Minimum Spanning Trees and Kruskal’s Algorithm

Recall that a **tree** is a graph with no cycles.
A **spanning tree** is a subset of edges in a graph that creates a unique path between all vertices in a graph. The spanning tree is a **minimum spanning tree** when the edges are weighted and the sum of the weights of the spanning tree edges are less than the sum of the weights on any other spanning tree.
In **Kruskal’s Algorithm**, we find the minimum spanning tree in the following way:

1. Sort the edges in increasing order
2. Take the lowest-weight edge. Put it in the tree if doing so would not create a cycle. Throw it away otherwise
3. Repeat 2 until all the vertices are connected

Let’s find the minimum spanning tree for this graph:

![Graph Image]

Application of DFS

We’re going to write a version of DFS that returns the length of the longest non-doubling path from one node to any other node in the graph.
Our representation of graphs is as an adjacency list (**adjlists** are linked list nodes,) where we have the following at our disposal:

typedef unsigned int vertex;
typedef struct graph_header* graph;
typedef struct adjlist_node adjlist;

struct adjlist_node{
    vertex vert;
    adjlist *next;
};

adjlist *graph_connections(graph G, vertex v); // returns the neighbors of v in G

struct graph_header {
    unsigned int size;
    adjlist *adj[]
};

unsigned int uint_max(unsigned int x, unsigned int y);
The following function will work assuming we have a tree. Why won’t it work for graphs with cycles?

```c
unsigned int dfs_tree_distance(graph G, vertex parent, vertex child) {
    REQUIRES(G != NULL);
    REQUIRES(parent < graph_size(G) && child < graph_size(G));

    unsigned int distance = 0;

    for (adjlist *L = graph_connections(G, child); L != NULL; L = L->next)
        if (L->vert != parent) {
            unsigned int new_distance = dfs_tree_distance(G, child, L->vert) + 1;
            distance = uint_max(distance, new_distance); // did we find a longer path?
        }
    return distance;
}
```

Consider the following graph, \( G \), with size 9 and adjacency list as given. What do you get when you start at 0, \( \text{dfs_tree_distance}(G, 0, 0) \)? If you start at 5? If you start at 6?

![Graph Image]

What about this graph? It has (is) a cycle. What happens when we start at 0?

![Graph Image]

The reason it won’t work for cyclic graphs is that we don’t know which vertices have already been visited for each recursive call in the graph. We only keep track of whether or not the parent is a neighbor. Try writing a new version of the function that handles cycles.

**Graph Diameter**

The **diameter** of a graph is the length of the longest shortest path between any two vertices in the graph. This can be a confusing description, so think of it in the following way:

1. For every pair of vertices in the graph, \( u \), and \( v \), we find the length of the shortest path from \( u \) to \( v \) and store that length in a 2-D array.

2. We iterate over all the elements in that array and return the maximum element. That is the **diameter**: the length of the longest shortest path.

Suppose we evaluate \( \text{dfs_tree_distance} \) on an arbitrary vertex \( v \) in a graph \( G \) and we get \( distance \) as a result. Fun fact: if the diameter of \( G \) is \( d \), then \( distance \leq d \leq 2 * distance \).