Efficient Algorithms

Competetive Programming

Lab Report

Yash Patel

University of Bonn
Computer Science Department

Introduction

The lab "Efficient algorithm for selected problems" was about coding programs for certain problems. It's main aim was to train us exclusively for competitive programming contests which take on platforms like HackerRank, CodeChef, Topcoder and many others. We solved an array of problems from data structures to graph algorithms and basic problems begetting from computational geometry. In this report, we give highlights of some of the important problem solving paradigms we used to solve the problems assigned to us in the lab.

Paradigms

We describe four problem solving paradigms commonly used to solve problems in competitive programming, namely brute force, divide and conquer, the greedy approach, and dynamic programming. We would like to remark that tackling every problem with *brute force* solutions will not always work in the favor of the programmer. To shed some more light on it, we make an attempt to explain what happens if we attempt every problem by brute force. For instance, consider four simple tasks involving an array A containing $n \leq 10,000$ small integers $a_i \leq 1,00,000$ (i.e. $\mathsf{A} = \{10,7,3,5,8,2,9\}; n = 7$).

1. Find the smallest and the largest element of A. *(10 and 2 for the given example)*
2. Find the k^{th} smallest element in A. *(if k = 2, the answer is 3 for the given example)*.
3. Find the largest difference $diff$ such that $a_i, a_j \in \mathsf{A}$ and $diff = \mid a_i - a_j \mid$. *(8 for the given example)*.
4. Find the longest increasing subsequence (LIS) of A. *({3,5,8,9} for the given example)*.

The first task is simple in the sense that we can check every element of A and see whether the current element is largest (smallest) seen so far. This is a brute force strategy running in $O(n)$.

The second one is a bit harder: we can use the solution from first task to find the smallest value and replace it with a value larger than the largest element of the array. We can then proceed with the algorithm of first task as a subroutine and find the smallest value again (second smallest of the original array) and replace with the largest item. Repeating this action k times gives us the k^{th} smallest value. This works totally fine but in the case when $k = \frac{n}{2}$, this way of solving the problem by brute force isn't efficient, that is runs in $O(\frac{n}{2} \times n) = O(n^2)$. Alternatively, we can sort A in $O(n \log n)$ and then return the solution as $\mathsf{A}[k-1]$. This latter technique is called divide and conquer technique.

The third task can also be solved using brute force technique by considering all pairs a_i and a_j in A and check whether the difference is the largest for each pair. This runs in $O(n^2)$, which is slow and inefficient. More intelligently, we can prove that $diff$ can be obtained by finding the difference between the smallest and the largest element of A. Both these elements can be efficiently found from solution to first task in $O(n)$. This is a greedy approach.

Lastly, for the fourth task, trying all $O(2^n)$ possible subsequences to find LIS is not a feasible solution technique for all $n \leq 10K$. Later, we discuss the $O(n^2)$ (DP) and $O(n \log n)$ (greedy) strategies for this task.

Dynamic Programming

It is an algorithmic paradigm that solves a complex problem by dividing it into smaller subproblems and stores the results to subproblems in a table, which might be used to compute new solutions. That being said, dynamic programming is not useful when there are no overlapping subproblems as there seems no point storing solutions if they are of no need in future. For instance, binary search doesn't have overlapping subproblems. The results to the subproblems can be stored in two ways, namely memoization (top-down), and tabulation (bottom-up). In the former, we initialize a lookup table initialized to zero. Whenever a solution to subproblem is needed, we first look into the lookup table: if the value has been precomputed is present, we return the value; otherwise, we calculate the value and update the lookup table accordingly so that the value can be reused again. While the latter builds the lookup table in bottom-up fashion and returns the last entry to the table.

Longest Increasing Subsequence

Task : The LIS problem is to find the length of the longest subsequence of a given sequence $\{\mathsf{A}[0], \ldots, \mathsf{A}[n-1]\}$ where all the elements of the subsequence are sorted in increasing order. Note that the subsequence might not be contiguous or unique.

