CMSC 27200 - Winter 2023 — Andrew Drucker, Haifeng Xu

due feb. 3, 5:00 pm via gradescope

H3. Bayard Walsh

- The assignment is due at Gradescope on Friday, Feb. 3 at 5:00p.
- You can either type your homework using LaTex or scan your handwritten work. We will provide a LaTex template for each homework. If you writing by hand, please fill in the solutions in this template, inserting additional sheets as necessary. This will facilitate the grading.
- You are permitted to study with up to 2 other students in the class (any section) and discuss the problems; however, *you must write up your own solutions, in your own words*. Do not submit anything you cannot explain. If you do collaborate with any of the other students on any problem, please do list all your collaborators in your submission for each problem.
- Similarly, please list any other source you have used for each problem, including other textbooks or websites. *Consulting problem solutions on the web is not allowed.*
- *Show your work.* Answers without justification will be given little credit.

PROBLEM 1 (25 POINTS) *The Running Sums problem is defined as follows:*

Input: *a sequence* $(a_1, \ldots, a_n)$*, where each* a_i *is either* 1 *or* -1*.*

Desired output: *a sequence* $(b_1, \ldots, b_n)$*, where each* b_i *is either* 0 *or* 1*. Your goal is to* ***minimize*** $\sum_{1 \le i \le n} b_i$*, subject to the* ***constraints*** *that*

$$\sum_{1 \le i \le j} (a_i + b_i) \;\ge\; 0\,, \qquad \text{for all} \quad j \in \{1, 2, \ldots, n\}\,.$$

Give an algorithm for this problem that runs in $O(n)$ *steps. Prove correctness and efficiency.*

(Note: reading/writing from memory, addition, comparison, and logic operations can all be considered unit-cost for this problem.)

Solution:

Algorithm$(a_1, \ldots, a_n)$**:**
$S = \emptyset$ // *ordered set building* $(b_1, \ldots, b_n)$
$T = 0$ // *maintain current sum*
for every j in $(a_1, \ldots, a_n)$
  if $T = 0$ **and** $a_j = -1$**:** // *check if* $T < 0$ *for* $T + a_j$
    $b_j = 1$ // *add* 1 *so* $T = 0$.
      // *No need to update* T *as* $T = 0$ *before and after iteration*
  else:
    $b_j = 0$ // *add* 0 *to minimize*
    $T = T + a_j$ // *Update* T
  $S = S \cup b_j$ // *Build set*
return S // *return ordered set,* $(b_1, \ldots, b_n)$

Halting + Run time: The algorithm builds a set, maintains variable T, and loops through the set $(a_1, \ldots, a_n)$ which is $O(N)$, performing $O(1)$ basic operations in the loop. Therefore the run time is $O(N)$

Correctness:
Feasibility: The algorithm follows the constraints on run time, meaning that it runs in $O(n)$ steps. Furthermore, $T = \sum_{1 \le i \le j}(a_i + b_i)$ for any j, because $T = T + a_j$ whenever $T \neq 0$ ($b_j = 0$ in this case) and adding nothing when $T = 0$ and $a_j = -1$, which is because $b_j = 1$ in this case, meaning $T = T + 0$. Therefore, by tracking the running sum, and adding a 1 whenever $T = 0$ and $a_j = -1$, there is no j in $(a_1, \ldots, a_n)$ where $T < 0$. Because the sum can only increase or decrease by 1, and the algorithm adds $b_j = 1$ whenever it would be negative, the sum is always ≥ 0 for every value in the sequence, meaning that the $\sum_{1 \le i \le j}(a_i + b_i) \;\ge\; 0\,, \qquad \text{for all} \quad j \in \{1, 2, \ldots, n\}$ is followed.

Optimality: Proof by induction: Let there be some optimal solution $F_O(a_1, \ldots, a_n)$ and let there be the greedy solution proposed by the algorithm $F_G(a_1, \ldots, a_n)$ that minimizes $\sum_{1 \leq i \leq n} b_i$. Assume that for every j in $(a_1, \ldots, a_n)$, $F_O(a_1, \ldots, a_n) \geq F_G(a_1, \ldots, a_n)$.

