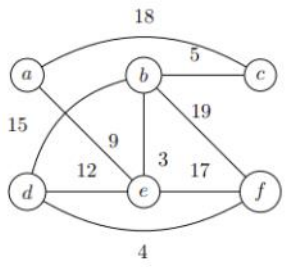
Exercise 1: Kruskal (5 Points)

Consider the graph G given below.



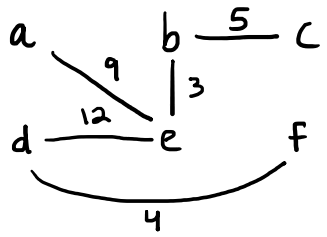
(a,c)	18
~~(a,e)~~	~~9~~ 4.
~~(b,c)~~	~~5~~ 3.
(b,d)	15
~~(b,e)~~	~~3~~ 1.
(b,f)	19
~~(d,e)~~	~~12~~ 5.
~~(d,f)~~	~~4~~ 2.
(e,f)	17

At each step of Kruskal's Algorithm

1. draw the minimum spanning forest
2. give a cut that justifies the inclusion of each edge

Step		Cut T	V-T
1.	b | 3 e	b	acdef
2.	b | 3 d e f, 4	bde	acf
3.	b — 5 — c | 3 d e f, 4	c	abdef
4.	a, 9, b — 5 — c | 3 d e f, 4	a	bcdef

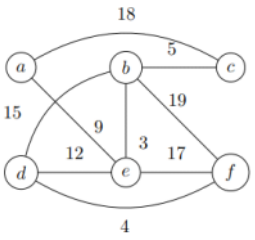
4

5. 

ebac df

Exercise 2: Prim (5 Points)

Consider the graph G given below.

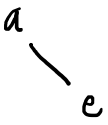


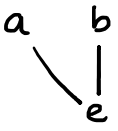
(a,c)	18
~~(a,e)~~	~~9~~
~~(b,c)~~	~~5~~
(b,d)	15
~~(b,e)~~	~~3~~
(b,f)	19
~~(d,e)~~	~~12~~
(d,f)	4
(e,f)	17

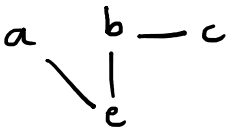
At each step of Prim's Algorithm

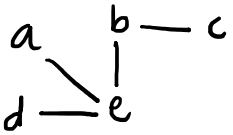
1. draw the minimum spanning tree
2. give *the* cut $(T, V-T)$ that justifies the inclusion of each edge

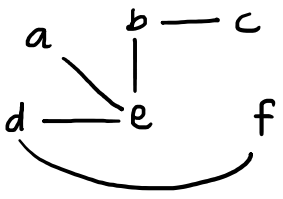
Step	T	V-T	Add Edge
1	a	b c d e f	(a,e): 9
2	ae	bcdf	(b,e): 3
3	aeb	cdf	(b,c): 5
4	aebc	df	(d,e): 12
5	aebcd	f	(d,f): 4

1. 

2. 

3. 

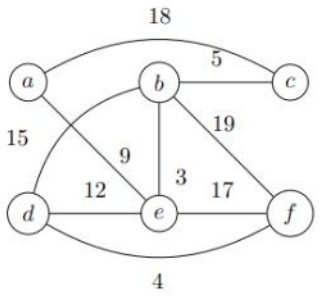
4. 

5. 

Exercise 3: SPT (10 Points)

Consider the graph G given below.

18



(a, c)	18
(a, e)	9
(b, c)	5
(b, d)	15
(b, e)	3
(b, f)	19
(d, e)	12
(d, f)	4
(e, f)	17

For each of the following, with vertex a as the root, draw the shortest path tree.

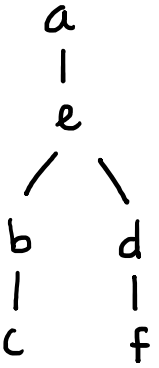
1. Dijkstra's Algorithm
2. the Bellman Ford Algorithm

1. Dijkstra's

Step	a	b	c	d	e	f
0	0	∞	∞	∞	∞	∞
1	0	∞	18	∞	9	∞
2	0	12	18	21	9	26
3	0	12	17	21	9	26
4	0	12	17	21	9	26
5	0	12	17	21	9	25
6	0	12	17	21	9	25

node	Parent
a	
b	e
c	b
d	e
e	a
f	d

SPT:



At each step of Dijkstra's we pop the next unvisited node with the lowest known distance from the source. The blue square represents the node popped at each step. Then we look at all the neighbors of the popped node and see if we can update any of the distances.

Remember, we maintain that once a node has been popped we know we have its shortest distance. So after a node is popped we don't have to worry about it anymore and you can just ignore it when considering updates in the following steps.

The professor recommends keeping track of the parents of each node so that you can easily draw the SPT at the end.

