Algorithm design
Aryan Gholami
40030973
JAN 15

TSP is one of the hardest problems in CS, meaning there isn't any polynomial time algorithm to solve this problem. In this project, we focus on both deterministic and approximate methods to solve this problem. Moreover, the deterministic method cannot solve this problem in an acceptable time.

An **approximate algorithm** is a method used to find near-optimal solutions to problems that are computationally hard to solve exactly, especially for large input sizes. Unlike exact algorithms, which guarantee finding the best possible solution (often in an optimal time for smaller cases), approximate algorithms provide solutions that are close to optimal but with a lower computational cost.

For problems like **TSP (Travelling Salesman Problem)**, where the goal is to find the shortest possible route visiting all cities, solving the problem exactly can be extremely time-consuming for large numbers of cities, as it is an NP-hard problem. Approximate algorithms, therefore, are used to find solutions that are "good enough" within a reasonable amount of time.

In this project folder, there are some header files, a cpp file as the main program, and a Python file that handles a simple GUI. The header files contain the DP algorithm implementation, MST-based algorithm, and My_algorithm. The DP and brute force algorithms solve this problem deterministically, while the MST_base algorithm solves it approximately with polynomial time complexity I will describe the algorithm that use in my project

1. DP and Brute Force approach
2. MST_Base
3. An algorithm that works on the branches of the MST

DP and Brute Force approach

In brute force approach we determine all hamiltonian path in graph then return the minimum of them it can be very slow due to the O(n!) time complexity.

The **Dynamic Programming (DP)** approach for solving TSP is based on breaking the problem into overlapping subproblems and solving them efficiently by storing intermediate results. One of the most popular DP algorithms for TSP is the **Held-Karp Algorithm**. This algorithm systematically explores all possible paths while avoiding redundant computations by using memoization.
It pseudocode blow

```
function algorithm TSP (G, n) is
    for k := 2 to n do
        g({k}, k) := d(1, k)
    end for

    for s := 2 to n−1 do
        for all S ⊆ {2, ..., n}, |S| = s do
            for all k ∈ S do
                g(S, k) := min_{m≠k,m∈S} [g(S\{k}, m) + d(m, k)]
            end for
        end for
    end for

    opt := min_{k≠1} [g({2, 3, ..., n}, k) + d(k, 1)]
    return (opt)
end function
```

It takes O(n*2^n) time.

My implement

The find_all_Hamiltonian_path function is a recursive method to generate all possible Hamiltonian paths (complete tours visiting each city exactly once). The key steps are:
Use a visited set to keep track of the cities already included in the current path.
Recursively add unvisited cities to the path.

Once a complete path is formed, append it to the res vector.
The compute_val function evaluates all generated paths, calculates their costs, and identifies the optimal path with the minimum cost.

Key Characteristics:

Time Complexity: due to the enumeration of all possible permutations.
Suitable for small problem sizes where (number of cities) is manageable.

The dynamic programming function (totalCost) optimizes the TSP solution by using memoization and bit masking. The approach involves:

State Representation:

A bitmask representing visited cities.
The current city.

Stores the minimum cost of visiting all cities in mask starting from curr.

Recursive Relation: For every unvisited city , compute the cost of visiting from the current city and recursively calculate the cost of visiting the remaining cities.

Base Case: When all cities are visited (i.e.,), return the cost of returning to the starting city.

Path Reconstruction: The path matrix stores the next city to visit for each state, allowing reconstruction of the optimal path.

The tsp function initializes the DP structures (memo and path) and calculates the minimum cost. It also reconstructs and prints the optimal path.

Key Characteristics:

Time Complexity: , significantly faster than brute force for small-to-medium .

Space Complexity: for the memoization table.

MST_base

There are a lot of algorithm to approximate the TSP . In my project I implemented the below algorithm

1. Compute the MST using by Prime algorithm
2. Performing a preorder traversal on a tree
3. Make a hamiltonian path

This algorithm has a prove that compute the path that its cost double the optimal solution.**(in metric graph)**

My implement

This implementation provides an approximate solution to the TSP using a Minimum Spanning Tree (MST) approach. Below is a brief explanation of the key components:

find_min Function: This helper function identifies the unvisited node with the smallest edge weight.
Inputs: A vector of distances, a set of visited nodes, and references to store the minimum value and corresponding index.
Purpose: Ensures the MST construction always adds the least costly edge to the tree.

prim_algorithm Function: This function constructs an MST using Prim's algorithm.
Steps:
Initializes the distance vector with infinity except for the starting node.
Iteratively selects the minimum edge connecting a visited node to an unvisited one.
Updates distances for neighboring nodes and adds the edge to the MST.
Output: The MST is stored in an adjacency list (list_adj).

Mst_base Function: This function leverages the MST to approximate the TSP solution.
Steps:
Constructs the MST using prim_algorithm.
Performs a **preorder traversal** of the MST to determine the order of visiting cities.
Calculates the total cost of the approximate TSP tour and prints the path.
Key Advantages:
Time Complexity: The MST-based approximation runs in , making it suitable for larger instances compared to exact methods.
Produces a feasible, though suboptimal, solution with significantly reduced computational effort.

