CE204 Lab 7: Graph Algorithms

David Richerby

Week 23, 2020–21

The lab exercises are not assessed and you are not required to complete all of them, though I recommend that you attempt them all. Feel free to work with others and talk about your answers.

Solutions will be released on Moodle, on the Friday after the labs.

1 Kruskal's algorithm

Write a method `Graph minimumSpanningTree (Graph g)` that implements Kruskal's algorithm, as described in lecture 7. You can test it by downloading the class `GraphOfEssex` from Moodle.

2 The Floyd–Warshall algorithm

This is a more challenging exercise, relating to a question that came up in lecture 7. Note that material such as this, which is not covered in the lectures, is also **not** on the exam.

In Lecture 7, somebody asked about finding shortest paths between all pairs of vertices in a weighted graph. Using Dijkstra's algorithm with each vertex in turn as the source takes time $O(ne \log n)$ for a graph with n vertices and e edges. The following algorithm, due to Floyd and Warshall[1] solves the problem in time $O(n^3)$, which is faster for graphs with a lot of edges ($e = \Omega(n^2/\log n)$).

The Floyd–Warshall algorithm is an example of the technique of *dynamic programming*, which is a useful technique to be aware of. Recall that the vertices of our graphs are the integers $0, \dots, n-1$. Define $\mathrm{dist}(x, y, k)$ to be the length of the shortest path from x to y that doesn't pass through any vertex with number k or higher (except for x itself, if $x \geq k$, and y if $y \geq k$), or ∞ if no such path exists. Thus, $\mathrm{dist}(x, y, 0)$ is either

- zero if $x = y$,

[1] And Roy – this is another case of people 60 years ago rediscovering things that had already been published because searching the literature was hard, in those days.

- the weight of the edge (x, y) if that edge exists, or
- ∞ if $x \neq y$ and (x, y) is not an edge.

On the other hand, $\mathrm{dist}(x, y, n)$ is just the length of the shortest path from x to y, since we're allowed to go via any vertex we want. This is exactly what we want to compute.

The key to the technique is that we can compute $\mathrm{dist}(x, y, 0)$ easily from the definition of the graph, and we can compute $\mathrm{dist}(x, y, k+1)$ easily from values of $\mathrm{dist}(i, j, k)$. We do this using the observation that the shortest path from x to y via vertices $0, \ldots, k$ either goes through vertex k or it doesn't (well, duh, but there's a point to this tautology).

- If the shortest path via $0, \ldots, k$ doesn't go through vertex k then $\mathrm{dist}(x, y, k+1) = \mathrm{dist}(x, y, k)$;
- if it does go through k, then it must be composed of the shortest path from x to k, followed by the shortest path from k to y, so
$$\mathrm{dist}(x, y, k+1) = \mathrm{dist}(x, k, k) + \mathrm{dist}(k, y, k)\,.$$

This means that

$$\mathrm{dist}(x, y, k+1) = \min\left(\mathrm{dist}(x, y, k), \mathrm{dist}(x, k, k) + \mathrm{dist}(k, y, k)\right).$$

Dynamic programming is this technique of building up a solution by gradually allowing more and more of the space to be used in intermediate calculations.

Write a method `double[][] shortestPaths (Graph G)` that returns the $n \times n$ array `D` such that `D[x][y]` $= \mathrm{dist}(x, y, n)$. Start by computing the array for $k = 0$ and then use a loop to calculate the array for $k = 1, 2, \ldots, n$.