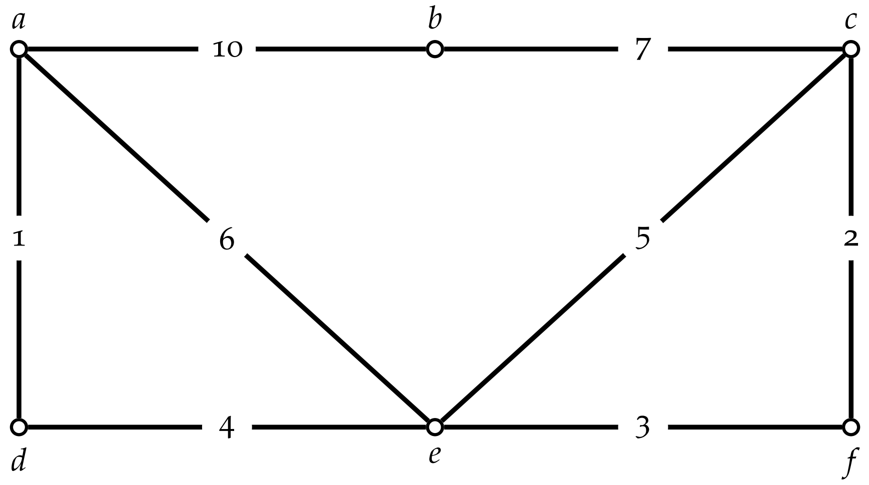
Warm-up

Problem 1. Consider Dijkstra's shortest path algorithm for undirected graphs. What changes (if any) do we need to make to this algorithm for it to work for directed graphs and maintain its running time?

Solution 1. The only thing we need to ensure is that when updating $D[z]$, we need to consider only the vertices where there is a directed edge from u to z. In the pseudocode in the lecture, this is implicitly handled by only considering the neighbors of u (i.e., the vertices that can be reached by following an edge from u), so we don't need to change anything.

Problem 2. Consider the following weighted undirected graph G:



Your task is to compute a minimum weight spanning tree T of G:

a) Which edges are part of the MST?

b) In which order does Kruskal's algorithm add these edges to the solution.

c) In which order does Prim's algorithm (starting from a) add these edges to the solution.

Solution 2.

a) $(a,d),(b,c),(c,f),(d,e),(e,f)$

b) The edges are considered in order of their weight, so they are added in the order: $(a,d),(c,f),(e,f),(d,e),(b,c)$. Note that when (c,e) is considered (c,f) and (e,f) are already present, so it isn't added. Similarly, (a,e) and (a,b) aren't added.

c) Prim's algorithm grows the MST from the starting vertex, each time adding the cheapest edge that connects the MST to a new vertex, so the edges are added in the order: $(a,d),(d,e),(e,f),(c,f),(b,c)$.

Problem solving

Problem 3. Let $G = (V, E)$ be an undirected graph with edge weights $w : E \to R^+$. For all $e \in E$, define $w_1(e) = \alpha w(e)$ for some $\alpha > 0$, $w_2(e) = w(e) + \beta$ for some $\beta > 0$, and $w_3(e) = w(e)^2$.

a) Suppose p is a shortest s-t path for the weights w. Is p still optimal under w_1? What about under w_2? What about under w_3?

b) Suppose T is minimum weight spanning tree for the weights w. Is T still optimal under w_1? What about under w_2? What about under w_3?

Solution 3.

a) For w_1 we note that for any two paths p and p' if $w(p) \leq w(p')$ then

$$w_1(p) = \alpha w(p) \leq \alpha w(p') = w_1(p').$$

so if a path is shortest under w it remains shortest under w_1.
This is now longer the case with w_2 and w_3. Consider a graph with three vertices and three edges, basically a cycle. Suppose one edge has weight 1 and the other edges have weight 1/2. Consider the shortest path between the endpoints of the edge with weight one. The two possible paths have total weight 1, so both of them are shortest. Now imagine adding 1 to all edge weights; that is, consider w_2 with $\beta = 1$. The shortest path in w_2 is unique—the edge with weight 1 in w. Now imagine squaring the edge weights; that is, consider w_3. The shortest path in w_3 is unique—the path with two edges of weight 1/2, which under w_3 have weight 1/4 each.

b) We claim that the optimal spanning tree does not change. This is because the relative order of the edge weights is the same under w, w_1, w_2, and w_3. Therefore, if we run Kruskal's algorithm the edges will be sorted in the same way and the output will be the same in all cases.

Problem 4. It is not uncommon for a given optimization problem to have multiple optimal solutions. For example, in an instance of the shortest s-t path problem, there could be multiple shortest paths connecting s and t. In such situations, it may be desirable to break ties in favor of a path that uses the fewest edges.

