cycle starting from an arbitrary start point s, and initialize the set of visited vertices to just s. At each step, if we are currently at vertex u and our cycle has not visited all the vertices yet, add the shortest edge from u to an unvisited vertex v to the cycle, and then move to v and mark v as visited. Otherwise, add an edge from the current vertex to s to the cycle, and return the now complete cycle.

(b) In the maximum matching problem, we have an undirected graph $G(V,E)$ and our goal is to find the largest matching E' in E, i.e. the largest subset E' of E such that no two edges in E' share an endpoint.

One greedy algorithm is: While there is an edge $e = (u,v)$ in E such that neither u or v is already an endpoint of an edge in E', add any such edge to E'. (Challenge: Can you prove that this algorithm still finds a solution whose size is at least half the size of the best solution?)

Solution:
Note: For each part, there are many counterexamples.

(a) One counterexample is to have a four vertex graph with vertices a,b,c,d, where the edges $(a,b),(b,c),(c,d)$ cost 1, the edge (a,d) costs 100, and all other edges cost 2. If the greedy algorithm starts at vertex a, it will add the edges $(a,b),(b,c),(c,d)$ to the cycle, and then be forced to add the very expensive edge (a,d) at the end to find a cycle of cost 103. The optimal cycle is $(a,b),(b,d),(d,c),(c,a)$ which costs 6. The key idea here is that by using a path of low-weight edges, we forced the algorithm into a position where it had to pick a high-weight edge to complete its solution.

(b) The simplest example is a path graph with three edges. The greedy algorithm may pick the middle edge, the optimal solution is to pick the two outer edges.

To show this algorithm always finds a matching at least half the size of the best solution, let the size of the solution found by the algorithm be m. Any edge in the best solution shares an endpoint with one of the edges in the algorithm's solution (otherwise, the greedy algorithm could have added it). None of the edges in the best solution can share an endpoint, and the m edges in the algorithm's solution have $2m$ endpoints, so the best solution must have at most $2m$ edges.