2025 Fall CSE5025

Combinatorial Optimization

组合优化

Instructor: 刘晟材

Lecture 3-2: Metric TSP and Routing

Agenda for Today's Lecture

In this lecture, we will focus on

- Metric TSP and its approximation algorithms
- Generalizations of TSP: Routing Problems

Learning Objectives for this lecture

- Know routing problems and classic approximation/heuristic algorithms for them
- Master the concept of the approximation ratio and be able to analyze it

What is an Approximation Algorithm?

An **approximation algorithm** is an algorithm that finds an approximate (i.e., not necessarily optimal) solution to an NP-hard optimization problem.

Key Properties:

- It must run in polynomial time.
- It must output a feasible solution
- There must be a provable guarantee on the quality of the solution

The Approximation Ratio (ρ):

This guarantee is quantified by the approximation ratio (or factor), denoted by ρ (rho).

For a minimization problem:

- An algorithm is a ρ-approximation if: Solution's Cost $\leq \rho \cdot$ Optimal Cost

For a maximization problem:

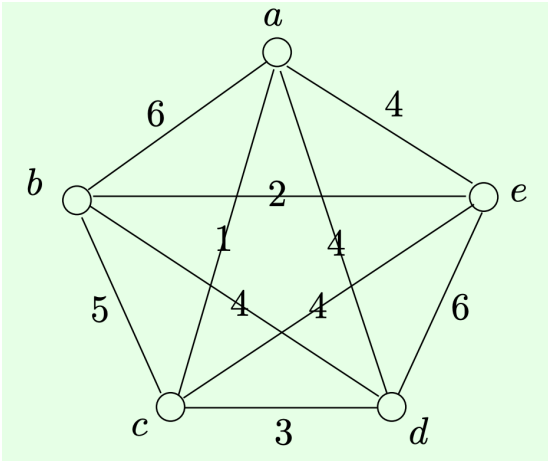
- An algorithm is a ρ-approximation if: Optimal Value $\leq \rho \cdot$ Solution's Value

In the above definition, the ratio ρ is always ≥ 1.

A 1-approximation algorithm is an exact algorithm.

Metric TSP

TSP: Given a complete weighted undirected graph with n vertices, and a cost function $c(u, v)$ that gives the the weight (cost) of each edge (u, v), **find a Hamiltonian cycle (tour) of least weight (cost)**.



An optimal solution: $a \to c \to d \to b \to e \to a$
with cost 14

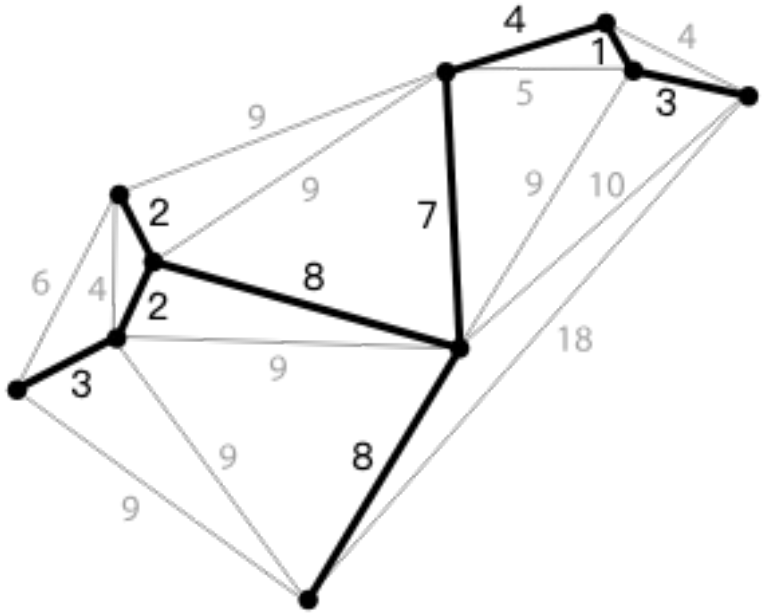
We saw in last lecture **General TSP is hard to approximate**.

Metric TSP: The edge costs satisfy the triangle inequality: for any three vertices u, v and w, the direct cost $c(u, w)$ is no more than the indirect cost via v, i.e., $c(u, w) \leq c(u, v) + c(v, w)$.

Minimum Spanning Tree (1)

Finding the optimal TSP tour is hard. Finding the optimal way to simply connect all vertices is easy.

That is to say, we can efficiently find a Minimum Spanning Tree (最小生成树, MST), i.e., the minimum-cost subgraph that connects all vertices.



Minimum Spanning Tree (2)

Definition of MST: A subgraph of a **connected, undirected, edge-weighted graph** that **links all vertices** together with the **minimum possible total edge weight** and **no cycles (**by definition of tree**)**.

Two well-known efficient greedy algorithms that find the MST:

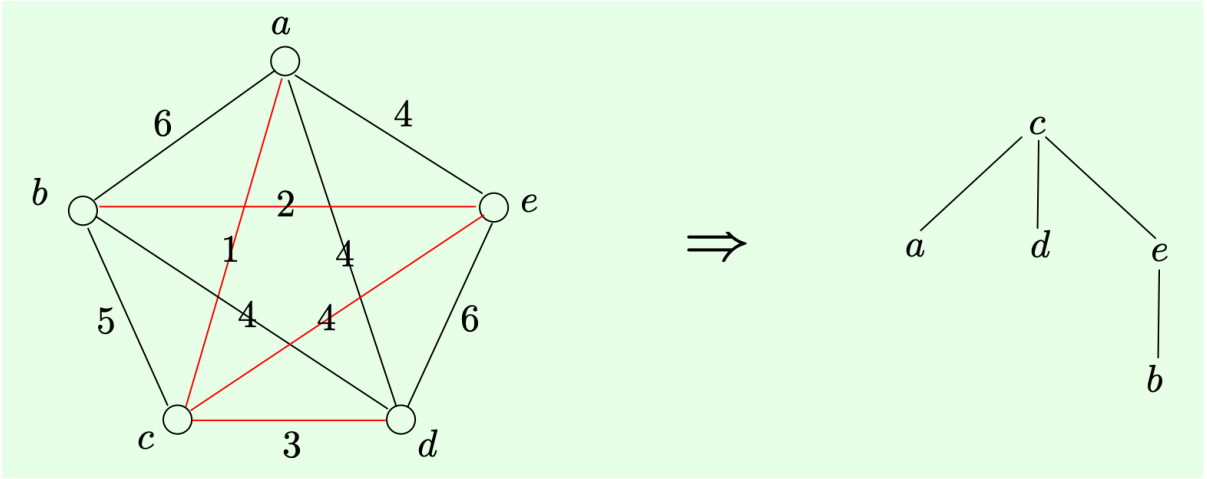
- Kruskal's Algorithm: Sorts all the edges by weight in increasing order and adds an edge to the growing forest if it does not form a cycle.
- Prim's Algorithm: Starts from an arbitrary vertex and continuously adds the cheapest edge that connects a vertex already in the tree to a vertex not yet in the tree.

