Midterm 1

Name:

TA:

Section Time:

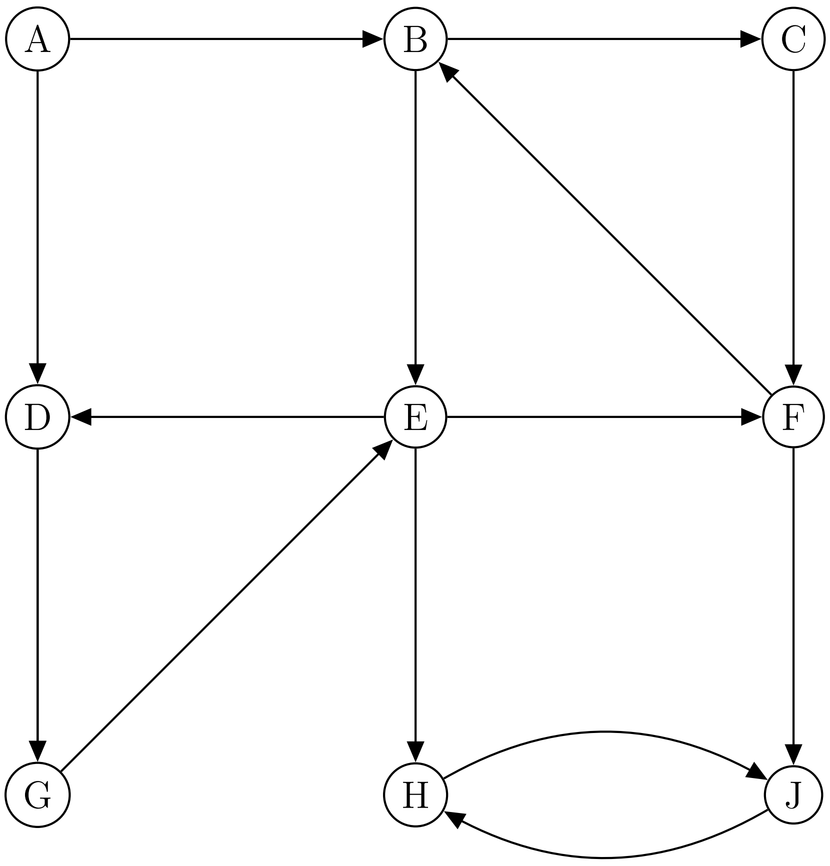
Course Login:

Answer all questions. Read them carefully first. Be precise and concise. The number of points indicate the amount of time (in minutes) each problem is worth spending. Write in the space provided, and use the back of the page for scratch. Good luck!

1	
2	
3	
4	
5	
6	
total	

Problem 1 ***(20 points)***

(a) Do a depth-first search of the graph below. Process nodes (and edges out of a node) in alphabetic order. Show the pre-visit and post-visit order numbers.



Answer:

v	$pre(v)$	$post(v)$
A	1	18
B	2	17
C	3	10
D	12	15
E	11	16
F	4	9
G	13	14
H	6	7
J	5	8

(b) Give the strongly connected components of this graph in the order in which they would be discovered by the algorithm (DFS in the reversed graph, then DFS of the graph). Give the components in the form ABC, DEFG, etc., and give the DAG of the strongly connected components.

Answer: The strongly connected components are found in the order $HJ, BCDEFG, A$. The associated DAG is

$$\begin{tikzcd}\text{A} \arrow[r] & \text{BCDEFG} \arrow[r] & \text{HJ}\end{tikzcd}$$

(c) In a directed graph with $k > 1$ strongly connected components, you are asked to add edges, as few as possible, so that the resulting graph is strongly connected. What is the maximum number of edges that may be required? The minimum possible number of edges? What is the actual number of edges that you need to add in this particular graph? Justify your answers very briefly.

Answer: *The maximum number of edges required is k.* If the k strongly connected components are completely disconnected from each other, we can make the graph strongly connected by adding k edges to form a cycle among the k components. In such a case, adding $k-1$ edges will only give us a tree on the components, which is not enough to make the graph. strongly connected.

The minimum number of edges required is 1. This happens when there is a single source and a single sink: adding an edge from any node in the sink to any node in the source makes the graph strongly connected.

In the graph from part (b), you need to add 1 edge. For example, add an edge (J, A).

(d) Could this be the directed graph from a 2SAT instance? Justify briefly your answer.

Answer: No. For each variable x, the graph of a 2SAT instance has a node for x and $\bar{x}$ and therefore has an even number of nodes. This graph has an odd number of nodes.

Problem 2 *(15 points)*

(a) You want to multiply a polynomial of degree 1 with a polynomial of degree 2. On how many points will you do FT (smallest appropriate power of two)? What is the corresponding ω?

On how many points will you do FFT (smallest appropriate power of two)? What is the corresponding ω?

