FOUNDATIONS OF ALGORITHM DESIGN AND MACHINE LEARNING

DATE: 11 Feb 2025

Intractable Problems

These are problems for which no efficient (polynomial-time) solution is known. They are often NP-hard, meaning that as the problem size grows, the time required to solve it increases exponentially.

Examples: The Traveling Salesperson Problem (TSP), the Knapsack Problem, and the Hamiltonian Cycle Problem are classic examples of intractable problems. For large instances of these problems, finding an exact solution is computationally expensive or impossible within a reasonable time frame.

Approaches to Solving Intractable Problems:

When exact solutions are infeasible, we use the following approaches:

A. Efficient Search: Utilize algorithms that efficiently explore the solution space, such as dynamic programming or branch-and-bound techniques, to optimize combinatorial problems by reducing time complexity.

B. Suboptimal Solutions (Approximation Algorithms)

Approximation algorithms are used to find near-optimal solutions to problems that are computationally hard (e.g., NP-hard problems like TSP). For many real-world problems, finding the exact optimal solution is too time-consuming or impossible for large inputs. Approximation algorithms provide good-enough solutions in a reasonable amount of time. Trade-off: They sacrifice some accuracy (the solution might not be perfect) for efficiency (the algorithm runs quickly).

Types of Approximation Algorithms:

1. Polynomial-Time Approximation Scheme (PTAS):

PTAS is a type of approximation algorithm that allows you to get a solution arbitrarily close to the optimal solution. - For any given $\varepsilon > 0$, PTAS provides a solution that is within a factor of $(1 + \varepsilon)$ of the optimal solution.

Example: If $\varepsilon = 0.1$, the solution will be at most 10% worse than the optimal solution.

The algorithm's running time depends on ε (how close you want the solution to be to optimal). For smaller ε (better approximation), the algorithm might take longer to run. However, for a fixed ε, the running time is polynomial in the input size, meaning it scales efficiently with larger inputs.

The **Euclidean TSP has a PTAS**. This means you can find a solution that is, say, 1% worse than the optimal solution, and the algorithm will still run efficiently for large inputs.

2. Constant-Factor Approximation:
These algorithms provide solutions that are always guaranteed to be within a constant factor of the optimal solution.
Example: The MST-based approximation for TSP is a 2-approximation algorithm. It guarantees that the solution is at most twice the cost of the optimal solution.
The algorithm runs in polynomial time (efficiently). Unlike PTAS, the approximation factor is fixed and does not depend on a parameter like ε.

C. Almost Correct Solutions (Randomization):
Uses probabilistic methods to find solutions that are close to optimal.
Example: Randomized algorithms like Monte Carlo methods, which provide approximate solutions with a certain probability.

Traveling Salesperson Problem (TSP):

The TSP is one of the most well-known combinatorial optimization problems. Finds the shortest possible route that visits each city exactly once and returns to the starting city.
- **Input**: A set of cities and the distances between them.
- **Output**: A cycle (route) that minimizes the total travel distance.

Types of TSP:

1. Euclidean TSP:
The distances between cities follow the Euclidean metric (straight-line distances).
- Satisfies the triangle inequality, meaning the direct path between two cities is always the shortest.
- Admits PTAS, allowing solutions arbitrarily close to the optimal solution.

2. Non-Euclidean TSP:
- The distances do not follow the Euclidean metric.
- Does not necessarily satisfy the triangle inequality.
- No PTAS exists for general metric TSP. However, approximation algorithms like Christofides' algorithm provide solutions within a factor of 3/2 of the optimal solution.

When exact solutions are infeasible, approximation algorithms provide near-optimal solutions efficiently.

MST-Based Approximation for TSP:
A subset of the edges of a connected, undirected graph that connects all the vertices together without any cycles and with the minimum possible total edge weight.
MST in TSP:
- Step 1: Find the MST of the graph representing the cities.

- Step 2: Duplicate the edges of the MST to create an Eulerian graph.
- Step 3: Use shortcuts (triangle inequality) to convert the Eulerian graph into a Hamiltonian cycle (TSP solution).

Approximation Bound: The TSP solution is at most twice the length of the MST.

Lower and Upper Bounds:

1. $MST(G) < L^*$

The cost of the MST is strictly less than the cost of the optimal TSP solution L*. Because,
- The MST connects all cities (vertices) with the minimum total edge weight, but it does not form a cycle. It is a tree, so it has no closed loops.
- The TSP solution, on the other hand, is a cycle (Hamiltonian cycle) that visits every city exactly once and returns to the starting city.
- To convert the MST into a TSP solution, you would need to add at least one more edge to create a cycle. This additional edge increases the total cost.

The TSP path will be only **slightly more than MST(G)**, this happens when **the shortcuts are highly effective**, meaning that almost every vertex can be visited using direct edges rather than following the full Eulerian path. **MST is already close to a Hamiltonian cycle**: and when **Graph is dense** (i.e., complete or nearly complete). The more edges available, the better the shortcuts we can take.