Double-Tree Algorithm

Insight: Given a complete weighted undirected graph, the cost of an MST is always a lower bound on the cost of the optimal TSP tour $C(MST) \leq C(OPT)$.

Why? Because an optimal tour is a connected graph; removing one edge leaves a spanning tree, which must be at least as expensive as the MST.

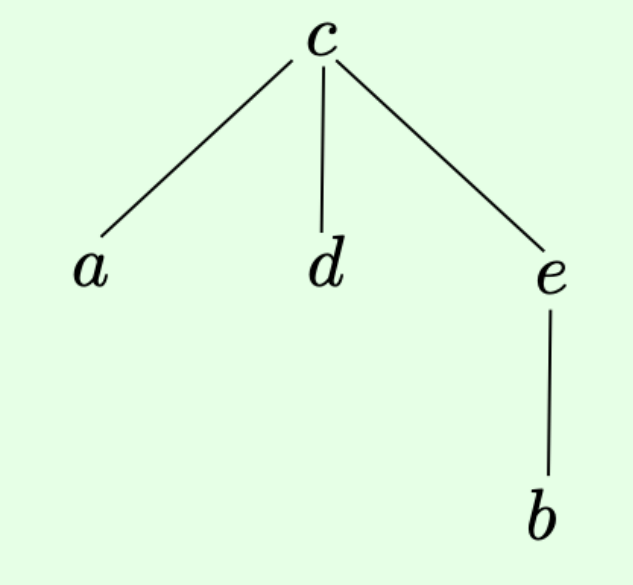
Question: Can we use the MST as a "skeleton" to build a low-cost tour?



Step 1: From a Tree to A Walk

An MST is not a TSP tour: vertices can have degrees $\neq 2$, and it's not a cycle.

However, we can easily create a walk (途/游走，允许重复经过顶点和边) that traverses the entire MST by "doubling" every edge: performing a full depth-first traversal of the MST (it must traverse every edge exactly twice).



The resulting walk:

$$c \to a \to c \to d \to c \to e \to b \to e \to c$$

Step 2: From a Walk to A Tour

The walk we created visits every vertex at least once, but it's not a valid TSP tour because it **revisits vertices**.

We can convert this walk into a valid tour by taking shortcuts.

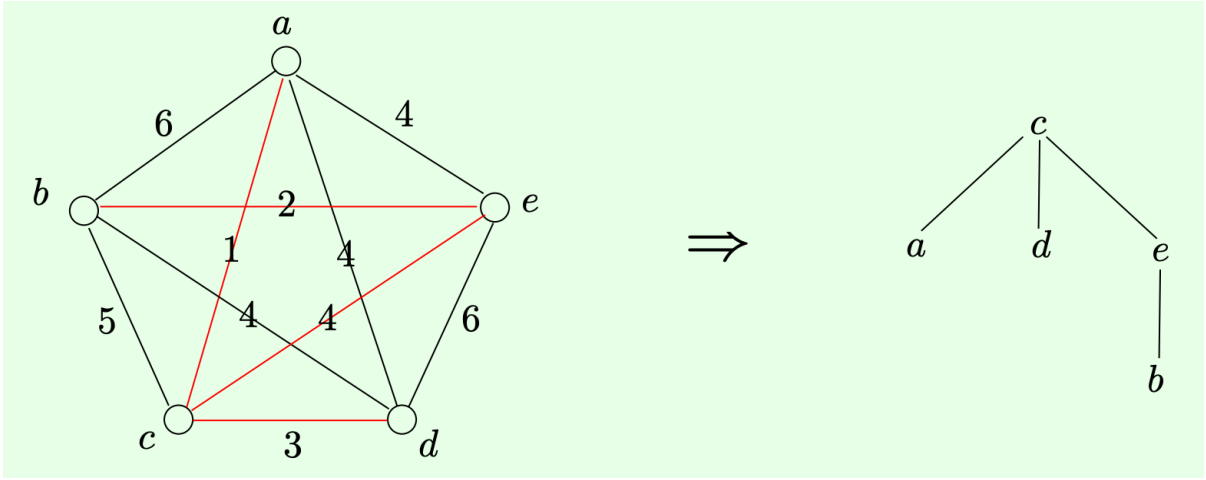
- We traverse the walk and build our tour by adding vertices in the order of their first appearance.
- When the walk would revisit a vertex, we instead take a direct path ("shortcut") to the next new vertex in the walk sequence. And it finally would give us **a valid tour (why?)**.
- **Why this tour can be low-cost**: The triangle inequality guarantees that the cost of a shortcut is never more than the cost of the path it replaces. Hence skipping a vertex can help reduce cost

Double-Tree Algorithm: Pseudocode

```
Algorithm Double_Tree_TSP(Vertices V, Cost_function c):
    // 1. Compute the Minimum Spanning Tree on the graph.
    MST = Compute_MST(V, c)

    // 2. Perform a depth-first traversal starting from an arbitrary vertex (e.g., V[0]) and record the order of vertices
    Walk = Get_DFS_Traversal_Order(MST, V[0])

    // 3. Create the final tour by taking shortcuts (removing duplicates) from the walk.
    Tour = []
    Visited_Set = {}
    for u in Walk:
        if u not in Visited_Set:
            Tour.append(u)
            Visited_Set.add(u)

    // 4. Add the starting vertex to the end to close the tour.
        Tour.append(V[0])

  return Tour
```

Example

Step 1:



Step 2: The resulting walk: $c \to a \to c \to d \to c \to e \to b \to e \to c$

Step 3: Tour: $c \to a \to d \to e \to b \to c$ with total cost 18

Double-Tree Algorithm: Complexity Analysis

Let n be the number of vertices.

Step-by-Step Breakdown:

1) Compute MST: Using Prim's algorithm on a dense graph takes $O(n^2)$.

