Question 1. *[70 points] Recall the* Traveling Salesperson Problem *(TSP). Given a complete, undirected graph $G = (V, E)$ with non-negative, real-valued edges costs $c : E \to \mathbb{R}$, the goal is to find a minimum cost cycle that visits every vertex in V exactly once. The cost of the cycle is the sum of the costs of the edges in the cycle.* Metric TSP *refers to a class of TSP instances where the edge costs obey the triangle inequality. This means that shortcuts between two cities don't exist; it's best to take the direct route. Formally the triangle inequality means that*

$$c(u, v) \leq c(u, w) + c(w, v) \qquad \text{for all } u, v, w \in V.$$

We will develop a polynomial-time 3/2-approximation for Metric TSP. In all parts, $G = (V, E)$ is the complete, undirected input graph. V is the set of nodes and E is the set of edges where $|V| = n$ and $|E| = m$. c gives the real-valued edge costs that obey the triangle inequality. Sometimes we extend c to a set of edges $E' \subseteq E$ so that $c(E') = \sum_{e \in E'} c(e)$. Please answer the following questions clearly and concisely. If you get stuck on a particular part, move on — you can assume previous results and still make progress.

(a) Let C^* be the cost of the minimum cost tour in G. Let T be a minimum spanning tree in G with cost $c(T)$. Show that $c(T) \leq C^*$.

Proof. Since the minimum cost cycle visits each vertex exactly once, by definition, the cycle has exactly n edges. If we remove any edge from the minimum cost cycle, then we end up $n - 1$ edges, which is a tree, call it T^*, by a theorem from class notes (If a a graph with n nodes has n-1 edges, then it is a tree). Since T^* has 1 less edge, $c(T^*) \leq C^*$; since T is the minimum spanning tree, by definition, $c(T) \leq c(T^*)$. So $c(T) \leq c(T^*) \leq C^*$. □

(b) Use structural induction to show that all trees have an even number of odd-degree nodes.

Proof. We will prove by induction:

base case: tree with 1 node and 0 edges, so we have 0 odd-degree vetices. 0 is is an even number – Check.

Now assume the inductive hypothesis, that all trees with up to N nodes have an even number of odd-degree nodes.

For a tree with $N + 1$ nodes, remove a leaf, now we know that the remaining tree has an even number of odd-degree nodes. Now we add back that leaf, if we connect it to a node with odd degree, then we do not change the total number of odd-degree nodes, since that odd-degree node is now an even node, and our new node is an odd node; if we connect it to a node with even degree, then we have just changed that node to an odd degree node and in effect increased the number of odd-degreed nodes by 2, which means the parity of the total number of odd-degree nodes will remain constant.

So all trees have an even number of odd-degree nodes. □

An undirected multigraph H is an undirected graph with parallel edges. That is, pairs of nodes may have multiple edges between them. An Euler tour *of a connected, undirected multigraph $H = (V', E')$ is a cycle that traverses each edge of H exactly once, although it may visit a vertex more than once. We denote a cycle as a list of vertices, $u_0 \ldots u_{m'}$ where $u_0 = u_{m'}$ and for all $0 \leq i < m'$, (u_i, u_{i+1}) is an edge in E'. Note that an Euler tour has length $m' = |E'|$.*

(c) Show that an undirected, connected multigraph H has an Euler tour if and only if the degree of v is even for each vertex $v \in V'$. (Hint: In the forward direction, think about the cycle representation above. For the backward direction, think about any walk in the graph. What happens eventually? What happens when you remove the walk and consider the smaller problem?)

Proof. For the forward direction, take the cycle represented by a list of vertices $u_0...u_{m'}$. If x appears in this list n times, then it has degree exactly $2n$ because each time it appears as u_i, we get 2 additional incident edges: (u_{i-1}, u_i) and (u_i, u_{i+1}). For the backward direction, if we take a walk from any vertex v_0 until we get stuck, then we will certainly be back at v_0. Because suppose we are stuck on vertex $v_i \neq v_0$, then we would have saw an odd number of incident edges upon v_i, since for each occurrence of v_i before the last we get 2 additional incident edges, and for last occurrence we only get 1. Since v_i by assumption has an even number of edges, there must be another edge incident upon v_i we have not seen before, and this contradicts with the fact that we are stuck at v_i. So $v_i = v_0$.