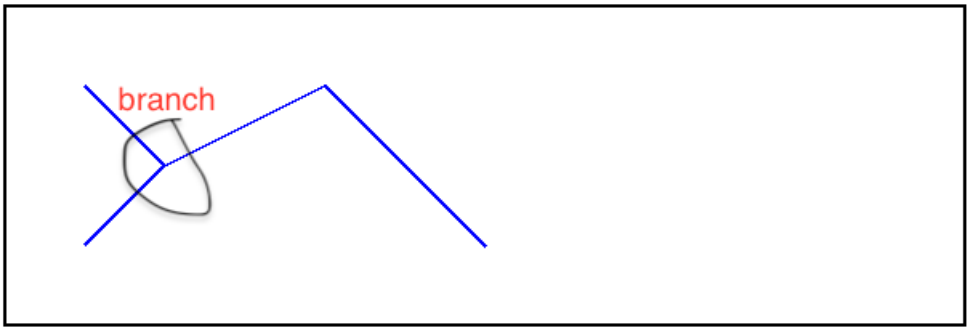
An algorithm that works on the branches of the MST(https://www.mdpi.com/2076-3417/11/1/177)

The idea of this algorithm like Christofides algorithm but it works in branch of the mst
MST is quite similar to TSP; however, in MST, there may be multiple branches, which is why an MST is not a solution to TSP. However, by eliminating the branches using a greedy technique, the problem can be resolved.

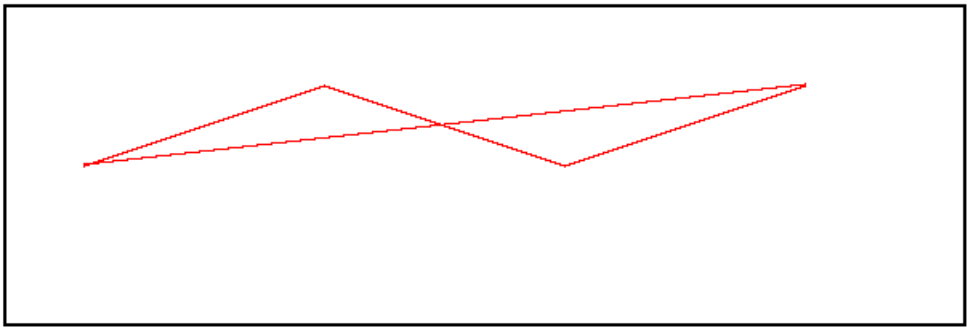
The trick is

I. Compute mst by Prime algorithm
II. While there is a branche
   - Find branches in tree
   - Delete the most cost edge (make two subtree)
   - Merge the subtrees by connecting the leaf with the minimum cost to the rest.

MST with branches



TSP solution (branches eliminated)



The branch node is anode that its degree more
than 2
A tsp cycle cannot have a branch.

My implementation

`primMST`:
Implements Prim's algorithm to find the Minimum Spanning Tree (MST) of a graph. It initializes the key values of nodes, selects the node with the minimum key, and adds it to the MST. The function updates the MST edges based on the node connections.

`compute_degress`:
Computes the degree (number of connections) of each node in the MST (represented by matrix `T`) and stores it in the `degress` vector.

`compute_branches`:
Identifies branch nodes in the MST. A branch is a node with a degree greater than 2. These nodes are stored in the `branches` vector.

`delete_branch`:
Removes a branch node from the MST by identifying the node with the maximum edge weight (cost) and deleting it from the graph.

`leaves_visit`:
Recursively traverses the MST starting from a given root node and collects leaf nodes (nodes with degree 1) into the `leaves` vector.

`leaves_visit_helper`:
A helper function that initializes the visited set and clears the `leaves` vector before calling `leaves_visit` to collect leaves.

`merge_trees`:
Merges two trees in the MST by connecting the leaf nodes from both trees with the minimum-cost edge in the original graph. This creates a single tree.

`make_path_helper`:
Recursively creates a path by visiting nodes in the MST, starting from the root node, and adds the nodes to the `path` vector.

`make_path`:
Creates the final path for the TSP solution by combining paths from different branches, ensuring that all nodes are visited. It also computes the total cost of the path.

`print_res`:
Prints the resulting TSP path and its total cost. It outputs the sequence of nodes and the total cost of traveling through them.

`tspApproximationMST`:
The main function implementing the TSP approximation algorithm. It uses MST, deletes branches, merges trees, and finally constructs and prints the approximate TSP path.

In project file:

Files:

AdjacencyListMatrix.h
Implementation the adjacency list to keep graph data and this header file useful in mst_base algorithm

dp_and_brute_force.h
implementation of the deterministic algorithm (dp and brute force)

input.h
In this header file there are three functions

`read`:

Reads lines from a user-specified file and stores them in a vector of strings (`my_in`).

`make_adj`:

Converts the input strings into an adjacency matrix, representing the graph with edge weights.

`init`:

Initializes an empty adjacency matrix of appropriate size based on the number of edges in the input data.

mst base.h
implementation of mst_base algorithm

my_Algorithm.h
implementation of the branches Algorithm

GUI
This application supports a very basic GUI implemented using Tkinter in Python. To use the GUI, you need to convert the C++ file into an executable. I achieved this using the Clang compiler with C++20 support. Finally, it shows the path and its cost.