Let $\mathsf{L}[i]$ be the length of the LIS ending at index i. Also, $\mathsf{L}[0] = 1$ as the first element of A is a subsequence. For $i \geq 1$, we need to find an index j such that it $j < i$ and $\mathsf{A}[j] < \mathsf{A}[i]$ and $\mathsf{L}[j]$ is the largest. Once we have such an index j, we have that $\mathsf{L}[i] = \mathsf{L}[j] + 1$. The recurrence can be written as:

- $\mathsf{L}[0] = 1$ (base case)
- $\mathsf{L}[i] = \max L[j] + 1, \forall j \in [i-1]$ and $\mathsf{A}[j] < \mathsf{A}[i]$

Then the answer is the largest value of $\mathsf{L}[k]$ $\forall k \in [n-1]$.

We can clearly see that there are many overlapping subproblems as in order to compute $\mathsf{L}[i]$, we need to compute $\mathsf{L}[j]$ $\forall j \in [i-1]$. However, we only need to

compute n distinct values of LIS ending at index i, $\forall i \in [n-1]$. As it takes $O(n)$ time to compute each value, the total running time is $O(n^2)$.

Interestingly, there also exists a more efficient way to solve LIS in $O(n \log n)$ time. The trick is to use an ensemble of greedy + divide and conquer algorithm by maintaining an array that is always sorted and therefore amenable to binary search. Let Aux be an auxiliary array such that $\mathsf{Aux}[i]$ represents the smallest ending value of all length-i LIS' found so far. Although the definition is quite confusing, it is easy to see that it is always ordered, that is $\mathsf{Aux}[i-1] < \mathsf{Aux}[i]$ as the second last element of any LIS (length-i) is smaller than it's last element. We can now use the means of binary search to determine the LIS: we update by appending the current element $\mathsf{A}[i]$– by finding an index of the last element in Aux that is less than $\mathsf{A}[i]$.

Looking for oil

Task : Given an $n \times n$ quadratic land, we need to subdivide the land into rectangles parcels and find the rectangle parcel with maximum profit.

The naive solution is to check for every possible rectangle in the quadratic land, which requires four nested loops leading the running time of the algorithm to $O(n^4)$.

To reduce the running time, we should first discuss the solution to the analogous problem in 1D, that is maximum sum contiguous subarray problem. The idea here is to maintain a cumulative sum of the elements seen so far and greedily reset that to zero if the cumulative sum goes below zero. This makes sense as restarting from zero is always better than continuing from negative cumulative sum. In the context of DP, we can either leverage the previously accumulated maximum sum, or start with a new subarray. Let $dp[i]$ represent the maximum sum of subarray that ends with element $\mathsf{A}[i]$. Hence, the final answer is the maximum over all $dp[i]$, where $i \in [n-1]$. The running time is $O(n)$.

Now, for the case in 2D, we extend the logic for 1D by employing the inclusion-exclusion principle. First, we calculate the cumulative sum for all columns. Assuming that the maximum subarray is between row r_1 and r_2 (inclusive) try all $O(n^2)$ (r_1, r_2) pairs such that $r_1 < r_2$. Since, we have already computed the cumulative sum of all columns, the sum of elements in $\mathsf{A}[r_1 \dots r_2][c]$ for column c can be computed in $O(1)$ time. Intuitively, this means that each cumulative sum for a column is an element of a 1D array across all columns, that is 1D array with one row and n columns. Using the 1D counterpart ($O(n)$) as a subroutine inside each rows r_1 and r_2 combined gives an algorithm with running time $O(n^3)$.

Longest Common Subsequence

Task : Given two sequences $\mathsf{X}[0 \dots n-1]$ and $\mathsf{Y}[0 \dots n-1]$, find the length of the longest subsequence that is present in both of them.

Let $\mathsf{L}(\mathsf{X}[i], \mathsf{Y}[j])$ be the length of LCS of two sequence of first i elements of X and the first j elements of Y. The recurrence can be written as:

- $\mathsf{L}(\mathsf{X}[i], \mathsf{Y}[j]) = 1 + \mathsf{L}(\mathsf{X}[i-1], \mathsf{Y}[j-1])$, if $\mathsf{X}[i] = \mathsf{Y}[j]$.
- $\mathsf{L}(\mathsf{X}[i], \mathsf{Y}[j]) = \max\{\mathsf{L}(\mathsf{X}[i-1], \mathsf{Y}[j]), \mathsf{L}(\mathsf{X}[i], \mathsf{Y}[j-1]\}$

We can see clearly that there are many overlapping subproblems in order to compute $\mathsf{L}(\mathsf{X}[n], \mathsf{Y}[n])$. Using, tabulation we can update the tables and the running time is $O(n^2)$ and $O(n^2)$ space.

