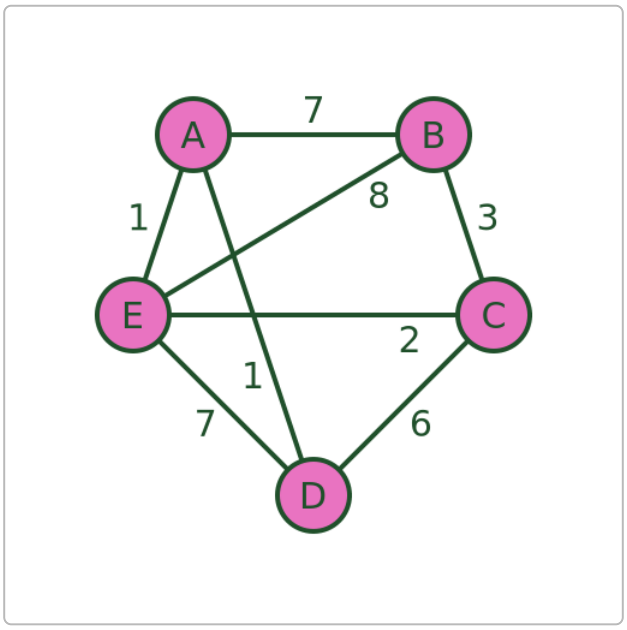
ChatGPT

The Traveling Salesman Problem (TSP): Theory, Algorithms, and Applications

The **Traveling Salesman Problem (TSP)** asks: given a list of *n* cities and the distances (or costs) between each pair, find the shortest possible route that visits each city exactly once and returns to the starting city. In graph-theoretic terms, we seek a minimum-cost Hamiltonian cycle in a weighted complete graph. A standard integer linear programming (ILP) formulation introduces binary variables $x_{ij}=1$ if the tour goes from city *i* to *j*, and 0 otherwise, and minimizes the total cost $\sum_{i,j} c_{ij}x_{ij}$. To ensure each city is entered and left exactly once, we enforce for each city *j*: [\sum_{i\neq j} x_{ij} = 1, \quad \sum_{k\neq j} x_{jk} = 1.] These constraints mean every city has exactly one incoming and one outgoing edge [1] [2]. An example graph with 5 cities (A–E) and edge weights is shown below:



Example: A small TSP instance on 5 nodes with weighted edges (cities labeled A–E). The goal is to find the minimum-weight tour visiting all nodes exactly once and returning to the start.

Additional **subtour-elimination** constraints are needed to prevent disconnected cycles; there are exponentially many such constraints (on the order of 2^n) [3]. Altogether, the ILP is: [\text{minimize} \;\sum_{i,j} c_{ij}x_{ij}, \quad x_{ij}\in{0,1},] subject to the “enter/leave once” constraints above and subtour-elimination constraints [1] [2].

Because of this combinatorial complexity, the TSP is *NP-hard* (the decision version is NP-complete) [4]. The naive brute-force approach checks all $(n-1)!$ tours in $O(n!)$ time [5] [6], which is only feasible for very small *n*. Even smarter exact methods (like the Held–Karp dynamic programming) take $O(n^2 2^n)$ time [7]. In practice, one often uses branch-and-bound or branch-and-cut methods (solving LP relaxations and adding cuts) to prune the search space. However, in the worst case the complexity remains exponential, since there are about $(n-1)!$ possible tours [3].

Convex Geometry Applications

In the **Euclidean TSP** (points in the plane with Euclidean distances), geometry provides some useful properties. For example, if all cities lie on the convex hull of the point set, they will appear in the optimal tour in the same cyclic order as on the hull [8]. In fact, one heuristic is to first compute the convex hull of the points and use it as a backbone tour, then insert interior points greedily. The figure below illustrates the convex hull (green) and interior points (gray); the optimal tour visits hull points in hull order.

(Image: Illustration of a convex hull on random points (not included).)

More generally, the **TSP polytope** is the convex hull of all incidence vectors of tours. Optimization over this polytope (finding cutting planes that separate a fractional solution from the hull) was pioneered by Dantzig, Fulkerson and Johnson [9] [10]. In fact, solving the LP relaxation of the ILP with cuts is a classic example of leveraging convex relaxations: each LP solution is an extreme point of a convex polytope, and if it is not integral (not a valid tour), one finds a hyperplane (cut) that separates it [11]. However, because the subtour-elimination facets grow exponentially, the LP relaxation alone does not make TSP easy; it still requires searching or generating cuts to reach integrality.

