FADML Scribe

Week - 06

11-15 Feb 2025

Intractable Problem:

There are 3 ways to solve intractable problem:

1) **Efficient search (Combinatorial Optimization)** – This approach explores the solution space by using techniques like branch and bound, dynamic programming or heuristic search to prune unnecessary computations. This methods tries to improve efficiency by avoiding exhaustive brute-force searches.

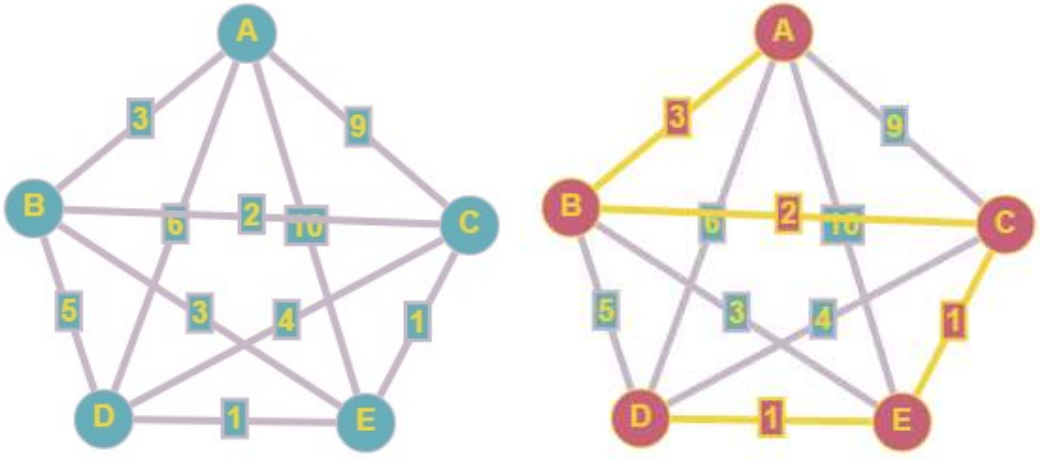
2) **Suboptimal Solution (Approximation)** – When finding the exact solution is infeasible computationally, algorithms provide near optimal solution within **a guaranteed error bound**. It uses technique like Greedy algorithms. As finding the exact optimal solution is infeasible, we trade accuracy for efficiency. The solution is not necessarily optimal, but it comes with a performance guarantee.

3) **Almost correct solution (Randomization)** – It introduces probabilistic techniques to find a solution that is likely correct or close to optimal. They do not necessarily compromise the solution's quality but introduce probabilistic elements, meaning the solution may be correct with high probability.

➔ Time complexity is minimized in efficient search
➔ Optimality is played with in suboptimal solutions
➔ Quality is usually not compromised in randomization, but correctness or runtime may vary probabilistically.

Travelling Salesman Problem: (Euclidean TSP)

Complete tour from start city to all other cities such that each city is visited only once and return back to start city at minimum cost.



Solution:

- ➔ Find a minimum spanning tree using Kruskal's algorithm. And then return back to starting vertex.
- ➔ Minimum spanning tree for the given graph is A -> B -> C -> E -> D (3 + 2 + 1 + 1 = 7)
- ➔ The best possible cost of traveling salesman tour is never less than the cost of MST because after visiting every vertex, the salesman has to return to the origin.

$$L^* > MST(G)$$

Suppose, for other graph as shown below, we may need to travel from M to K using backtrack. So, number of back edges can be at maximum number of total edges.

Assume that the distance between K to L is x, L to M is y and M to K is z. Then, for the Euclidean travelling salesman problem satisfying the in equality z < x + y, algorithm can't give worst solution than 2 * MST (G).

$$2 * MST(G) < Lsol < L^* < MST(G)$$

Where, Lsol is solution of the algorithm, L* is the best solution and MST(G) represents minimum spanning tree of a graph.

➔ For Euclidean TSP, PTAS (Polynomial-Time approximation scheme) exists and for non-Euclidean TSP, PTAS doesn't exist.