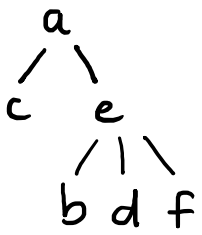
2. Bellman Ford

Step	a	b	c	d	e	f
0	0	∞	∞	∞	∞	∞
1	0	∞	18	∞	9	∞
2	0	12	18	21	9	26
3	0	12	17	21	9	25
4	0	12	17	21	9	25

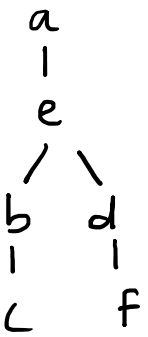
We can stop at step 4 since there were no updates.

Step 1

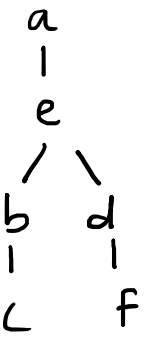
Step 2



Step 3



Step 4



At each step of Bellman Ford's algorithm, we look at the neighbors of EVERY node and update the distance if we found a shorter path.

We repeat this step n-1 times, or until there are no updates. If we have an update in all n-1 steps, we do an additional nth iteration to check for a negative weight cycle. If there is an update on this iteration we know there is a negative weight cycle.

Exercise 4: Cut Property (2 Points)

The Cut Property states that the minimum weight edge crossing a cut belongs to an MST.

1. Prove the cut property. You may assume the edge weights are unique.
2. Prove the correctness of Prim's algorithm, assuming the cut property. (Hint: prove by induction that every edge added to T belongs to the MST. After $n-1$ edges are added, what must T be?)

---

1. By way of contradiction assume the smallest weight crossing edge, e, does not belong to the MST.
   But, since the MST spans the entire graph, the MST must contain some other crossing edge e'.

   Let the weight of the MST T be W.
   We know T contains e' and does not contain e.

We can take T, remove e' and add e to create a new tree. We know the new tree is still spanning because we removed e' and added e. This new spanning tree has with a lower weight than T, since the weight of e is less than the weight of e', contradicting that T is the minimum spanning tree.

This proves that an MST must contain the smallest weight crossing edge.

Exercise 5: Simple Path (2 Points)

A *simple* path is a path that does not revisit any vertices (equivalently, a path that contains no cycles).

Given a weighted digraph G as an adjacency matrix and distinct vertices u and v:

1. Give an $\mathcal{O}(n \cdot n!)$ algorithm to determine the shortest simple path from u to v
2. What restrictions can we place on G to improve this to polynomial time? Justify your answer.

1.

First, how many ways are there to go from u to v?
Let's say we have a graph with the nodes a b c u d e f g v.

We would have to reach v from u through some path of vertices (this path can be directly from u to v or through a sequence of vertices).

So we would to take a sequence like {u, a, b, c, d, e, f, g, v} and check the adj matrix to see if there's an edge from u to a. Once we've reached a, check if there's an edge from a to b, then check if there's an edge from b to c, and so on until we reach v. Checking the matrix for an edge between each vertex in the sequence takes O(n) time

We also have to check all possible sequences from the length of 2 (the case where we can go directly from u to v) to length n (when the path contains all n vertices).

Then for each of those sequences we have to look at all possible permutations of the vertices. We know that in the worst case the shortest simple path contains n vertices, so the number of possible permutations is n!

This gives a total runtime of O(n * n!) in the worst case.

Whichever path successfully brings us from u to v with the minimum distance is the shortest simple path.

Main Idea:
Any simple path from u to v is a sequence of vertices. There are n! possible sequences.
And for each of those sequences we use the adj matrix to check if it's a valid path which takes O(n) time

Total runtime: O(n*n!)

2. The problem is essentially asking us to find the shortest path. In general we would use Dijkstra's algorithm to solve this problem. But Dijkstra's only works when the graph has no negative weight edges. So if we place the restriction that G must not contain any negative weight edges, we can use Dijkstra's, which is O(mlogn), polynomial time to find the shortest simple path.

   If we wanted to use Bellman Ford for O(mn) time, we would have to restrict the graph to contain no negative weight cycles, since the "shortest path" is not well defined whenever there are negative weight cycles.

   We could also use Floyd Warshall for $O(n^3)$ time, if there are no negative weight cycles.

Exercise 6: Shortest Path Table (1 Points)

Fill in the following table with the runtime of each algorithm used to solve each problem.

algorithm	runtime	
	SSSP	APSP
Dijkstra's	mlogn	mnlogn
Bellman Ford	mn	mn^2
Floyd Warshall	n^3	n^3
Modified Dijkstra's	n^2	n^3

Dijkstra's and Bellman Ford are SSSP algorithms. In order to solve APSP using them, simply run the algorithm n times, so that each node gets a chance to be the source node.

Floyd Warshall is an APSP algorithm. To use this algorithm to solve SSSP simply run the algorithm once and only return distances for the desired source. The algorithm is inherently $O(n^3)$, so using it to solve a simpler

problem does not change the runtime, it remains $O(n^3)$.

The modified Dijkstra's is just dijkstra's but instead of using a heap to pop the next node with the lowest distance, we manually search for the next node with the lowest distance. This is $O(n^2)$.