We will color the vertices in this walk red and then remove the edges in this walk. A vertex will also be removed if all the edges incident to it is removed. The the remaining graph would still have all even-degree vertices, since a cycle has all even-degree vertices, so we can repeat by starting another walk on a red vertex, and repeat the algorithm. If any vertex is left unremoved, a red vertex must exist because otherwise the vertices in our walk would be disconnected from the rest of the graph, which violates our connected assumption.

We repeat algorithm until all all vertices are removed, then we can we can then combine all the cycles we have encountered into one large Euler cycle. This is definitely possible since the graph is connected. The can combine all cycles by first combining two intersecting cycles and then use the combined cycle to combine with other cyles. Even though 2 cycles may have more than one intersection, it is always possible to combine them into one large cycle, by hopping from one cycle to another whenever we encounter an intersection point. See the attached pictoral demonstration on how to combine 2 intersecting cycles into 1. We can also prove that this always works mathematically, given 2 cycles $a_1a_2...x_1...x_2...x_3...x_ka_na_1$and $b_1b_2...x_k...x_5...x_2...b_mb_1$, noting that intersection points may occur in arbitrary order, and that they may not be unique. We will denote $seg(x_i, a)$ as the longest segment ending with x_i in the first cycle that does not contain any other intersection points. $seg(x_i, b)$ is similarly defined for the second cycle. I will define $next(x_i, a)$ as the next intersection point in the first cycle after x_i. Similarly define $next(x_i, b)$, for the second cycle. Our recursive solution is simply $seg(x_1, a)seg(next(x_1, a), b)seg(next(next(x_1, a), b), a)seg(next(next(next(x_1, a), b), a), b)....$, We can see that this recursion will terminate in $2k$ steps given k intersections, since for each x_i function seg is called twice, once for a, once for b. This means we have also traversed all of the k segments in the first cycle and the k segments in the second cycle. □

(d) *Describe (in prose) an $O(|V'| + |E'|)$-time algorithm to find an Euler tour of H if one exists. (Hint: Use your proof of the converse of part (c) to develop the algorithm.) Prove that your algorithm is correct.*

Proof. A modified version of depth-first search would suffice to implement the walk procedure above. Depth-First search has the nice property that when we finished one walk, it will automatically backtrack into a vertex we have already seen – a "red" vertex – and continue on finding another walk/cycle. The only additional work we will need to do is to "remember" the various cycles we have walked through between backtracking steps. One way to do so is to keep a hash-table type linked-list structure. Let c_i represent the i-th Eulerian cycle we find. For the first cycle, we might have $c_1[v_1] = v_2$, $c_1[v_2] = v_3$,..., $c_1[v_{m_1}] = v_1$. After backtracking to finding the second cycle, we will have $c2[v_k] = v_{m_1+1}$, ... where $k \leq m_1$. When we finish, we might have j cycles. We can also keep a $cycle$ hash table with intersection points as key and cycles as value, so that way we can use our algorithm defined in (c). Depth-First search has time complexity $O(|V'| + |E'|)$, and we can extend our algorithm in (c) to combine arbitrary number of cycles into one Euler cycle. We simply start with $while(len(cycle[v_i]) < 2) v_i = c_1[v_i]$, once we exit the loop by hitting an intersection point, we enter another cycle in the list of cycles in $cycle[v_i]$ that we have not visited yet and repeat. Since we are essentially a backtracking procedure and it travels through each edge exactly once. This procedure is $O(|E'|)$, and storing key values in c_i and $cycle$ simply gets absorbed upon each step in the depths first search, so our overall time complexity is still $O(|V'| + |E'|)$. □

A minimum weight perfect matching in a graph G' with n nodes and non-negative edge costs is a matching M of size $n/2$ with minimum cost $c(M)$. Edmonds showed in the 60's how to find a minimum weight perfect matching of a graph in $O(n^4)$ time. Gabow recently improved this running time to $O(n(m + n \log n))$.

