CSCI 570 - HW 4

Solutions

1. Suppose we define a new kind of directed graph in which positive weights are assigned to the vertices but not to the edges. If the length of a path is defined by the total weight of all nodes on the path, describe an algorithm that finds the shortest path between two given points A and B within this graph.

Solution. We have to reduce a vertex weighted graph G into an edge weighted graph G_1. Create a new graph G_1 as follows. Split each vertex v in G into two vertices v_{in} and v_{out}, with an edge weight between them equals to the vertex weight. Edges coming to v in G will come to v_{in} in G_1. Edges leaving v in G will leave v_{out} in G_1. Run Dijkstra's algorithm. In the shortest path tree T_1 for G_1, an edge between v_{in} and v_{out} means that vertex v is in the shortest path tree T for G. Collapse all such edges (v_{in}, v_{out}) in T_1 by merging v_{in} and v_{out}. This will create the shortest path tree T.

2. A new startup FastRoute wants to route information along a path in a communication network, represented as a graph. Each vertex represents a router and each edge a wire between routers. The wires are weighted by the maximum bandwidth they can support. FastRoute comes to you and asks you to develop an algorithm to find the path with maximum bandwidth from any source s to any destination t. As you would expect, the bandwidth of a path is the minimum of the bandwidths of the edges on that path; the minimum edge is the bottleneck. Explain how to modify Dijkstra's algorithm to do this.

Solution. Data structure change: we'll use a max heap instead of a min heap used in Dijkstra's.

Initialization of the min heap. Initially all nodes will have a distance (bandwidth) of zero from s, except the starting point s which will have a bandwidth of ∞ to itself.

Change in relaxation step. Based on the definition of a path's bandwidth, the bandwidth of a path from s to u through u's neighbor v will be d(u) = min (d(v), weigth(v,u)), because the bandwidth of a path is equal to the bandwidth of its lowest bandwidth edge. Therefore, in the relaxation step, we will be replacing:

d(u) = min (d(u), d(v)+ weigth(v,u))

with

d(u) = **max** (d(u), min (d(v) , weigth(v,u))

3. Solve the following recurrences by the master theorem

$$T(n) = 3\,T\left(\frac{n}{2}\right) + n\log n$$

Solution. $T(n) = \Theta(n^{\log 3})$

$$T(n) = 8\,T\left(\frac{n}{6}\right) + \log n$$

Solution. $T(n) = \Theta(n^{\log_6 8})$

$$T(n) = \sqrt{7}\,T\left(\frac{n}{2}\right) + n^{\sqrt{3}}$$

Solution. $T(n) = \Theta(n^{\sqrt{3}})$

$$T(n) = 10\,T\left(\frac{n}{2}\right) + 2^n$$

Solution. $T(n) = \Theta(2^n)$

4. We know that binary search on a sorted array of size n takes O(log n) time. Design a similar divide-and-conquer algorithm for searching in a sorted singly linked list of size n.

   a) Describe the steps of your algorithm in plain English

   b) Write a recurrence equation for the runtime complexity

   c) Solve the equation by the master theorem

Solution.

a) Find the middle element of the list. This can be done in O(n). Compare it with the search element. Recurse on the proper sublist.

b) $T(n) = T(n/2) + O(n)$

c) $T(n) = \Theta(n)$

5. Given a sorted array of n integers that has been rotated an unknown number of times, give an O(log n) algorithm that finds an element in the resulting array. Note, after a single rotation, the array is not sorted anymore, so we cannot use the binary search. An example of rotations, sorted array A = [1, 3, 5, 7, 11], after first rotation A= [3, 5, 7,11, 1], after second rotation A= [5, 7, 11, 1, 3].

Solution. The resulting array (after several rotations) will consist of two sorted parts. Run a binarysearch on each part. That splitting point in the array (rotation index) is a local minimum. How fast can we find it? Use divide-and-conquer. Here is the O(log *n*) algorithm for finding a rotation index:

1. Compare it with the middle k = n/2.
2. If A[k] ≤ A[k +1] and A[k] ≤ A[k -1] return k
3. If A[k] < A[1] recurse on A[1..k]
4. Else recurse on A[k…n]