By contrast, **convex optimization** typically deals with continuous variables and convex feasible sets, guaranteeing global optimality and polynomial-time solvability. The TSP's feasible region (Hamiltonian tours) is **non-convex** and discrete. Nonetheless, convex optimization ideas are used via relaxations: e.g. one can formulate a linear relaxation or even semidefinite relaxations of TSP. These provide lower bounds (like the Held–Karp bound via Lagrangian relaxation) but do not directly yield integer tours.

Algorithms for TSP

A wide range of algorithms have been developed:

- **Brute-force enumeration:** Try all possible permutations of cities (or $(n-1)!/2$ distinct tours for undirected TSP). This guarantees the optimum but is $O(n!)$ time [5] [6] and only works for very small *n*.

- **Dynamic Programming (Held–Karp):** Uses subsets and bitmask DP. With $O(2^n n^2)$ time and memory, it is much faster than brute force for moderate *n* [7], but still exponential. It is often used to compute exact solutions or strong lower bounds for moderate-size instances.

- **Branch-and-Bound / Branch-and-Cut:** Systematically explores a search tree of partial tours, using lower bounds (e.g. from linear programming relaxations) to prune. Modern exact solvers (such as *Concorde*) use advanced branch-and-cut, adding cutting planes (subtour elimination cuts, comb cuts, etc.) dynamically to improve bounds. These methods solved TSP instances with tens of thousands of cities in practice.

- **Greedy Heuristics:** Simple quick algorithms that build a tour. Examples include:

- *Nearest Neighbor:* Start at a city and repeatedly travel to the nearest unvisited city. This takes $O(n^2)$ time. It is fast but not optimal; typically the resulting tour is about 25% longer than optimal on average [12].

- *Cheapest Insertion / Best Insertion:* Start with a small subtour (e.g. two cities) and repeatedly insert the next city that causes the least increase in tour length. Runs in $O(n^2)$ and often gives better tours than nearest-neighbor.

- **Local Search and Improvement:** Given any initial tour, iteratively improve it by local moves. Common moves:

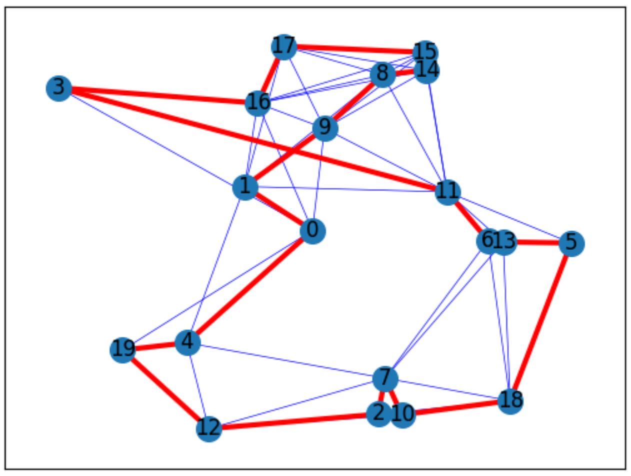
- *2-opt:* Remove two edges and reconnect the two paths in the other way. Repeat until no improvement. Usually yields a tour within ~5% of optimal [13].
- *3-opt:* Generalizes 2-opt by removing three edges and reordering. Often improves further (~3% above optimal on average [13]).

- *Lin–Kernighan:* A sophisticated variable-k opt heuristic, still one of the most effective heuristics for large TSPs (often used within Concorde).

- **Approximation Algorithms:** For metric TSP (distances satisfy the triangle inequality), Christofides' algorithm (1976) finds a tour at most 1.5 times the optimal in polynomial time. It works by computing an MST, finding a minimum-weight matching on odd-degree nodes, and shortcutting. Christofides' algorithm remains the best known approximation ratio for metric TSP [14].

- **Metaheuristics:** These probabilistic algorithms often perform well in practice:

- *Genetic Algorithms (GA):* Maintain a population of tours, combine them (crossover) and mutate to evolve better tours. No worst-case guarantees, but GAs can find good solutions for large instances.
- *Simulated Annealing (SA):* Starts with a tour and makes random local changes; sometimes accepts worse moves according to a temperature schedule. Over time the "temperature" cools. SA can escape local optima and often finds near-optimal tours.
- *Ant Colony Optimization (ACO):* Simulate many "ants" building tours based on pheromone trails that favor shorter edges. ACO was originally demonstrated on TSP and is effective for various routing problems.
- *Others:* Tabu Search, Particle Swarm, etc., have also been applied to TSP.