2) Perform Depth-First Traversal: $O(n)$ on the MST (n vertices, $n-1$ edges)

3) Create final tour (using shortcuts): A single pass through the walk ($2n-2$ edges in total) takes $O(n)$.

The total complexity is dominated by the MST computation.

$$O(n^2) + O(n) + O(n) = O(n^2).$$

Double-Tree Algorithm: Approximation Ratio

Theorem: The Double-Tree algorithm is a **2-approximation** algorithm for the Metric TSP

We first define some notations before proving the theorem:

- T_{OPT}: An optimal TSP tour, with cost $C(OPT)$.
- MST: The Minimum Spanning Tree, with cost $C(MST)$.
- W: The walk traversing every MST edge twice, with cost $C(W)$.
- T: The final tour produced by the algorithm, with cost $C(T)$.

Proof of the Approximation Ratio (1)

Step 1: Relating MST Cost to Optimal Tour Cost

1) An optimal tour T_{OPT} is a cycle that connects all n vertices.

2) If we remove any single edge from T_{OPT}, the result is a spanning tree (a connected graph with n vertices, $n-1$ edges, and no cycles).

3) The cost of this spanning tree must be greater than or equal to the cost of the MST. Therefore, we have the inequality:

$$C(MST) \leq C(OPT)$$

Proof of the Approximation Ratio (2)

Step 2: Relating Algorithm's tour to the MST

1) The algorithm creates W by traversing every edge of the MST exactly twice. The total cost of this walk is therefore: $C(W) = 2 \cdot C(MST)$

2) The final tour T is constructed from this walk W by using "shortcuts". A shortcut replaces a sub-path in W (e.g., u → v → w) with a single direct edge (u → w).

3) Because this is Metric TSP, the triangle inequality holds: $c(u, w) \leq c(u, v) + c(v, w)$. This guarantees that taking a shortcut can never increase the total cost of the walk. Since T is formed by a series of such shortcuts on W, its total cost cannot be greater than the cost of W: $C(T) \leq C(W)$

Proof of the Approximation Ratio (3)

Step 3: Combining the Inequalities

We now chain our established facts together:

$$C(T) \leq C(W) = 2 \cdot C(MST) \leq 2 \cdot C(OPT)$$

We have proven that the cost of the tour produced by the Double-Tree algorithm is at most twice the cost of the optimal tour.

The Christofides Algorithm

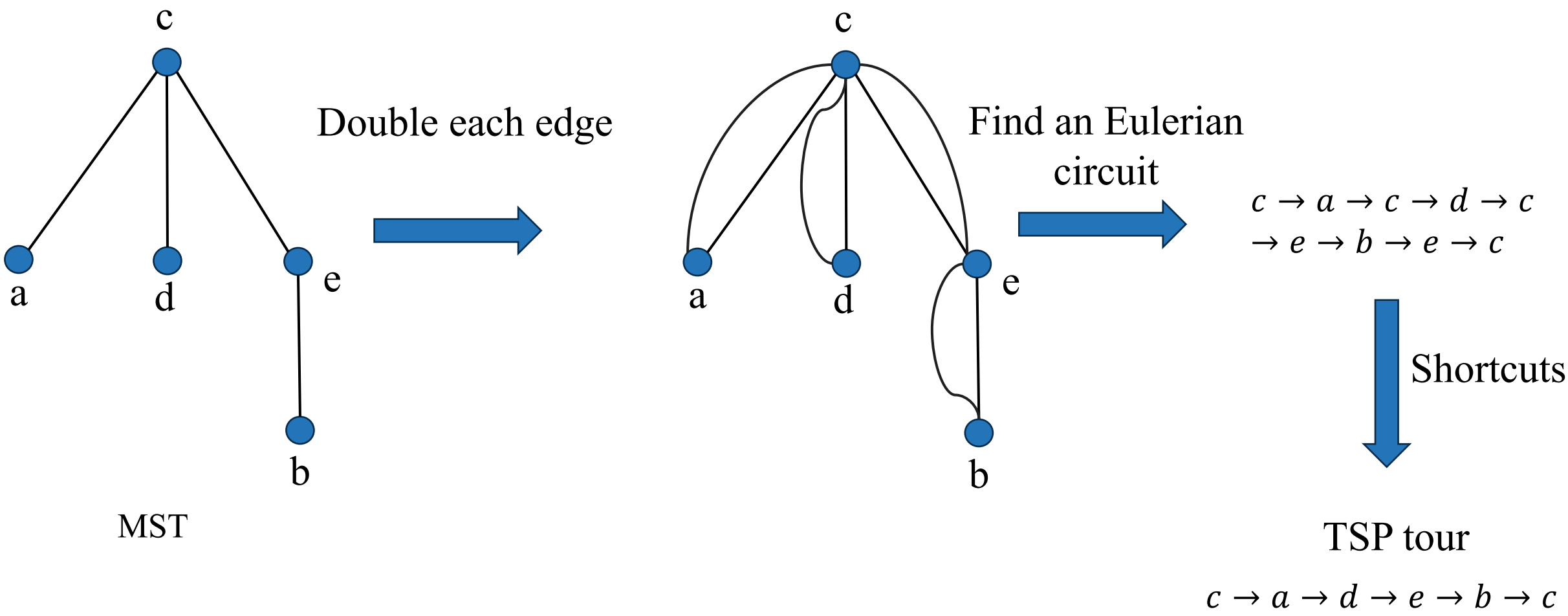
Motivation: Can we do better than a factor of 2?

The Double-Tree algorithm's 2-approximation comes from the step where we **double every edge of the MST to create an Eulerian circuit (欧拉回路).**

Definition of Eulerian Circuit (欧拉回路): A trail (迹,可以重复访问顶点，但不能重复访问边) in a graph that visits every edge exactly once and starts and ends at the same vertex.

By using shortcuts, an Eulerian Circuit on the doubled MST can be transformed into a TSP tour on the complete graph.

