

Programming Assignment II

Yohandi [SID: 120040025]

1. Special Shortest Path

In this problem, we are asked to find the minimum energies required from the first node to all nodes in a weighted undirected graph. We are given n nodes and m edges that represent the whole graph. However, we are also given a constant K . There are a special case that can be rephrased as following: > If there are two edges, edge[1] connects $u[1]$ and $v[1]$ with weight $w[1]$ and edge[2] connects $u[2]$ and $v[2]$ with weight $w[2]$, then it is possible to achieve a shortcut from $u[1]$ to $v[2]$ with weight $w[2]$ if $v[1]$ equals to $u[2]$ and $w[2]$ equals to $K \times w[1]$.

To solve this, we can first add all m edges to the graph two times with different direction (since the edge is defined bidirectionally). Then, to create a shortcut we can use a method that is similar to Breadth First Search (BFS) algorithm. Each node in BFS contains 3 main states: the beginning node, the current node, and the current weight. The starting condition for the BFS is all the $2m$ edges where the beginning node and the current node is directly connected (at first). Then, for a state, we track all other nodes that are also directly connected to the current node, if the weight that connects them is exactly $K \times weight_{current}$ then we add a new state $node_{beginning}, node_{next}, weight_{next}$ to the queue and add a new shortcut from $node_{beginning}$ to $node_{next}$ with a weight $weight_{next}$ into the main graph for the later distance calculation.

The shortcuts finding can be pruned furthermore by only adding edges that is directly divisible by K to the queue. For each edges that are connected in a path, we can also track the shortest path required from the beginning node to the current node. With that, we don't have to add another condition states to the BFS queue. Last part is to run Dijkstra's algorithm to the main graph and obtain the distance results. The implementation details can be found directly in `A4_P1_120040025.cpp` file. Note that the Dijkstra's class header template is copied from my own GitHub repository.

As for the time complexity analysis, we first split the case when $K = 1$ and when $K \neq 1$. - When $K = 1$, In a worst case scenario we might encounter all equal weights. This turns the graph into a complete graph (which means the total edges are $\frac{n \times (n-1)}{2}$). As a consequence of that, the BFS will have to slow down the entire process by running itself with $\mathcal{O}(\frac{n \times (n-1)}{2} - m) = \mathcal{O}(n^2)$ operations to find all possible shortcuts. After adding the shortcuts, running the complete graph with Dijkstra's algorithm requires a time complexity of $\mathcal{O}((n + m_{new}) \log m_{new})$. In here m_{new} is defined as the total edges and shortcuts. Hence, the total time complexity is $\mathcal{O}((n + \frac{n \times (n-1)}{2}) \log \frac{n \times (n-1)}{2}) = \mathcal{O}(n^2 \log n^2) = \mathcal{O}(2n^2 \log n) = \mathcal{O}(n^2 \log n)$. Since the constraints for n is set low enough when $K = 1$, the solution works pretty well. - When $K \neq 1$, In a worst case scenario we might encounter a condition where the graph contains a very long path and the weights

are increased K times from the previous one. However, since K is an integer, the worst case scenario is encountered when $K = 2$. Since all weights are restricted to 10^4 , it can be proved that a half-complete subgraph may occur with size maximum of $\log_2(10^4) \approx 13$. With that, we have the number of shortcuts as $\sum_{i=0}^{\frac{n}{13}} \binom{13 \times (13-1)}{4} = 3n$. Then, running the graph with Dijkstra's algorithm requires a time complexity of $\mathcal{O}((n + m_{new}) \log m_{new})$. In here m_{new} is defined as the total edges and shortcuts. Hence, the total time complexity is $\mathcal{O}((n + 3n + 2m) \log(3n + 2m)) = \mathcal{O}((n + m) \log(n + m))$.

2. Median Search Tree

In this problem, we are asked to do m operations: insert an element, output median $2k$ values, and delete an element from the median $2k$ values. We are first given $2k$ values.

To solve this, we can simply use two heaps (or priority-queues) that each has size $\frac{n}{2}$, where n denotes the number of existing elements. The first heap contains $a_1, a_2, \dots, a_{\lceil \frac{n}{2} \rceil}$ and the second heap contains $a_{\lceil \frac{n}{2} \rceil + 1}, a_{\lceil \frac{n}{2} \rceil + 2}, \dots, a_n$. However, note that a is already assumed to be sorted (if not yet sorted, simply sort it first). Moreover, the first heap is set to maximum heap while the second heap is set to minimum heap.

