Data Structures and Algorithms

Spring 2014

Cesar Agustin Garcia Vazquez

February 22, 2014

1 Problem Set 03

1. Explain how to solve the following two problems using heaps. (No credit if you're not using heaps!). First, give an $O(n \log k)$ algorithm to merge k sorted lists with n total elements into one sorted list. Second, say that a list of numbers is k-close to sorted if each number in the list is less than k positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an $O(n \log k)$ algorithm for sorting a list of n numbers that is k-close to sorted.

   Answer: The idea to solve the first part of the problem, is to treat each sorted list as a node in a heap. To build the heap we consider the the first element of each list to do the comparison. Since each list is sorted, the first element in the heap will be the minimum and then we extract the min from the heap and insert it into a new array of size n will be sorted and end of the extraction of the n elements.

   To build the heap we used BUILD-MIN-HEAP, which builds the min-heap in $O(k)$. We are going to extract the minimum element of the heap n times, and the extraction of each minimum takes $O(\log k)$, so we have an $O(n \log k)$ algorithm to merge k sorted lists with n total elements into one sorted list.

   To solve the problem of k-close list, we just consider the same idea. We start with the first element in the array. This element is less than k positions from its actual place in the sorted list, so it can be in either of $[1, 2, 3, \ldots, k-1]$. So we construct a heap with the first $k-1$ elements, using BUILD-MIN-HEAP, and then we extract the minimum element of the heap, which also takes $O(k)$ time. After deleting the minimum element, we insert the element in the k position and repeat the process again.

   We only create the heap once, at the beginning and that takes $O(k)$. Then we are going to extract-min in $O(\log k)$, place the element in its corresponding place in

$O(1)$ using an index, inserting the next element takes $O(\log k)$ and this is going to be done n times. So the complexity of the algorithm is $O(k + n\log k) = O(n\log k)$.

2. Design an efficient algorithm to find the longest path in a directed acyclic graph. (Partial credit will be given for a solution where each edge has weight 1; full credit for solutions that handle general real-valued weights on the edges, including *negative* values.)

   Answer: To solve this problem, we consider the fact that the graph is acyclic, so we don't have negative cycles and the concept of shortest path is valid. Also, due to the acyclic property of the graph, we don't have positive cycles, which would make the concept of longest path invalid, since we can simple loop through the cycle and have a path of infinite distance.
   Currently if we have the path between node A and B, P_1 of cost 3 and P_2 of cost 10, the Dijkstra's algorithm will give as an answer P_1 as the shortest path between A and B. So we have to modify Dijstra's algorithm in a way that P_2 ends being the best option.
   We just need to consider a number M, and then instead of comparing the weight of the edge $(u, v) = a$, we consider $M - a$. For instance, if we consider $M = 10$, an edge $e_1 = (u_1, v_1) = 3$ and $e_2 = (u_2, v_2) = 10$, then when Dijstra's algorithm is comparing the weights, instead of choosing $e_1 = 3$ over $e_2 = 10$, the algorithm picks $e_2 = M - 10 = 0$ over $e_1 = M - 3 = 7$.

   Since we are going to look for the shortest path, we are going to traverse more than $|V|$ vertices because that would imply that we fall into a cycle which is not possible in an acyclic graph. This is going to be helpful to determine the value of M, which is computed by retrieving the $|V|$ greatest edges and add them, so no path is going to be greater than this value.

   To retrieve the $|V|$ greatest edges, we just need to have the edges sorted, which can be done in $O(|E|\log|E|)$, which when $|E| = |V|^2$ then the running time of the algorithm would be dominated by $O(|V|^2\log|V|)$, which depending on the implementation, might be the expected running time, or it would be greater than expected if it is something like $|V|\log|V|$.
   In order to simplify the problem, we are just going to obtain the greatest edge weight e_{MAX}, which is in $\Theta(|E|)$, and then multiply this value by the number of vertices $|V|$. This way we obtain $M = |V| \cdot e_{\text{MAX}} \geq \sum_{i=1}^{|V|} e_{i_{\text{MAX}}}$ in $\Theta(|E|)$, which even though is a bigger threshold, it helps for our purposes.

   Finally, we just need to change the values of each edge e_i to $M - e_i$, which is going to be in $\Theta(|E|)$ and at the end of Dijstra's algorithm we revert back the values and we will have the longest path from node A to every other. For the cases when $e_i = \infty$, we can just consider $e_i = \text{nil}$.

3. Giles has asked Buffy to optimize her procedure for nighttime patrol of Sunnydale. (This takes place sometime in Season 2.) Giles points out that proper slaying technique would allow Buffy to traverse all of the streets of Sunnydale in such a way that she would walk on each side of each street, exactly once, going up the street in one direction and down the street in the other direction. Buffy now has slayer homework: how can it be done? (If you have to assume anything about the layout of the city of Sunnydale, make it clear!)