Base case 1 **:** $n = 1$. Assume $a_1 = 1$. Therefore $b_1 = 0$ by $F_G(a_1)$, and we have $F_G(a_1) = 1$. As b_1 could only be 1 or 0, $F_G(a_1)$ is optimal so therefore $F_O(a_1) \geq F_G(a_1)$ for $n = 1$ and $a_1 = 1$.

Base case 2 **:** Then, assume $a_1 = -1$. Therefore $b_1 = 1$ by $F_G(a_1)$, and we have $F_G(a_1) = 0$. As b_1 could only be 1 or 0, if $b_1 = 0$, we have $-1 \ngeq 0$, which violates the constraints. Therefore it must be that $b_1 = 1$, and therefore $F_O(a_1) \geq F_G(a_1)$ for $n = 1$ for either -1 or 1.

Induction: Assume $F_O(a_1, \ldots, a_n) \neq F_G(a_1, \ldots, a_n)$. Therefore there must be some point where the $(b_1, \ldots, b_n)$ given by $F_O(a_1, \ldots, a_n) \neq F_G(a_1, \ldots, a_n)$. Let j be the final b before the two sequences first diverge. Let b_{j+1} be the $j+1$ element from $(b_1, \ldots, b_j, b_{j+1})$ produced by $F_G(b_1, \ldots, b_j, b_{j+1})$ and let b'_{j+1} be the $j+1$ element from $(b'_1, \ldots, b'_j, b'_{j+1})$ produced by $F_O(b'_1, \ldots, b'_j, b'_{j+1})$. Therefore, $F_O(b_1, \ldots, b_j) = F_G(b_1, \ldots, b_j)$ and $b_{j+1} \neq b'_{j+1}$.

Assume that $T > 0$ or $a_{j+1} \neq -1$. Therefore $b_{j+1} = 0$, meaning that $b'_{j+1} = 1$. We have $F_G(b_1, \ldots, b_j) \leq F_O(b_1, \ldots, b_j)$ by inductive hypothesis, and as $b_{j+1} = 0$, we have:

$$F_G(b_1, \ldots, b_j) = F_G(b_1, \ldots, b_j, b_{j+1}) \leq F_O(b_1, \ldots, b_j) + 1 = F_O(b_1, \ldots, b_j, b_{j+1})$$

Therefore, we have $F_G(b_1, \ldots, b_j, b_{j+1}) \leq F_O(b_1, \ldots, b_j, b_{j+1})$, when $T > 0$ or $a_{j+1} \neq -1$.

Therefore, assume that $T = 0$ and $a_{j+1} = -1$. Therefore $b_{j+1} = 1$ and as $b_{j+1} \neq b'_{j+1}$, $b'_{j+1} = 0$. As T equals the current sum, $\sum_{1 \leq i \leq j}(a_j + b_j) = 0$. As $a_{j+1} = -1$ and $b'_{j+1} = 0$, $\sum_{1 \leq i \leq j+1}(a_{j+1} + b'_{j+1}) = -1$ at $j+1$ for $F_O(b_1, \ldots, b_j, b_{j+1})$. Therefore for $j+1$, $\sum_{1 \leq i \leq j+1}(a_{j+1} + b'_{j+1}) \ngeq 0$, which is a violation of the constraints. As $a_{j+1} = -1$ and $b_{j+1} = 1$, $\sum_{1 \leq i \leq j+1}(a_{j+1} + b_{j+1}) = 0$, $\sum_{1 \leq i \leq j+1}(a_{j+1} + b_{j+1}) \geq 0$, which is not a violation of the constraints. Therefore, when $T = 0$ and $a_{j+1} = -1$, it must be that $b_{j+1} = 1$, meaning that

$$F_G(b_1, \ldots, b_j) + 1 = F_G(b_1, \ldots, b_j, b_{j+1}) \leq F_O(b_1, \ldots, b_j) + 1 = F_O(b_1, \ldots, b_j, b_{j+1})$$

Therefore we have $F_G(b_1, \ldots, b_j, b_{j+1}) \leq F_O(b_1, \ldots, b_j, b_{j+1})$, when $T = 0$ and $a_{j+1} = -1$ and when $T > 0$ or $a_{j+1} \neq -1$. Therefore for every $j+1$, the greedy algorithm is equal to or better than the optimal algorithm, meaning it stays ahead and is optimal.