Example: A sample solution (red path) on 20 random points found by an approximation algorithm (here using NetworkX's Christofides implementation [15]). The red route visits every node once. Such visualizations help compare algorithms: one can plot the cities and the tour connecting them.

In summary, exact algorithms give optimal solutions (with exponential worst-case time), while heuristics/metaheuristics run in polynomial time and produce good (but not guaranteed optimal) tours. For instance, a recent source notes that standard heuristics like nearest-neighbor give tours "within 15–25% of optimal," and local k-opt improvements reduce the gap to around 3–5% [12] [13]. The famous 1.5-approximation of Christofides is the **best-known bound** for metric TSP [14].

Algorithmic Inspirations

The TSP has been a driving force in optimization and algorithm design [10]. It motivated the development of **cutting-plane methods** and branch-and-bound in integer programming: Dantzig, Fulkerson and Johnson's 1954 breakthrough used LP relaxations and cuts to solve a 49-city TSP [9]. Over decades, researchers have built on these ideas (as in the Concorde TSP solver) to solve very large instances. TSP also inspired classic heuristic ideas: local search (2-opt, Lin–Kernighan), and metaheuristics like simulated annealing (first introduced on TSP examples) and genetic algorithms (one of GA's early benchmarks was TSP).

As one recent survey notes, **"the TSP is the most popular and most studied combinatorial problem"** and it "has driven the discovery of several optimization techniques such as cutting planes, branch-and-bound, local search, Lagrangian relaxation, and simulated annealing" [10]. More recently, TSP has been used to test new approaches like constraint programming, integer programming solvers, and even machine learning (graph neural networks or transformer models trained to predict good tours [10]). Thus, studying TSP has had broad impact: from theoretical advances (P vs NP theory) to practical optimization frameworks used in many fields.

Real-World Applications

Though abstract, TSP models many real scenarios. In *logistics* and *route planning*, a single-vehicle delivery can be viewed as a TSP: find the shortest route through all required locations. Extensions with multiple vehicles lead to the Vehicle Routing Problem (VRP). In *manufacturing* and *circuit design*, TSP appears in tasks like **drilling printed circuit boards**: a drill must visit many hole positions once. Optimizing the order of holes (for each drill bit size) is a TSP [16]. Similarly, *wire routing* can be modeled by TSP: connecting pins with minimal total wire length [17].

In scheduling, TSP captures sequence-dependent tasks. For example, single-machine *job sequencing* with setup times between jobs is equivalent to finding a minimum-tour through jobs (nodes are jobs, edge costs are setup times) [18]. In *science*, TSP-like scheduling arises in experiments: e.g. determining the order to make successive measurements or telescope pointings to minimize movement [19]. In *genomics*, the **Shortest Common Superstring** problem (an NP-hard model of DNA assembly) can be mapped to TSP: overlapping reads form a graph, and finding a shortest common superstring is like finding a shortest Hamiltonian path [20].

Other examples include planning a museum or city tour, robotic arm path optimization, and even image processing (e.g. *TSP art* draws images with one continuous path). In all these cases, approximate or heuristic TSP algorithms provide practical solutions. As noted, the TSP has "a diverse range of applications, in fields including mathematics, computer science, genetics, and engineering" [21], reflecting its wide relevance.

Example Python Implementations

Below are two illustrative Python code samples for solving TSP-like problems on small point sets. Visualization of the resulting tours can be done using, for example, Matplotlib (not shown here).

Nearest Neighbor Heuristic

This greedy heuristic builds a tour by always going to the nearest unvisited city. It is simple but often yields a reasonably good tour quickly.

```python
import math

# Example set of points (x, y coordinates)
points = [(0,0), (1,5), (5,2), (6,6), (8,3)]
n = len(points)

# Function to compute Euclidean distance
def dist(a, b):
    return math.hypot(a[0]-b[0], a[1]-b[1])

# Nearest Neighbor tour starting from index 0
def nearest_neighbor(points, start=0):
    n = len(points)
    unvisited = set(range(n))
    tour = [start]
    current = start
    unvisited.remove(current)
    # Build tour by repeatedly visiting nearest unvisited city
    while unvisited:
        next_city = min(unvisited, key=lambda j: dist(points[current],
points[j]))
        unvisited.remove(next_city)
        tour.append(next_city)
        current = next_city
    return tour

tour = nearest_neighbor(points, start=0)
print("Nearest-Neighbor tour order:", tour)
print("Tour length:", sum(dist(points[tour[i]], points[tour[(i+1)%n]]) for i
in range(n)))
```

This code starts at city 0 and at each step moves to the closest unvisited city. The resulting tour is printed (as a sequence of point indices) along with its total length. For the small example above, you might get a tour like `[0, 1, 3, 4, 2]`. Plotting the points and connecting them in this order would show a closed path visiting each point once.

