CS170–Fall 2012 — Solutions to Homework 6

Ben Augarten, section 106, `cs170-bo`

October 12, 2012

1. 4.8 No this is not a valid method. Lets say there are two paths from s to t. Even in the case of no negative edges, lets say these two paths have length n and m, $n < m$. The first path of length n goes through 10 edges, the second through 1. If we add a constant factor k to each edge, then the first path now has cost $n + 10k$, while the second has $m + k$. Well, now $n + 10k > m + k$ so Professor F. Lake would get the incorrect answer using his algorithm. Of course, we can construct many counter examples that follow this pattern

   4.9 Yes Dijkstra's will work (assuming no edges enter s as Prof Rao said on piazza). The lack of incoming edges to s prevents a negative cycle. Now, any shortest path between s and any u has to go through an outgoing edge of s exactly once (because s has no incoming edges). Because of this, we can add a constant factor to every outgoing edge of s as proposed in 4.8. This constant factor is added exactly once to each shortest path and makes all of the outgoing edges positive, so we know that Dijkstra's works on this graph. Because its added only once to each path, we don't run into problems as posed in 4.8. Finally, subtract the constant factor from the length of the final path found.

2. Go through the shortest path tree and label each node on the graph with their shortest path. (This is trivial). Once we label all of the vertices with their potentially shortest paths, for all edges, e, between (u, v), dist$(u) + l(e)$ >=dist(v) – that is, a node can't be farther than all of its ancestors plus the length of the edge from the ancestor. Therefore, we can run one iteration of bellman-ford and if any vertex is updated, then we know that its not a shortest path tree. If not, then we know it is a shortest path tree by the same argument as bellman-ford being correct (we know its correct when there is no update preformed). This runs in linear time.

3. (a) Cost: 19

   (b) 2 different MSTs

   (c) Added in this sequence: AE, EF, FB, FG, GH, CG, GD
   For AE: Cut: A and B..H
   For EF: Cut: AE and BCDFGH
   For FB: Cut: AEF and BCDGH
   For FG: Cut: AEFB and CDGH
   For GH: Cut: AEFBG and CDH
   For CG: Cut: AEFBGH and CD
   For GD: Cut: AEFBGHC and D

4. (a) Why would you make me do this
   I used code from rosettacode.org: http://rosettacode.org/wiki/Huffman_coding#Python
   empty 111
   e 010
   a 1010
   h 0001
   i 0111
   n 0110
   o 1000
   r 0000
   s 0011
   t 1100
   c 00101
   d 10111
   l 10110
   u 00100
   b 100100
   f 110100
   g 100111
   m 110111
   p 100101
   w 110101
   y 100110
   v 1101100
   k 11011010
   x 110110110
   j 1101101110
   q 11011011110
   z 11011011111

   (b) 4.201

   (c) Entropy is 4.15, only slightly smaller than out solution, to be expected.

   (d) I don't think this is a the limit – we can also assign sequences to especially common words or sequences of characters. I.e. the, an, or even (space)I(space) would be especially common sequences of characters that could get special encodings.

5. (a) Degree sequence 3,1,1,3

   (b) i. If the neighbors of v_1 are not $v_2, ..., v_{d_1+1}$, then v does not share an edge with one of its first d_1 neighbors. But, in order for v_1 to have degree d_1 it must share edges with d_1 other vertices (no multiple edges). Therefore, there must be at least one vertex not in $v_2, ..., v_{d_1+1}$ such that $\{v_1, v_j\} \in E$. Now we have just shown that there must be $\{v_1, v_i\} \notin E$. Now we must show that there has to exist a u that doesn't share an edge with v_j. This is easy – Because v_1 doesn't share edges with all n neighbors, each vertex can have degree only as high as v_1. Therefore, its obvious that v_j cannot have edges to all other vertices, there must exist a u such that v_j doesn't share an edge with u. There must exist a u such that v_i shares an edge with it and v_j does not because v_i has equal or higher degree than v_j. In either case, v_j shares an edge with v_1, so it has $d_j - 1$ other outgoing edges. v_i has at least d_j outgoing edges, one of which cannot go to the same vertices has one in the set of $d_j - 1$ edges (the sets aren't equal). Therefore, there must exist a u that shares an edge with v_i and not with v_j.

      ii. First, connect v_1 with v_i and remove the connection between v_1 and v_j. Now v_j has degree $d_j - 1$ and v_i has degree $v_i + 1$. The good news is we know that a certain u exists that shares an edge with v_i and not with v_j. We can simply change the edge $\{v_i, u\}$ to be $\{v_j, u\}$. Now the degrees are fixed and $(v_1, v_i) \in E$

      iii. We can repeat the above procedure, picking out v_i's one at a time and replacing them with vertices that are connected to v_1 until all $v2, ..., v_{d_1+1}$ are neighbors of v_1. This will produce a graph with the same degree sequence (because each iteration doesn't change the degree sequence) but v_1 neighbors come after it in the ordering

   (c) Sort the degrees in non-decreasing order, much like we had in part b. Now we have $d_1, d_2, ..., d_n$. For the elements $d_2, ..., d_{d_1+1}$, decrement their degree by 1. We can do this because these elements must have or we can rearrange the graph for them to share edges with d_1. Now remove d_1 and recurse on the remaining graph.

   If we get to an element that has degree higher than the remaining number of elements, then the graph doesn't exist. If we get to the last element and its degree is 0 then awesome! the graph exists, if not, the graph doesn't exist.

   Correctness: If we got a nonzero degree value for the last element, then there are either too few incoming edges (positive) or too many incoming edges (negative). Therefore, the degree sequence couldn't possibly exist because the higher degrees either connect too many times with the lower degrees or not enough. If we get a 0 for the last degree, then the number of incoming edges matches the number of outgoing edges for each node, so the handshaking lemma holds for each vertex,

so the graph must exist. If we reach a node that has degree higher than the remaining number of nodes, that means it has too many outgoing edges left and couldn't possibly connect to all of the remaining nodes, so the graph cannot exist.
Run time: each time we have to sort n elements. We run the algorithm n times, each time looking at at most n elements, so it runs in $O(n^3 \log(n))$ time. The sort should be relatively fast though, considering we shouldn't change the ordering much between iterations.