Midterm take-home assignment

CSED331 Algorithms
49003157 최우석 ChoiWooseok
f2sound@postech.ac.kr
010-6830-6360

Problem 1 - Shortest Path

Describing The Algorithm

I used Dijkstra algorithm, which is based on BFS algorithm. For this algorithm, each nodes are denoted as ascending integer from 0 to N-1, and @dist[i] has the shortest path from node 0 to node i. Also, @prev[i] has the previous node of node i in the shortest path from 0 to i. For example, if the shortest path from node 0 to node 8 is {0-4-2-3-7-8} and its weight is 25, then @dist[8]=25 and @prev[8]=7, @prev[7]=3, @prev[3]=2, @prev[4]=0. Dijkstra algorithm is the process of finding the value of all @dist and @prev, and the results of Problem 1 is in @prev.

Firstly, the algorithm travel each nodes, and the next node to be searched is in priority queue. Assume that searching node is ‘now’ and its adjacent nodes are ‘adj’. For all ‘adj’ of ‘now’, if @dist[adj] > @dist[now] + weight(now-adj), then the shortest path from ‘now’ to ‘adj’ is found(for this, we also assume that the initial value of @dist[1~N-1] is MAX value). Therefore, @dist[adj] = @dist[now] + weight(now-adj) and @prev[adj] = now. Then, this ‘adj’ is pushed into priority queue.

When some node is pushed into the priority queue, its weight is compared and more less weight is to be first. For example, ‘now’ is 0 and ‘adj’ are 2, 6, 4 with weights are 40, 15, 26 each, then the next searching sequence will be node 6, node 4, and node 2, if all ‘adj’ are pushed into the priority queue.

If the very first ‘now’ in the algorithm is 0, then @dist[N-1] is the shortest path from 0 to N-1. For get nodes in the shortest path, get nodes from @prev[N-1] recursively.

Analyzing The Time Complexity

STL’s priority_queue takes $O(logN)$ in insertion, where the N is size of priority_queue. V is number of nodes and E is number of edges in graph. The number of nodes which can be pushed into priority_queue is at most V^2, so traversing nodes to find the shortest path takes $O(logV^2) = O(2logV) = O(logV)$. Also, checking edges is at most E, so all taking times to find the shortest path from node 0 to node N-1 takes $O(ElogV)$. Lastly, getting @prev[N-1] recursively in order to get nodes in the path takes $O(V)$. Therefore, the total time complexity is $O(E(logV) + V)$.

Analyzing The Correctness

Note that this algorithm works when all weights of edges in graph are non-negative value. Also, all @dist[i] has MAX value as its initial value, except @dist[0]. For this feature, the adjacent node ‘adj’ is always compared to the existing @dist[adj] and the distance of the currently discovered new path whenever it is searched, and the shortest path is maintained by taking the shorter value as its @dist[adj]. Therefore, as long as all weights are positive, this algorithm always finds the shortest path and writes it to @dist. At this time, whenever the @dist is updated, the @prev is also updated, so that nodes existing on the shortest path can be recorded together.

Problem 2 - K-Jump MST

Describing The Algorithm

Using transformed Kruskal algorithm. For this algorithm, all edges E have to be sorted in ascending order in its weights. Also, all nodes V have to be in its set. In the process of the algorithm, select an edge in order. Then compare two sets of nodes in picked edge. If the two sets are not the same, then union two sets and this picked edge is an element of result edges. If the difference between the currently selected edge and the previously selected edge is less than K, or if the set of two nodes at the currently selected edge belongs to the same set, then discard the currently selected edge and select the edge in the next sequence. After traversing all nodes, and if the number of elements in the existing set equals the number of all nodes in the graph, then we have successfully found the K-Jump MST in the graph G. It returns its total weight. If there is no MST, then it returns -1.

Suppose that the smallest weight among the given edges is m. Also suppose the algorithm described above has the starting point m. For a given edges, we need to perform the above algorithm starting with all edges which weights are less than $m + k$, respectively. For example, if the ascending order of the weights of the given edges in E is {1, 2, 4, 7, 9, 15, …} and $k = 5$, then total three process above should work with edges {1, 2, 4, 7, 9, 15, … } , {2, 4, 7, 9, 15, … } , {4, 7, 9, 15, … }. After that, each returned weight is compared. The smallest value that is not -1 is the final correct answer.

Analyzing The Time Complexity

I used heap sort at the input of the edges, so the sorting of E takes $O(ElogE)$. Also, I used STL unordered_set to represent the set, which takes $O(l)$ for the union of the two sets, where l is the number of elements in the smaller of the two sets. Therefore, $O(l) = V/2$. Each nodes in V has pointer to its set. Therefore, it takes a constant time to get the set to which a node belongs, and to compare the set to another set. To investigate every edges in E takes $O(E)$ times. Therefore, total kruskal takes $O(ElogE + E(\frac{V}{2})) = O(E(logE + \frac{V}{2}))$

Let t be the number of edges with weights less than $m + K$ in E, where m is the smallest weight in E. $O(t) = K$, so finding K-Jump MST algorithm takes $O(KE(logE + \frac{V}{2}))$.

Problem 3 - Extreme Stairs

Describing The Algorithm

Assume that the position on stairs now is k. Then the next position can be $k + 1$,$k + 2$,$k + 3$, and this can be formed with a recursive function. Like Fibonacci recursion, recursive function 'proc(k)' returns proc($k - 3$) + proc($k - 2$) + proc($k - 1$). This can be done more efficiently with dynamic programming. If the input is n, there need to be additional space n. I denoted it as @cache. The answer when $n =$ i is stored in @cache[i]. Then the proc($k - 3$) + proc($k - 2$) + proc($k - 1$) above can be denoted as @cache[k-3] + @cache[k-2] + @cache[k-1]. Without recursion, just fill @cache array sequentially to get the correct answer.

Analyzing The Time Complexity

It takes $O(n)$, for traversing @cache once.

Analyzing The Correctness

First, @cache[0]=1, @cache[1]=1, @cache[2]=2. If input n is less than 3, then return each value. Otherwise, traversing @cache update the value of each @cache[i] = @cache[i-3] + @cache[i-2] + @cache[i-1], and return when @cache[i=N]. Therefore there is no possibility the wrong answer comes out if the base case(i=0, i=1, i=2) is not changed.