The Travelling Salesman Problem

The Ultimate Route Planning Challenge - When Efficiency Meets Real World!

CS 101 - Fall 2025

What is the Travelling Salesman Problem?

The Ultimate Route Challenge

The Problem: A salesperson needs to visit every city exactly once and return home, using the **shortest possible route**.
Real-World Analogy: * Like a **delivery driver** planning the most efficient route * **GPS apps** finding the fastest path through multiple stops * **Amazon trucks** optimizing deliveries to save time and fuel * **Tour guides** planning the best sightseeing route

Why It Matters

- **Delivery Services**: UPS saves millions by optimizing routes
- **Circuit Board Manufacturing**: Drilling holes efficiently
- **DNA Sequencing**: Finding optimal gene arrangements
- **School Bus Routes**: Getting kids to school faster

The Challenge

- Seems simple with 3-4 cities
- Gets **impossibly hard** very quickly
- With just 10 cities: **3,628,800** possible routes!
- With 20 cities: More routes than atoms in the universe!

Key Question: *"How do we find the best route without checking every possibility?"*

Interactive TSP Demo: Plan Your Route!

The Mathematics Behind The Factorial Explosion!

ℹ **For n cities, there are (n-1)!/2 unique routes to check** Why? We fix the starting city and divide by 2 since routes can go clockwise or counterclockwise.

Route Count Growth

```python
def calculate_routes(n_cities):
    """Calculate number of TSP routes"""
    if n_cities <= 1:
        return 0

    # (n-1)! / 2 unique routes
    factorial = 1
    for i in range(1, n_cities):
        factorial *= i

    return factorial // 2

# Let's see the explosion!
for cities in range(2, 11):
    routes = calculate_routes(cities)
    print(f"{cities} cities: {routes:,} routes")
```

Real Numbers:

- 3 cities → 1 route, 4 cities → 3 routes

- 5 cities → 12 routes, 10 cities → 181,440 routes
- 15 cities → 43,589,145,600 routes!

The Pattern:

- Each new city **multiplies** the complexity
- Not addition - **factorial growth**!
- This is why TSP is so challenging
- Real-world problems have 100+ cities!

Time Complexity: When Mathematics Meets Reality

The Scary Truth About Computation Time

Even with the world's fastest computers, brute force TSP becomes impossible very quickly!

Time Complexity Analysis

```python
import time
import math

def time_estimate(n_cities):
    """Estimate computation time for brute force TSP"""
    routes = math.factorial(n_cities - 1) // 2

    # Assume 1 million routes per second
    seconds = routes / 1_000_000

    if seconds < 60:
        return f"{seconds:.2f} seconds"
    elif seconds < 3600:
        return f"{seconds/60:.2f} minutes"
    elif seconds < 86400:
        return f"{seconds/3600:.2f} hours"
    elif seconds < 31536000:
        return f"{seconds/86400:.2f} days"
    else:
        return f"{seconds/31536000:.2f} years"

# The scary truth
print("Time to solve TSP by brute force:")
for n in [10, 15, 20, 25]:
    print(f"{n} cities: {time_estimate(n)}")
```

Reality Check

- **10 cities**: 0.18 seconds
- **15 cities**: 21.8 days

- **20 cities**: 77 billion years
- **25 cities**: Longer than universe exists!

Why This Matters

- **UPS trucks** visit 100+ stops daily
- **Amazon deliveries** optimize thousands of routes
- **GPS systems** need real-time solutions
- **Need better algorithms** than brute force!

Python Implementation: Brute Force Approach

Warning: This Gets Slow Fast!

Our brute force solution checks every possible route. It works great for small examples, but becomes impossible for real-world problems.