   Answer: For this problem, we use the Depth-First-Search algorithm. We assume that each intersection of a street with another is a node and the streets are the edges. In this case, when traversing a street going up is considered as one directed edge, and traversing the the street going down is considered as another directed edge.

   We have to assume that Sunnydale is a city that can be traversed by Buffy, i.e., it is finite, other wise she might be able to walk going up all the way and never come. From any starting point, give Buffy a painting device so she can mark the corners of the street that she has already visited, and mark the corner of the street going up as explored. When Buffy reaches an intersection, if she can turn right and if none of the corners have been previously explored then she has to move forward.
   When Buffy reaches the end of the street, then she just crosses the street to reach the other side walk and starts walking in the opposite direction, i.e., going back the previous intersection and repeat the process of turning to the right whenever possible.

4. Consider the shortest paths problem in the special case where all edge costs are non-negative integers. Describe a modification of Dijkstra's algorithm that works in time $O(|E| + |V|m)$, where m is the maximum cost of any edge in the graph.

   Answer: Just as mentioned in problem 2, if we have $|V|$ nodes, then m is the maximum cost of any edge, if it is possible to go from node A to B, then the cost is at most $(|V| - 1)m$. So any distance $\delta(v_i, v_j) \leq (|V| - 1)m$.

   We can use the bucket sort to maintain our vertex with their distance in an array. The initialization takes $O(|E|)$, and insertions take $O(1)$, the costs are different. To insert a new element in an already use bucket can take up to $O(|E|)$, in case all of the edges have the same cost. To solve the problem, we maintain a linked list with a reference to the last element, so the insertion is done in constant time.

   So far we have that initialization takes $O(|E|)$ and during the cycle we are going to do $|V|$ deletes and $O(|E|)$ inserts, which as mentioned above takes constant time, so our algorithm's complexity is $O(|E|)$ for initialization and $O(|E|)$ for inserts.
   Hence we need to design the *deletemin* operation in a way that it takes $O(|V|m)$.

To delete an element from the linked list takes $O(1)$ since we are going to delete either the first element of the last element, for which we already have references. So the only costly operation is moving from one bucket to another the list of a buck becomes empty. This means that if bucket i has deleted all of the elements of its corresponding list, we have to traverse our bucket array until we find another bucket with its list not empty. Traversing the bucket array takes $O(m)$, so for each node, *deletemin* takes $O(m)$ and for $|V|$ nodes, that is going to take $O(|V|m)$.
Finally we have that the complexity of out algorithm is $O(|E|)+O(|V|m)+O(|E|) = O(|E|+|V|m)$

5. The *risk-free currency exchange* problem offers a risk-free way to make money. Suppose we have currencies $c_1, \ldots, c_n$. (For example, c_1 might be dollars, c_2 rubles, c_3 ye, etc.) For every two currencies c_i and c_j there is an exchange rate $r_{i,j}$ such that you can exchange one unit of c_i for $r_{i,j}$ units of c_j. Note that if $r_{i,j} \cdot r_{j,i} > 1$, then you can make money simply by trading units of currency i into units of curency j and back again. This almost never happens, but occasionally (because the updates for exchange rates do not happen quickly enough) for very short periods of time exchange traders can find a sequence of trades that can make risk-free money. That is, if there is a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdots r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$, then trading one unit of c_{i_1} into c_{i_2} and trading that into c_{i_3} and so on will yield a profit.
   Design an efficient algorithm to detect if a risk-free currency exchange exists. (You need not actually find it.)

   Answer: This kind of problem resembles to problems used to show the benefits of using greedy algorithms or dynamic programming. In this case, a greedy algorithm will work but not as straightforward as it might seem. To solve this problem using a graph algorithm, we consider each currency c_i as a vertex and the rate from one currency to another, the weight of the edge. So we need to change $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdots r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$ into a sum of $r_{i,j}$. This can be done if we applied the logarithm in both sides so we get $\log_2 r_{i_1,i_2} + \log_2 r_{i_2,i_3} + \ldots + \log_2 r_{i_{k-1},i_k} + \log_2 r_{i_k,i_1} > 0$.

   Since we can change from one currency to any other, we have a complete directed cyclic graph. Hence we know there are cycles, but we are only interested to know if there are cycles that will lead us to a profit. Each edge has weight $\log_2 r_{i,j}$, if this rate is less than one, then we have a negative weight. So if we change $\log_2 r_{i_1,i_2} + \log_2 r_{i_2,i_3} + \ldots + \log_2 r_{i_{k-1},i_k} + \log_2 r_{i_k,i_1} > 0$ into $-\log_2 r_{i_1,i_2} - \log_2 r_{i_2,i_3} - \ldots - \log_2 r_{i_{k-1},i_k} - \log_2 r_{i_k,i_1} < 0$, we just applied Dijkstra's algorithm considering $e_{i,j} = -\log_2 r_{i,j}$the modification of iterating over n instead of just $n-1$, if we detect a change, then we have found a negative cycle and we return true. We do this for each vertex, so the running time is $O(|C|^3)$, where C is the set of currencies.