15-122: Principles of Imperative Computation

Recitation 25 Josh Zimmerman

Kruskal's algorithm

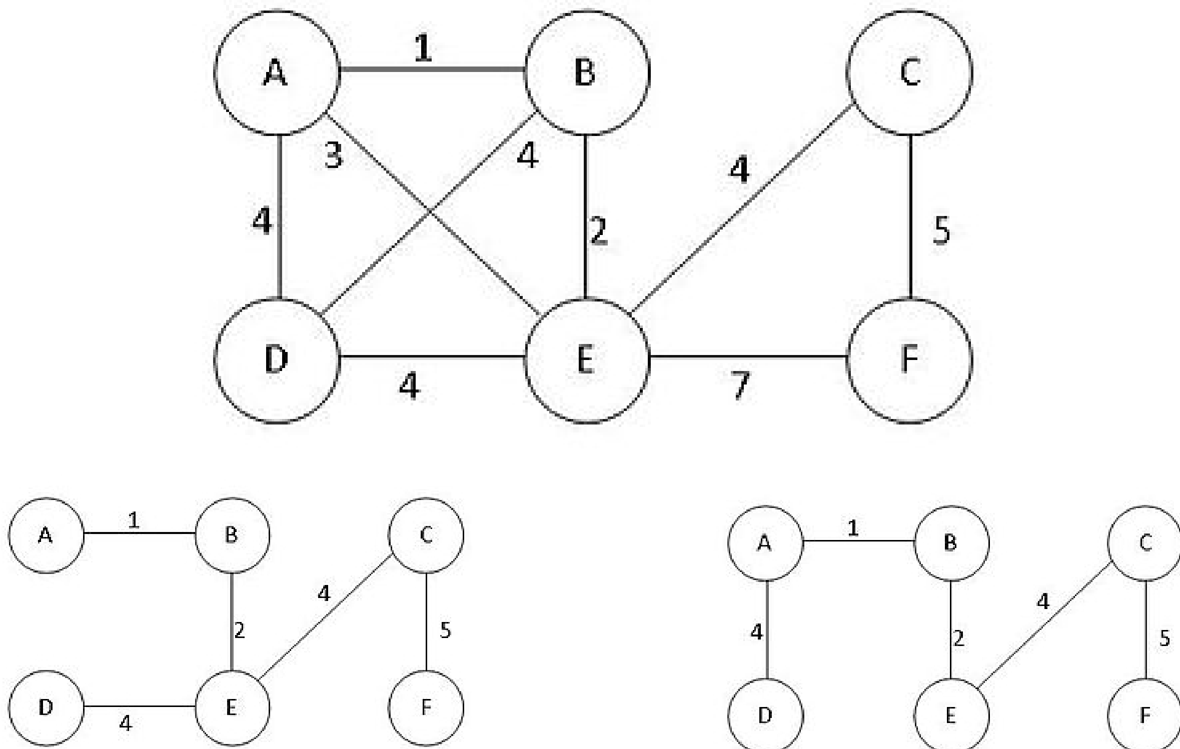
Kruskal's algorithm is an algorithm to find a *minimum weight spanning tree* (often called a *minimum spanning tree*) on a graph.

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

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

For any given graph, there may be multiple different MSTs.

For instance, on the graph below we have two different minimum spanning trees, which are shown below it.



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 in terms of amount of wire, and doesn't want to use any additional wiring. If the phone company makes a graph where the vertices are houses and the edges are all possible places the company could lay wires, a 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`. (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.)

BFS recap

Breadth-first search is very similar to depth-first search, but instead of using a stack to store the nodes we want to look at, we use a queue.

The main difference from DFS is that we explore all neighbors of a vertex before exploring their neighbors, and so we'll always find the shortest path from the source to the target.

Let's spend some time to walk through an example of this with the following code for BFS.

```
1 bool bfsearch(graph G, vertex source, vertex target) {
2  REQUIRES(source < graph_size(G) && target < graph_size(G));
3
4  queue Q = queue_new();
5  unsigned int size = graph_size(G);
6  bool mark[size];
7  for(unsigned int i = 0; i < size; i++)
8   mark[i] = false;
9
10  enq(Q, (void*)(uintptr_t)source);
11  mark[source] = true;
12  while(!queue_empty(Q)) {
13   vertex v = (vertex)(uintptr_t)deq(Q);
14   printf("Visiting %d\n", v);
15   if (v == target) {
16    queue_free(Q, NULL);
17    return true;
18   }
19   for(adjlist *L = graph_connections(G, v); L != NULL; L = L->next) {
20    if(!mark[L->vert]) {
21     enq(Q, (void*)(uintptr_t)L->vert);
22     mark[L->vert] = true;
23    }
24   }
25  }
26
27  queue_free(Q, NULL);
28  return false;
29 }
```

`dfs_tree_distance`

Let's 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.

(Remember 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.)

```
1 unsigned int dfs_tree_distance(graph G, vertex parent, vertex child) {
2    REQUIRES(G != NULL);
3    REQUIRES(parent < graph_size(G) && child < graph_size(G));
4    // G must be a tree (no cycles!!!)
5
6    unsigned int distance = 0;
7
8    for (adjlist *L = graph_connections(G, child); L != NULL; L = L->next) {
9       if (L->vert != parent) {
10         unsigned int vert_distance = dfs_tree_distance(G, child, L->vert);
11         distance = uint_max(distance, vert_distance + 1);
12      }
13   }
14   return distance;
15 }
```

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` 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.