First Name: Zhicheng
Last Name: Yan
Student ID: 1002643142

First Name: Zhihong
Last Name: Wang
Student ID: 1002095207

First Name: Kecheng
Last Name: Li
Student ID: 1001504507

We declare that this assignment is solely our own work, and is in accordance with the University of Toronto Code of Behaviour on Academic Matters.

This submission has been prepared using LaTeX.

Question 1

(a). Solution:

```
1       AddMST(G, w, T, e1, w1):
2           e1 = {u, v}
3           Generate path P form u to v by BFS
4           em = max(all edges in P with weight)
5           if w(em) <= w1:
6               return T
7           else:
8               return T union {e1} - {em}
```

Time complexity:
By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.
Assume $n = |V|$ and $m = |E|$
By line 3, BFS costs $O(n + m)$.
By line 4, getting em costs $O(m)$.
Therefore, the worst-case running time is $O(n + m)$.

Show correctness:
prove by contradiction:

Note: Define a MST T contains a path P between u and v by definition of spanning tree. So for every edge $e \in P$, $T \cup \{e_1\} - \{e\}$ is another spanning tree. Because T is MST in G, for all $e \in P$, $w(e) \leq w(e_1)$ (using the original value of $w(e_1)$). Otherwise, we could swap e_1 for some edge $e \in P$ to get a spanning tree whose total weight less than $w(T)$. Let $T_1 = AddMST(G, w, T, e_1, w_1)$. Either $T_1 = T$, in this case $w(T) = w(T_1)$; or $T_1 = T \cup \{e_1\} - \{e_m\}$, in this case $w(T_1) < w(T)$ (when $w_1 < w(e_m)$).

Let $T_1 = AddMST(G, w, T, e_1, w_1)$ and, assume T_1 is not MST in G_1.
(i.e. there is some spanning tree T' in G_1 with $w(T') < w(T_1)$. Either $e_1 \notin T'$ or $e_1 \in T'$.)

Case 1:
If $e_1 \notin T'$, then T' is a spanning tree in the orignal graph G with $w(T') < w(T_1) \leq w(T)$.
Contradiction since T is a MST in G.

Case 2:
If $e_1 \in T'$, then consider $T' - \{e_1\}$. This consists of two connected components X and Y, each containing one endpoint of e_1.

Claim: at least one edge $e' \in P$ has one endpoint in X and another endpoint in Y. Otherwise, T' would contain every edge in P in addition to e_1, which will cause a cycle. We know $w(e') \leq w(e_m)$ because e_m is maximum weight edge on path P. Then:
$w(T') < w(T_1)$
$\rightarrow w(T') - \{e_1\} < w(T_1) - \{e_1\}$
$\rightarrow w(T') - \{e_1\} \cup \{e'\} < w(T_1) - \{e_1\} \cup \{e_m\} = w(T)$
(i.e. $T' - \{e_1\} \cup \{e'\}$ is a spanning tree in the graph G, $w(T') < w(T)$)
Contradiction since T is a MST in G.

In both cases, we reach a contradiction.
Therefore, T_1 is a MST in G_1.

(b). Solution:

```
1     DelMST(G, w, T, e0):
2         if e0 not in T:
3             return T
4         else:
5             e0 = {u, v}
6             run BFS for all edges in T − {e0}, begin at u;
7                 assign white to every vertex encountered
8             run BFS for all edges in T − {e0}, begin at v;
9                 assign black to every vertex encountered
10            for all edges in E − {e0}:
11                find a minimum−weight edge e1 with one white
                      endpoint and one black endpoint.
12            if there is no e1:
13                return NIL
14            else:
15                return T − {e0} union {e1}
```

Time complexity:
By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.
Assume $n = |V|$ and $m = |E|$
By line 6 and 8, BFS costs $O(n + m)$.

By line 10 and 11, finding e_1 costs $O(m)$.
Therefore, the worst-case running time is $O(n + m)$.

