Xi Liu, xl3504, Homework 10

Problem 1

(a)

find_span_tree() is a function to find the minimum spanning tree, it runs depth first search and populates the *tree* array with vertices visited, runtime of find_span_tree() is $O(|V| + |E|)$ since depth first search function dfs() requires $O(|V| + |E|)$, and find_span_tree() calls dfs(). dfs() requires $O(|V| + |E|)$ since the recursion is on $\Theta(V)$ vertices, and all vertices in the adjacency list is traversed once and the time of sizes of adjacency lists is $\sum_{v \in V} |Adj[v]| = \Theta(|E|)$, which shows dfs() and find_span_tree() both requires $\Theta(|V|) + \Theta(|E|) = \Theta(|V| + |E|)$

correctness justification: since tree is a connected acyclic graph, and spanning tree of graph G is a subgraph and is a tree that that contains all vertices of G and with minimum number of edges. if all edge weights of an undirected connected graph G are 1, all of the spanning trees are minimum, since every spanning tree has $n - 1$ edges, and each edge cost 1. so for $G = (V, E)$, all acyclic subset $T \subseteq E$ has total weight

$$w(T) = \sum_{(u,v) \in T} w(u, v) = \sum_{i=1}^{n-1} 1 = (n-1)(1) = n - 1$$

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <list>
using namespace std;

struct vtex
{
    int vtex_id;
};

struct graph
{
    int n_v;
    bool * visited;
    list<void *> * adj;
```

```
    graph(int n_v);
    void add_undirect_edge(void * v1, void * v2);
    void dfs(void * v, void ** tree, int * tree_sz);
    void ** find_span_tree(void * v, int * tree_sz);
};

graph::graph(int n_v)
{
    this->n_v = n_v;
    adj = new list<void *>[n_v];
    visited = (bool *)malloc(n_v * sizeof(bool));
}

void graph::add_undirect_edge(void * v1, void * v2)
{
    adj[((vtex *)v1)->vtex_id].push_back(v2);
    adj[((vtex *)v2)->vtex_id].push_back(v1);
}

void graph::dfs(void * v, void ** tree, int * tree_sz)
{
    int v_id = ((vtex *)v)->vtex_id;
    visited[v_id] = true;
    tree[(*tree_sz)++] = v;
    for(list<void *>::iterator i = adj[v_id].begin();
    i != adj[v_id].end(); ++i)
        if(!visited[((vtex *)*i)->vtex_id])
            dfs(*i, tree, tree_sz);
}

