Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

Graphs Cheatsheet 2.0

Bellman-Ford Algorithm

```
def bellman_ford(G=(V, E), s):
    for all v in V:
        dist(v) = infinity # distances
        par(v) = none # parents in shortest paths tree

    dist(s) = 0

    # since all paths contain at most |V| - 1 edges, we relax all edges |V| - 1
        times to ensure we've computed all distances correctly
    repeat |V| - 1 times:
        for each edge (u, v) in E:
            if dist(v) > dist(u) + w(u, v):
                dist(v) = dist(u) + w(u, v)
                par(v) = u

    # if we are still able to update distances, then there must be a negative
        cycle
    for each edge (u, v) in E:
        if dist(v) > dist(u) + w(u, v):
            return "Negative Cycle detected."

    return dist, par
```

$\hookrightarrow$ Runtime of Bellman-Ford: $O(|V||E|)$.

MST Important Definitions/Theorems

Minimum Spanning Tree (MST): is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. To find the MST of a graph, you can use either Kruskal's algorithm or Prim's algorithm.

Cut Property: the lightest edge e across a given cut belongs to *some* MST. If that e is uniquely lightest (i.e. it's lighter than all other edges across the cut), then it belongs to *all* MSTs.

Cycle Property: the uniquely heaviest edge in a given cycle cannot belong to any MST. Note that if the heaviest edge is *not* unique, then it may belong to some MST.

Kruskal's Algorithm

```
def kruskals(G=(V, E)):
    T = []
    for each v in V:
        make_set(v) # create a disjoint set structure for each vertex v
    sorted_edges = sorted(E)
    for each edge (u, v) in sorted_edges:
        if find(u) != find(v): # if adding the edge (u, v) doesn't create a cycle
            T.append((u, v))
            union(find(u), find(v)) # merge the components of u and v together
    return T
```

↪ Runtime of Kruskal's: $O(|E| \log |E|)$.

Prim's Algorithm

General Idea:

```
def prims(G=(V, E), s):
    T = []
    added = set()

    while added != V:
        (u, v) = "lightest edge where u in added and v in (V / added)"
        T.append((u, v))
        added.add(v)

    return T
```

Optimized Implementation

```
def prims(G=(V, E), s):
    h = heap()
    for each v in V:
        h.insert(v, inf) # each vertex v is given a priority value, initialize
            at inf
        prev(v) = v

    h.decrease_key(s, 0)
    while h is not empty:
        v = h.delete_min()
        for each edge (v, u) in E:
            if h.get_priority(u) > w(v, u): # minimize edge weight to u
                prev(u) = v
                h.decrease_key(u, w(v, u))

    T = []
    visited = set()
    for each v in V:
        curr = v
        while curr != prev(curr):
            if visited(curr):
                break
            T.append((curr, prev(curr))
            curr = prev(curr)

    return T
```

↪ **Runtime of Prim's:**

- $O((|E| + |V|) \log |V|)$ using a binary min-heap
- $O(|E| + |V| \log |V|)$ using a fibonacci min-heap

1 True and False Practice

For the following problems, if your answer is True, justify your answer; if you answer is False, provide a counterexample.

(a) True or False: Kruskal's algorithm works with negative edges.

Solution: True. The cut property holds even with negative edges. We can add a constant to each edge to make all positives and run Kruskal's if we want positive edges.

(b) We modify a graph G with negative edges by adding a large positive constant to each edge, making all edges positive. Let's call this modified graph G'.

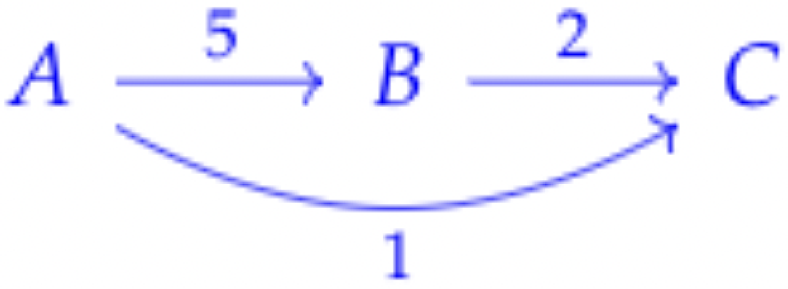
True or False: If we run Dijsktra's on G', the resulting shortest paths on G' are also the shortest paths on G.

Solution: False. Longer paths are penalized in G'. This means that a shortest path with many edges in G is no longer the shortest path in G'.

(c) Let $G = (V, E)$ be a DAG with positive edge weights. We first run Dijkstra's algorithm to compute the distance from the source s to every other vertex v. Afterwards, we store the vertices in increasing order of their distance from s.

True or False: this sequence of vertices be a valid topological sort of G.

Solution: False. Consider the graph below: a topological sort of the vertices would be A, B, C, but using the method in this question, we get A, C, B.