PROBLEM 2 (25 POINTS) *Here we look at attempts to solve graph problems with negative edge weights, by solving a related problem with nonnegative edge weights.*

For purposes of both parts of this problem, graphs have no self-loops or multiple edges, except that a directed graph may contain both (u, v) and (v, u). ***We will consider undirected graphs in part (a), and directed graphs in (b).***

Given a directed or undirected graph $G = (V, E)$ with edge weights $\{w_e\}_{e \in E(G)}$ (some of which may be negative), let $C > 0$ be chosen sufficiently large that $w_e + C > 0$ for all $e \in E(G)$. Define new weights $\{w'_e\}_{e \in E(G)}$ by $w'_e = w_e + C$.

(a) *Let G be an* ***undirected*** *graph, and $T \subseteq E(G)$ be a Minimum Spanning Tree (MST) for G with respect to the weights $\{w'_e\}$. Must T also be an MST for G with respect to the original weights $\{w_e\}$? Prove or give a counterexample.*

*(**Note:** if you give a counterexample in either (a) or (b), you may briefly explain, rather than carefully prove, why it refutes the claim.)*

(b) *Now we are given a* ***directed*** *graph G. We assume that the weight function $\{w_e\}_{e \in E(G)}$ may have negative values, but has* no *cycles of negative total weight.*

(In this problem the terms "path" and "cycle" both allow for repeated use of edges and vertices, but edges in a path or cycle must follow the edge's direction.)

Suppose that P is a path in G from one vertex s to a distinct vertex t, and among all such s-t paths, P has minimal total weight[1] with respect to the modified edge weights $\{w'_e\}$ as above. Must P have minimal total weight among s-t paths with respect to $\{w_e\}$? Prove or give a counterexample.

Solution:

A:

Let G be an undirected graph, and $T \subseteq E(G)$ be a Minimum Spanning Tree (MST) for G with respect to the weights $\{w'_e\}$. In order to prove that T must be a Minimum Spanning Tree (MST) for G with respect to the weights $\{w_e\}$: proof by contradiction. Therefore, assume that T is not an MST for $\{w_e\}$. As the only difference between $\{w'_e\}$ and $\{w_e\}$ is the weights of the edges of tree, then there must be some edge for the weights of $\{w'_e\}$ which produces an MST for $\{w'_e\}$ but does not produce an MST for $\{w_e\}$. Let this edge be $e' = (u, v)$. Since T is a spanning tree, adding any edge will create a loop. Therefore suppose that there is a cutset S in between (u, v) such that an edge in T, e, could be removed to remove the loop and maintain the spanning tree property. If T is a minimum spanning tree for $\{w'_e\}$, we know that $\{w'_e\} \leq \{w'_{e'}\}$, otherwise we would have a violation of the MST property. However, we know that $(w'_e = w_e + C) \forall w'_e$ by the weight function definition. Therefore, we have

$$\{w'_e\} = w_e + C \leq \{w'_{e'}\} = w_{e'} + C$$

[1] (but is not necessarily the unique minimal-weight path)

Which implies

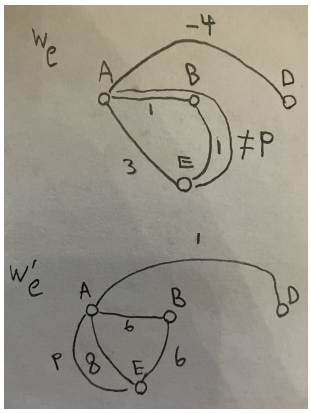
$$w_e + C \leq w_{e'} + C$$

And therefore

$$w_e \leq w_{e'}$$

Therefore, for any arbitrary edge e', it is impossible for $w_e > w_{e'}$ if $\{w'_e\} \leq \{w'_{e'}\}$, meaning that no edges $\notin T$ can be added to decrease the overall weight of T. Therefore, the MST T with respect to $w'_{e'}$ must also be an MST for G with respect to the original weights $\{w_e\}$