Hamiltonian cycle reduction to Non Euclidean TSP

Hamiltonian cycle problem:

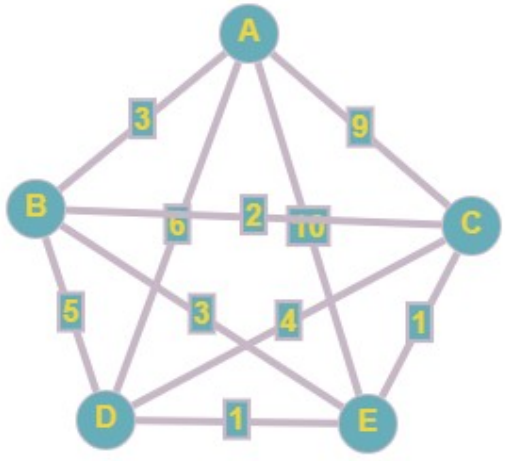
➔ Given an input instance of an undirected graph and problem is to find a path that visits every vertex in the graph exactly once and ends at the same vertex it began with.

Steps:

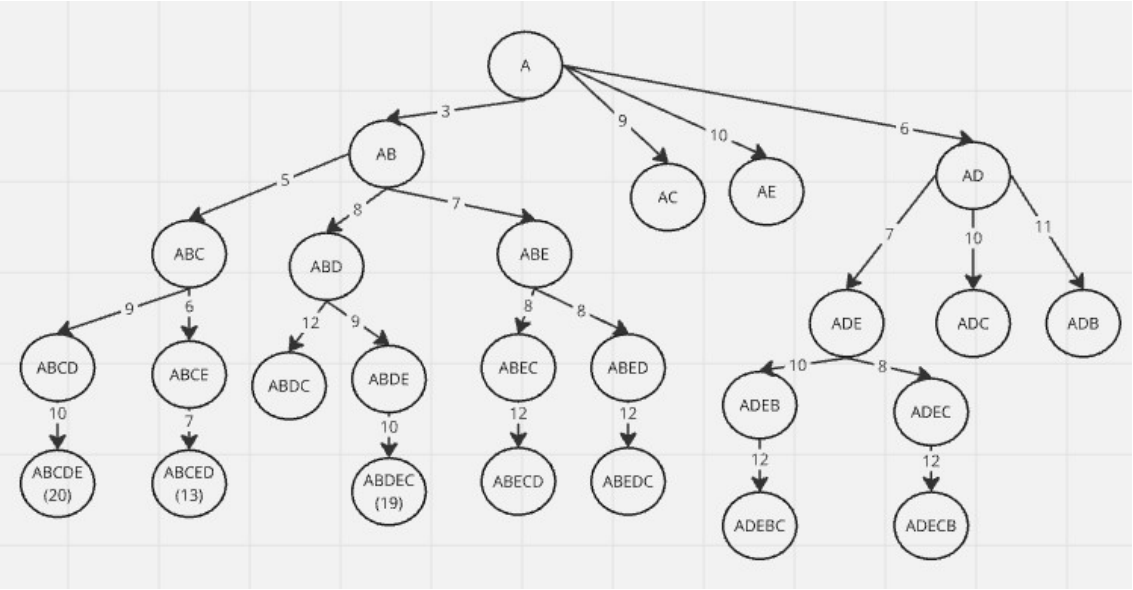
1) Assign weight 1 to all the edges
2) Produce all the other edges that doesn't exists and assign it a weight infinity.
3) If polynomial time solution exists for Non Euclidean TSP, then
   i) If there is a tour of length inf or
   ii) If there is a tour of length < inf
4) If the minimum cost of the tour is inf, then the Hamiltonian cycle doesn't exist and if the minimum cost of the tour is < inf, then the Hamiltonian cycle exists.

➔ So, if the Hamiltonian cycle is NP-complete, this reduction implies that Non-Euclidean TSP is at least as hard as the Hamiltonian cycle.

Combinatorial Optimization (Trying all combinations)



Solution: Start with vertex A and try all possible combinations. And use pruning when path of particular path is already more than minimum cost obtained till that point.



Here, first A -> B -> C -> D -> E -> A path is explored according to DFS and total cost is 20.

