Kruskal’s algorithm

Kruskal’s algorithm is an algorithm to find a *minimum weight spanning tree* (often called a *minimum spanning tree*) on a graph.

A spanning tree of a graph is a subgraph that is a tree and that connects all vertices of the graph. (Recall that a tree is a connected graph with no cycles.)

For any given graph, there may be multiple different spanning trees.

For instance, on the graph below we could delete the edge from the top left vertex to the second-to-top left vertex, and add an edge from the top left vertex to the top second-to-left vertex and we’d have another spanning tree.

A *minimum* spanning tree is simply a spanning tree whose edges have the minimum total weight of any spanning tree on the graph.

Minimum spanning trees are useful in a variety of applications. One classic example is if a phone company is laying cables to connect its customers to the phone network. It wants to connect them all with the minimum possible cost of wire, and doesn’t want to use any additional wiring. If the phone company makes a minimum spanning tree where the vertices are houses and the edges are all possible places the company could lay wires, the minimum spanning tree of the graph will give the places they should lay wires to connect everyone to the phone network with minimum possible cost.

Kruskal’s algorithm is an algorithm that finds the minimum spanning tree for a graph. The algorithm works as follows:

1. Sort all edges by weight, from smallest weight to largest weight.

2. Go through the edges in order. If adding an edge would not create a cycle in the graph, add it. When we have a minimum spanning tree (when the number of edges we’ve added is one less than the number of vertices), we’re done.

Let’s look at a visualization of this algorithm, available at [http://www.cs.usfca.edu/~galles/visualization/Kruskal.html](http://www.cs.usfca.edu/~galles/visualization/Kruskal.html) (Don’t worry about the leftmost part of the visualization for now — it’s a way of visualizing the union-find algorithm, which we’ll discuss in lecture on Tuesday.)
Let's now look at an algorithm which, given a tree and a start vertex, gets the length of the longest path that starts at that vertex and that doesn't double back.

(Recall that we have typedef unsigned int vertex;, that graph_connections returns the set of neighbors of a given vertex, and that uint_max takes two unsigned ints and returns whichever is greater.)

```c
unsigned int dfs_tree_distance(graph G, vertex parent, vertex child) {
  \requires(G != NULL);
  \requires(parent < graph_size(G) && child < graph_size(G));
  // G must be a tree (no cycles!!!)

  unsigned int distance = 0;

  for (adjlist *L = graph_connections(G, child); L != NULL; L = L->next) {
    if (L->vert != parent) {
      unsigned int vert_distance = dfs_tree_distance(G, child, L->vert);
      distance = uint_max(distance, vert_distance + 1);
    }
  }

  return distance;
}
```

This code is a bit hard to understand at first, so let’s step back and examine it. The function takes two arguments besides the graph: parent and child. The only place we use parent is on line 9 to make sure we never double back. When we first start, there’s no way we can double back since we haven’t gone anywhere yet, so we can just pass in the start for both the parent and child initially.

One important note here: since the graph is undirected, the relationship between parent and child is fairly arbitrary.

Now, let’s look at the preconditions. In this case, they’re fairly simple: we just require that G isn’t NULL so we can safely dereference it (this isn’t enough to fully ensure that — G could be some other arbitrary pointer we can’t dereference) and that the vertices we’re given are actually in the graph (we label vertices in the graph in increasing order). We also need to require that G has no cycles, but didn’t write code to check this so we just have it as a comment.

Now we get into the meat of the code. The for loop is iterating over the neighbors of child. For each neighbor that wouldn’t cause us to double back (so, for each neighbor that isn’t parent), we see if the path that goes through that neighbor is longer than a path that we already know about from one of the already-examined neighbors of this vertex. If it is, we update the distance to reflect that new maximum.

This code is still a bit tricky, so let’s work through some examples.

**Recursive DFS — let’s write it!**

It’s possible to do DFS on a graph recursively. This function should look for a path from start to
target in the graph. It should return true if there is a path and false otherwise. Your solution should not use stacks, priority queues, or any data structure that emulates the behavior of a stack.

Take a few minutes and work on completing this function. Hint: the structure of your code will be very similar to dfs_tree_distance. The solution I have in mind is 10 lines, including two blank lines and lines with braces. I've purposefully given you more space than that so you can use large handwriting and/or cross things out.

```c
bool dfs (graph G, bool *visited, vertex start, vertex target) {
    REQUIRE(G != NULL & visited != NULL);
    REQUIRE(start < graph_size(G) & target < graph_size(G));
    //REQUIRE(graph_size(G) == length(visited));
    visited[start] = true; // Mark this vertex as visited

    if (start == target) {
        // We’ve found a path to the target vertex!
        return true;
    }
}
```