Illustration



The Christofides Algorithm

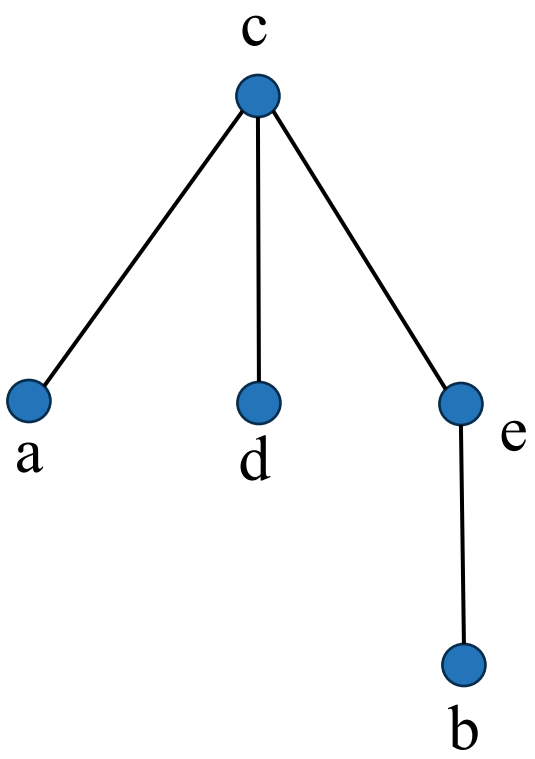
Existence Condition (according to graph theory): A connected **multigraph (多重图, 允许任意两个顶点之间存在多条边)** has an Eulerian circuit if and only if **every vertex has an even (偶数) degree**.

Once the condition is met, an Eulerian circuit can be found efficiently **in linear time using Hierholzer's algorithm**.

In an MST, some vertices will have an odd (奇数) degree. The Double-Tree algorithm “fixes” this by doubling everything, making all degrees even.

Key question: Can we do better?

Illustration



MST

?

Find an Eulerian circuit

An Eulerian circuit

Shortcuts

TSP tour

The Christofides Algorithm

The Christofides algorithm makes every vertex have an even degree smartly.

The Strategy:

1) Start with the MST, just like before.

2) Identify the set *S* of all vertices that have an odd degree in the MST. (It's a graph theory fact that there must be an **even number** of such vertices).

3) Find **a minimum-cost way to pair up these odd-degree vertices by adding new edges**. This is precisely the Minimum-Weight Perfect Matching problem on the set *S*.

4) Combine the MST edges and the new matching edges. In the resulting multigraph, every vertex now has an even degree.

5) Find the Eulerian circuit on this new, cheaper graph and use shortcuts to get the final tour.

Minimum-Weight Perfect Matching

Definition of Perfect Matching: Given an even number of vertices, a perfect matching is a set of edges where no two edges share a vertex and every vertex is an endpoint of exactly one edge.

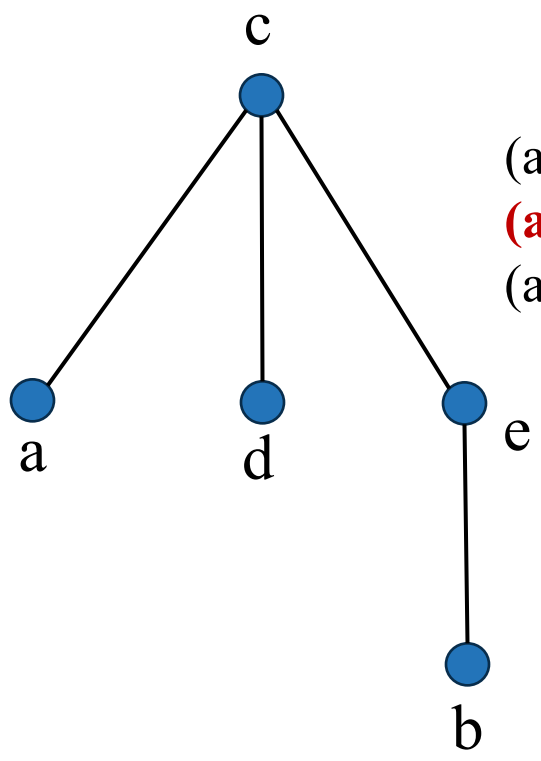
Minimum-Weight Perfect Matching (MWPM): Given costs on the edges, find a perfect matching with the **minimum possible total cost**.

The Blossom Algorithm: The MWPM problem can be solved in polynomial time (typically $O(n^3)$).

The Christofides Algorithm: Pseudocode

```
Algorithm Christofides_TSP(vertices V, Cost_function c):
    // 1. Compute the Minimum Spanning Tree T of the graph.
    T = Compute_MST(V, c)

    // 2. Find the set S of all vertices with an odd degree in T.
    S = Find_Odd_Degree_Vertices(T)

    // 3. Find a Minimum-Weight Perfect Matching M on S
    M = Min_Weight_Perfect_Matching(S, c)

    // 4. Combine the MST and the Matching to form a new multigraph H.
    H = T U M // H's edges are the union of edges in T and M

    // 5. Find an Eulerian circuit in H (guaranteed to exist).
    Euler_Circuit = Find_Eulerian_Circuit(H)

    // 6. Convert the circuit to a tour by taking shortcuts.
    Tour = Create_Tour_from_Circuit_with_Shortcuts(Euler_Circuit)
    return Tour
```

Example

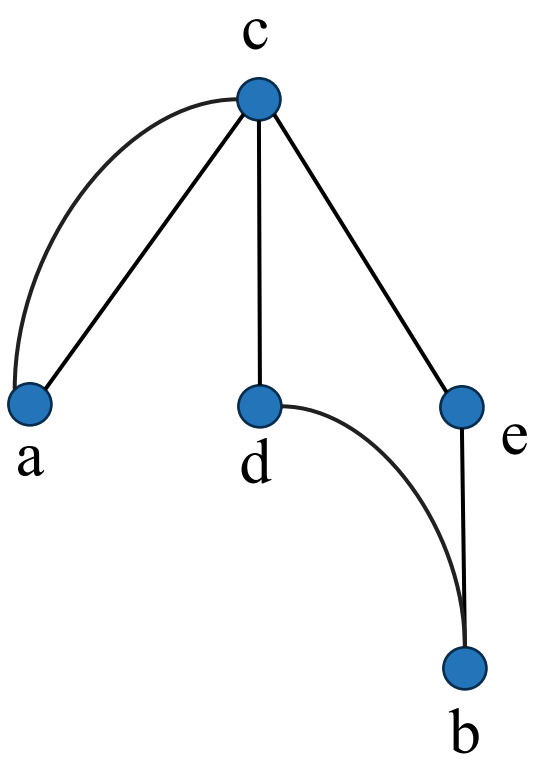


MST

(a,b) + (c,d) = 6+3 = 9

(a,c) + (b,d) = 1+4 = 5

(a,d) + (b,c) = 4+5 = 9



The Christofides Algorithm: Complexity Analysis

Let n be the number of vertices.