Then, second A -> B -> C -> E -> D -> A path is taken for which total cost is 13.

Then, while exploring the path A -> B -> D -> C, total cost encountered is 12 and there are still 2 nodes to visit and so the total cost can't be less than 13. So, this branch is not explored further.

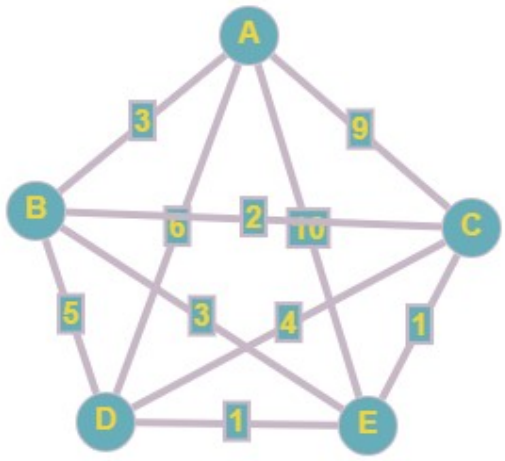
Same way pruning is applied after considering all the combinations, (i.e. A -> C and A -> E are not explored as it can't give less than 13 cost solution and there are other pruned branches as well)

A* algorithm (Best-First Search)

- ➔ In this algorithm, heuristic values for reaching to the goal is also included in the graph which helps in pruning more branches quickly.
- ➔ Function used for A* algorithm is: f(n) = g(n) + h(n), where g(n) = cost from start node to node n, h(n) is heuristic cost from n to goal and f(n) is total estimated cost of the path through n
- ➔ Heuristic values are determined based on domain knowledge.
- ➔ A* algorithm works best when heuristic value h(n) never overestimates and as close as possible to actual cost. In that case, we might not need to explore extra nodes and hence computation will not increase.
- ➔ If h(n) overestimates the actual cost, as A* is kind of greedy algorithm, we potentially skip better paths at earlier stages and hence need to explore all combinations.
- ➔ The best you can under approximate, the more branches you can prone.

Hill Climbing Algorithm

- ➔ In this algorithm, we start with an initial path which is random initialization.
- ➔ And then generate neighbouring paths by making small changes (Swapping two cities) and evaluate the cost of each neighbouring path. Based on the cost, we move the best neighbour.
- ➔ And stopping criteria is also set like if there is not improvement in the last 3 iterations, then we stop. (Local minima is reached)



- ➔ For this graph, we initialize random solution A B C D E and then come back to A. Total cost for the path is 20. In the next iteration, we change position of nodes and check if any improvement is made.
  Initialization: A B C D E (20)
  Iteration 1: A B E D C (20)
  Iteration 2: A B D E C (19) and so on.

Problems in Hill climbing:

1) Local Minima – The algorithm may stuck in suboptimal solution and can't find better global solution as no neighbout is better. In this case, random restarts could help in overcoming the problem.
2) Plateau – A flat region where all neighbouring solutions have same value and hence no progress can be made. In this case, random jumps can help.

- ➔ Hill climb problem uses similar technique like stochastic gradient descent.

Simulated Annealing

- ➔ Unlike hill climbing, in simulated annealing move towards worst solution is accepted occasionally with probability P = $A = e^{\frac{-t}{z}}$. Initially, there are more shakes or fluctuations in the solution which helps in jumping out of deep but suboptimal valleys (Local Minima) but it gradually decreases and couldn't climb the mountain.

Tic-Tac-Toe

- ➔ It is a 2 player game played on 3 X 3 grid where player take turns marking X or O. And the goal is to finish row, column or a diagonal with three of the same symbol.
- ➔ The game can be solved using Minimax algorithm, where one player (X) tries to maximize his score and at the same time, opponent (O) is trying to minimize X's score.
- ➔ Optimization for Minimax algorithm is Alpha-Beta pruning which pruns unnecessary branches of the game tree. Where, alpha is best score maximizer can guarantee and beta is best score, minimizer guarantees. And if at any node, alpha > beta, then it is not eplored further as opponent will never allowed to reach this state.
- ➔ **Alpha-Beta Pruning** is an optimization for the Minimax Algorithm that prunes unnecessary branches of the game tree, reducing computation. Alpha **(α)** represents the best score the maximizer can guarantee, while Beta **(β)** represents the best score the minimizer can guarantee. If at any node, **α ≥ β**, further exploration is stopped because the opponent will never allow reaching this state.