B: Brief Explanation: The idea is that when you have two edges with weights 1 and 1 on a path and another edge with weight 3 on a path you have $1+1$ and 3 for the respective paths, however when you add by constant C to every edge, you have $3+C$ and $1+1+2C$, which for some C, $1+1+2C$ can be greater than $3+C$, but when C is removed the two edge path is shorter, providing a counter example.
Formal Counter example: Let G be a directed graph with the following edges (where (u, v) indicates an edge from u to v: $G = (A, B), (B, E), (A, E), (A, D)$. Let G have the following edge weights, $w_e : (A, B)_w = 1, (B, E)_w = 1, (A, E)_w = 3, (A, D)_w = -4$. Therefore, as $C > 0$ must be chosen sufficiently large that $w_e + C > 0$ for all $e \in E(G)$, as $min(\{w_e\}_{e \in E(G)}) = -4$, and $-4 + C > 0$, $C > 4$. Therefore, let $C = 5$. Therefore, we have the following new edge weights $w'_e : (A, B)_{w'} = 6, (B, E)_{w'} = 6, (A, E)_{w'} = 8, (A, D)_{w'} = 1$ by $w'_e = w_e + C$. Let $s = A$ and let $t = E$, therefore path P must have the minimal total weight with respect to modified edges weights above. There are two $A - E$ paths; (A, E) and $(A, B), (B, E)$. By their weight functions, we have $(A, E)_{w'} = 8$ and $(A, B)_{w'} = 5, (B, E)_{w'} = 5$, or $(A, B, E)_{w'} = 10$. As $10 > 8$, the minimum total weight path for (A, E) for w'_e is 8, so $P = (A, E)$. However, for minimum total weight among $A - E$ paths for w_e, we have the same two $A - E$ paths; (A, E) and $(A, B), (B, E)$, and we have the following weights: $(A, E)_{w'} = 3$ and $(A, B)_{w'} = 1, (B, E)_{w'} = 1$, or $(A, B, E)_{w'} = 2$. As $2 > 3$, the minimum total weight path for (A, E) for w_e is 2, which is (A, B, E). As $(A, B, E) \neq (A, E)$, P is not the minimal total weight among $A - E$ paths with respect to w_e. Drawing below.


We
A
B
D
-4
1
1
≠P
3
E
We'
1
A
B
D
6
P
8
E
6

PROBLEM 3 (25 POINTS) *Solve exercise 19 (on "bottleneck rates") from Chapter 4 in the Kleinberg-Tardos textbook.*

Solution:
Input: A connected undirected graph G, and a bandwidth function b_e which gives an integer value for all $e \in G$

Desired Output: A spanning tree which for each edge $u, v \in G$, the bottleneck rate of the $u - v$ path in T is equal to the best achievable bottleneck rate (the max minimum value of b_e along the path from edge $u - v$) for the pair u, v in G.

Algorithm(G, b_e)

Invert all weights or $* - 1 \forall E \in G$
Sort edges by cost and label so that $b_e(e_1) \leq b_e(e_2) \leq \cdots \leq b_e(e_n)$
"Un-Invert" all weights, or $* - 1 \forall E \in G$
$T = \emptyset$
for all $v \in V$ do:
    Make set $\{v\}$
for i from 1 to n do:
    Let e_i be (u, v)
    if u and v are not in the same set **then:**
        $T = T \cup \{e_i\}$
        Take the union of the sets containing u and v
return T

Run time: Assume that we are using the union-find data type introduced in class for Kruskal's algorithm. Therefore, we have the following operations and run-times to consider: make-set$(v) = O(1)$ time. find-set$(v) = O(logn)$ time. union$(u, v) = O(1)$ time. Running time should be $O(mlogn)$, where $m = |E|$ and $n = |V|$, which will be explained as follows. First, we invert every edge, which takes $O(N)$. Then we sort the edges by weight from largest to smallest weight, which takes $O(m \cdot logm)$ time. Then we "un-invert" every edge which takes $O(n)$. Then, we perform n make-set operations, which takes $O(n)$ time, as it is $O(1)$ for n iterations. We then loop and examine each edge, meaning that the actions in the loop will be multiplied by m. We check if the endpoints of each edge are in the same set. When edges don't share the same set, we add that edge to the tree and take the union of the two sets, gradually combining all edges. Therefore, we perform two find operations for each edge and either one or zero union operations. As union(u, v)= $O(1)$ and find-set(v) =$O(logn)$, and we loop through m edges, this operation will be a total of $O(m \cdot logn)$ time. Therefore, we have $O(m \cdot logm) + O(n) + O(n) + O(n) + O(m \cdot logn)$