Show correctness:
prove by contradiction:

Note: Define a MST T in original graph G.
If $e_0 \notin T$: We have $T_0 = T$, then the algorithm is correct because it returns T at line 3.
If $e_0 \in T$:
If there is no edge in E that connects the trees which created by deleting e_0 in T (for example, let's say X and Y), then removing e_0 makes G disconnected. In this case, it is impossible to build a spanning tree, and there is no solution, which is returned at line 13 in the algorithm.(i.e. it returns NIL if loop through the edges and cannot find an edge connecting X and Y.)
Else, removing e_0 from T gives us two connected components (for example, let's say C_1 and C_2) since T is MST in G. Our algorithm is able to find the minimum cost edge e_1 between C_1 and C_2, and adds this to $T - \{e_0\}$ as result.

Assume that the result tree $T_0 = DelMST(G, w, T, e0) = T - \{e_0\} \cup \{e_1\}$ is not a MST in G_0. Then there must exist some other spanning tree $T' \in E - \{e_0\}$, which is MST, such that $w(T') < w(T_0)$.

Case 1:
If $e_1 \notin T'$, then there must be some other edge e' connects a vertex from C_1 to a vertex from C_2. Otherwise there will be a disconnection and T' will not spanning. Then adding e_0 back and removing e' from T', we can get a spanning tree T'' for original graph G such that $w(T'')$ satisfies the following:
$w(T'') = w(T') - w(e') + w(e_0) < w(T_0) - w(e') + w(e_0)$
$\rightarrow w(T'') < w(T) - w(e_0) + w(e_1) - w(e') + w(e_0)$
(#$w(T_0) = w(T) - w(e_0) + w(e_1)$ based on algo and $w(T') < w(T_0)$)
$\rightarrow w(T'') < w(T) + w(e_1) - w(e')$
$\rightarrow w(T'') < w(T) + w(e_1) - w(e') \leq w(T)$
(#e_1 is the minimum cost edge connecting a vertex from C_1 to a vertex from C_2, so $w(e_1) \leq w(e')$)

Contradiction since T is a MST in G.

Case 2:
If $e_1 \in T'$, then $E - e_0 = C_1 \cup C_2 \cup \{e_1\}$. ($e_1$ connects C_1 and C_2)
Since $e_1 \in T'$, T' is a spanning tree for $E - e_0$, choose one endpoint of e_1 as root of T', then at least one of subtrees in T' contains same vertices in C_1 or C_2, otherwise T' is not a spanning tree.
Use C_1' and C_2' represent subtrees.
Assume $w(C_1') < w(C_1)$ or $w(C_2') < w(C_2)$ because $w(T') < w(T)$, at least one of C_1' and C_2' will has less weight than its counterpart. Let's say $w(C_1') < w(C_1)$ now.
Then $w(C_1' \cup C_2 \cup e_0) < w(C_1 \cup C_2 \cup e_0) = w(T)$.
Contradiction since T is a MST in G.

In both cases, we reach a contradiction.
Therefore, T_0 is a MST in G_0.

Question 2

(a). Solution:
No.
Counter-example:
$G = (V, E)$ with $V = \{x, y, z\}$, $E = \{(x, y), (y, z)\}$, $c(x, y) = c(y, z) = 1$, and $L = \{y\}$.
Since G is already a tree, there is only one spanning tree of G, which is G itself, and y is not a leaf in G.

(b). Solution:
When $G = (V, E)$ with $V = \{x, y, z\}$, $E = \{(x, y), (y, z), (z, x)\}$, $c(x, y) = c(y, z) = 1, c(x, z) = 2$, and $L = \{y\}$, then G contains exactly one MST: $T = \{(x, y), (y, z)\}$, but T is not an solution to the MST with Fixed Leaves problem because y is not a leaf of T.
Also, there are 2 optimal solutions to the MST with Fixed Leaves problem: $T_1 = \{(x, z), (x, y)\}$ and $T_2 = \{(x, z), (y, z)\}$. They both are not MST in G.