Genetic Algorithm

- ➔ Genetic Algorithm is an optimization technique inspired by natural selection. The key steps are selection, crossover and mutation.
- ➔ Crossover (Recombination): It is a process where two parent solutions combine to create offspring. By acquiring characteristics from both parents, it facilitates the exploration of new areas of the solution space.
- ➔ Mutation: It prevents early convergence and preserves variety by introducing random changes in offspring. It facilitates genetic algorithm's exploration of novel solutions and escape from local optima.

Longest Palindromic Subsequence

Problem:

Given a string S, find the length of the longest subsequence that is a palindrome.

Input:

A string of length n where $1 \leq n \leq 1000$ and string contains lowercase letters only.

Output:

Integer representing the length of the LPS.

Example 1:

Input : "bbbab" => Output : 4 ("bbbb")

Example 2:

Input – S = “cbbd” => Output – 2 (“bb”)

Solution

Using Top down memoization by slightly modifying longest common subsequence memoization solution. This algorithm maintains dp table to store intermediate results to avoid reduandant calculations. If the first and last characters match, then LPS is extended by 2 plus the remaining substring. Otherwise, it takes the maximum LPS by excluding either end.

```
n <- len(s)

Create memoization table dp[n][n] and initialize all
elements to -1

LPS(S):

# Base case

If i > j: (Invalid range)

        Return 0

If i == j: (Only one character – filling diagonal elements)

        Return 1

If dp [i][j] != -1 (Already computed)

        Return dp [i][j]

If s[i] == s[j]:

        dp[i][j] <- 2 + LPS(i + 1, j – 1)

Else:

        dp[i][j] <- max (LPS[i + 1, j), LPS(i, j-1))

Return dp[i][j]
```

Length 2 subsequence:

dp[0][1] = 2 (“bb”, s[0] == s[1], so 2 + LPS(“”) = 2)

dp[1][2] = 2 (“bb”, s[1] == s[2], so 2 + LPS(“”) = 2)

dp[2][3] = 1 (“ba”, s[2] != s[3], max(LPS(“b”), LPS(“b”) = 1)

dp[3][4] = 1 (“ab”, s[3] != s[4], max(LPS(“a”), LPS(“b”) = 1)

Length 3 subsequences:

dp[0][2] = 3 (“bbb”, s[0] == s[2], so 2 + LPS(“b”) = 3)

dp[1][3] = 2 (“bba”, s[1] != s[3], so max(LPS(“bb”), LPS(“ba”))= 2)

dp[2][4] = 3 (“bab”, s[2] == s[4], so 2 + LPS(“a”) = 3)

Length 4 subsequences:

dp[0][3] = 3 (“bbba”, max(LPS(“bbb”), LPS(“bba”)) = 3)

dp[1][4] = 3 (“bbab”, max(LPS(“bba”), LPS(“bab”)) = 3)

Length 5 subsequences:

dp[0][4] = 4 (“bbbab”, 2 + LPS(“bba”) = 4)

	j	**0**	**1**	**2**	**3**	**4**
i		b	b	b	a	b
0	b	1	2	3	3	4
1	b	-	1	2	2	3
2	b	-	-	1	1	3
3	a	-	-	-	1	1
4	b	-	-	-	-	1

Time Complexity:

There are O(n^2) subproblems as n X n DP table is filled. And each subproblem is taking O(1) time. Hence, the time complexity is O(n^2)

Alternative Solution:

Reverse the given string S and then find the length of longest common subsequence (Using memoization table) between original and reversed string. In this case, reversing the string will take O(n) time and finding longest common subsequence will take O(n^2) time. Hence, total time complexity of the problem will be O(n^2).