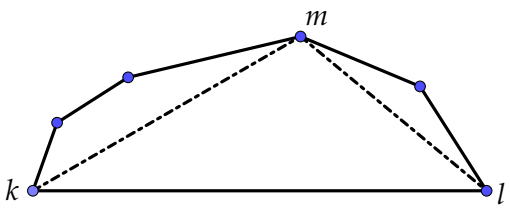
We would like to make a remark that we can also optimize the space used by the DP algorithm according to the recurrence mentioned above. The optimization of space is possible due to the following observation: if we look closely at the lookup table, then in order to compute the values of current row, we only need values from all columns of previous row. Hence, there is no need to store all rows in the lookup table, we can just store two rows at a time and use them. In this way, the used space reduces by n-fold (i.e. $O(n)$).

Triangula

Task : Given a convex polygon with n vertices, we need to divide the polygon into triangles in such a way that the partition minimizes the total length of shared lines (diagonals). This is a variant of minimum cost triangulation of a convex polygon, where the objective function to minimize is the sum of the diagonals.

Assuming the vertices of the polygon are either given in clockwise direction, we proceed by defining the subproblem. For $1 \leq k \leq l \leq n$, let $\mathsf{T}(k, l)$ be the minimum cost triangulation of the polygon spanned by vertices $[k, l]$. By formulating the problem in this fashion, we enumerate all the possible diagonals that do not cross each other. The recurrence can now be written as:

- $\mathsf{T}[k, l] = \min_{k \leq m \leq l}\{\mathsf{T}[k, m] + \mathsf{T}[m, l] + dist(k, m) + dist(m, l)\}$



The base case $\mathsf{T}[i, i]$ is always zero as a diagonal can't form due to same vertex. Also, the diagonal only exists between vertices whose order differ by two. The distance is measured as follows:

- $dist(k,l) = \begin{cases} \sqrt{(x_k - x_l)^2 + (y_k - y_l)^2}, & \text{for all } k - l \geq 2 \\ 0, & \text{otherwise} \end{cases}$

The algorithm is as follows:

ALGORITHM : TRIANGULA

FOR $i = 1$ to n DO

$\mathsf{T}[i,i] = 0$

DO

FOR $j = 1$ to $n - 1$ DO

FOR $k = 1$ to $n - j$ DO

$l = k + j$

$\mathsf{T}[k,l] = \min_{k \leq m \leq l}\{\mathsf{T}[k,m] + \mathsf{T}[m,l] + dist(k,m) + dist(m,l)\}$

OD

RETURN $\mathsf{T}[1,n]$

OD

The computation of $\mathsf{T}[k,l]$ takes $O(l-k)$ time which is $O(n)$. The outer two *for* loops takes $O(n^2)$. Hence, the total running time is $O(n^3)$.

Graph Algorithms

Breath First Search

BFS is one of the most fundamental algorithms for graphs. It traverses through all vertices reachable from the starting vertex s. The traversal is done by scanning all neighboring vertices layer by layer, starting with vertices at distance one to s, then distance two to s, and so on.

```
ALGORITHM : BFS(s) // all the vertices are initially white colored
    color[s] = gray // gray:after we visit the vertex but before
    scanning all it's neighbors
    d[s] = 0
    ENQUEUE(Q, s)
    WHILE Q not empty DO
        DEQUEUE(Q, u)
        FOR (u, v) ∈ E DO
            IF color[v] = white THEN // white:before we start
               color[v] = gray
               d[v] = d[u] + 1
               parent[v] = u
               ENQUEUE(Q, v)
            FI
            color[u] = black // black:after we have visited the
            vertex and it's neighbors
        OD
    OD
```

The aforementioned algorithm runs in $O(V + E)$ time. Note that if the graph is not connected, we will have to change the starting vertex (source) as many times as the number of different connected components. This adds an additional $O(V)$ time but the overall running time still remains $O(V + E)$.

Bipartite or not?

Task : Given an undirected graph $G = (V, E)$, we want to check whether the underlying graph is bipartite or not?