(d) Let $G = (V, E)$ be an undirected graph. Let $G' = (V', E')$ where $V' = V \bigcup \{u\}$ and $E' = E \bigcup E_u$, where E_u is some set of edges that include u.

True or False: any MST of G is the subset of some MST of G'.

Solution: False. Consider the case when the new vertex u in G' has edges to all other vertices in G'. If each of these edges from u are lighter than any edge of G, then the MST of G' is the set of all edges from u.

2 Bellman Ford Properties

Recall that Dijkstra's algorithm relies on the fact that we will visit vertices in increasing order of distance. So if the shortest path to a vertex u goes through v, we will visit v before u because $dist[v] < dist[u]$. This is not true when we have negative edges, since it can happen that $dist[v] > dist[u]$. In this problem, we will see how Bellman Ford algorithm avoids this issue.

Let $G = (V, E)$ be a directed weighted graph with possibly negative weights, such that $|V| = n$ and $|E| = m$. We want to find the shortest path from a fixed source s to every vertex in V.

(a) Suppose G has no cycles of negative total weight. Prove that for any pair of vertices u, v, there is a shortest path that has most $n-1$ edges. *Hint: If the shortest path has more than $n-1$ edges, what can we say about the number of vertices on the path?*

(b) Let $d_k[v]$ denote shortest distance from s to v using **at most k edges**. Prove that if $d_n[v] < d_{n-1}[v]$, then there has to be negative weight cycle in G.

(c) For the remaining parts, assume that G has no negative weight cycles. What is $d_k[s]$ for any value $k \geq 0$? What is the value of $d_1[v]$ if v is a neighbor of s?

(d) For a fixed $k > 1$, suppose you are given the values $d_{k-1}[v]$ for all $v \in V$, give an algorithm to compute $d_k[v]$ for all v in $O(n + m)$ time.

(e) Given a value of $K < N$, describe a $O(n + K \cdot m)$ algorithm to find the length of the shortest path from s to every v that uses at most K edges. Prove that your algorithm is correct.

Solution:

(a) This is because if the path has more than $n-1$ edges, then some vertex has been repeated on the path. Which means the path contains a cycle. However, since the cycle cannot have negative weight, we can remove it without increasing the length of the path.

(b) Follows from part (a), because there cannot be a shorter path between any two pairs of vertices using more than $n-1$ edges.

(c) $d_k[s] = 0$ because there are no negative weight cycles. If (s, v) is an edge then, $d_1[v] = weight(s, v)$ otherwise it is ∞.

(d) Initialize $d_k[v] = \infty$. Loop over all the edges and for an edge (u, v), make the following update:

$$d_k[v] = \min(d_{k-1}[v], d_{k-1}[u] + weight(u, v))$$

(e) Starting with $d_0[s] = 0$ and $d_0[v] = \infty$ for all other v, repeatedly apply part (d) for $k = 1, 2, \ldots K$. Correctness can be proved using induction. Assuming we have $d_{k-1}[\cdot]$ correct,

$$d_k[v] = \min\left\{d_{k-1}[v], \min_{(u,v)\in E} d_{k-1}[u] + weight(u, v)\right\}$$

This is what the algorithm from part (d) does. The only difference is that here, we don't have to explicitly look over all neighbors of each v, and instead compute all $d_k[v]$ simultaneously by traversing through all edges in E.

3 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 (i.e. G is a complete graph). 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?)

(c) In the interval scheduling problem from lecture, our greedy algorithm instead repeatedly picks the interval with the smallest number of conflicts.

Solution:

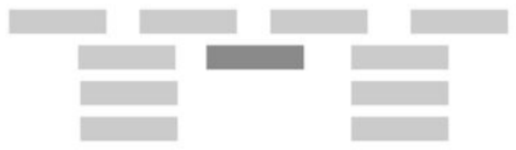
Note: For each part, there are many counterexamples.

(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.

(c) Consider the following counterexample:



Here we would end up choosing the dark gray interval first since it has the fewest conflicts. However, this yields only 3 intervals at the end of our smallest-conflict algorithm, which is less than the optimal solution of 4 intervals (i.e. the top 4 intervals).

4 Horn Formula Practice

(a) Find the variable assignment that solves the following horn formula:

$$(x \wedge z) \Rightarrow y, z \Rightarrow w, (y \wedge z) \Rightarrow x, \Rightarrow z, (\bar{z} \vee \bar{x}), (\bar{w} \vee \bar{y} \vee \bar{z})$$

(b) Show that any implication clause of the form $(x_i \wedge x_j \wedge \cdots) \Rightarrow \mathsf{True}$ is always satisfiable.
Hint: what disjunction clause is this equivalent to?

(c) Show that any implication clause of the form $\mathsf{False} \Rightarrow x_k$ is always satisfiable.

Solution:

(a)
- z must be true since we have the statement $\Rightarrow z$.
- w must be true since $z \Rightarrow w$
- x and y need not be changed, as all our implications are satisfied.
- All negative clauses are now satisfied, so we've found our satisfying assignment.