Closest Pair Problem (2-D)

Input:

Set of n points in 2D [(x1, y1), (x2, y2), (x3, y3), …]

Output:

Find the closest pair [(xi, yi), (xj, yj)]

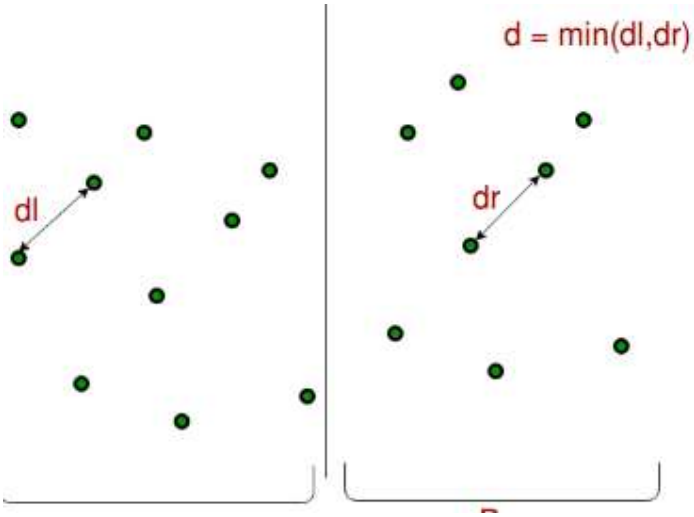
Brute – Force solution:

Calculate distance between each pair and find the smallest out of that. Which is O(n^2).

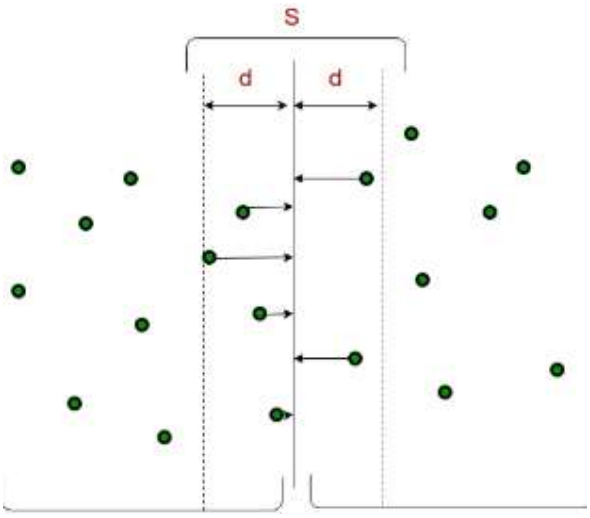
Divide and Conquer Approach O(n $log^2 n$):

1) Find the middle point in the sorted array based on x coordinate and divide the array in two halves.
2) Find the closest pair in the left subarray. And calculate the minimum distance between 2 points as dl.

3) Find the closest pair in the right subarray. And calculate the minimum distance between 2 points as dr. And d = min(dl, dr)



4) Check if there exists a pair such that one point is on the left and the other point is on the right and the distance between them is) less than d. For this follow the below steps:
   - Consider the vertical line passing through middle point and get all points whose x coordinates are closer than d to the middle vertical line. Build an array window for all these points.
   - Sort all these points according to y coordinates.
   - Find the smallest distance among these points. It takes O(n^2) time for large n but it is geometrically proven, that at most 7 points are needed to be checked for every point in the strip. Hence it is O(n) and not O(n^2).
   - Return the minimum from the minimum distance calculated using strip and d.



Why 7 points and not more?

- ➔ Consider a strip, its width is 2d (Where, d is the minimum distance between left and right halves)
- ➔ And then points are sorted by y-coordinates.
- ➔ So, in 2d X d box, if there are 8 or more points, at least two of them must be closer than d, which contradicts the earlier assumption that d is the minimum distance. Hence, only 7 points can be packed while maintaining a distance of at least d apart.

Time Complexity:

$T(n) = 2\,T(n/2) + O(n) + O(n \log n) + O(n)$

$T(n) = 2\,T(n/2) + O(n \log n)$

$T(n) = O(n\ log^2 n)$