(c). Solution:

```
1     GreedyAlgo(V, E, L):
2         if |V| == 2, |E| == 1, |L| == 2:
3             return E
4         if |L| > log|E|:
5             return NIL
6         init a MST T
7         for v in L:
8             let c = infinity
9             for (v,u) in E:
10                E = E - {(v,u)}
11                if (u not in L) and (c(v,u) < c):
12                    e = (v,u)
13                    c = c(v,u)
14            T = T union {e}
15        run BFS to check if discovered vertices = |V|-|L|:
16            if not, return NIL
17        run Kruskal on E
18        return T
```

Algo description:
Our algo is a variation of Kruskal's algorithm: we construct the tree edge-by-edge, starting with the nodes in L. We handle the only special case

when two nodes of L must be connected to each other at very beginning. Also, if $|L| > log|E|$, then the given G contains no solution. Then, we handle the general case. After initialized a spanning tree T, we select one edge (v, u) for each node $v \in L$, where $u \notin L$ and $c(v, u)$ is minimum (by assign c as infinity at first to make sure it can be updated). Then, we remove all edges adjacent to v from E, to ensure v is a leaf in T. After removing, e is a minimum-cost edge connecting v to $V - L$. Then, we use BFS to check the case if the given G contains no solution. After checking, run Kruskal's algorithm on the remaining graph, starting from the edges already in T. In the end, return T as result.

Time complexity:
By lectures, tutorials and textbooks, assume all lines except recursions or loops only require constant time.
Assume $n = |V|$, $l = |L|$ and $m = |E|$($l < logm$ after line 4)
By line 7 and 9, the loop costs $O(lm)$
By line 15, the BFS costs $O(n + m)$
By line 17, Kruskal costs $O(mlogm)$
Therefore, the worst-case running time is $O(mlogm)$.

(d). Solution:
Let $G_0 = G - L$, which means G_0 is G but removed every node and every edge that contains the node in L.
Define any optimal solution T to the MST with Fixed Leaves problem on input G, L.
Define $T' = T - \{(v, u) : v \in L\}$ (T' is T with its leaves removed).

Claim 1: T' is a MST in G_0.
Prove by contradiction:
Assume T^* is a spanning tree of G_0 and $c(T^*) < c(T')$. Then $T^* \cup \{(v, u) \in T : v \in L\}$ forms a spanning tree T_c of the original graph G. The total cost of T_c is smaller than that of T and where each node of L is a leaf.
Contradiction: T is an optimal solution for the original input.

Claim 2: The spanning tree generated by our algorithm is an optimal solution to the MST with Fixed Leaves problem on input G, L.
Proof:
For every optimal solution, each node of L must be connected to $V - L$

by only one edge. Any none minimum-cost edge will increase the cost of the result, which leads to less optimal. Also, connecting one node of L to another node of L will make them impossible to be connected to the rest of the graph, which means these nodes will still be leaves (unless G has only one edge). Once these edges have been selected, and other edges to the vertices of L removed (by our algo, in order to guarantee every node of L is a leaf), use G_0 to represent the rest of graph and Kruskal's algorithm is guaranteed to find a MST of G_0. Therefore, there is no spanning tree of G where each node of L is a leaf and with a smaller cost.

Question 3.
(i) Algorithm
Description: At first, we use the Edmonds-Karp Algorithm (A way to perform Ford Fulkerson algorithm from section 26.7 in CLRS) to find the max flow of the graph G by using the residual graph G_f. The graph G_f concurrently modified when executing Edmonds-Karp Algorithm. The edges that the capacity has reduced to 0 are automatically removed by this algorithm. Then, use do the DFS on the graph G_f from s and put all discovered node to the set S, the remaining is set T(i.e. $T = V - S$), which is the same as the process of min-cut. Finally, find an edge in the original graph G, which is from S to T, and then return this edge.