- Suppose we want to insert an element with value w to the heaps. Then, we can simply add it into a heap that is smaller than the other. If both heaps have the same size, we prioritize the first heap. However, a bug may occurs if we add the value that is smaller than equal to median of all elements to the second heap, or the value that is larger than median of all elements to the first heap. To solve this, we can simply keep swapping the maximum element in the first heap (denote this as X) and the minimum element in the second heap (denote this as Y) such that $X > Y$. Getting the top element requires only 1 operation (2 operations for both heaps), while deleting and inserting it back to the respective heaps require $\mathcal{O}(\log n)$ operations. Hence, inserting an element requires a total of $\mathcal{O}(2 + 2c \log n) = \mathcal{O}(\log n)$ operations.
- Suppose we want to display the median $2k$ values to the heaps. Then, we can simply pop the top elements and save it to a temporary vector k times to both heaps. Since each pop and insert requires $\mathcal{O}(\log n)$ operations, we notice that displaying the median $2k$ values requires a total of $\mathcal{O}(2k \log n + 2k \log n) = \mathcal{O}(k \log n)$ operations.
- Suppose we want to delete the p -th value among the median $2k$ values. Then, we must first determine which heap does contain the p -th value. This can be done by comparing p with $\lceil \frac{k}{2} \rceil$. Then, we can simply pop the top elements of the involved heap k times and saves all the elements to a temporary vector. However, we only want to insert back the first $k - 1$ values that are popped. This requires a total of $\mathcal{O}((2k - 1) \log n) =$

$\mathcal{O}(k \log n)$ operations.

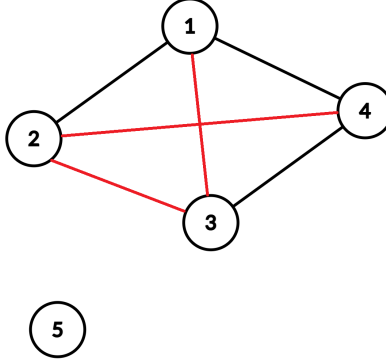
The implementation details can be found directly in `A4_P2_120040025.cpp` file.

To conclude the total time complexity for the m queries, we can simply multiply it with the largest operation query. Hence, we have the total executions as $\mathcal{O}(2k \log 2k + m \max(\log n, k \log n, k \log n)) = \mathcal{O}(k \log k + mk \log n) = \mathcal{O}(k \log k + mk \log(m + 2k)) = \mathcal{O}(mk \log(m + k))$.

3. Football Match

In this problem, we are asked to determine the maximum sum of attractions from $n - 1$ matches of FIFA World Cup. The attraction is defined as $(a_i \times a_j) \bmod M$, where a_i denotes the popularity of the i -th team. However, we can't simply take the highest attraction and multiply it $n - 1$ times as one of the involved team must be eliminated from the cup.

Suppose we only want to consider the possible matches that can be made through particular states. In a cup where there are n teams, we then draw them in a graph, where node i portrays the i -th team. For example, if we have 5 teams and matches between $1 - 2$, $1 - 4$, and $3 - 4$ are already conducted, we have the graph as following:



The red line indicates that it is impossible to conduct a new match between the nodes. Let's consider all cases in the above example. - Match between $2 - 4$: Notice that it is impossible to make a match between $2 - 4$. There are two cases for the result match between $1 - 4$: 1 won (4 getting kicked from the cup; hence, a new match for 4 will not be possible) and 4 won (1 must win in previous $1 - 2$ match so that $1 - 2$ can exist; hence, a new match for 2 will not be possible). - Match between $1 - 3$: Similar case with match between $2 - 4$. - Match between $2 - 3$: Notice that it is also impossible to make a match between $2 - 3$. Assume that it is possible, then both 2 and 3 must win the respective match between $1 - 2$ and $3 - 4$. If that was the case, a match $1 - 4$ won't happen as 1 and 4 are kicked out from the cup.

