CSE200A : Competitive Programming I (Summer 2023) Reading 4

FooBar: Competitive Programming Club of IIITD

July 2023

1 Minumum Spanning Trees

1.1 Introduction

A spanning tree T of an undirected graph G is a subgraph of G such that T has the same vertex set as G (i.e., V (T) = V (G)), and T is a tree. A minimum spanning tree T of a weighted undirected graph G is a spanning tree such that the sum of edge weights in T is minimum.

1.2 Finding the MST of a Graph

Problem: Given a weighted graph G, find its minimum spanning tree (MST). Solution: There are 2 famous efficient algorithms to determine the MST of a graph: Prim's algorithm, and Kruskal's Algorithm

1.3 Prim's Algorithm

Basic Idea: Start with a tree T with one edge: the edge with minimum weight. At every iteration, select an edge having minimum weight connected to T and not in T , and add it to T (along with the corresponding vertex not in T). Repeat until T contains all vertices in G.

Resources : Prim's Minimum Spanning TRee

1.4 Kruskal's Algorithm

Basic Idea: Start with a subgraph T having all vertices of G and no edges (This is essentially a forest of n trees where n = V (G)). At every iteration, select an edge not in T having minimum weight, such that it connects 2 disconnected components in T , and add that edge to T . Repeat until T is a connected graph.

Resources : Kruskal's Minimum Spanning Tree (Geeksforgeeks)

1.5 Problems

Minimum Spanning Tree

Mr. President

Gift(Hard)

New Year and Rainbow Roads (Hard)

2 Disjoint Set Union

2.1 Introduction

A disjoint set is a data structure that keeps track of set of elements portioned into a number of disjoint (non-overlapping) subsets.

A disjoint set offers the following two functions:

- **find(int a):** find the set that element a belongs to.
- **unite(int a, int b):** Combine the sets of element a and element b.

Using optimisation techniques, both of the above operations can be made highly efficient.

2.2 Implementation

The union-find structure can be conveniently implemented using arrays. In the following implementation, the array **link** indicates for each element the next element in the path or the element itself if it is a representative, and the array **size** indicates for each representative the size of the corresponding set.
Initially, each element belongs to a separate set:

```
    for (int i = 1; i <= n; i++) link[i] = i;
    for (int i = 1; i <= n; i++) size[i] = 1;
```

The function **find** returns the representative for an element x. The representative can be found by following the path that begins at x.

```
    int find(int x) {
        while (x != link[x]) x = link[x];
        return x;
    }
```

The function **same** checks whether elements a and b belong to the same set. This can easily be done by using the function **find**:

```
    bool same(int a, int b) {
        return find(a) == find(b);
    }
```

The function **unite** joins the sets that contain elements a and b (the elements have to be in different sets). The function first finds the representatives of the sets and then connects the smaller set to the larger set.

```
    void unite(int a, int b) {
        a = find(a);
        b = find(b);
        if (size[a] < size[b]) swap(a,b);
        size[a] += size[b];
        link[b] = a;
    }
```

The time complexity of the function **find** is $O(logn)$ assuming that the length of each path is $O(logn)$. In this case, the functions **same** and **unite** also work in $O(logn)$ time. The function **unite** makes sure that the length of each path is $O(logn)$ by connecting the smaller set to the larger set.

2.3 Path Compression

Here is an alternative way to implement the **find** operation:

```
        int find(int x) {
            if (x == link[x]) return x;
            return link[x] = find(link[x]);
        }
```

This function uses **path compression:** each element in the path will directly point to its representative after the operation.

It can be shown that using this function, the union-find operations work in amortized $O(\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function which grows very slowly (it is almost a constant).

However, path compression cannot be used in some applications of the union-find structure, such as in the dynamic connectivity algorithm.

2.4 Additional Resource

Disjoint Set Union (HackerEarth)

2.5 Practice Problems

Roads Not only in Berland
Road Reform
Strange Food Chain
Fight In Ninja World

3 Trie Data Structure

3.1 Introduction

A trie is an efficient information retrieval data structure. It is most commonly used to store and search strings, as each can be done in O(length(s)) time.

3.2 Implementation

A trie is stored as a rooted n-ary tree, i.e., it is stored as a rooted tree where each node points to n 'child' nodes. A trie node is stored as the following structure. Trie is essentially a rooted tree. To search for a word, we traverse down the tree.

3.3 Resources

Trie Implementation in C – Insert, Search and Delete (techiedelight)

Trie data structure (javatpoint)

Basics of trie (Video tutorial, tech dose)

4 Segment Trees

4.1 Introduction

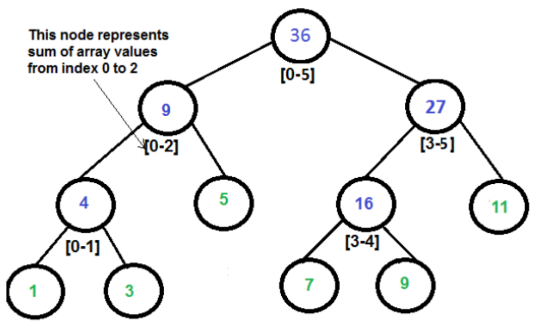
Segment trees are a data structure used for efficiently querying and updating ranges of an array. They are particularly useful for problems that involve range queries or range updates, such as finding the minimum, maximum, or sum of elements in a range of an array.

A segment tree is a binary tree where each node represents a segment of the array. The root node represents the entire array, and each child node represents a half of the parent segment. The leaf nodes represent individual elements of the array.

To build a segment tree, we start with the root node, which represents the entire array. We then recursively divide the array into two halves and create child nodes for each half, until we reach the leaf nodes.

Each node of the segment tree stores information about its corresponding segment of the array, such as the minimum, maximum, or sum of the elements in that segment. This information can be used to answer queries on the array efficiently. To answer a query for a range of the array, we traverse the segment tree and combine the information stored in the nodes that correspond to the segments that overlap with the query range.

Segment trees can be updated efficiently as well. Whenever an element of the array is updated, we can traverse the segment tree from the leaf node corresponding to the updated element, all the way up to the root, updating the stored information in each node as we go.



Segment Tree for input array {1, 3, 5, 7, 9, 11}

4.2 Implementation

Like Heap, the segment tree is also represented as an array. The difference here is, it is not a complete binary tree. It is rather a full binary tree (every node has 0 or 2 children) and all levels are filled except possibly the last level. Unlike Heap, the last level may have gaps between nodes. Below are the values in the segment tree array for the above diagram. For the above tree, the array representation is as follows: st[] = [36, 9, 27, 4, 5, 16, 11, 1, 3, DUMMY, DUMMY, 7, 9, DUMMY, DUMMY]

This implementation has some wasted space, and can take upto 4n elements to represent an array with n elements.

4.3 Problems

Sum of given range

Range Maximum Query with Node Update

XOR of given range

Tree query

Travelling between cities

Counting the number of intervals