Pseudo-code:

```
1       Find_critical_edge(G):
2           Let G_f = residual graph of G
3           Let V = set of nodes in graph G //Also set of nodes in
                graph G_f
4           Find_Maximum_Flow_EK_algo(G, G_f) // using Edmonds–Karp
                algorithm, the edges with capacity has reduced to 0
                are removed from G_f
5           Do DFS on G_f begin with s and put discovered node in
                the set S
6           Let T = V - S
7           Satisfied_edges = set of edges (u, v) in G that u is in
                S and v is not in S // v is in T
8           edge = pick an edge (u, v) from satisfied_edge
9           return edge
```

(ii) Show correctness
The edge in the graph G is critical if and only if the current flow of this edge is the same as the capacity of this edge(i.e. $f(u, v) = c(u, v)$), when it is on its maximum flow. The reason is that if this edge reduced its capacity, it cannot afford the current flow on this edge. Then the flow has to be reduced, which also affects the maximum flows come out from the source. The algorithm above find the maximum flow by using the Edmond-Karp Algorithm to find the maximum flow. Since it removed the edges with capacity has reduced to 0 from the residual graph G_f. Then the G_f has modified and no path exists from s to t by the Max-Flow Min-Cut Theorem. By the same theorem, it also implies $|f| = c(S, T)$, where S is the set

of node that there exists a path from s to this node and $T = V \backslash S$(S can be get from the DFS from s, we can get S and T by line 6 and line 7, which is the same as min-cut). Since the max flow is the same as the min cut, then $\Sigma_{u \in S} \Sigma_{v \in T} = f(S, T) = c(S, T)$. Then for every edge (u, v) satisfies $u \in S$ and $v \in T$, $f(u, v) = c(u, v)$. Then each edge from node in S to a node in T is a critical edge. So, the algorithm above picks the edge which is from node in S to the node in T, and return this edge which has already shown that this edge return is critical.
(iii) Show complexity
The complexity of the algorithm above is $\mathcal{O}(|V||E|^2)$ where $|V|$ is the number of nodes and $|E|$ is the number of edges, the Edmonds-Karp Algorithm to find the max flow in line 3 takes $\mathcal{O}(|V||E|^2)$ time. The DFS in line 5 takes $\mathcal{O}(|V| + |E|)$ time. Also, getting satisfied edges from line 7 takes $\mathcal{O}(|E|)$ time. Any other steps takes not more complex than $\mathcal{O}(|V|)$. So, the complexity of the whole algorithm takes $\mathcal{O}(|V||E|^2)$ time.

Question 4

(a). Solution:
Note: By CLRS and lecture, $|f|$ = total flow out of the source minus the flow into the source (i.e. $|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$). So, $|f| = 1$ means there is only one path from node s to node t. According to Q4, there are no capacity constraints for $pseudo - flow\ f$, which means all edges have chance to be one of the edges in final path from s to t.

We have directed graph $G = (V, E)$ with edge lengths $l_e > 0$ for each $e \in E$. Want to compute the shortest path form node s to node t in G.

Define: $pseudo - flow$ is a function f: $V \times V \to R$ that satisfies the following constraints for all nodes u and v in V (s is source node, t is sink node) and for all $e \in E$:
1). $|f| = 1 = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$
2). No capacity constraints. For all nodes $u, v \in V, 0 \leq f(u,v) < \infty$
(i.e. For all $e \in E, 0 \leq f(e) < \infty$)
3). For all nodes $u \in V - \{s,t\}, \sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$
4). When $e \notin E, f(e) = 0$
5). Let P be current path. If $e \in P$, define $f(e) = 1$; if $e \notin P$, define $f(e) = 0$.

Let set X = visited edges, which is also edges in current path P; set Y = unvisited edges. $X, Y \in E, X + Y = E$. If $e \in X, f(e) = 1$. If $e \in Y, f(e) = 0$.