(b) Suppose that the two polynomials are 1 and x^2. What are the two outputs of the FT?

Answer:

$$FFT(1) = FT(1) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & 1 & i \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

$$FFT(x^2) = FT(x^2) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & 1 & i \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}$$

(c) Describe briefly how you will find the product polynomial from those outputs (no need to actually find it).

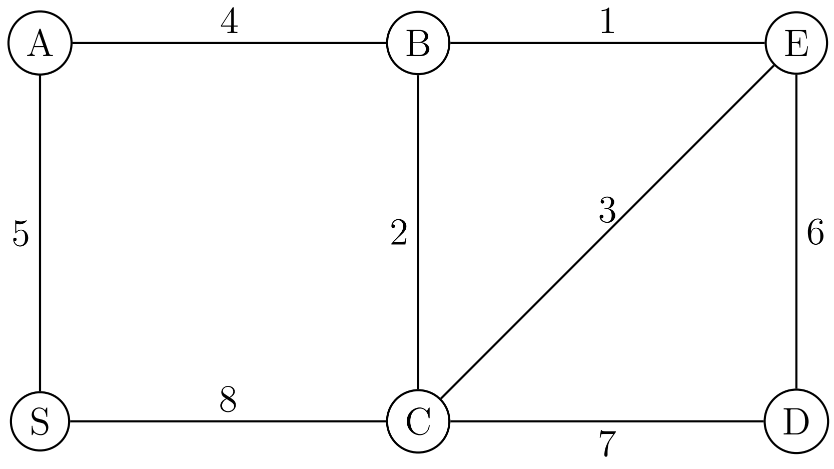
Answer: Use the FT of each and do a pointwise multiply of the elements $C[i] = A[i] * B[i]$. Then, calculate the inverse Fourier transform of that vector as follows: $FT^{-1}(C) = \frac{1}{n} M_n(w^{-1})C$. The output of the inverse FT is the product of the two polynomials.

(d) True or false? If the input vector has $O(1)$ entries that are not zero, then the FT can be done faster than $n \log n$ time. Explain briefly your answer.

Answer: The naive algorithm requires n^2 operations to calculate the Fourier transform. This is because we need n multiplications to calculate each element in the output vector and there are n elements in that vector. If the input vector has a constant number of elements, then we only need a constant number of multiplications for each element in the output vector. This allows us to calculate the FT in O(1*n) time rather than O(n*n) time, making it faster than $n \log n$.

Problem 3 *(15 points)*

For the graph below:



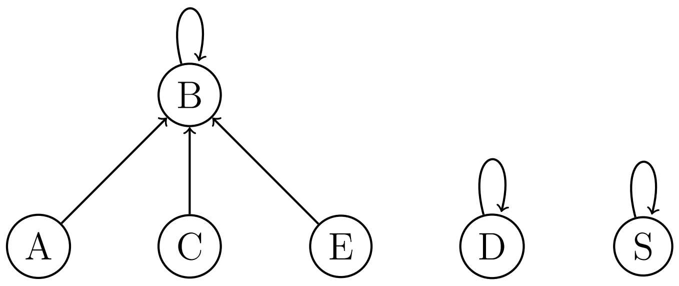
1. Which would be the third vertex, besides the starting vertex S, that would be reached by Prim's algorithm for the MST?

   Answer: E. The algorithm adds nodes A, B, then E.

2. Which would be the third edge added to the tree by Kruskal's (greedy) algorithm for the MST?

   Answer: AB. The algorithm adds edge BE, BC, skips CE because it creates a loop, then adds AB.

3. How would the trees of the union-find data structure (the data structure used in class to maintain disjoint sets) look at that point? (Use union-by-rank, but do not do path compression.)

   Note: Since the edge BE was added first and no tie breaker was specified, nodes B and E could be swapped in the graph above.

4. Which would be the third vertex, besides the starting vertex S, finalized by Dijkstra's algorithm for shortest paths?

   Answer: *B*

   Dijkstra's algorithm runs as follows:

v	Init	Iteration 1	Iteration 2	Iteration 3
S	0	0	0	0
A	∞	5	5	5
B	∞	∞	∞	9
C	∞	∞	8	8
D	∞	∞	∞	∞
E	∞	∞	∞	∞

Problem 4 *(20 points)*

True or false? Circle the right answer. No justification is needed. No points will be subtracted for wring answers, so it is to your best interest to guess all you want.

[✓ **T**] **F** If a directed graph is a dag, then it has a source (a vertex with no incoming edges) and a sink (a vertex with no outgoing edges).

T [✓ **F**] If a directed graph has a source and a sink, then it is a dag.

T [✓ **F**] $T(n) = 4T(n/4) + n$ has solution $\Theta(n)$.