Step-by-Step Breakdown:

1) Compute MST: Using Prim's algorithm on a dense graph takes $O(n^2)$.

2) Find Odd-Degree Vertices: $O(n)$.

3) Compute MWPM: This is the bottleneck. The Blossom algorithm runs in $O(n^3)$.

4) Find Eulerian Circuit & Shortcut: $O(m+n) = O(n)$. (m is #edges)

Overall Complexity: $O(n^3)$. It is slower than the Double-Tree algorithm but provides a significantly better approximation guarantee.

Christofides Algorithm: Approximation Ratio

Theorem: Christofides algorithm is a **1.5-approximation algorithm** for the Metric TSP.

We first define some notations before proving the theorem:

- T_{OPT}: An optimal TSP tour, with cost $C(OPT)$.
- MST: The Minimum Spanning Tree, with cost $C(MST)$.
- S: The set of odd-degree vertices in the MST.
- M: The Minimum-Weight Perfect Matching on S. $C(M)$ is its cost.
- H: The multigraph formed by the union of edges in MST and M.
- E_C: The Eulerian circuit found in H. $C(E_C)$ is its cost.
- T: The final tour produced by the algorithm. $C(T)$ is its cost.

Proof of the Approximation Ratio (1)

Step 1: Bounding the MST Cost

This step is identical to the proof for the Double-Tree algorithm.

$$C(MST) \leq C(OPT)$$

Proof of the Approximation Ratio (2)

Step 2: Bounding the Matching Cost (The key step)

1) Consider the set S of odd-degree vertices from the MST. Consider the optimal tour T_{OPT}. Now, create a new tour, T'_{OPT}, that visits only the odd-degree vertices in S, following the same order they appear in T_{OPT}.

2) By the triangle inequality, the cost of T'_{OPT} cannot be more than the cost of the full optimal tour: $C(T'_{OPT}) \leq C(T_{OPT})$.

3) T'_{OPT} is a cycle on an even number ($|S|$) of vertices. We can decompose this cycle into two disjoint perfect matchings, M_1 and M_2, by taking alternating edges of the cycle. The sum of their costs is the cost of the tour: $C(M_1) + C(M_2) = C(T'_{OPT})$.

Proof of the Approximation Ratio (3)

4) By the pigeonhole principle, at least one of these matchings must have a cost no more than half the total: $\min\{C(M_1), C(M_2)\} \leq \frac{1}{2} C(T'_{OPT})$.

5) The algorithm finds M, the Minimum-Weight Perfect Matching on S. Its cost must be less than or equal to the cost of any other perfect matching on S. Therefore:

$$C(M) \leq \min\{C(M_1), C(M_2)\} \leq \frac{1}{2} C(T'_{OPT}) \leq \frac{1}{2} C(T_{OPT})$$

Proof of the Approximation Ratio (4)

Step 3: Combining to Get the 1.5 Ratio

1) The final tour T is created by shortcutting an Eulerian circuit E_C on the graph $H = MST \cup M$. By definition, the cost of the Eulerian circuit is exactly the sum of its edge costs. Due to the triangle inequality, the cost of the final tour is no more than the cost of the circuit.

$$C(T) \leq C(E_C) = C(MST) + C(M)$$

2) Now, we substitute the bounds into this expression:

$$C(T) \leq C(T_{OPT}) + \frac{1}{2}C(T_{OPT}) = 1.5 \cdot C(T_{OPT})$$

It is proved the algorithm's tour cost is at most 1.5 times the optimal tour's cost.

Nearest Neighbor (NN) Heuristic

NN is a heuristic algorithm, but **without constant approximation ratio** (see assignment 1).

Core Idea: A simple, intuitive, and fast greedy strategy.

Algorithmic Steps:

1) Start at an arbitrary vertex.
2) Repeatedly move to the **nearest unvisited** vertex.
3) When all vertices have been visited, return to the starting vertex.

This algorithm is widely used and easy to implement, but its “greedy” nature can lead it into traps.

Nearest Neighbor (NN): Pseudocode

```
Algorithm Nearest_Neighbor(vertices V, Cost_function c, start_vertex):
    Tour = [start_vertex]
    unvisited = V \ {start_vertex}

    current_vertex = start_vertex
    while unvisited is not empty:
        // Find the nearest unvisited vertex
        nearest_vertex = find_min_cost_vertex(unvisited, current_vertex, c)

        Tour.append(nearest_vertex)
        unvisited.remove(nearest_vertex)
        current_vertex = nearest_vertex

// Return to start
  Tour.append(start_vertex)

  return Tour
```

Nearest Neighbor (NN): Complexity Analysis

Let n be the number of vertices.

Step-by-Step Breakdown:

1) The main while loop runs $n - 1$ times.

2) Inside the loop, the find_min_cost_vertex step must scan all remaining unvisited vertices. In the worst case, this is an $O(n)$ scan.

Overall Complexity: $O(n^2)$. This is very fast.

Cheapest Insertion (CI) Heuristic

CI is a heuristic algorithm, **and an approximation algorithm** for the metric TSP.

Core Idea: Start with a small, trivial sub-tour and iteratively "grow" it. In each step, **find the unvisited vertex that is "cheapest" to insert into the existing sub-tour.**

What does "cheapest" mean?

- To insert a vertex *k* into an edge *(i, j)* of the current sub-tour, we remove edge *(i, j)* and add edges *(i, k)* and *(k, j)*.
- The insertion cost is: $\text{Cost}(i,k,j) = c(i,k) + c(k,j) - c(i,j)$.
- The algorithm finds the combination of k and (i,j) that minimizes this insertion cost.

Cheapest Insertion (CI): Pseudocode

```
Algorithm Cheapest_Insertion(vertices V, Cost_function c):
    // 1. Start with a sub-tour containing two vertices
    T = (v1, v2, v1)
    unvisited = V \ {v1, v2}

    while unvisited is not empty:
        // 2. Find the cheapest insertion
        find k in unvisited and (i, j) in Tour T that minimizes insertion_cost = c(i, k) + c(k, j) - c(i, j)

        // 3. Insert k into the tour
        insert k into T between i and j
        unvisited.remove(k)

  return T
```

Cheapest Insertion (CI): Complexity Analysis

Let n be the number of vertices.