2. $L^* \leq L_{\text{sol}}$

The cost of the optimal TSP solution L* is always less than or equal to the cost of the approximate TSP solution L_{sol}. Because

- The approximate solution L_{sol} is obtained using an approximation algorithm (e.g., the MST-based approach or Christofides' algorithm).
- By definition, the optimal solution L* is the best possible solution, so any approximate solution will have a cost greater than or equal to L*.

3. $L_{\text{sol}} < 2 \times MST(G)$

The cost of the approximate TSP solution L_{sol} is strictly less than twice the cost of the MST. Combining the Inequalities and Putting it all together:

$$MST(G) < L^* \leq L_{\text{sol}} < 2 \times MST(G)$$

The TSP solution remains close to 2×MST(G), when

1. **MST is a long, branching structure (not cyclic)**:
    - The TSP path has to revisit many nodes to form a valid cycle.
2. **Graph is sparse** (few edges available):
    - If the graph doesn't have many edges, **shortcuts may not exist**.
    - We are forced to take nearly the full Eulerian Walk.
3. **Triangle inequality is weakly satisfied**:
    - If distances between nodes don't allow efficient skipping (e.g., graph is not fully connected or distances vary too much), shortcutting does not help much.

Hamiltonian Cycle Problem:

Hamiltonian Cycle Problem

- **Definition**: Given an undirected graph G=(V, E) the Hamiltonian Cycle Problem asks whether there exists a cycle that visits every vertex exactly once and returns to the starting vertex.
- **Nature**: It is a **feasibility problem** (yes/no question).
- **Complexity**: It is **NP-complete**, meaning no known polynomial-time algorithm solves it efficiently for all cases.

Traveling Salesperson Problem (TSP)

- **Definition**: Given a set of n cities, the distances between them, and an integer k, TSP asks whether there exists a tour of length at most k that visits each city exactly once and returns to the starting city.
- **Nature**: It is an **optimization problem** (finding the shortest possible tour).
- **Complexity**: It is **NP-hard**, meaning it is at least as hard as the hardest problems in NP.

The Traveling Salesperson Problem (TSP) is NP-hard, not NP-complete.

- Decision Version of TSP (Exact Path Constraint): The decision version of TSP asks whether there exists a tour of length at most k. This version is in NP because a given tour can be verified in polynomial time.
- Optimization Version of TSP: The standard TSP is an optimization problem, not a decision problem. Since NP-completeness applies to decision problems, TSP (as an optimization problem) is NP-hard because:
    - There is no known polynomial-time algorithm to solve it.
    - It is at least as hard as any problem in NP.

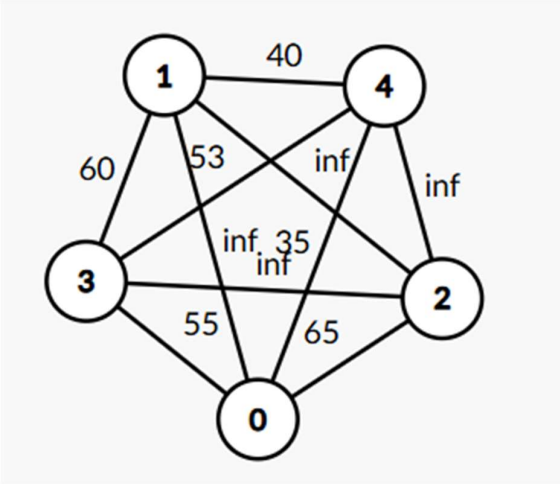
Hamiltonian Cycle to Non- Euclidean TSP:

If we could solve TSP optimally in polynomial time, we could solve all NP-complete problems like Hamiltonian Cycle.

- The Traveling Salesman Problem (TSP) assumes a complete graph, where every pair of vertices is connected by an edge. This means that in TSP, you can travel directly from any city (vertex) to any other city.

- However, the Hamiltonian Cycle (HC) problem is defined on a general graph, which may not be complete. In HC, not all pairs of vertices are necessarily connected by edges.

- So, to provide the input to TSP, the HC graph should be transformed into a Non-Euclidean TSP instance by assigning an infinite cost (`inf`) to edges that do not exist in the original Hamiltonian graph.



Step 1: Converting the Graph to a Non-Euclidean TSP Instance

1. **Start with the given Hamiltonian cycle input graph**: The input is an undirected graph with weighted edges.
2. **Modify the graph for Non-Euclidean TSP**:
    - If an edge exists in the original Hamiltonian graph, retain its cost.
    - If an edge does not exist in the original graph (i.e., the two nodes were not directly connected), assign its cost as **infinity (`inf`)**.
    - This forces the shortest path computation to avoid such edges unless no other valid path exists.