T [✓ **F**] The fastest algorithm to find the median is $\Theta(n \log n)$

[✓ **T**] **F** Suppose that in a weighted graph there is a unique edge, e, that has smallest weight. The minimum spanning tree always contains e.

T [✓ **F**] Suppose that in a weighted graph there is a unique edge, e, that has smallest weight. The shortest path from S to T always contains e.

T [✓ **F**] Suppose that in a weighted graph there is a unique edge, e, that has largest weight. Then the minimum spanning tree cannot contain e.

[✓ **T**] **F** There is a family of hash functions from a set S to $[0, \ldots, n-1]$ such that the probability that any two distinct items from S collide is $1/n$.

[✓ **T**] **F** There are more families of universal hash functions besides the one based on modular arithmetic we discussed in class.

[✓ **T**] **F** For every set of keys to be hashed into a hash table larger than the set, there is a hash function that has no collisions.

Problem 5 *(25 points)*

Consider a directed graph G with one negative length one edge, call it (A, B).

(a) Use Dijkstra's algorithm to detect whether there is a negative cycle in G. Describe your algorithm; explain why it is correct and why its running time is asymptotically the same as that of Dijkstra's.

Answer:

Algorithm: Remove (A, B) from G to get the graph G'. Run Dijkstra's algorithm in G' starting at B to get distances $d'_B(v)$. Say there exists a negative-length cycle if and only if $d'_B(A) < |l(A, B)| = -l(A, B)$, that is, if the shortest path from B to A is less than the absolute value of the length of edge (A, B).

Correctness: We argue that G contains a negative cycle if and only if there is a path $B \to A$ of length less than $|l(A, B)|$. Clearly, if there is a path from B to A, then we can construct a cycle by adding (A, B), and the cycle will have length $d'_B(A) + l(A, B)$. If $d'_B(A) < |l(A, B)|$, then $d'_B(A) + l(A, B) < 0$ and this cycle has negative length.

Proving the other direction, a negative-length cycle must contain a negative-length edge, and thus it must include (A, B). Removing (A, B) from such a cycle leaves a path $B \to A$. If the total length of the cycle is negative, we know that the length of this path from B back to A is less than $|l(A, B)|$.

Running Time: (A, B) can be found and removed in time $O(|E|)$ by looping through the edges of G, looking for an edge with negative length, and removing it from the linked list storing it. After removing (A, B) we call Dijkstra's algorithm once and then perform a single comparison, giving an overall running time of

$$O(|E|) + T_{Dijkstra}(|V|, |E|) + O(1) = O(T_{Dijkstra}(|V|, |E|)) \ .$$

Alternative Answers: Another way to solve this problem is to run Dijkstra's algorithm multiple times as follows:

(a) Initialize vertices as in Dijkstra's algorithm.

(b) Build a queue as in Dijkstra's algorithm.

(c) Run the loop of Dijkstra's algorithm, popping out each node and updating.

(d) Repeat steps (2) and (3). When running step (3) the second time, say there is a negative-length cycle if the algorithm ever attempts to update a vertex that was already frozen/popped. (Alternatively, let this step run to completion and run Dijkstra's algorithm a third time, saying that there is a negative-length cycle if any distance is changed in this third iteration.)

Many students responded along these lines but did not run Dijkstra's algorithm enough times, either running it once and looking for a frozen node to be updated (not twice) or running it twice and looking for any node to be updated (not three times). If these answers were correct, then they would essentially imply that running Dijkstra's algorithm once would compute correct shortest paths even with a negative edge, for which the following graph is a counterexample:

$$\begin{tikzcd} & B \arrow[dr, "-2"] & & \\ A \arrow[ur, "2"] \arrow[rr, "1"] & & C \arrow[r, "1"] & D \end{tikzcd}$$

(b) Show how to find the shortest paths from a node S in the case there are no negative cycles in G. Again use Dijkstra's algorithm as a subroutine or guide. Describe your algorithm; explain why it is correct and why its running time is asymptotically the same as that of Dijkstra's.

Answer: The main idea is that the shortest path from S to T either uses edge (A, B) or it doesn't. In the first case, the path must take the shortest path $S \to A$, then edge (A, B), then the shortest path

$B \to T$. For the second case, we can simply compute the shortest $S \to T$ path ignoring (A, B). None of these shortest paths will use (A, B), so we can compute them all by removing (A, B) and running Dijkstra's algorithm on the resulting graph (in which all edges have nonnegative weights).

Algorithm:

(a) Remove (A, B) from G to get G'.

(b) Run Dijkstra's algorithm on G' starting at S. Use $d'_S(v)$ denote the computed distances.

(c) Run Dijkstra's algorithm on G' starting at B. Let $d'_B(v)$ denote the computed distances.