Step-by-Step Breakdown:

1) The main while loop runs $O(n)$ times (once for each vertex to be inserted).

2) Inside the loop, we must find the best vertex k to insert ($O(n)$ choices) and the best edge (i, j) to insert it into ($O(n)$ choices). Finding the minimum cost takes $O(n^2)$ time in each iteration.

Overall Complexity: $O(n) \cdot O(n^2) = O(n^3)$.

Cheapest Insertion (CI): Approximation Ratio

Theorem: The CI algorithm is a **2-approximation algorithm** for Metric TSP.

We first define some notations before proving the theorem:

- T_{OPT}: An optimal TSP tour, with cost $C(OPT)$.
- $C(T_k)$: the cost of the sub-tour after k cities are in the tour.
- $C(MST)$: the cost of the MST on all n cities.
- $C(T)$: the cost of the tour produced by the CI algorithm

Proof Sketch of the Approximation Ratio (1)

Goal: To show that the cost of the tour generated by Cheapest Insertion ($C(T)$) is at most twice the cost of the optimal tour ($C(T_{OPT})$).

Key Intermediate Structure: The Minimum Spanning Tree (MST). We know that $C(MST) \leq C(T_{OPT})$.

Proof Strategy: We will show that $C(T) \leq 2 \cdot C(MST)$. Combining these gives the desired result.

How the Algorithm Builds Cost:

- The algorithm starts with an initial edge (cost C_{init}).
- In each step k (from $k = 3$ to n), it inserts the vertex v_k that requires the *minimum increase* in tour cost. Let this minimum insertion cost be Δ_k.
- The total cost is $C(T) = C_{init} + \sum_{k=3}^{n} \Delta_k$.

Proof Sketch of the Approximation Ratio (2)

Bounding the Insertion Cost (The Core Lemma Idea):

- Consider the step where we insert vertex v_k. Let v_a be the vertex already *in* the current partial tour (T_{k-1}) that is *closest* to v_k. The cost is $d(v_k, T_{k-1}) = c(v_a, v_k)$.
- Let v_b be a neighbor of v_a in the tour T_{k-1}.
- We *could* insert v_k between v_a and v_b. The cost increase would be $\Delta' = c(v_a, v_k) + c(v_k, v_b) - c(v_a, v_b)$.
- Using the triangle inequality ($c(v_k, v_b) \leq c(v_k, v_a) + c(v_a, v_b)$), we can show that this specific insertion cost Δ' is at most $2 \cdot c(v_a, v_k)$.
  - $\Delta' \leq c(v_a, v_k) + (c(v_k, v_a) + c(v_a, v_b)) - c(v_a, v_b) = 2 \cdot c(v_a, v_k)$.
- Since the algorithm chooses the *cheapest possible* insertion Δ_k, it must be that $\Delta_k \leq \Delta'$.
- **Key Lemma Result:** $\Delta_k \leq 2 \cdot c(v_a, v_k) = 2 \cdot d(v_k, T_{k-1})$.
  - (The cost of the cheapest insertion is no more than twice the cost of simply connecting the new vertex to its nearest neighbor in the current tour).

Proof Sketch of the Approximation Ratio (3)

Summing the Bounds:

- From the previous slide, we have $\Delta_k \leq 2 \cdot d(v_k, T_{k-1})$, where $d(v_k, T_{k-1})$ is the cost of the cheapest edge connecting v_k to the existing partial tour T_{k-1}.
- The total cost of the algorithm's tour is:

$$C(T) = C_{init} + \sum_{k=3}^{n} \Delta_k \leq C_{init} + \sum_{k=3}^{n} 2 \cdot d(v_k, T_{k-1})$$

$$C(T) \leq 2 \left(\frac{C_{init}}{2} + \sum_{k=3}^{n} d(v_k, T_{k-1}) \right)$$

Proof Sketch of the Approximation Ratio (4)

Connecting the Sum to the MST (The Crucial Insight - High Level):

- Let $S = \frac{C_{init}}{2} + \sum_{k=3}^{n} d(v_k, T_{k-1})$. This sum represents the cost accumulated by connecting each new vertex to the existing partial tour via the cheapest available edge *at that moment*.
- The structure formed by these "cheapest connection" edges is indeed a spanning tree.
- **A more advanced argument**, often involving **comparing the sequence of edges added by the insertion process to the edges added by Prim's algorithm for constructing an MST**, demonstrates that the sum of these cheapest connection costs (`S`) is, in fact, bounded by the cost of the Minimum Spanning Tree ($C(MST)$) on all `n` vertices.
- This rigorous comparison yields the crucial inequality:

$$S \leq C(MST)$$

Proof Sketch of the Approximation Ratio (5)

Final Result:

- Substitute this inequality back into the cost bound for $C(T)$:

$$C(T) \leq 2 \cdot S \leq 2 \cdot C(MST)$$

- And since we know from previous proofs that $C(MST) \leq C(T_{OPT})$:

$$C(T) \leq 2 \cdot C(T_{OPT})$$

- This correctly outlines why the algorithm achieves a 2-approximation ratio, linking the sum of insertion costs (bounded via a comparison to Prim's algorithm) to the cost of the MST.

Summary

Algorithm	Core Idea	Time Complexity	Approx. Ratio	Remark
Double-Tree	Based on MST, create a "doubled-edge" walk, then shortcut	$O(n^2)$	2	- Simple, guaranteed. - Often outperformed by heuristics
Christofides	Based on MST, fix odd degrees using Min-Weight Perfect Matching, find Eulerian tour, shortcut.	$O(n^3)$	1.5	- Best known approx. ratio. - More complex (needs matching). Use when guarantee is crucial.
NN	Always go to the closest unvisited vertex.	$O(n^2)$	None	- Very fast, very simple. - No quality guarantee, can be very bad.
CI	Start small tour, repeatedly insert the cheapest unvisited vertex.	$O(n^3)$	2	- Often better than NN. - Conceptually simpler. - Good constructive heuristic.

Chinese Postman Problem (CPP)

A Shift in Perspective:

- Hamiltonian Cycle (TSP): Visit every vertex exactly once. (NP-hard).
- Eulerian Circuit: Visit every edge exactly once.

This second problem is the basis for tasks like:

- Snow plowing (clearing every street).
- Mail delivery (walking every block in a neighborhood).
- Road-seam inspection (checking every meter of highway).