$\sum_{e \in E} l_e f(e) = \sum_{e \in X} l_e f(e) + \sum_{e \in Y} l_e f(e) = \sum_{e \in X} l_e$

So, if we minimize $\sum_{e \in X} l_e$, then all lengths of edges in current path P are minimized, which means P is a shortest path.

(b). Solution:

Minimize: $\sum_{e \in E} l_e f(e)$
Constraints:
1). $|f| = 1 = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$
2). No capacity constraints. For all nodes $u, v \in V, 0 \leq f(u,v) < \infty$
(i.e. For all $e \in E, 0 \leq f(e) < \infty$)

3). For all nodes $u \in V - \{s, t\}, \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$
4). When $e \notin E, f(e) = 0$
5). Let P be current path. If $e \in P$, define $f(e) = 1$; if $e \notin P$, define $f(e) = 0$.

Justification:
1). According to Q4, we made $|f| = 1$. By CLRS 26.1 and lecture, $|f| =$ total flow out of the source minus the flow into the source (i.e. $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$). So, $|f| = 1$ means there is only one path from node s to node t.
2). According to Q4, there are no capacity constraints for $pseudo - flow$ f, which means all edges have chance to be one of the edges in final path from s to t. So we make the capacity does not matter at all (by making capacity incredibly large). Also, by the definition and properties of flow f (CLRS 26.1), there is no negative flow $f(e)$ in flow network G for any edge $e \in E$. (i.e. For all $e \in E, 0 \leq f(e)$)
3). This is the Flow Conservation of the properties of any flow f. (by CLRS 26.1)
4). We need to handle the case when edge e is not in flow network G.
5). This is the most important part for our $pseudo - flow$. Because all flow can only be 1 and there is no upper capacity by Q4 definition. In order to get the shortest path in Q4, the only thing we need to worry about is edge lengths l_e for each $e \in E$. By part(a), we know $\sum_{e \in E} l_e f(e) = \sum_{e \in X} l_e f(e) + \sum_{e \in Y} l_e f(e)$ where set $X =$ visited edges, which is also edges in current path P; set $Y =$ unvisited edges, $X, Y \in E, X + Y = E$. So it obviously that we only need to care about edges in current path P and ignore edges not in P, because our goal is to find a existing shortest path, not predict a shortest path. So, we decide to let $\sum_{e \in Y} l_e f(e) = 0$ by define $f(e) = 0$ if $e \notin P$. Also, define $f(e) = 1$ if $e \in P$ because all flow can only be 1 and l_e is the key element to get the shortest path. In the end, $\sum_{e \in E} l_e f(e)$ can be simplified to $\sum_{e \in X} l_e$ by this constraint. If we minimize $\sum_{e \in X} l_e$, then all lengths of edges in current path P are minimized, which means P is a shortest path.

Question 5
(a)
With the given information, we have a directed graph $G = (V, E)$ with an integer weight $w(e)$ for each edge $e \in E$. For any set $S \subset V$, we let δ^+ denote the set of directed edges going from S to V \ S, and δ^- denote the set of directed edges going in the reverse direction. We also define a binary variable x_e for each directed edge $e \in E$, taking the value 1 if and only if the directed edge e is in the tour. Finally, for any set of edges $F \subset E$, let $x(F)$ denote $\sum_{e \in E} w_e x_e$ which counts the number of used edges in set F.