(e) *Use parts (a) and (b) along with Gabow's algorithm, to produce first, a minimum weight perfect matching M, and second, a multigraph $H = (V, E')$ where all the nodes in V have even degree. Note that V refers to the same set of nodes as the input graph. Also, you may use Gabow's algorithm as a black box. (Hint: Can you find a perfect matching M that combines with T to form a multigraph H where all the nodes have even degree?)*

Proof. First find the minimum spanning tree T. Since there are an even number of odd-degree nodes in T, we can apply Gabow's algorithm on the subgraph of G consisting of just the odd-degree nodes in T. This way, every one of these nodes will accumulate one more degree. We can also think of this procedure as combining the edges in T with edges in M to form the multigraph H, since every odd-degree node in T has just accumulated one more degree from M, all nodes in H will have even degree. □

(f) *Show that $c(M) \leq 1/2 \cdot C^*$. Using part (a), conclude that $c(E') \leq 3/2 \cdot C^*$.*

Proof. Given the minimum cost cycle $v_1v_2v_3...v_nv_1$, with cost C^*. If n is even, then let $E_1 = \{(v_1, v_2), (v_3, v_4), ..., (v_{n-1}, v_r$, $E_2 = \{(v_2, v_3), (v_4, v_5)...(v_{n-2}, v_{n-1})$. Since $c(E_1) + c(E_2) = C^*$, and that both E_1 and E_2 are matchings, we can pick one, E_i, with lesser cost. Clearly, $c(E_i) \leq \frac{1}{2}C^*$, and since M is a minimum cost matching, we must have $c(M) \leq c(E_i) \leq \frac{1}{2}C^*$. If n is odd, then we can let $E_1 = \{(v_1, v_2), (v_3, v_4), ..., (v_{n-2}, v_{n-1})\}$, $E_2 = \{(v_2, v_3), (v_4, v_5)...(v_{n-1}, v_n)$, and use the same argument. □

(g) *Use parts (c) and (d) along with the multigraph H (and the fact that c obeys the triangle inequality) to produce a cycle $v_0v_1 \ldots v_n$ that visits every vertex in V exactly once. Conclude that this cycle is a 3/2-approximation for the metric TSP problem.*

Proof. We now have an Euler cycle that visits every edge in H once so it may visit some vertex twice. We can trim this path down to Hamiltonian cycle by the following method. When ever we see a duplicate node in our Euler cycle, for instance in $a_1a_2...a_ma_1a_{m+1}....$, we can simply remove all occurences of duplicate a_1 nodes, since graph G is complete, we will still have a cycle. Since graph is Euclidean, $c(a_m, a_{m+1}) \leq c(a_m, a_1) + c(a_1, a_{m+1})$, so this procedure can only decrease the overall cost. Let E' be the Euler cycle in H , and E'' be the Hamiltonian cycle derived from trimming down E'. We must have $C(E'') \leq C(E^{'}) \leq C(M) + C(T) \leq \frac{1}{2}C^* + C^* = \frac{3}{2}C^*$. So we have just found a 3/2-approximation to for the metric TSP problem. □

Question 2. *[30 points] Given a graph $G = (V, E)$ and a natural number k, we define a relation $\overset{G,k}{\rightarrow}$ on pairs of vertices of G as follows. If $x, y \in V$, we say that $x \overset{G,k}{\rightarrow} y$ if there exist k mutually edge-disjoint paths from x to y in G.*

Is it true that for every G and every $k \geq 0$, the relation $\overset{G,k}{\rightarrow}$ is transitive? That is, is it always the case that if $x \overset{G,k}{\rightarrow} y$ and $y \overset{G,k}{\rightarrow} z$, then we have $x \overset{G,k}{\rightarrow} z$? Give a proof or a counterexample.