run time. However, as we have $m \leq n^2$, we also have $\log m \leq 2 \log n$, and subsequently $O(logm) \leq O(logn)$. Therefore the largest term is $O(m \cdot logn)$, so we have $O(m \cdot logn)$ as the overall run time.

Correctness:
Feasibility: Proof that the algorithm produces a spanning tree: Assume that T is not a spanning tree. Therefore, there must be a cycle in the graph. However, it is impossible to add an edge which forms a cycle, because this would imply that we added an edge which joins two endpoints which were already in the same tree, which is avoided in the algorithm by the line "if u and v are not in the same set **then:**". Therefore suppose that T is not connected. However, as we know that G is a connected graph, and we check every edge for connection then there would be an edge added to T which would eventually add all edges. Therefore the algorithm produces a spanning tree.

Optimality: In order for the algorithm to be optimal, it must construct a spanning tree T which for each $u, v \in V$, the bottleneck rate of the $u - v$ path in T is equal to the best achievable bottleneck rate for the pair any pair u, v in G.
Lemma: The tree produced by T is a "maximum-spanning-tree": Assume that we invert all tree weights, or multiply by -1 and attempt to minimize. Therefore, in order to maintain our sorting order, we would sort by $b_e(e_1) \leq b_e(e_2) \leq \cdots \leq b_e(e_n)$. This gives us Kruskal's algorithm, which we know gives a minimum spanning tree. Therefore, invert all weights and minimize, giving a spanning tree. Then we un-invert weights, maintaining the ordering. Let T_p be the set of all possible spanning trees for G. Let b_i be the set of weights for T after inversion and let b_m be the set of weights for T when un-inverted. Since T is a minimum spanning tree when inverted, we have $\sum_{e \in T} b_i = min_{T \in T_p} \sum_{e \in T} b_i$, and after inversion it implies, $\sum_{e \in T} b_m = max_{T \in T_p} \sum_{e \in T} b_m$. Therefore, T after un-inversion is a maximum spanning tree.

Lemma: As T is a spanning tree, there is only one path between two arbitrary nodes u and v. As T is a spanning tree, then the nodes u and v must be connected, via path p of edges $\{e_1, e_2 \cdots e_k\}$ from $u - v$. Assume that there is another path p' of edges $\{e'_1, e'_2 \cdots e'_j\}$ from $u - v$. However, this would produce a cycle, $\{e_1, e_2 \cdots e_k, e'_j \cdots e'_2, e'_1\}$ from $u - v$ and then $v - u$. As cycles cannot occur in trees by definition, this is impossible. Therefore in T, there is only one path between two arbitrary nodes u and v.
Proof of best bottleneck rate for every $u - v$ by contradiction: As proved previously T is a maximum spanning tree produced by the algorithm. Consider T after the algorithm runs, and suppose that it is not optimal. Therefore there must be some path p' between two arbitrary nodes $u - v$ where the best achievable bottleneck rate is better than that produced by the path p between $u - v$ in T. As there is only one path p in between $u - v$, as proved by the