(b) This clause is satisfied no matter the assigment to the variables because it is equivalent to $(\mathsf{True} \vee \overline{x_i} \vee \overline{x_j} \ldots)$.

(c) This clause is equivalent to $(x_k \vee \mathsf{True})$ so is always satisfied no matter what x_k is.

5 Longest Huffman Tree

Under a Huffman encoding of n symbols with frequencies $f_1, f_2, \ldots, f_n$, what is the longest a codeword could possibly be? Give an example set of frequencies that would produce this case, and argue that it is the longest possible.

Solution: The longest codeword can be of length $n-1$. An encoding of n symbols with $n-2$ of them having probabilities $1/2, 1/4, \ldots, 1/2^{n-2}$ and two of them having probability $1/2^{n-1}$ achieves this value. No codeword can ever by longer than length $n-1$. To see why, we consider a prefix tree of the code. If a codeword has length n or greater, then the prefix tree would have height n or greater, so it would have at least $n+1$ leaves. Our alphabet is of size n, so the prefix tree has exactly n leaves.

6 Doctor

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 (i.e. exchange argument).

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

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

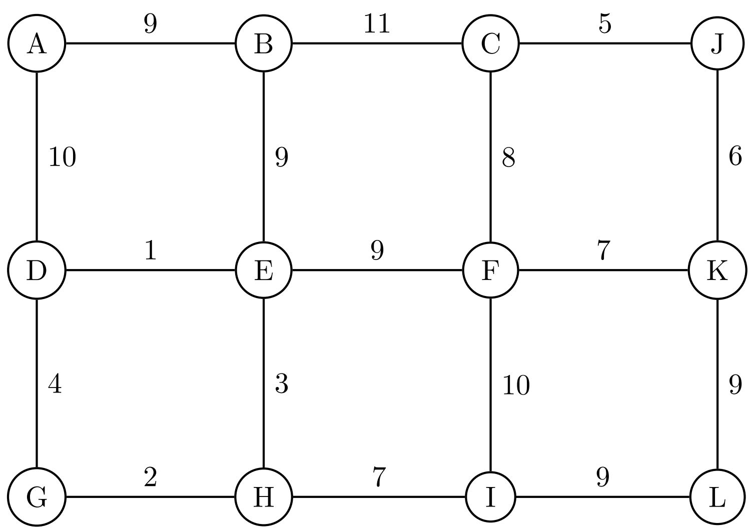
and the waiting time after the swap is

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

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.

7 MST Tutorial



1. List the first **six** edges added by Prim's algorithm in the order in which they are added. Assume that Prim's algorithm starts at vertex A and breaks ties lexicographically.

   Solution: AB, BE, DE, EH, GH, HI

2. List the first **seven** edges added by Kruskal's algorithm in the order in which they are added. You may break ties in any way.

   Solution: (DE, GH, EH, CJ, JK, FK, HI) **or** (DE, GH, EH, CJ, JK, HI, FK)

8 MST Basics

For each of the following statements, either prove or give a counterexample. Always assume $G = (V, E)$ is undirected and connected. Do not assume the edge weights are distinct unless specifically stated.

(a) Let e be any edge of minimum weight in G. Then e must be part of some MST.

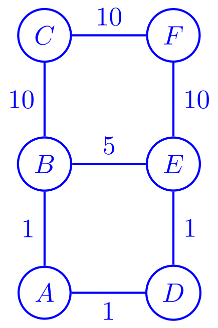
(b) If e is part of some MST of G, then it must be a lightest edge across some cut of G.

(c) If G has a cycle with a unique lightest edge e, then e must be part of every MST.

(d) For any $r > 0$, define an r-path to be a path whose edges all have weight less than r. If G contains an r-path from s to t, then every MST of G must also contain an r-path from s to t.

Solution:

1. True, e will belong to the MST produced by Kruskal.
2. True, suppose (u, v) is the edge. Let one side of the cut be everything reachable in the MST from u without using the edge (u, v). If this cut has an edge lighter than (u, v) then we could add this edge to the MST and remove (u, v). We know this edge is not already in the MST because otherwise both its endpoints would be reachable from u without using (u, v).
3. False. Let e be also the heaviest edge of a different cycle; then, we know that e can't be part of the MST. Concretely, in the following graph, edge (B, E) satisfies this condition, but will not be added to the graph.



4. True. Let $v_1, v_2, \ldots, v_n$ denote the r-path in G from $s = v_1$ to $t = v_n$. Consider the greatest i such that the MST has an r-path from s to v_i and assume for contradiction that $i < n$. Then the edge (v_i, v_{i+1}) is not in the MST. Adding this edge to the MST forms a cycle, which must have edges of length less than r since otherwise we could replace one of them with (v_i, v_{i+1}).

Thus there is an r-path from v_i to v_{i+1} and hence from s to v_{i+1}, contradicting our initial assumption.