Again, this problem can be modeled into $2 - coloring$ problem. We use BFS algorithm as a subroutine and first assign black color to the source. Then color white to all the neighbors of the source vertex. Similarly following this procedure to do m-coloring with the constraint that $m = 2$, and checking at the end, if we find two vertices with same color, then it's non-bipartite. Else, it is a bipartite graph. The running time of the algorithm is $O(V^2)$ if adjacency matrix is used to represent the graph, while we get $O(V + E)$ running time for the adjacency list representation of the graph.

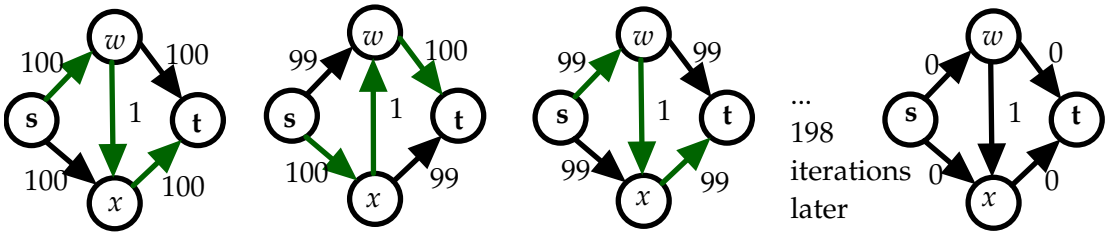
Maximum Flow

Task : Given a directed graph $G = (V, E)$ and a capacity function $u : E \to \mathbb{R}_{\geq 0}$. Given multiple sources ($S = \{s_1, \ldots s_a\}$) and multiple sinks ($T = \{t_1, \ldots t_b\}$), we want to compute the maximum flow between the sources and the sinks. This problem is commonly referred to as multi-source multi-sink max flow problem.

Note that this variant is no harder than the original max flow problem with single source and single sink. For solving this problem, create a super source s^* and super sink t^*. Connect s^* with all $s_i \in S$ with infinite capacity and doing the same on the sink side where the edge is directed from $t_i \in T$ towards t^*. Then run Ford Fulkerson/Edmond Karp's algorithm as normal. We now give a brief description about them.

Ford Fulkerson's algorithm iteratively finds an augmenting path p, that is a path from source to sink that passes through positive weighted edges in the residual graph [1]. After finding the augmenting path p that has f as the minimum edge weight along the path p, the algorithm now keeps track of two important things: decrease/increase the capacity of forwards and backward edges along path p by f, respectively. It goes on repeating this process until there is no possibility of augmenting path from source to sink, implying that the total flow so far is the maximum flow.

The reason behind decreasing the capacity of forward edge is that by doing so, we decrease the remaining capacities of the forward edges used in p. The not quite obvious reason behind increasing the capacity of backward edge is that by doing so, it allows $flow$ to cancel some part of capacity used by forward edge that might be incorrectly used by earlier flow. This is the crux for the correctness of this algorithm.



Ford Fulkerson's algorithm implemented using DFS has a worst case running time of $O(|f^*|E)$, where $|f^*|$ is the maximum flow value. This can be explained by analyzing the above figure where we may encounter a scenario where two augmenting paths $s \to w \to x \to t$, and $s \to x \to w \to t$ only decrease the edge capacities along the path by one. This is repeated for 200 times ($|f^*|$ times) in

[1] The residual graph of the flow network indicates how much additional flow is possible. The initial state of the residual graph is with all edges having their original capacity. But if an edge (i, j) is used by the augmenting path and a flow $f < u(i, j)$ passes through this edge, then the remaining capacity of the edge is $u(i, j) - f$.

worse case. Moreover, the DFS runs in $O(E)$ time on flow graph[2], the overall running time is $O(|f^*|E)$. This implies that it is a pseudo-polynomial time algorithm and the convergence is never guaranteed (in the case where the edge capacities are rational).

A more clever implementation of the above algorithm is known as Edmonds Karp algorithm. This algorithm is identical to Ford Fulkerson's algorithm in many ways, except the search order for finding augmenting paths is differently defined. The augmenting path must be the shortest path with available capacity. This is done by giving weights of 1 to each edge and additionally applying BFS gives us such path. Notice that each augmenting path here can be found in $O(E)$ time and that every time at least one of the $|E|$ edges become saturated (maximum capacity used). Also the distance from the saturated edges to source along the augmenting path must be longer than last time it was saturated, and the length is at most $|V|$. This shows that the running time of the algorithm is $O(VE^2)$.