previous lemma, and $p \neq p'$ for at least one edge $\in p$ (otherwise $p = p'$ and p would have the best achievable bottleneck rate and we would be done) then p' must contain some edge not in p and subsequently not in the tree of T where adding this edge would achieve a better achievable bottleneck rate. Let e' be this edge on the path p'. Furthermore, by adding an edge to a spanning tree, we have a loop, and a corresponding cutset with two edges on the two paths between $u - v$. Therefore, let e be the corresponding edge on p such that e could be exchanged for e' and maintain the T property of maximum spanning tree. Therefore we have a loop, L, made from the two paths to $u - v$ which contains both e and e'. However, if $b_e < b_{e'}$, then T would contain the edge with the lowest bottleneck rate in a cycle, which is a definition violation of the maximum spanning tree. Therefore this is a contradiction, and adding e' cannot improve the bottleneck rate for the path between any two nodes, meaning that our path p created by T has the best bottleneck rate between u, v in G. We can apply this to any set of nodes, meaning that for all arbitrary (u, v), T has the best achievable bottleneck rate. Therefore, for each $u, v \in V$, the bottleneck rate $u - v \in T$ provides the best achievable bottleneck rate $\in G$. Therefore, the maximum spanning tree we proved we created earlier is equivalent to the tree with the best achievable bottleneck rate for every pair of nodes. Therefore the solution exists and we have produced it in our algorithm. Note that there is also the possibility an edge of equal weight to one $\in T$ which is not in T, therefore T is not unique, although there are no edges that could be swapped out to create a greater bottleneck amount.

PROBLEM 4 (25 POINTS) *Solve exercise 28 (on X- and Y-edges in spanning trees) from Chapter 4 in the Kleinberg-Tardos textbook.*

Solution:
Inputs: Graph G and integer K which is the desired amount of X edges in spanning tree.

Desired Output: A spanning tree T from G with exactly K X edges or reports correctly that no such tree exists.

START OF ALGORITHM

Algorithm($G = (V, E), K$ **)**

Let all edges with x have weight 1 and let all edges with y have weight 2.
Sort edges in E such that $w_e(e_1) \leq w_e(e_2) \leq \cdots \leq w_e(e_n)$, so all x come before y and relabel so that $E = \{x_1, x_2, \cdots, x_q, y_{q+1}, y_{q+2} \cdots, y_n\}$ *// Note that there is no hierarchy in $x_1, x_2, \cdots, x_q$, just that all x are before any y*
    $x_c = 0$ *// counts X edges*
    $T = \emptyset$ *// spanning tree of edges*
for all $v \in V$ do:
    Make set $\{v\}$
for i from 1 to n do: *//Create spanning tree which prioritizes all possible X edges*
    Let e_i be (u, v)
    if u and v are not in the same set **then:**
        $T = T \cup \{e_i\}$
        Take the union of the sets containing u and v
        if $e_i = X$ **then:** *// Check for X edge value*
            $x_c = x_c + 1$ *// Increment when x edge is added*
if $x_c < K$ **do:** *// Check if enough X edges to produce T*
    return **no tree possible. terminate.**
if $x_c = K$ **do:** *// Terminate if exactly X edges in T*
    return T. **terminate.**

for i from $q + 1$ to n do: *//Iterate through Y edges again for potential swaps*
    If $e_i \notin T$ **do:** *//Ignore Y edges already in set*
        $e_i' = null$ *//Initialize exchange edge*
        $L = \emptyset$ *//Initialize loop set*
        $T = T \cup e_i$ *// Add e_i to T*
        Find loop in T. Add all edges in loop to L
        for every j in L **do:**
            $T = T - e_j$ *//Remove edge*
            Check if T has a loop *//check if "right edge" is removed*
            If T does not have a loop **do:**
                $e_i' = e_j$
                exit "for every j in L do: " loop
                // do not add back edge, as e_j = "right" e_i', so this is the exchange step
            else:
                $T = T \cup e_j$ *// add back edge, as $e_j \neq$ "right" e_i'*

        If $e_i' = x$ **do:** *// Check if X edge*
            $x_c = x_c - 1$ *// Decrement edge counter*
            if $x_c = K$ **do:** *// Terminate if exactly X edges in T*
                return T. **terminate.**
return **no tree possible. terminate.**