This problem is famously known as the Chinese Postman Problem (CPP).

The CPP: Definition

Input: A connected, undirected graph $G = (V, E)$ with edge cost function c.

Output: A tour starting and ending at the same vertex that **traverses every edge in *E* at least once**.

Objective: Find a tour with the minimum total cost.

Key Insight:

- If the graph has an Eulerian circuit (i.e., all vertices have an even degree), the optimal solution is the Eulerian circuit. The cost is the sum of all edge costs.
- If a graph has odd-degree vertices, it means we must re-traverse some edges to make the total "walk" on each vertex an even degree.
- The problem is equivalent to: "Which edges should we traverse twice to make all vertex degrees even, at a minimum total cost?"

The exact algorithm for CPP (1)

Core Idea: The only "problematic" vertices are those with an odd degree. An edge must be added (re-traversed) for each. To solve the problem, we must find the cheapest way to "pair up" all the odd-degree vertices.

The Strategy:

1) Identify the set S of all vertices with an odd degree. (The number of such vertices is always even). Compute all-pairs shortest paths between the vertices in S.

2) Find a Minimum-Weight Perfect Matching (MWPM) on the set S, using the shortest path distances (costs) as the edge costs.

The exact algorithm for CPP (2)

2) Create a new multigraph H by adding the edges from the matching M to the original graph G. (Adding an edge from the matching means we plan to traverse that path in G an extra time).

3) The new graph H is guaranteed to have all-even-degree vertices. Find the Eulerian circuit on H. This circuit is the optimal CPP tour.

CPP Algorithm: Pseudocode

```
Algorithm Solve_CPP(Graph G=(V, E), Cost function c):
    // 1. Find all odd-degree vertices
    S = Find_Odd_Degree_Vertices(G)

    // 2. Compute all-pairs shortest paths (APSP) for all vertices in S
    Distances = APSP_on_G(S)

    // 3. Find the minimum-weight perfect matching M on S
    M = Min_Weight_Perfect_Matching(S, Distances)

    // 4. Create the final multigraph H by adding the matching paths
    H = G U M // Add the paths from M as new edges

    // 5. Find the Eulerian circuit on H
    Tour = Find_Eulerian_Circuit(H)

  return Tour
```

CPP Algorithm: Complexity Analysis (n=|V|, m=|E|)

Step-by-step breakdown:

1) Find S: $O(m)$.

2) APSP: Using Johnson's algorithm: $O(n(m + n \log n))$.

3) MWPM: Using the Blossom algorithm on $k = |S|$ vertices (where $k \leq n$).

Complexity: $O(n^3)$. This is the bottleneck.

4) Find Eulerian circuit: $O(m + n)$.

Overall Complexity: $O(n^3)$, dominated by the matching step.

From TSP to Routing

We now consider a generalized problem of TSP: the **Capacitated Vehicle Routing Problem (CVRP)**.

This is arguably one of the most commercially important combinatorial optimization problem, forming the core of all logistics and delivery operations (JD, CAINIAO, Meituan, Amazon, etc.).

Capacitated Vehicle Routing Problem (CVRP)

CVRP is a **generalization of TSP**.

- TSP: 1 vehicle, infinite capacity.
- CVRP: m vehicles, finite capacity Q.

CVRP is also **a generalization of Bin Packing**.

- Bin Packing: How to partition items into minimum bins?
- CVRP: How to partition customers (items) into minimum-cost routes (bins), where the cost of a "bin" (route) is not fixed, but is the cost of the TSP tour on the items inside it.

Capacitated Vehicle Routing Problem (CVRP)

Input:

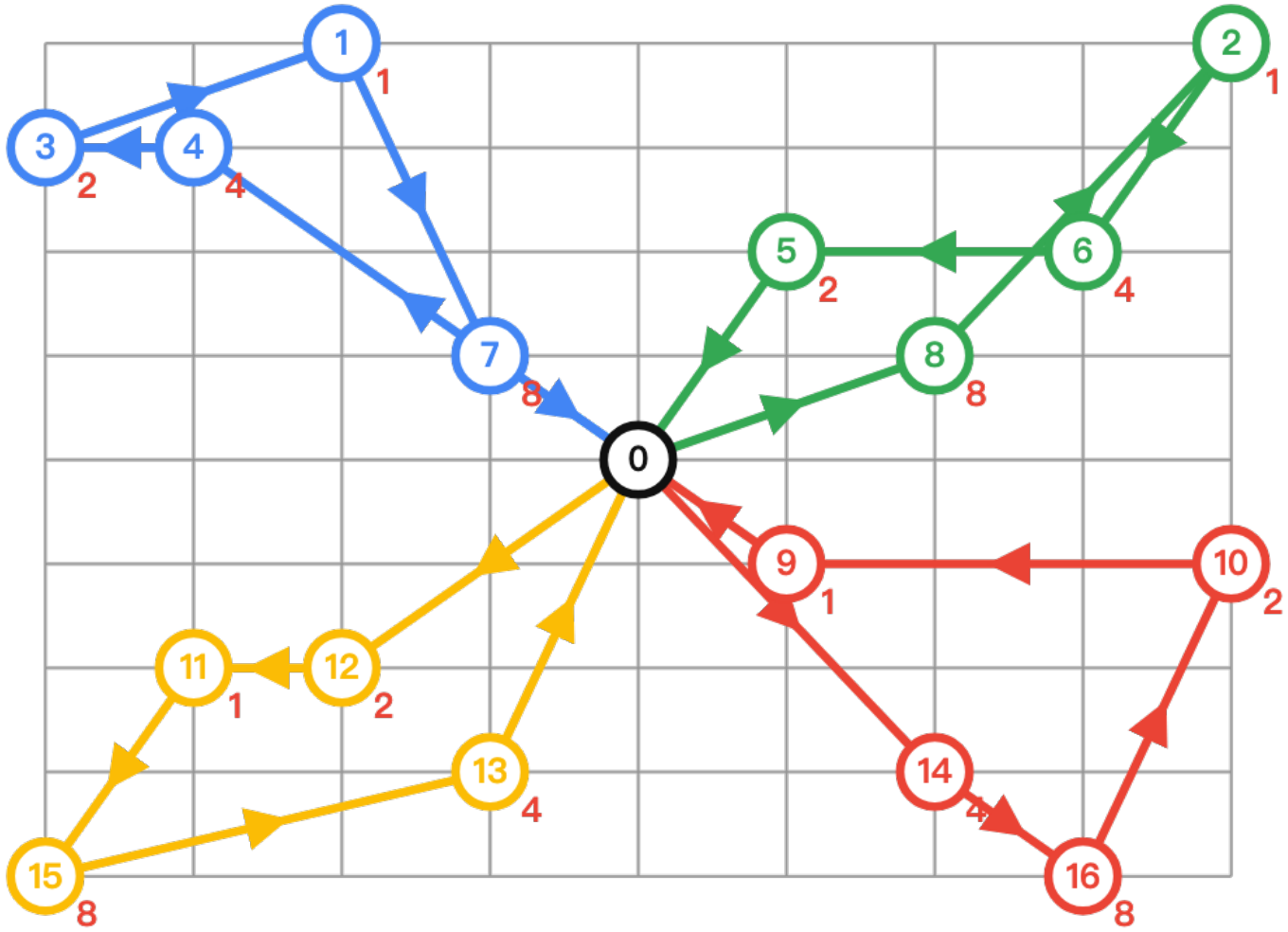
- A central depot (vertex 0).
- A set of n customers $V = \{1, \dots, n\}$.
- A demand d_i for each customer i.
- A fleet of m identical vehicles, each with capacity Q.
- A cost matrix c_{ij}.