1984

Task : Given an undirected railway network with track (edge) costs and station (node) costs. We need to calculate a minimum cost in order to destroy tracks and stations in such a way that there is no connection from one station s to another station t. Note that theses two stations cannot be destroyed. More formally:

Input

- An undirected graph $G = (V \cup \{s, t\}, E)$, where V is the complete set of stations and E is the set of tracks.
- A cost function $c : V \cup E \to \mathbb{N}$

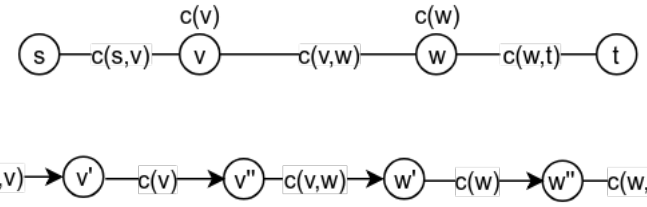
Output

- Find $stations \subset V$ and $tracks \subset E$ such that
  - s and t are in different connected components of graph defined on $V \setminus stations$ and $E \setminus tracks$.
  - $c(stations \cup tracks)$ is minimum

In order to solve this problem, we transform the original graph into a new graph by converting stations into edges without changing the costs. Then the problem just boils down to solving minimum $s - t$ cut in the transformed graph G'. We define G' as follows:

[2]In flow graph, edges must follow $|E| \geq |V| - 1$ to ensure the existence of at least one $s - t$ flow. This implies that BFS and DFS–using adjacency list–runs in $O(E)$ instead of $O(V + E)$.

$$V' = \{v' \mid v \in V \setminus \{s,t\}\} \cup \{v'' \mid v \in V \setminus \{s,t\}\} \cup \{s,t\}$$
$$E' = \{(v',v'') \mid v \in V \setminus \{s,t\}\} \cup \{(v'',w') \mid \{v,w\} \in E \wedge v,w \in V \setminus \{s,t\}\} \cup$$
$$\{(s,v') \mid \{s,v\} \in E \wedge v \in V \setminus \{s,t\}\} \cup \{(v'',t) \mid \{v,t\} \in E \wedge v \in V \setminus \{s,t\}\} \cup$$
$$\{(s,t) \text{ if } \{s,t\} \in E\}$$



The new cost function $c' : E' \to \mathbb{N}$ is induced from the cost function c. Clearly, finding a minimum $s-t$ in the new graph would solve the purpose of the problem. The algorithm for min-cut is the same for the max-flow algorithm in the sense that min-cut is always the byproduct of computing max-flow. That is, after the max-flow algorithm (either Ford Fulkerson or Edmond Karp) terminates, we run BFS from the source again. All reachable vertices from source s using positive weighted edges in the residual graph belong to s-component, while the unreachable belong to t-component. Moreover, all edges connecting s-component to t-component belong to the cut-set. Hence, min-cut is equal to max-flow value.

Miscellaneous

Collinearity

Task : Given n points in $\mathbb{R}^2$, find the maximum number of points lying on the line, when a line is drawn through $\mathbb{R}^2$. Note that the coordinates are given in $float$ datatype.

This setting of the problem is slightly difficult implementationally. The overall logic remains the same, that is for every point p calculate it's slope with other points and use a map to keep a log on how many points have the same slope, which enables us to compute how many points are on the same line with p being one of them. We do this for each point by updating the maximum count of points found so far.

Nevertheless, we would like to mention few tricks while implementing this logic when coordinates are given in $float$ datatype:

- For two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, the slope $m = \frac{y_2 - y_1}{x_2 - x_1}$ which can be of *double* datatype, and can cause precision problems. As a remedy to

this problem, we treat the pair $(y_2 - y_1, x_2 - x_1)$ as slope and reduce it to the lowest order by computing their *gcd* after which it is inserted into the map. We also treat vertical and repeated points separately.

- Using unordered map of C++ for storing the slope pair defined above, the running time of the algorithm is $O(n^2)$.

Summary

All the programs were written in Python 2, Python 3 and C++. They were submitted to the Lab's website. The problems required knowledge of many different algorithms and computer science techniques. These were also discussed and explained during the meetings. It was really fun to solve these problems and really helped me by forcing me to learn C++ as sometimes my Python codes gave TLE.