END OF ALGORITHM

Run time: Assume that we are using the union-find data type introduced in

class for Kruskal's algorithm. Therefore, we have the following operations and run-times to consider: make-set$(v) = O(1)$ time. find-set$(v) = O(logn)$ time. union$(u, v) = O(1)$ time. Running time should be $O(m^3)$, where $m = |E|$ and $n = |V|$, which will be explained as follows. First, we sort the edges by x or y, which takes $O(m \cdot logm)$ time. *note: because the requirements are simpler here (only two options* 2, 3*), sorting could be faster, but we will stick with given sorted time*. Then, we perform n make-set operations, which takes $O(n)$ time, as it is $O(1)$ for n iterations. Then we perform a modified Kruskal's loop (only difference is maintaining a constant for X edges, but that is $O(1)$). Therefore we will follow that approach for run time analysis. We then loop and examine each edge, meaning that the actions in the loop will be multiplied by m. We check if the endpoints of each edge are in the same set. When edges don't share the same set, we add that edge to the tree and take the union of the two sets, gradually combining all edges. Therefore, we perform two find operations for each edge and either one or zero union operations. As union(u, v)= $O(1)$ and find-set(v) =$O(logn)$, and we loop through m edges, this operation will be a total of $O(m \cdot logn)$ time. This marks our run time until we begin the second loop, which considers Y variables. Exchange argument (inner loop) is explained as follows. Here, potentially for each Y edge we add, we first find a loop, which takes $O(m)$ time. Second, we add the value to the loop set, and remove each value and then attempt to find if there is a still a loop for each removed value. Therefore, this exchange argument takes $O(m^2)$ time for each edge e_i added, inside a loop which potentially has M iterations. Therefore, we have $O(m^3)$ run time. In the second part of the loop there are constant time operations of updating the set and the counter and checking equivalences. Therefore, in the loop we have m^2 for y edges, which is worst case run time m^3. Therefore, we have $O(m \cdot logm) + O(n) + O(m^3)$. Therefore the largest term is $O(m^3)$, so we have $O(m^3)$ as the overall run time.

Correctness:
Feasibility: Similar to above, proof that the algorithm produces a spanning tree: Assume that T is not a spanning tree. Therefore, there must be a cycle in the graph. However, it is impossible to add an edge which forms a cycle, because this would imply that we added an edge which joins two endpoints which were already in the same tree, which is avoided in the algorithm by the line "if u and v are **not in the same set** then:". Therefore suppose that T is not connected. However, as we know that G is a connected graph, and we check every edge for connection then there would be an edge added to T which would eventually add all edges. One different aspect from this algorithm compared to the algorithm in problem three is that this algorithm removes and subsequently adds edges in the second loop. However, as the algorithm only adds one edge at a time and then removes the edge which create a loop, we maintain our spanning tree property at every step. To see this, recall that the addition of an edge e_i, creates a unique cycle with e_i', but we consequently break that cycle by removing e_i' once we identify it, meaning that the algorithm maintains a spanning tree with no cycles at every step. The

graph T remains connected because any pair of vertices connected by a path that contained e_i' will now be connected by a path that contains e_i, as they form a loop through the two connected components. In other terms, they are in the same cutset, meaning they connect the same connected components. Therefore the algorithm produces a spanning tree if a spanning tree can be returned.