Proof. We will use the following theorems from the book, note that theorem (7.43) assumes a weight of 1 on each directed edge in the graph G for the underlying network flow problem.

p. 376 (7.43) There are k edge-disjoint paths in a directed graph G from s to t if and only if the value of the max $s - t$ flow in G is at least k.

p. 350 (7.13) In every flow network, the maximum value of an $s - t$ flow is equal to the minium capacity of an $s - t$ cut.

Let us assume a weight of 1 on each directed edge of G, and consider the underlying flow network problem. Let (A_x, B_z) be a minimum capacity $x - z$ cut, then we know that either $(A_x, B_z) \sim (\{x, y...\}, \{z, ...\})$[meaning $y \in A_x$], or $(A_x, B_z) \sim (\{x, ...\}, \{y, z...\})$[meaning $y \in B_z$]. It turns out that for either case we can bound $c(A_x, B_z)$ from below using the capacities of the min-capacity $x - y$ cut and $y - z$ cut respectively. Reference the following table as you read onwards.

	(A_x, B_y)	(A_y, B_z)	(A_x, B_z)	conclusion
Case 1	$(\{x, ...\}, \{y, z, ...\})$		$(\{x, ...\}, \{y, z...\})$	$c(A_x, B_z) = c(A_x, B_y) \geq k$
Case 2	$(\{x, z...\}, \{y, ...\})$		$(\{x, ...\}, \{y, z...\})$	$c(A_x, B_z) \geq c(A_x, B_y) \geq k$
Case 3		$(\{x, y...\}, \{z, ...\})$	$(\{x, y...\}, \{z, ...\})$	$c(A_x, B_z) = c(A_y, B_z) \geq k$
Case 4		$(\{y...\}, \{x, z, ...\})$	$(\{x, y...\}, \{z, ...\})$	$c(A_x, B_z) \geq c(A_y, B_z) \geq k$

In case 1, since we require $z \in B_y$ and we have discovered that $y \in B_z$, we must have $c(A_x, B_z) = c(A_x, B_y)$, because in both cases of finding (A, B) we are reduced to solving the same question, which is: given $x \in A$ and $y, z \in B$, how do we assign other vertices to be either in A or in B so that $c(A, B)$ is minimized?

In case 2, for (A_x, B_y) we have solved the broader question of: given $x \in A$, $y \in B$, how to assign other vertices to minimize $c(A, B)$, and we have found that $z \in A$. Since we require that $y \in B_z$ as a condition, and that $(A_x, B_y) \sim (\{x, z...\}, \{y, ...\})$ implies that, given $y \in B$, $x \in A$, the cut is minimized when $z \in A$, our $x - z$ cut is must have strictly higher capacity than our $x - y$ cut, save the degenerate case when moving z from A to B makes no difference in cut capacity.

Another way to think about these two cases is that since we required $y \in B_z$, the problem of finding min-cut (A_x, B_z) is simply a more restrictive version of finding min-cut (A_x, B_y), the additional restriction being $z \in B$, so if given $y \in B$, $c(A_x, B_y) \leq c(A_x, B_z)$.

For cases 3 and 4, we explore what happens if we require $y \in A_x$. By the exact same argument, we conclude that, if given $y \in A$, $c(A_y, A_z) \leq c(A_x, B_z)$.

Since in all possible solutions to min-cut (A_x, B_z), we must have either $y \in A_x$or $y \in B_z$, so we have certainly covered all possibilities in our 4 cases. By (7.43) we know that the max $x - y$ flow and the max $y - z$ flow is at least k, and by (7.13) we know that their min-cut capacity is also at least k, so now we know that the min $x - z$ cut capacity is also at least k, so using (7.13), we know that the max $x - z$ flow must be at least k, and by (7.43) we know that there has to be at least k edge-disjoint paths going from x to z.

So the aforementioned relation (in the question) is transitive. $\square$