The Core Algorithm

```python
def tsp_brute_force(cities, current_city=0,
                   visited=None, path=None):
    """
    Solve TSP by checking ALL possible routes
    Time Complexity: O(n!) - FACTORIAL!
    """
    # Initialize first call
    if visited is None:
        visited = {current_city}
        path = [current_city]

    # Base case: visited all cities, return home
    if len(visited) == len(cities):
        complete_path = path + [0]  # Return to start
        total_distance = calculate_distance(complete_path)
        return complete_path, total_distance

    # Try every unvisited city next
    best_path = None
    best_distance = float('inf')

    for next_city in range(len(cities)):
        if next_city not in visited:
            # Recursive magic: solve for remaining cities
            new_path, distance = tsp_brute_force(
                cities, next_city,
                visited | {next_city},
                path + [next_city]
            )

            # Keep the best route found so far
            if distance < best_distance:
                best_distance = distance
                best_path = new_path

    return best_path, best_distance
```

Distance Calculation

```
def calculate_distance(path):
    """Calculate total route distance"""
    # Example distance matrix for 4 cities
    distances = {
        (0, 1): 10, (1, 0): 10,  # City 0  City 1
        (0, 2): 15, (2, 0): 15,  # City 0  City 2
        (0, 3): 20, (3, 0): 20,  # City 0  City 3
        (1, 2): 25, (2, 1): 25,  # City 1  City 2
        (1, 3): 30, (3, 1): 30,  # City 1  City 3
        (2, 3): 35, (3, 2): 35,  # City 2  City 3
        # Distance to self is 0
        (0, 0): 0, (1, 1): 0, (2, 2): 0, (3, 3): 0
    }

    total = 0
    for i in range(len(path) - 1):
        current = path[i]
        next_city = path[i + 1]
        total += distances.get((current, next_city), 999)

    return total

# Real-world: use Euclidean distance
def euclidean_distance(city1, city2):
    """Calculate straight-line distance between cities"""
    x1, y1 = city1
    x2, y2 = city2
    return ((x2 - x1)**2 + (y2 - y1)**2)**0.5
```

Build a Project From This Code?!

Left to the reader...

> Copy these blocks into two files (`main.py`, `examples.py` and `tsp.py`) to run this larger demo from the commmand line.
> **tsp.py**

```
def tsp_brute_force(cities, current_city=0,
                   visited=None, path=None):
    """
    Solve TSP by checking ALL possible routes
    Time Complexity: O(n!) - FACTORIAL!
    """
    # Initialize first call
    if visited is None:
        visited = {current_city}
        path = [current_city]

    # Base case: visited all cities, return home
    if len(visited) == len(cities):
        complete_path = path + [0]  # Return to start
        total_distance = calculate_distance(complete_path)
        return complete_path, total_distance

    # Try every unvisited city next
    best_path = None
    best_distance = float('inf')

    for next_city in range(len(cities)):
        if next_city not in visited:
            # Recursive magic: solve for remaining cities
            new_path, distance = tsp_brute_force(
                cities, next_city,
                visited | {next_city},
                path + [next_city]
            )

            # Keep the best route found so far
            if distance < best_distance:
                best_distance = distance
                best_path = new_path

    return best_path, best_distance

def calculate_distance(path):
    """Calculate total route distance"""
    # Example distance matrix for 4 cities
    distances = {
        (0, 1): 10, (1, 0): 10,  # City 0   City 1
        (0, 2): 15, (2, 0): 15,  # City 0   City 2
        (0, 3): 20, (3, 0): 20,  # City 0   City 3
        (1, 2): 25, (2, 1): 25,  # City 1   City 2
        (1, 3): 30, (3, 1): 30,  # City 1   City 3
        (2, 3): 35, (3, 2): 35,  # City 2   City 3
        # Distance to self is 0
        (0, 0): 0, (1, 1): 0, (2, 2): 0, (3, 3): 0
    }

    total = 0
    for i in range(len(path) - 1):
```

examples.py

```
#!/usr/bin/env python3
"""
```

main.py

Output

`python3 examples.py`

Note

```
 TSP Solver - Usage Example
========================================

1  Using built-in distance matrix:
   Input: [0, 1, 2, 3]
   Best path: [0, 1, 2, 3, 0]
   Distance: 90
   Time: 0.0001s

2  Using custom city coordinates:
   Coordinates: [(0, 0), (1, 3), (4, 1), (2, 4)]
   Best path: [0, 1, 3, 2, 0]
   Distance: 12.31
   Time: 0.0001s

3  Performance for different city counts:
   3 cities:      2 routes, 0.0000s
   4 cities:      6 routes, 0.0000s
   5 cities:     24 routes, 0.0001s
   6 cities:    120 routes, 0.0007s
   7 cities:    720 routes, 0.0041s

 Solving a specific problem:
   Visiting 5 offices in a city
   Offices: ['HQ', 'Branch A', 'Branch B', 'Branch C', 'Branch D']
   Best route: HQ → Branch B → Branch D → Branch A → Branch C → HQ
   Total distance: 16.03 km
   Solve time: 0.0001 seconds

 Want more examples? Run 'python3 main.py' for interactive mode!
```

TSP is Everywhere!