Optimality: We will break this proof into a few lemma:
Lemma: After the completion of the first loop, we have a spanning tree with the max amount of X edges possible in a spanning tree.
Proof: As the edges are sorted so that all x come before y (with assigning a value where $x < y$ to each edge and running Kruskal's), when the algorithm iterates through i from 1 to n, every x edge will be added or is redundant before a single y edge is considered. Similar to Kruskal's algorithm proved in class, the algorithm will gradually join sub-trees with every x edge being considered before any y edge. Note that with the weight function assigning 1 to each x edge and 2 to each y edge, and considering that the first loop is a modified version of Kruskal's algorithm (x_c counter is added), this promises that an MST will be produced. In this sense, an MST is a spanning tree with the most x edges (because $w_x < w_y$ for all edges). Therefore, a spanning tree with max X edges will be produced in the first step.
Lemma: If $x_c < K$ by the line "if $x_c < K$ do:", no spanning tree with K amount of X edges is possible
Proof: As proved above, there is no way to add another x edge to T after every edge is considered, because T has the maximum amount of x edges already. As x_c increments for every x edge added, if $x_c < K$, there is no way to create a spanning tree that covers K X edges, meaning that early termination is appropriate.
At this point if $x_c = K$, return T, as T is a spanning tree with K X edges.
If the program has not yet terminated, we know that $x_c > K$, as we have already had terminating conditions for $x_c \geq K$. Therefore, we will attempt to prune X edges until we reach $x_c = K$.
Lemma: The second loop, "for i from $q+1$ to n do:" will exchange x edges until we have $x_c = K$ edges or until the loop terminates.
Proof: Similar to the minimum value in a cutset proof belonging to an MST, we will form a sort of cutset for every potential $e_i \in \{e_{q+1}, e_{q+2} \cdots, e_j\}$. Note that based on sorting and starting at $q+1$ **only Y edges are considered to be added**. First, we don't consider y edges already $\in T$. For an edge $\notin T$, e_i assume that it connects two arbitrary nodes (u, v). As proved previously, as T is a spanning tree, adding any edge creates a cycle. Therefore we add e_i and intentionally create a cycle. Therefore, there must be some edge between (u, v), e_i' such that both e_i' and e_i connect the same two connected components. Add e_i, and use a BFS algorithm to find the cycle. Once we find the cycle, we add all nodes from that cycle to the "loop set" L. For every j in L, $e_1, e_2 \ldots e_r$, where $r = |L|$ we follow this strategy: first, remove the chosen edge e_j from T. Second, attempt to find a cycle in T. If there is still a cycle, we know that we have don't have the "right edge", which is the edge that is in the same

cut set as e_i, so we re add e_j and repeat until an edge from j can be removed such that T no longer has a cycle. Not having a cycle indicates that we have removed the correct edge, making an exchange and maintaining the spanning tree property. Eventually, this edge will be found, because adding any edge to a spanning tree will create a cycle which implies that there must be some edge which can be exchanged for that given edge to maintain the spanning tree property. Therefore, once found we don't re add e_j and instead set it equal to e_i', and terminate the "for every j in L do: " loop. We check if $e' = x$ (or if e' is an x edge). If it is, we know that we have made a swap which has decreased the amount of overall x edges, and therefore we decrement our counter x_c. If $e' = y$, then we have exchanged a y edge for another y edge, meaning our amount of overall y edges is the same. Note also that every time we exchange e_i' for e_i, T still maintains its spanning tree property since any pair of vertices connected by a path that contained e_i' is now connected by a path that contains e_i. Every time we remove an x edge we check for $x_c = K$ to see if we can terminate the algorithm and return a T spanning tree with x edges. Therefore, exchanging every Y edge not in the initial T and seeing if it can be exchanged for an x edge will only cause the overall amount of x edges to either stay the same or decrease with every iteration. As we check the counter with each iteration, every spanning tree from the maximum amount of x edges to the termination of the loop will be considered. Therefore, either the loop returns a tree with k x edges or the loop terminates.

Lemma: Once the "for i from $q+1$ to j do:" loop terminates we have a spanning tree T with the maximum amount of y edges, and therefore the minimum amount of x edges.

Proof: Assume that this is not true. Therefore, there must be some edge y_o that could be added to T by the time the loop terminates to increase the overall amount of y edges. As every y edge $\in G$ is considered and only not added if y is already $\in G$, and edges are only removed by another y connecting the same connected components, then there is no way to add y_o while maintaining its spanning tree status in order to increase the overall amount of y edges. Therefore, T is a spanning tree with the maximum amount of y edges. As edges are only the y or x edges, then the maximum amount of y edges corresponds to the minimum amount of x edges (x edges= k, y edges=$n-k-1$).

Proof of optimality: As shown previously, T begins as a spanning tree with the maximum amount of x edges, then it reduces its x edges by 1 (checking if $k = |x| \in T$ each time) until it has the minimum amount of x edges. As each edge is only an x or y edge, if $x_c \neq k$ for every permutation between the maximum x edges to the minimum amount of x edges, or for k, **X edges when MAX**x_c, (the upper bound) $< k$ or $k <$ **X edges when MIN**x_c, (the lower bound), then there is no spanning tree producible where $|x| = k$. Therefore the algorithm will return a spanning tree if possible and **no tree possible** if not.