Step 2: Finding the Minimum Cost Tour

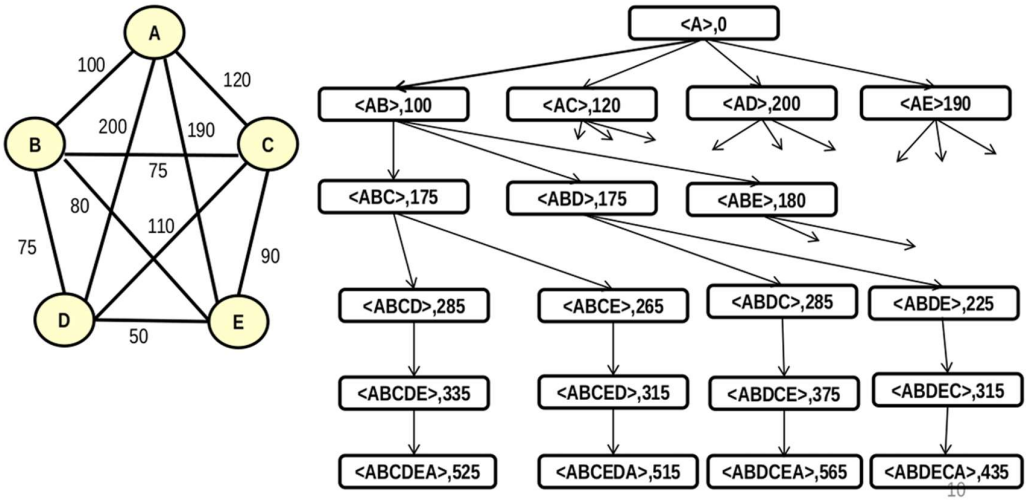
- The **goal** of the **TSP solver** is to find the shortest possible tour that visits all nodes exactly once and returns to the starting point.
- If a **finite-cost cycle exists**, then a Hamiltonian cycle is present.
- If the **minimum cost tour has an infinite (`inf`) cost**, then no Hamiltonian cycle exists because at least one node is unreachable in the required sequence.

The Hamiltonian Cycle Problem is NP-complete, meaning it is computationally hard. TSP (non-Euclidean) is NP-hard, but for our transformation, we assume a polynomial-time solution exists. The assumption allows us to check Hamiltonicity efficiently by solving the transformed Non-Euclidean TSP in polynomial time.

Solving TSP by (Branch and Bound):

The graph with nodes labelled **A, B, C, D, E** and provides the costs of traversing between certain nodes**.**

Branch and Bound (BnB) is an optimization algorithm that systematically explores possible solutions while eliminating unpromising branches early. For TSP, it ensures we find the shortest Hamiltonian cycle.



Steps in Branch and Bound for TSP:

1. Start from the Root Node (Initial City)
    - Begin at a starting city (e.g., A) with cost 0.
2. Expand Partial Paths (Branching Step)
    - From the current city, generate paths to all unvisited cities.
    - Track the accumulated travel cost.
3. Use a Lower Bound (Bounding Step)
    - Calculate a lower bound (minimum cost required to complete the tour).
    - If a path's lower bound exceeds the current best solution, discard it (pruning).
4. **Keep exploring paths until all cities** are visited and return to the start.
5. Update Best Solution
    - If a new completed tour has a lower cost than the previous best, update the solution.
6. Terminate When All Paths Are Pruned or Explored
    - The shortest valid tour is the optimal solution.

Solving TSP by (Heuristic Search _ A* Algorithm):

A* introduces a **heuristic function (h(n))** to estimate the remaining cost of a tour, which makes searching more efficient. So, you don't need to see the entire path.

1. **Start at the Root Node (A) with Cost 0**
    - Instead of expanding all paths equally, A* prioritizes paths with the lowest f(n)=g(n)+h(n).
    - g(n) = cost so far
    - h(n) = estimated cost to complete the tour (heuristic).
2. **Calculate the Heuristic h(n): Minimum Spanning Tree (MST) Lower Bound**
    - The heuristic estimates the **minimum** remaining cost using an MST.
    - Example: If unvisited cities form an MST with cost 250, we set h(n)=250.
3. **Expand the Most Promising Path First**

    - Instead of exploring all paths equally, A* **prioritizes** paths with the lowest f(n).
4. **Avoid Unnecessary Exploration**
    - A* often finds the optimal solution **without checking all possible paths**, unlike Branch and Bound.

A* is effective for small instances of TSP but becomes impractical for large datasets due to its exponential complexity. If the heuristic is good, A* will find the optimal path much faster than exploring all possibilities. But if the heuristic is bad, A* might not perform much better than Branch and Bound.

Other Heuristic Methods

1. **Nearest Neighbour Algorithm**:
    - Start at a random node. Always move to the nearest unvisited node. Return to the starting node at the end.
    - **Pros**: Fast, simple.
    - **Cons**: Often produces suboptimal solutions.
2. **Branch and Bound**:
    - Systematically explores all possible tours and eliminates paths that exceed the best-known solution.
    - **Pros**: Guarantees optimality.
    - **Cons**: Computationally expensive.
3. **Genetic Algorithms (GA)**:
    - Uses principles of natural selection (mutation, crossover) to evolve better solutions over generations.
    - **Pros**: Effective for large problems.
    - **Cons**: No guarantee of finding the absolute best solution.
4. **Simulated Annealing (SA)**:
    - Inspired by the cooling process of metals. Gradually reduces the probability of accepting worse solutions as the algorithm progresses.
    - **Pros**: Can escape local minima.
    - **Cons**: Requires careful tuning of parameters.