Graphs: Runtime and Mininum Spanning Trees

Kathi Fisler and Milda Zizyte

October 26, 2022

Objectives

By the end of this lecture, you will:

- Be able to reason about runtime for BFS/DFS/Dijkstra's
- Know what a minimum spanning tree is
- Use Jarnik/Prim's algorithm to construct a minimum spanning tree
- Use Kruskal's algorithm to construct a minimum spanning tree
- Have an overview of Sollin's/Boruvka's algorithm for constructing a minimum spanning tree in parallel

1 Runtime of the graph algorithms we've seen so far

1.1 BFS and DFS

Consider the non-recursive (stack/queue) implementation of either BFS or DFS. The outer loop executes $O(|V|)$ times, where $|V|$ is the number of vertices in the graph[1]. This is because, in the worst case, each vertex will be added (and therefore removed) once from the stack or queue. Since each vertex has no more than $|V|$ neighbors, this leads us to believe that code the inner loop will also execute at most $|V|$ times for each iteration of the outer loop, leading to a worst-case bound of $O(|V|^2)$.

While this is certain an *upper* bound of the algorithm, is it the tightest bound we can write down? We said that the inner loop happens $|V|$ times, because each vertex could have an edge to every other vertex. While this is possible, it isn't common. Usually graphs don't have edges between every pairs of vertex (such as a edge from Providence to every other city, without going through other cities). Indeed, the number of edges is often much smaller than $|V|^2$.

As a result, the complexity analyses of graph algorithms often account for vertices and edges separately. Let's use $|E|$ for the number of edges. Since the number of neighbors across all nodes is equal to the number of edges, the code in the inner loop will only execute a total of $|E|$ times over *all* iterations of the code (note the slight shift in reasoning here – we are essentially "unrolling" the outer loop to analyze how many times the code in the inner loop will run. Another way to think about this is that each vertex has an average of $|E|/|V|$ neighbors, which means the code in the inner loop will execute $|E|/|V|$ times for each execution of the outer loop. $|E|/|V| * |V| = |E|$). Accounting for the code in the outer loop that is not in the inner loop (which runs $|V|$ times total), the complexity for DFS and BFS is $O(|V| + |E|)$.

[1]This comes from math notation, where V is typically used to denote the *set* of vertices in the graph, and $|V|$ is the *magnitude* (or *size*) of the set

1.2 Dijkstra's

What is the performance of Dijkstra's algorithm? The initialization phase takes $O(|V|)$ because each vertice's routeDistance is initialized. The outer loop executes $O(|V|)$ times, since each vertex must be removed from the priority queue, and no vertex is returned to the queue once it is removed. `removeMin` will take $O(log|V|)$ operations, since there are at most $|V|$ items in the queue[2]. For the inner loop, just like for BFS and DFS, each vertex has no more than $|V|$ neighbors, and the `adjustPriority` operation takes $O(log|V|)$ time. This leads us to an initial complexity of $O(|V|^2 log|V|)$. Again, however, the number of neighbors across all nodes is equal to the number of edges, so the contribution of the outer loop is $O(|V|log|V|)$ and the contribution of the inner loop, across all vertices, is $O(|E|log|V|)$, which leads us to the tighter bound of $O((|V|+|E|)log|V|)$ [3]

1.2.1 The cost of Adjusting Priorities

Wait – how does `adjustPriority` take only $O(log|V|)$? Don't we have to both find the item for a specific node in the priority queue ($O(|V|)$), change the priority (constant), then move the node into its proper place ($O(log|V|)$)[4]? This sounds like $O(|V| + log|V|)$ which is $O(|V|)$, not $O(log|V|)$.

This is where details of your data structures and your specific implementation matter in computing runtime. Let's say you store vertices in a Priority Queue that was defined in a library, and the library doesn't give you access to the internal Priority Queue organization. Then you have to ask the Priority Queue implementation to search for the vertex whose priority you want to adjust. But instead, what if you built the priority queue manually, and had a way to get from the vertices you were storing to objects or array indices within the internal data structure in constant time (such as adding an extra hashmap with this info, or storing the element objects along with the nodes in your Priority Queue entries). Then you could, in principle, find the items in the heap in constant time.

When you look up explanations of Dijkstra's algorithm online, even from well-regarded sources, you will typically find the $O(log|V|)$ time reported rather than $O(|V|log|V|)$. You should simply be aware of the assumptions about the data structures that this depends on.

2 Minimum Spanning Trees

We've been talking about how to find shortest paths between two nodes in a graph. What if instead we needed to connect all nodes using the minimum total edge weight?

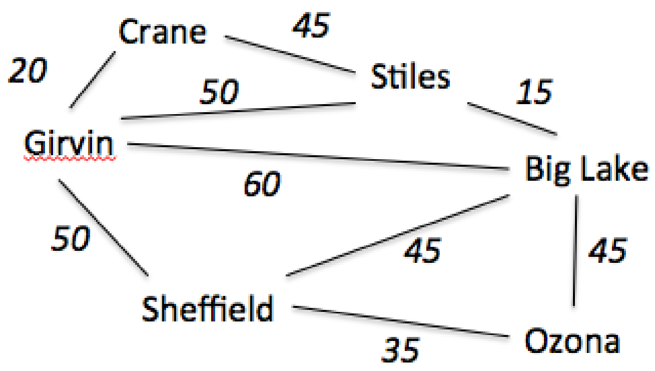
A typical motivating example for this is laying electrical wires among towns: given an undirected connected graph in which nodes are towns and weighted edges show the distance between pairs of towns, we need to select a subset of the edges such that (a) every town is included, (b) the subset of edges forms a connected graph, and (c) the total weight of the selected edges is minimal (relative to other sets of edges we might choose). Such a subset of edges is called a *minimum spanning tree*.

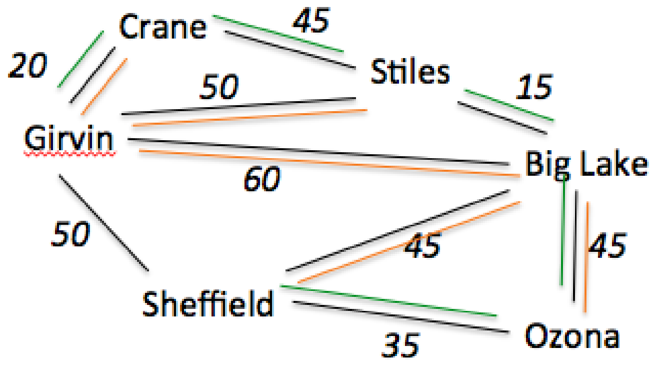
As an example, consider the following graph (using a collection of towns in rural Texas – the edge weights are only approximate). The upper figure shows the original graph. The lower figure shows two spanning trees for the graph: the orange has weight 220 and the green has weight 160. The green one is minimal.

[2]this is the priority queue runtime - for now, take our word for it, but in a few weeks, we'l talk about why this is

[3]There are other versions of Dijkstra's algorithm that are optimized for different situations, but these are beyond the scope of this course! If you take an algorithms class, you will likely see some alternatives that use more sophisticated data structures, such as a *Fibonacci Heap*.

[4]this operation is called *sifting*, and we will learn about it when we learn about heaps





For the rest of today's notes, we refer to section 19.8 of a textbook called Programming and Programming Languages (written by Brown CS Professor Shriram Krishnamurthi)

`https://papl.cs.brown.edu/2019/graphs.html`

We covered through 19.8.4 in this lecture, and will cover 19.8.5 in the next lecture.