Simulated Annealing

Simulated annealing is a metaheuristic that randomly perturbs the tour and occasionally accepts worse moves to escape local optima. Here is a simple implementation:

```python
import random
import math

# Re-use points and dist() from above.

def tour_length(tour):
    return sum(dist(points[tour[i]], points[tour[(i+1)%n]]) for i in
range(n))

# Generate an initial tour (e.g., random)
current_tour = list(range(n))
random.shuffle(current_tour)
current_len = tour_length(current_tour)

T = 100.0  # initial “temperature”
alpha = 0.995  # cooling rate
min_T = 1e-3

while T > min_T:
    # Random 2-opt swap: swap two cities to generate a new tour
    i, j = sorted(random.sample(range(n), 2))
    new_tour = current_tour[:]
    new_tour[i:j] = reversed(new_tour[i:j])
    new_len = tour_length(new_tour)
    # Decide whether to accept
    if new_len < current_len or random.random() < math.exp((current_len -
new_len) / T):
        current_tour, current_len = new_tour, new_len
    # Cool down
    T *= alpha

print("Simulated Annealing tour order:", current_tour)
print("Tour length:", current_len)
```

This code starts with a random tour and repeatedly performs a 2-opt swap (reversing the segment between two random indices). If the new tour is shorter, it is accepted; otherwise it may still be accepted with probability $\exp(-(d_{\text{new}}-d_{\text{current}})/T)$. As $T$ cools, worse moves become unlikely. The final tour is printed. Running this yields a tour often near the optimum. (Visualizing it by plotting the tour path would show a smooth loop.)

These examples illustrate how TSP heuristics can be implemented. In practice, one might compare the tours from nearest-neighbor vs. simulated annealing to see which is shorter. Further improvements (e.g. applying 2-opt locally) could also be coded.

Additional Resources

- **Concorde TSP Solver:** A highly optimized exact solver and library for TSP (code by Applegate *et al.*) [22]. (Website: math.uwaterloo.ca/tsp/concorde.html)
- **NetworkX TSP Example:** The NetworkX library includes an example using Christofides' algorithm [15]. (See NetworkX docs.)
- **Google OR-Tools:** Google's OR-Tools has a vehicle routing library and TSP example Google Developers.
- **TSPLIB:** A library of benchmark TSP instances (Symmetric TSP data at tsplib.com) to test algorithms.
- **Further Reading:** Classic references include *"The Traveling Salesman Problem"* by Lawler *et al.* (1985) and *"The Traveling Salesman Problem: A Computational Study"* by Applegate *et al.* (2006). The survey by Bresson & Laurent (2021) provides modern context [10].

Each of the above topics (formulation, convexity, algorithms, inspiration, and applications) is well-studied. The reader is encouraged to consult the cited sources and code repositories for deeper detail and open-source implementations.

---

[1] [2] [3] facultyweb.kennesaw.edu
https://facultyweb.kennesaw.edu/mlavrov/courses/lp/lecture28.pdf

[4] [6] [12] [13] [16] [17] [18] [19] [21] Traveling salesman problem - Cornell University Computational Optimization Open Textbook - Optimization Wiki
https://optimization.cbe.cornell.edu/index.php?title=Traveling_salesman_problem

[5] [7] math.nagoya-u.ac.jp
https://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf

[8] Traveling Salesman Problem
https://www2.isye.gatech.edu/~mgoetsch/cali/VEHICLE/TSP/TSP017__.HTM

[9] [11] www.math.uwaterloo.ca
https://www.math.uwaterloo.ca/tsp/methods/dfj/index.html

[10] [2103.03012] The Transformer Network for the Traveling Salesman Problem
https://arxiv.org/abs/2103.03012

[14] Approximating Metric TSP
https://www.cse.iitb.ac.in/~rgurjar/CS759/Metric_TSP.pdf

[15] Traveling Salesman Problem — NetworkX 3.4.2 documentation
https://networkx.org/documentation/stable/auto_examples/drawing/plot_tsp.html

[20] cs.jhu.edu
https://www.cs.jhu.edu/~langmea/resources/lecture_notes/assembly_scs.pdf

[22] Concorde Home
https://www.math.uwaterloo.ca/tsp/concorde.html