void ** graph::find_span_tree(void * v, int * tree_sz)
{
    size_t sz = n_v * sizeof(void *);
    void ** tree = (void **)malloc(sz);
    memset(tree, 0, sz);
    *tree_sz = 0;
    dfs(v, tree, tree_sz);
    return tree;
```

}

(b)
only 1 edge $e_0 = \{u_0, v_0\}$ is changed and all of the other edges still each have a weight of 1
below procedure uses find_span_tree() and dfs() defined in Problem 1 (a)
1. run find_span_tree() once on the graph G, store the returned spanning tree in $tre1$, and store the computed total weight in $wei1$, run time for step 1 is $O(|V| + |E|)$
2. create a temporary graph G' that is a copy of G with the edge $e_0 = \{u_0, v_0\}$ removed from the edge set. $G' = G \setminus \{u_0, v_0\}$, run time for step 2 is $O(|V| + |E|)$
3. run find_span_tree() or dfs() on the graph G', store the returned graph in $tre2$, and store the computed total weight in $wei2$, run time for step 3 is $O(|V| + |E|)$
4. if the result graph $tre2$ produced by the depth first search is not connected, this means the edge $e_0 = \{u_0, v_0\}$ must be included in the minimum spanning tree of G, so add the edge $e_0 = \{u_0, v_0\}$ to the result graph $tre2$ produced by the depth first search to produce the minimum spanning tree of G, then return $tre2$, run time for step 4 is $O(|V| + |E|)$
5. else if the graph $tre2$ produced by the depth first search is already connected, this means we need to compare the weights $wei1$ and $wei2$ to see whether we include the edge $e_0 = \{u_0, v_0\}$ in the output. if $wei2 \leq wei1$, return $tre2$; else if $wei1 < wei2$, return $tre1$, run time for step 1 is $O(1)$

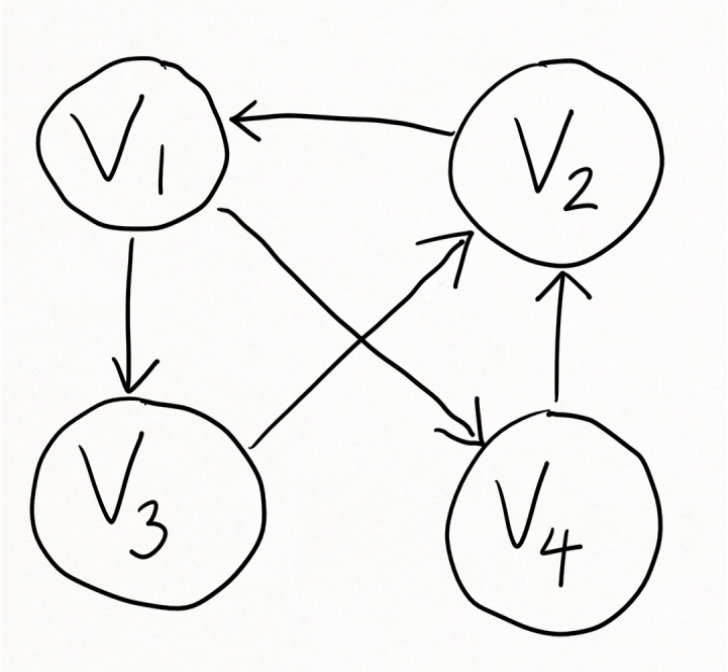
total run time is asymptotically the same with run time in part (a), since total run time is

$$\sum_{i=1}^{5} (\text{step } i) = 4O(|V| + |E|) + O(1) = O(|V| + |E|)$$

Problem 2
(a)
if the weight of 1 edge $e \in T$ is decreased:
let T be the original MST of G, T' be the new MST of G after decreasing the weight. the weight decrease be dec, and $e = \{v1, v2\}$, and weight of T be $weight(T)$, weight of T' be $weight(T')$
let $T1$ and $T2$ be subgraphs produced if e is removed from T
$T1'$ and $T2'$ be subgraphs produced if e is removed from T'
T' contains e, since T is a MST, T has e, and edge e's weight becomes lower means that any MST that includes e previously will have a lower weight when edge e's weight is decreased
for contradiction, assume after decreasing the weight, the new MST T' for the graph has a weight less than $weight(T) - dec$, or equivalently assume $weight(T') < weight(T) - dec$, since edge e is included in T', then either the vertices of $T1'$ are connected differently than the connections of $T1$ so that $weight(T1') < weight(T1)$, or the vertices of $T2'$ are connected differently than the connections of $T2$ so that $weight(T2') < weight(T2)$. however, $T1$ and $T2$ are MSTs for the vertices that they each connect, so $weight(T1') \not< weight(T1)$ and $weight(T2') \not< weight(T2)$, so $weight(T')$ is not less than $weight(T) - dec$

(b)
if the weight of 1 edge $e \notin T$ is increased, Kruskal's algorithm would produce the same MST, since Kruskal's algorithm involves a step that sort the edges of the graph's edge set into nondecreasing order by weight w, the weights involved in the original MST will have the same relative ordering in the sorted list used to produce the new MST, and Kruskal's algorithm makes the same choices of edges to be combined into the spanning tree by calling the union() function; since edge e originally does not belong to T, increasing the weight of e will not make e to be included in T

Problem 3



for the graph G shown above with vertices $v1, v2, v3, v4$
and edges $(v1, v4), (v1, v3), (v3, v2), (v4, v2), (v2, v1)$
$v3$ comes before $v4$ in the adjacency list of $v1$

if run a topological sort starting at $v2$, then the order of finish times from latest to earliest is $v2, v1, v4, v3$; "bad" edges here are $(v3, v2), (v4, v2)$

if instead the order of finish times from latest to earliest is $v1, v3, v4, v2$, then the "bad" edge is only $(v2, v1)$

so topological sort does not always produce a vertex ordering that minimizes the number of "bad" edges

Problem 4
(a)
1. call strongly connected components (SCC) function on G to obtain strongly connected components of G, contract all edges that are within the same strongly connected component to produce a result component graph G^{SCC}
2. call topological sort on the component graph G^{SCC} obtained in step 1 to obtain a sorted sequence of vertices sorted in decreasing order of finishing time
3. traverse through the sequence starting from the front to check if each pair of vertices v_i and v_{i+1} /* $\forall i \in [1, sequence.length] \cap \mathbb{N}$, where i is the index of the vertex in the sequence obtained from topological sort */ has an edge from v_i to v_{i+1}. if every pair of consecutive vertices satisfies this condition, then return true; else, return false
running time of algorithm would be running time of depth first search and the traversal of output of topological sort, so $T(n) = O(|V|+|E|)+O(|V|) = O(|V|+|E|)$

(b)
correctness: the algorithm checks whether there exists a directed edge between every consecutive vertices v_i to v_{i+1}. this is a necessary and sufficient condition for the graph to be semi-connected: suppose there are num vertices, and let $v_1, v_2, ..., v_{num}$ be the sequence generated by topological sort, then $\forall i \in [1, sequence.length] \cap \mathbb{N}$, the only path from v_i to v_{i+1} is a directed edge from v_i to v_{i+1}, which means $v_i \rightarrow v_{i+1}$ but not $v_i \leftarrow v_{i+1}$. if there is a path between every consecutive vertices in the sequence generated by topological sort, then any arbitrary pair of vertices in are semi-connected by following the path that go through all vertices

run-time: call strongly connected components (SCC) function on G needs 2 depth first search (DFS) time ($2O(|V| + |E|)$). call topological sort on G^{SCC} needs $O(|V| + |E|)$ since G^{SCC} contains no more than $|E|$ edges and $|V|$ vertices. in the traversal of the sequence, no more than $|V|$ vertices and no more than $|V| - 1$ edges are involved. so total run-time is

$$T(n) = 2O(|V| + |E|) + O(|V| + |E|) + |V| + |V| - 1 = O(|V| + |E|)$$