Show how to reduce this problem to a standard shortest path problem. You can assume that the edge lengths ℓ are positive integers.

a) Let us define a new edge function $\ell'(e) = M\ell(e)$ for each edge e. Show that if P and Q are two s-t paths such that $\ell(P) < \ell(Q)$ then $\ell'(Q) - \ell'(P) \geq M$.

b) Let us define a new edge function $\ell''(e) = M\ell(e) + 1$ for each edge e. Show that if P and Q are two s-t paths such that $\ell(P) = \ell(Q)$ but P uses fewer edges than Q then $\ell''(P) < \ell''(Q)$.

c) Show how to set M in the second function so that the shortest s-t path under ℓ'' is also shortest under ℓ and uses the fewest edges among all such shortest paths.

Solution 4.

a) Since the edge lengths are integer valued, if $\ell(P) < \ell(Q)$ then $\ell(Q) - \ell(P) \geq 1$. It follows then that $\ell'(Q) - \ell'(P) = M\ell(Q) - M\ell(P) = M(\ell(Q) - \ell(P)) \geq M$.

b) Let $|P|$ be the number of edges used in the path P. Then $\ell''(P) = M\ell(P) + |P|$. It follows that $\ell''(P) = M\ell(P) + |P| < M\ell(Q) + |Q| = \ell''(Q)$.

c) Set M to be the number of vertices in the graph.
From task (b) we know that if P is optimal under ℓ'' then it will have the fewest edges among paths that have length $\ell(P)$. Now suppose that there is another path Q such that $\ell(Q) < \ell(P)$; then $\ell(P) - \ell(Q) \geq 1$ and therefore $M\ell(P) \geq M + M\ell(Q)$. Notice that since $|Q| < n$, we have

$$M + M\ell(Q) > |Q| + M\ell(Q) = \ell''(Q)$$

and also

$$M\ell(P) \leq |P| + M\ell(P) \leq \ell''(P).$$

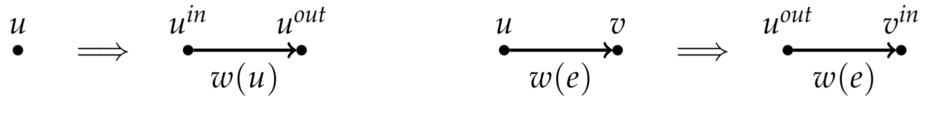
Putting all these inequalities together we get

$$\ell''(P) \geq M\ell(P) \geq M + M\ell(Q) > \ell''(Q)$$

which contradicts the optimality of P in ℓ''.

Problem 5. Consider the following generalization of the shortest path problem where in addition to edge lengths, each vertex has a cost. The objective is to find an s-t path that minimizes the total length of the edges in the path plus the cost of the vertices in the path. Design an efficient algorithm for this problem.

Solution 5. We reduce the problem of finding a path with minimum edges plus vertex weight to a standard shortest path problem with weights on the edges only. Before we do this, recall that an undirected graph can be modelled as a directed graph by replacing every undirected edge (u, v) by two directed edges (u, v) and (v, u). Hence, in the remainder, we'll focus on directed graphs. Given a directed graph $G = (V, E)$ and weights w we construct a new graph $G' = (V', E')$ and weights w'. For each vertex $u \in V$ we create two copies u^{in} and u^{out}, which we connect with an edge (u^{in}, u^{out}) with weight $w(u)$. For each edge $(u, v) \in E$ we create an edge (u^{out}, v^{in}) with weight $w(e)$.



It is trivial to check that for each path $p = \langle u_1, u_2, \ldots, u_k \rangle$ in the input graph G we have a path $p' = \langle u_1^{out}, u_2^{in}, u_2^{out}, \ldots, u_k^{in} \rangle$. Furthermore, the edge weight of p' equals the edge plus vertex weight of p.

It is easy to see that the graph G' can be constructed in $O(m+n)$ time. Furthermore, $|V'| = 2|V|$ and $|E'| = |V| + |E|$. Therefore Dijkstra's algorithm runs in $O(m + n\log n)$, where $m = |E|$ and $n = |V|$.

Problem 6. Consider the IMPROVING-MST algorithm for the MST problem.

```
1: function IMPROVING-MST($G, w$)
2:     $T \leftarrow$ some spanning tree of $G$
3:     for $e \in E \setminus T$ do
4:         $T \leftarrow T + e$
5:         $C \leftarrow$ unique cycle in $T$
6:         $f \leftarrow$ heaviest edge in $C$
7:         $T \leftarrow T - f$
8:     return $T$
```

Prove its correctness and analyze its time complexity. To simplify things, you can assume the edge weights are distinct.

Solution 6. Let $T_0, T_1, T_2, \ldots, T_m$ be the trees kept by the algorithm in each of its m iterations. Consider some edge $(u,v) \in E$ that did not make it to the final tree. It could be that (u,v) was rejected right away by the algorithm, or it was in T for some time and then it was removed. Suppose this rejection took place on the ith iteration. Let p be the u-v in path in T_i. Since (u,v) was rejected, all edges in p have weight less than $w(u,v)$. It is easy to show using induction that this is true not only in T_i but in all subsequent trees T_j for $j \geq i$.

