CS41 Lab 6

October 5, 2021

In typical labs this semester, you'll be working on a number of problems in groups of 3-4 students. You will not be handing in solutions; the primary purpose of these labs is to have a low-pressure space to discuss algorithm design. However, it will be common to have some overlap between lab exercises and homework sets.

1. In CS35, you likely saw the *shortest path on weighted graphs* problem. In this problem, each edge e has an edge length ℓ_e, and the length of a path $s \rightsquigarrow t$ is the sum of the edge lengths along the path. Your goal is to find, for a specific start vertex, the length of the shortest $s \rightsquigarrow v$ path for all vertices v.

   Problem: Shortest Paths

   Inputs:

   - a graph $G = (V, E)$
   - for each edge e a positive *edge length* ℓ_e
   - a start vertex $s \in V$

   Output: an array of distances d, where $d[v]$ is the length of the shortest $s \rightsquigarrow v$ path.

   Below is pseudocode for Dijkstra's Algorithm, which finds the shortest path in a graph G between a start vertex s and any other vertex.

   Dijkstra(G, s, ℓ)
   1 $S = \{s\}$.
   2 $d[s] = 0$.
   3 **while** $S \neq V$
   4   pick $v \in V \setminus S$ to minimize $\min_{e=\{u,v\}:u \in S} d[u] + \ell_e$.
   5   add v to S.
   6     $d[v] = d[u] + \ell_e$
   7 Return $d[\ldots]$.

   The greedy rule on line 4 is selecting a vertex $v \notin S$ which has an edge $e = \{u, v\}$ from a vertex $u \in S$ such that it minimizes the distance $d[u] + \ell_e$.

   (a) Show that Dijkstra's Algorithm solves the shortest path problem. (Hint: use the "stays ahead" method.)

   (b) What is the asymptotic running time of Dijkstra's algorithm?
   If you were to implement it (in, say, C++) what data structures would you need? Would you need any additional data structures beyond structures you've seen from CS35? If so, try to design an implementation for them.

   Note: the pseudocode given above is *high-level* pseudocode. One reason why this is high-level is because it doesn't specify how to compute the edge $e = (u, v)$ such that $u \in S$ that minimized $d[u] + \ell_e$. You'll need to understand how to compute this edge efficiently.

2. **Minimum Spanning Trees: edge weights.** In class we saw the cut property, which stated that for any nonempty subset $S \subseteq V$ of vertices, the edge $e = \{u, v\}$ of minimal weight such that $u \in S$ and $v \in V \backslash S$ is in every minimal spanning tree of the given graph.

   There may be more than one minimum spanning tree of a graph. The cut property is worded very carefully: the edge e is in *every* minimum spanning tree.

   However, if edge weights are not distinct then there might be two edges which are tied: both have the smallest weight.

   (a) Given a connected undirected graph G with edge weights from the set $\{1, 2, 3, 4, 5\}$, is there a minimum spanning tree that does not contain some edge e of weight 1 (the minimum weight)?
   If yes, give a graph where this is true. If no, argue why it is not true.

   (b) Given a connected undirected graph G with edge weights from the set $\{1, 2, 3, 4, 5\}$, is there a minimum spanning tree that does contain some edge e of weight 5 (the maximum weight)?
   If yes, give a graph where this is true. If no, argue why it is not true.

   (c) The problem with our cut property seems to be that when edge weights are not distinct, it does not say which edge should be in a minimum spanning tree. Rewrite the cut property so that it covers the case where edge weights are not distinct. (If there is not necessarily a *single* edge of minimum weight, then what should the cut property say?)

   (d) Use your new version of the cut property to prove that Prim's algorithm returns a minimal spanning tree, in the case when edge weights are not distinct.

3. **Minimum Spanning Trees: implementation.** Two common greedy MST algorithms are:

   - Prim's algorithm: Maintain a set of connected nodes S. Each iteration, choose the cheapest edge $\{u, v\}$ that has one endpoint in S and one endpoint in $V \setminus S$.
   - Kruskal's algorithm: Start with an empty set of edges T. Each iteration, add the cheapest edge from E that would not create a cycle in T.

   We saw high-level pseudocode for both these algorithms in class. Implementation details **matter a lot** in considering which of these algorithms to use.

   (a) What is the asymptotic running time of Prim's algorithm?
   If you were to implement it (in, say, C++) what data structures would you need? Would you need any additional data structures beyond structures you've seen from CS35? If so, try to design an implementation for them.

   (b) What is the asymptotic running time of Kruskal's algorithm?
   If you were to implement it (in, say, C++) what data structures would you need? Be specific. Would you need any additional data structures beyond structures you've seen from CS35? If so, try to design an implementation for them.