From the example, we can simply notice that a match between u and v can't be

conducted if node u and node v are connected in the graph. Since we want to find the sum attractions as maximum as possible, we then want to conduct all $n - 1$ matches. Because of that reason, the problem is now changed into finding the Maximum Spanning Tree of the graph where an edge between node u and node v is weighted $a_u \times a_v \bmod M$.

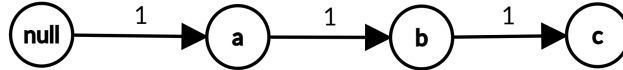
To implement the Maximum Spanning Tree, we can simply use either Prim or Kruskal's algorithm. However, instead of using greedy idea for the minimum weighted edge, we prioritize it for the maximum weighted edge first. In my implementation, a disjoint set union-find data structure is used to optimize the time complexity. The implementation details can be found directly in `A4_P3_120040025.cpp` file.

As for the complexity analysis, we must first notice that there are n nodes and $\frac{n \times (n-1)}{2}$ edges. We know that if we try to merge two connected subgraphs sizing A and B respectively, we may only use $\min(A, B)$ operations. Then, in a worst case scenario, merging all nodes requires at most $n \lceil \log n \rceil$ operations (the graph is similar to merge sort). Denote m as the number of edges. Therefore, the time complexity for the algorithm is $\mathcal{O}(m + n \log n + n) = \mathcal{O}(\frac{n \times (n-1)}{2} + n \log n + n) = \mathcal{O}(n^2)$.

4. Prefix

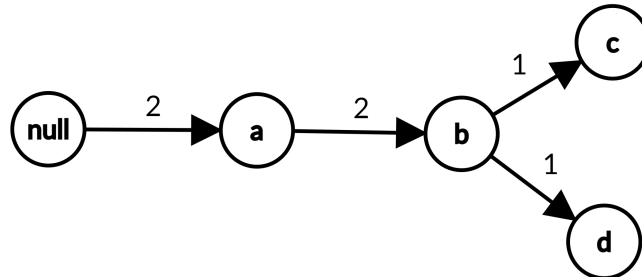
In this problem, we are asked to find the number of strings (from s_1, s_2, \dots, s_n) that begins with a string t_i . There are a total of 10^6 characters from all strings s_i and a total of 10^6 characters from all strings t_i . Since the length of all those strings could be 10^6 , a quick and efficient solution is expected.

This is a classical problem of Trie. All strings s_i can be drawn and merged into a tree-like data structure. For example, if we have $s_1 = abc$, we can have the tree



as following:

In the graph, we notice that the weights of the edge, which are all 1, indicate that the path is crossed 1 time each. If we have another string $s_2 = abd$, the tree is



now changed to


With this tree structure, we can simply go through all nodes according to char-

acters in t_i . If for example $t_1 = ab$, we move from *null* to a and from a to b . We notice that the last edge that we crossed was weighted 2; hence, we have 2 as the answer for $t_1 = ab$.

To implement the Trie, we can simply use pointers of node or use an array with the help of another variable pointing the unallocated empty space. The implementation details can be found directly in `A4_P4_120040025.cpp` file in the `main()` section.

As for the complexity analysis, we first notice that the number of nodes on the tree depends on s_i . Let's denote the number of nodes as N . Since $\sum_{i=1}^n |s_i| \leq 10^6$, we have $N \leq 10^6 + 1$ (1 addition node is used for *null* or the starting point node). To compute the tree itself, it takes exactly N operations; hence, $\mathcal{O}(N)$. For queries, each search takes at most $|t_i|$. Because of this reason, we can conclude that answering q queries require at most $\sum_{i=1}^q |t_i| = 10^6 = N$ operations. The number of operations take at most $2N$; hence, the final time complexity is $\mathcal{O}(2N) = \mathcal{O}(N)$.

Scoreboard Results (December 7, 2022)



Home

Problems

Submissions


Users

Contests

About

Status

Wiki

120040025

Edit profile

Log out

CSC3100 22Fall Assignment4

4 days 08:16:52

CSC3100 22Fall Assignment4

Info

Rankings

Participation

Leave contest

☐ Show organizations

Rank	Username	1 20	2 20	3 25	4 25	Points
???	120040025	20 108:04:59	20 12:40:08	25 12:45:47	25 13:27:55	90

proudly powered by [SchOJ](#), a modded version of [DMOJ](#) |

English (en)