(d) For each $v \in V$, the shortest path distance from S to v in G is given by

$$d_S(v) = \min(d'_S(v), d_S(A) + l(A, B) + d_B(v)) \ .$$

The actual path can be recovered by following the *prev* pointers from the appropriate run of Dijkstra's algorithm.

Correctness: Consider a fixed pair of nodes S and T. The shortest path $S \to T$ either uses edge (A, B) or it does not. After removing (A, B) and running Dijkstra's algorithm from S, the distances $d'_S(v)$ are precisely the lengths of the shortest $S \to v$ paths that do not use (A, B).

The shortest path that does use (A, B) will consist of the shortest path $S \to A$, then edge (A, B), then the shortest path $B \to T$. Since there are no negative-weight cycles, the paths $S \to A$ and $B \to T$ are acyclic (or, at least, acyclic shortest paths exist) and therefore will not use edge (A, B). Thus, the length of the shortest $S \to A$ path is $d'_S(A)$ and the length of the shortest $B \to T$ path is $d'_B(T)$. This gives an $S \to T$ path of length $d'_S(A) + l(A, B) + d'_B(T)$.

The actual shortest $S \to T$ path length is the minimum of $d'_S(T)$ and $d'_S(A) + l(A, B) + d'_B(T)$, which is precisely the value computed by the algorithm.

Running Time: As in part (1), removing (A, B) can be done in $O(|V|)$ time. After that, We call Dijkstra's algorithm twice, an then do an $O(1)$ operation for each $v \in V$. Thus, the runtime is

$$O(|E|) + 2T_{Dijkstra}(|V|, |E|) + |V| \times O(1) = O(T_{Dijkstra}(|V|, |E|)) \ .$$

Alternative Answers: As in part (1), this problem can also be solved by running Dijkstra's algorithm multiple times as follows:

(a) Initialize vertices as in Dijkstra's algorithm.

(b) Build a queue as in Dijkstra's algorithm.

(c) Run the loop of Dijkstra's algorithm, popping out each node and updating.

(d) Repeat steps (2) and (3). Interpret the result the same way you normally interpret the output of Dijkstra's algorithm.

Problem 6 *(25 points)*

You are given a weighted connected undirected graph, and a proper subset L of V. You are asked to find a spanning tree of the graph where the nodes in L are leaves and of minimum cost. If there is no spanning tree with all vertices in L as leaves, your algorithm should report this instead.

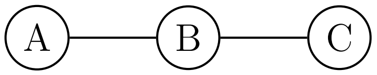
Give an algorithm for this which runs as fast as the MST algorithms that we know. Explain why your algorithm is correct, and why it is not asymptotically slower than the other MST algorithms.

Answer:

Algorithm: Call the set of nodes in V but not in L, L' or $L' = G/L$. The nodes in L' are not constrained as leaves, we can try to run an MST algorithm on this subset of nodes. We name the graph containing the nodes in L' and the edges that connect nodes in L' to other nodes in L' G', and run any MST algorithm on G' . Now, we must attach the nodes in L to the tree. Sort the edges that span from L to L' and and iterate through the list, for each edge (u, v) where $u \in L$ and $v \in L'$, add the edge only if there is not another edge in the tree containing u (this can be checked quickly by marking nodes as we add edges that contain them).

Now we must check that the spanning tree exists. There are a few cases to consider. First, if the graph containing only nodes and edges in L' might be unconnected. If L' is unconnected, then there is no way to create a spanning tree where all the nodes in L are leaves, because in any spanning tree some node $u \in L$ is needed to connect together the unconnected components in G'. This can be checked by running depth first search on the MST over G' returned by Kruskal's or Prim's algorithm.

For example, in the graph below, if $L = [B]$, there is no way to create a spanning tree over G' and therefore no spanning tree where all the nodes in L are leaves.



Second, it's possible that there is some node in L that is only connected to other nodes in L. In the graph above, if $L = [B, C]$, there is no way to connect B and C to a spanning tree without adding 2 edges to B. In the algorithm above, any node in $u \in L$ that only connects to other node in L will remain unconnected (there is never any opportunity add an edge that contains u). Therefore, checking connectivity at the end of the algorithm will tell us if this is the case.

Correctness: This algorithm guarantees to only add an edge incident to nodes in L once and uses a minimal edge to do so. The rest of the graph is connected by minimum weight since we are using an MST algorithm on G'. Therefore, if the algorithm doesn't return an error it will return a spanning tree that is minimal with the constraint that all the nodes in L are leaves.

Runtime analysis: This algorithm relies on MST, DFS to check connectivity, and iterating over the sorted list to attach nodes in L. The algorithm can sort the edges in $|E|log|E|$ time and then iterates over E edges, adding a constant time check to prevent the addition multiple edges containing some node $u \in L$. In total, this is dominated by the runtime of the MST algorithm and therefore cannot be asymptotically slower.