This TSP problem is now equivalent to the 0-1 LP problem formulated below:

$$
\begin{aligned}
& min \sum_{e \in E} w_e x_e \\
(1) \quad & \text{s.t } x(\delta^+(\{i\})) = 1 (\forall i \in V) \\
(2) \quad & x(\delta^-(\{i\})) = 1 (\forall i \in V) \\
(3) \quad & x(\delta^+(S)) \leq |V| - 1 (\forall S \in V : 2 \leq |S|) \\
(4) \quad & x(\delta^+(S)) \geq 1 (\forall S \in V : 2 \leq |S|) \\
(5) \quad & x \in \{0, 1\}^{|E|}
\end{aligned}
$$

The equation (1) and (2) are two degree equations to make sure that the salesman must arrive at and depart from each city exactly once. Equation (3) ensure the number of the path should less than number of the vertices minus one so that the path excludes subtours. Equation (4) ensure the number of the path should greater than 1 so that the path is connected.

If we assign every city to a number from 1 to n, and every edge e subscripted by two city number to indicate the cities and direction. The LP problem described above could give us a set of directed edges and a set of associated binary variable x_e presents if the edge is in the path or not. Edges with binary variable 1 given the solution of the question.

(b)
(i)

To convert a minimization linear program L into an equivalent maximization linear Program $L^{'}$. We can simply negate the coefficients in the objective function. Since L and $L^{'}$ have identical sets of feasible solutions and, for any feasible solution, the objective value in L is the negative of

the objective value in $L^{'}$, these two linear programs are equivalent. We can use the feasible solution of $L^{'}$ and the negative of the objective value for solution to original problem.

(ii)

We can convert the greater-than-or-equal-to constraints to less-than-or-equal-to constraints by multiplying these constraints through bu -1.

$$a_1x_1 + a_2x_2 + ... + a_nx_n \geq b$$
$$-a_1x_1 - a_2x_2 - ... - a_nx_n \leq -b$$

(iii)

We can replace an equality constraint by the pair of inequality constraints.We have a constraint of the form $a_1x_1 + a_2x_2 + ...a_nx_n = b$. Convert it to two inequality:(1)$a_1x_1 + a_2x_2 + ...a_nx_n \geq b$
(2)$a_1x_1 + a_2x_2 + ...a_nx_n \leq b$
We can convert inequality (1) to a less-than-or-equal-to constraints using the same method in (ii).

(iv)

If variable x_1 is negative. We can replace it by $x_1 = x^{'}{}_1 - x^{''}{}_1$ where $x^{'}{}_1 \geq 0$ and $x^{''}{}_1 \geq 0$. If our objective function has a term c_1x_1, we replace it by $c_1x_1 = c_1x^{'}{}_1 - c_1x^{''}{}_1$. If constraint i has a term $a_{i1}x_1$, we replace it by $a_{i1}x_1 = a_{i1}x^{'}{}_1 - a_{i1}x^{''}{}_1$. Any feasible solution $\hat{x_1}$ to the new linear program corresponds to a feasible solution $\bar{x_1}$ to the original linear program with $\bar{x_1} = \hat{x}_1^{'} - \hat{x}_1^{''}$.

(v)

```
1  Convert_to_MELPSE_form(LP):
2      if objective function is minimization problem:
3          for all C_i in objective function:
4              C_i = −C_i    #Convert to the form fits maximization
5      for a_constraint in LP:
6          if a_constraint is an equality:
7              convert it into two inequalities only change equal
                   sign to ≤ and ≥, variables and coefficients
                   remains the same
8      for a_constraint in LP:
9          if a_constraint with sign ≥:
10             multiply both side of the inequality by −1
11     for a_variable in LP:
12         if a_variable is negative:
13             initialize two non−negative variable p1 and p2 such
                   that a_variable = p1 − p2
14             replace a_variable by p1 and p2 in LP
```

If we have n variables and m constraints. The worst case is that every variable is negative and every constraint has an equal sign. Every variable needs to convert to two non-negative variables and every constrains can be split into two inequalities. Then we will at most have 2n variables and 2m constraints. Convert minimization to maximization will take time O(n). Convert equality to two inequalities takes time O(2m). Convert greater-or-equal-to sign takes time O(m). Replace negative variable by two non-negative variables takes time O(2n*m). Overall, time complexity is O(n+2m+m+2n*m) = O(n*m).