Let (x,y) be an edge in the final tree T_m. Remove (x,y) from T_m to get two connected components X and Y. The Cut Property states that if (x,y) is the lightest edge in the cut (X,Y) then every MST contains (x,y). Suppose for the sake of contradiction that there is some rejected edge $(u,v) \in \text{cut}(X,Y)$ such that $w(u,v) < w(x,y)$. This means that (u,v) is not the heaviest edge in the unique u-v path in T_m, which contradicts the conclusion of the previous paragraph. Thus, (x,y) belongs to every MST. Since this holds for every edge in T_m, it follows that T_m itself is an MST.

Regarding the time complexity, we note that finding the cycle in $T+e$ can be done $O(n)$ time using DFS. Similarly finding the heaviest edge and removing it takes $O(|C|) = O(n)$ time. There are m iterations, so the overall time complexity is $O(nm)$.

Problem 7. Consider the REVERSE-MST algorithm for the MST problem.

```
1: function REVERSE-MST($G, w$)
2:     Sort edges in decreasing weight $w$
3:     $T \leftarrow E$
4:     for $e \in E$ in decreasing weight do
5:         if $T - e$ is connected then
6:             $T \leftarrow T - e$
7:     return $T$
```

Prove its correctness and analyze its time complexity. To simplify things, you can assume the edge weights are distinct.

Solution 7. Suppose e was one of the edges the algorithm kept. If e belongs to the optimal solution we are done, so assume for the sake of contradiction that this is not the case.

If the algorithm decided to keep e then it must be that $T - e$ was not connected, say $T - e$ had two connected components X and Y. Notice that all edges in the cut (X, Y) have a weight larger than $w(e)$. Therefore, we can take the optimal solution, add e to form a cycle C. This cycle must contain one edge f from $\text{cut}(X, Y)$, so we could remove f and improve the cost of the optimal solution, a contradiction. Therefore, e must be part of the optimal solution. Since this holds for all e in the output the algorithm is optimal.

Regarding the time complexity, we note that checking connectivity can be done $O(n + m)$ time using DFS. There are m iterations, so the overall time complexity is $O(m(n + m))$, which is $O(m^2)$ assuming the graph G is connected.

Problem 8. A computer network can be modeled as an undirected graph $G = (V, E)$ where vertices represent computers and edges represent physical links between computers. The maximum transmission rate or *bandwidth* varies from link to link; let $b(e)$ be the bandwidth of edge $e \in E$. The bandwidth of a path P is defined as the minimum bandwidth of edges in the path, i.e., $b(P) = \min_{e \in P} b(e)$.

Suppose we number the vertices in $V = \{v_1, v_2, \ldots, v_n\}$. Let $B \in R^{n \times n}$ be a matrix where B_{ij} is the maximum bandwidth between v_i and v_j. Give an algorithm for computing B. Your algorithm should run in $O(n^3)$ time.

Solution 8. Let T be a maximum weight spanning tree, that is, the spanning tree maximizing $w(T) = \sum_{e \in T} w(e)$. We claim that in this tree for any two vertices $u, v \in V$, the unique u-v path in T is a maximum bandwidth path. Indeed, assume that there is a pair of vertices u and v contradicting our claim. That is, let e be the minimum bandwidth edge in the unique u-v path in T and let p be an alternative u-v path p in the graph whose bandwidth is $b(p) > w(e)$. Now remove e from T breaking the tree into two connected components A and B having u on one side and v on the other. Then the path p must contain at least one edge, call it f, with one endpoint in A and another endpoint in B. Since $b(p) > w(e)$ then $w(f) > w(e)$, and so $T - e + f$ is also spanning tree with greater weight. A contradiction.

The maximum spanning tree can be computed by modifying Prim's or Kruskal's minimum spanning tree algorithm. For Prim's algorithm, instead of picking the lightest edge in every iteration, we pick the heaviest edge. For Kruskal's algorithm, we sort the edges in decreasing weight.

After computing T we could loop over every pair of vertices v_i and v_j, computing the lightest edge in the unique v_i-v_j path in T. This takes $O(n)$ time per pair of vertices, so we get overall $O(n^3)$ time.

With a better approach, you can also do the final computation in $O(n^2)$ time. After computing T we can loop over the vertices. For each starting vertex v_i we will recursively visit its neighbours in T, keeping track of the minimum weight edge traversed so far. At each vertex v_j we visit, where W is the minimum weight seen so far, we will update B_{ij} to W, then visit all unvisited neighbours of v_j in T, using $\min\{W, w(v_j, u)\}$ as the minimum weight seen so far at u. Since T contains $O(n)$ vertices and edges, each traversal will take $\sum_{v \in V} O(\deg_T(v)) = O(n)$ time, and so all n traversals together will take $O(n^2)$ time in total. Since both Prim's and Kruskal's algorithm can be run in less than $O(n^2)$ time, the total running time is $O(n^2)$.