The Travelling Salesman Problem appears in countless real-world scenarios, often disguised as other optimization challenges.

Logistics & Transportation

```
# Delivery route optimization
delivery_stops = [
    "Warehouse",      # Start/end point
    "123 Main St",    # Customer 1
    "456 Oak Ave",    # Customer 2
    "789 Pine Rd",    # Customer 3
    "321 Elm St"      # Customer 4
]

# TSP finds shortest route visiting all stops
optimal_route = tsp_solver(delivery_stops)
print(f"Optimal delivery route: {optimal_route}")

# Real impact:
# - UPS saves $50M+ annually with route optimization
# - FedEx reduces fuel consumption by 10%
# - Amazon uses TSP for same-day delivery
```

School Bus Routing * Visit all bus stops efficiently * Minimize travel time for students * Reduce fuel costs and emissions

Manufacturing & Technology

```
# Circuit board drilling optimization
drill_points = [
    (10, 20),    # Hole 1 coordinates
    (30, 15),    # Hole 2 coordinates
    (25, 35),    # Hole 3 coordinates
    (40, 25)     # Hole 4 coordinates
]

# TSP minimizes drill head movement
optimal_drilling = tsp_solver(drill_points)
# Result: Faster manufacturing, less wear on equipment
```

DNA Sequencing * Arrange genetic fragments in correct order * Minimize overlapping regions * Critical for medical research

Video Game AI * NPCs planning efficient patrol routes * Resource gathering optimization * Strategy game unit movement

TSP Environmental Impacts: The Green Side of TSP

1. Positive Environmental Effects - TSP optimization reduces fuel consumption and lowers emissions - Decreases overall traffic congestion - Minimizes unnecessary vehicle miles traveled
2. The Consumption Paradox - Does efficient delivery encourage more online shopping? - Could TSP optimization actually *increase* total environmental impact? - Are we solving the right problem or enabling overconsumption?
3. Future Transportation Challenges - How should TSP algorithms adapt for electric delivery vehicles? - What about charging station stops and battery range limits? - How do we optimize for renewable energy usage timing?

Think: What is an Eco-Friendly TSP Design

If you could design a TSP system, what factors besides distance would you optimize for?
Consider these green factors: - Carbon footprint per route segment - Real-time traffic patterns to reduce idling - Air quality levels in different neighborhoods - Time-of-day energy grid efficiency
Research Task: Find one company using TSP for environmental benefits. What measurable impact have they achieved?

Summary: TSP - The Beautiful Impossible Problem

The Travelling Salesman Problem teaches us about the beauty and challenges of computer science!

What We Now Know

- **Real-world relevance**: TSP is everywhere
- **Computational complexity**: Problems grow!

- **Creative solutions**: When brute force fails, get creative
- **Economic impact**: Good algorithms save energy
- **Environmental benefits**: Efficiency is good

What We Also Know!

- **The Big Lesson**: Sometimes "good enough" solutions (heuristics) are better than perfect solutions that take forever!
- **Brute_force**: "Try everything - works for small problems",
- **Heuristics**: "Use smart shortcuts - good for most cases",

- **Approximation**: “Get close to optimal - practical for real world”,
- **Machine_learning**: “Learn from patterns to solve problems - modern AI approach”

And, Also Good to Know Too!

! Important

Remember:

- Not all problems have efficient exact solutions
- Creativity beats raw computational power
- Real-world constraints matter more than textbook perfection
- Technology should serve people and planet

i Note

Questions to Explore:

- How do we know when a “good enough” solution is actually good enough?
- What happens when we combine multiple optimization techniques?
- How is AI changing the way we solve impossible problems?