Output: A set of vehicle routes (#routes $\leq m$). Each route must:

- Start and end at the depot 0.
- The sum of demands of all customers on a single route must not exceed Q.
- Every customer must be visited exactly once by exactly one vehicle.

Objective: Minimize the total cost (e.g., total distance) of all routes.

An Example of CVRP



CVRP: Heuristics vs. Approximation

CVRP is NP-hard (**why**?). We need algorithms to find good solutions.

Heuristics (e.g., Clarke-Wright Savings):

- These are fast, effective algorithms that are used widely in practice.
- They follow greedy rules that make "common sense" merges.
- **Warning**: Most simple heuristics for CVRP do not have a constant approximation ratio. Their worst-case performance can be arbitrarily bad, even if their average performance is excellent.

Approximation Algorithms:

- These are more complex algorithms, often based on LP rounding or "cluster-first, route-second" frameworks.

CVRP Heuristic: The Clarke-Wright Savings

Core Idea: Start with the most expensive possible solution (one vehicle per customer) and iteratively merge routes that provide the largest "savings".

The "Saving" Calculation:

- Two routes: Depot → i → Depot and Depot → j → Depot.
- Total Cost: $(c_{0i} + c_{i0}) + (c_{0j} + c_{j0})$
- Merged State: One route: Depot → i → j → Depot.
- New Cost: $(c_{0i} + c_{ij} + c_{j0})$
- Saving: $(c_{i0} + c_{0j}) - c_{ij}$

The algorithm greedily merges the pair (i, j) with the highest saving, as long as the merged route does not violate the capacity Q.

CW: Pseudocode

```
Algorithm Clarke_Wright(Customers, Depot, Capacity Q):
    // 1. Calculate savings s_ij for all pairs (i, j)
    Savings_List = calculate_all_savings(s_ij)

    // 2. Sort the savings list in descending order
    sort Savings_List from largest to smallest

    // 3. Initialize: one route per customer: R_i = {0 -> i -> 0}
    Routes = { R_1, R_2, ..., R_n }

    // 4. Iteratively merge routes
    for each saving s_ij in the sorted Savings_List:
        let i and j be the customers.
        Find the routes R_i and R_j that contain i and j.
        if (R_i != R_j) AND (i and j are “exterior” points) AND (Demand(R_i) + Demand(R_j) <= Q):
            // Merge the two routes into one new route
            Merge(R_i, R_j)

    return Routes
```

CW: Complexity Analysis and Approximation Ratio

Complexity dominated by sorting the $O(n^2)$ savings: $O(n^2 \log n)$

Approximation Ratio: None for metric CVRP. Its worst-case performance can be arbitrarily bad.

The Vast World of VRP Variants

The standard CVRP is just the beginning. Researchers and companies have defined dozens of variants to match real-world operational constraints.

One of the most common extensions:

- Adding **Time Windows** (a scheduling component).

CVRP with Time Windows (VRPTW)

New Inputs:

- Each customer `i` has a **service time** s_i (how long it takes to unload).
- Each customer `i` has a **time window** $[e_i, l_i]$, where e_i is the earliest allowed arrival time and l_i is the latest.

New Constraints:

- A vehicle *can* arrive before e_i, but it must wait until e_i to begin service.
- A vehicle *must* arrive before l_i.

Difference from CVRP:

- This problem is no longer a pure routing problem; it is a **combined routing and scheduling** problem.
- The cost of a route is not just its distance, but also its feasibility with respect to time. A route that is short in distance might be infeasible if it arrives at a customer too late.

Other VRP Variants (Briefly)

Multi-Depot VRP (MDVRP):

- Vehicles are based at multiple depots.
- **Difference**: Adds a strategic decision of assigning customers to depots.

Stochastic VRP (SVRP):

- Customer demands, locations, or travel times are uncertain and modeled as random variables.
- **Difference**: Design routes that are “robust” or have the lowest expected cost.

VRP with Backhauls (VRPB):

- Routes are divided into two parts: “outbound” deliveries and “inbound” pickups, which must be done in that order.

CO Problems Studied so far

Problem	Core Decision / Characteristic	Typical Application Areas
Vertex Cover (VC)	Select minimum **vertices** to cover all **edges**. (NP-hard)	Network monitoring, security camera placement.
Set Cover (SC)	Select minimum **sets** to cover all **elements** in a universe. (Generalization of VC). (NP-hard)	Facility location, resource allocation (e.g., crew scheduling), feature selection.
Traveling Salesman Problem (TSP)	Find minimum cost **tour (sequence)** visiting each **vertex** exactly once. (NP-hard)	Basic route planning, manufacturing (e.g., circuit board drilling), genome sequencing.
Chinese Postman Problem (CPP)	Find minimum cost **tour** visiting each **edge** at least once. (**Polynomial-time solvable**)	Mail delivery, snow plowing, road inspection/maintenance, garbage collection routes, utility meter reading.
Capacitated Vehicle Routing Problem (CVRP)	Partition **vertices (customers)** into minimum cost **routes** for multiple **vehicles**, respecting **capacity**. (Generalization of TSP). (NP-hard)	Logistics & delivery (goods, packages, food), waste collection, bus routing