# Exploring other algorithms for Traveling Salesperson Problem (TSP)

**BY: Vivek Vittal Biragoni, 211AI041** 18/05/2023

The Traveling Salesperson Problem (TSP) is a well-known problem in computer science that seeks to find the shortest possible route for a salesperson to visit a set of cities and return to the starting city. It has practical applications in logistics, transportation, and manufacturing.

## Brute-force Method for TSP:

```
import sys

def tsp_brute_force(graph, start, path, visited, current_distance, min_distance, num_cities):
    if len(visited) == num_cities:
        # Complete the tour by returning to the start city
        path.append(start)
        current_distance += graph[path[-1]][start]
        print("Path:", path)
        print("Total Distance:", current_distance)
        print("")

        # Update the minimum distance if a shorter tour is found
        if current_distance < min_distance[0]:
            min_distance[0] = current_distance

        # Remove the start city to backtrack
        path.pop()
        return

    for city in range(num_cities):
        if city not in visited:
            visited.add(city)
            path.append(city)
            current_distance += graph[path[-2]][city]
            tsp_brute_force(graph, start, path, visited, current_distance, min_distance, num_cities)
            current_distance -= graph[path[-2]][city]
            path.pop()
            visited.remove(city)

# Example usage
graph = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

num_cities = len(graph)
start_city = 0
path = [start_city]
visited = {start_city}
current_distance = 0
min_distance = [sys.maxsize]

print("All Possible Tours:")
print("--------------------")
tsp_brute_force(graph, start_city, path, visited, current_distance, min_distance, num_cities)
print("Minimum Distance:", min_distance[0])
```

```
All Possible Tours:
--------------------
Path: [0, 1, 2, 3, 0]
Total Distance: 75

Path: [0, 1, 3, 2, 0]
Total Distance: 65

Path: [0, 2, 1, 3, 0]
Total Distance: 75

Path: [0, 2, 3, 1, 0]
```

```
Total Distance: 70

Path: [0, 3, 1, 2, 0]
Total Distance: 80

Path: [0, 3, 2, 1, 0]
Total Distance: 85

Minimum Distance: 65
```

## Notes

- Code Flow: The brute-force method for solving the Traveling Salesperson Problem (TSP) uses a recursive approach to generate all possible permutations of cities. It explores each permutation, calculates the total distance for each tour, and updates the minimum distance whenever a complete tour is found. The code backtracks by removing visited cities to explore other possibilities.
- Time Complexity Analysis: The brute-force method has an exponential time complexity of O(n!), where n is the number of cities. This is because it generates and evaluates all possible permutations, which grows factorially with the input size. As a result, the brute-force method becomes impractical for larger problem instances due to the exponential explosion of computations.
- Important Considerations:
    1. Efficiency: The brute-force method guarantees finding the optimal solution for the TSP by examining all possible tours. However, its computational complexity limits its practical use to small problem sizes.
    2. Trade-off: While the brute-force method provides an exact solution, it may not be suitable for real-world TSP instances due to its computational demands. In such cases, approximation algorithms or heuristics are often employed to find high-quality solutions within a reasonable amount of time.
    3. Problem Scaling: The brute-force method becomes infeasible as the number of cities increases. For example, even for just 20 cities, there are approximately 2.43 × 10^18 possible tours, making exhaustive enumeration impractical.
    4. Optimization: Various techniques can be applied to improve the efficiency of the brute-force method, such as memoization to store and reuse intermediate results, pruning techniques to eliminate redundant computations, or parallelization to utilize multiple processors.

Overall, while the brute-force method guarantees optimal solutions for the TSP, its exponential time complexity limits its practical use to small problem instances. Efficient approximation algorithms or heuristics are preferred for larger-scale TSPs, striking a balance between solution quality and computational resources.

## Genetic Algorithm for TSP:

```python
import random
import numpy as np

class Individual:
    def __init__(self, chromosome, distance):
        self.chromosome = chromosome
        self.distance = distance

def calculate_distance(city_order, distance_matrix):
    distance = 0
    num_cities = len(city_order)
    for i in range(num_cities):
        city1 = city_order[i]
        city2 = city_order[(i + 1) % num_cities]
        distance += distance_matrix[city1][city2]
    return distance

def create_initial_population(num_individuals, num_cities):
    population = []
    for _ in range(num_individuals):
        chromosome = random.sample(range(num_cities), num_cities)
        population.append(Individual(chromosome, 0))
    return population

def evaluate_population(population, distance_matrix):
    for individual in population:
        individual.distance = calculate_distance(individual.chromosome, distance_matrix)

def selection(population, num_parents):
        sorted_population = sorted(population, key=lambda x: x.distance)
        parents = sorted_population[:num_parents]
        return parents

    def crossover(parents, num_offspring):
        offspring = []
        num_cities = len(parents[0].chromosome)
        for _ in range(num_offspring):
            parent1, parent2 = random.sample(parents, 2)
            cut_point1, cut_point2 = sorted(random.sample(range(num_cities), 2))
            offspring_chromosome = [-1] * num_cities

            offspring_chromosome[cut_point1:cut_point2+1] = parent1.chromosome[cut_point1:cut_point2+1]
            remaining_cities = [gene for gene in parent2.chromosome if gene not in offspring_chromosome]
            offspring_chromosome[:cut_point1] = remaining_cities[:cut_point1]
            offspring_chromosome[cut_point2+1:] = remaining_cities[cut_point1:]

            offspring.append(Individual(offspring_chromosome, 0))
        return offspring

    def mutation(offspring, mutation_rate):
        num_cities = len(offspring[0].chromosome)
        for individual in offspring:
            if random.random() < mutation_rate:
                swap_indices = random.sample(range(num_cities), 2)
                individual.chromosome[swap_indices[0]], individual.chromosome[swap_indices[1]] = \
                    individual.chromosome[swap_indices[1]], individual.chromosome[swap_indices[0]]

    def tsp_genetic_algorithm(distance_matrix, num_cities, num_individuals, num_generations, num_parents, num_offspring, mutation_rate):
        population = create_initial_population(num_individuals, num_cities)

        for generation in range(num_generations):
            evaluate_population(population, distance_matrix)
            parents = selection(population, num_parents)
            offspring = crossover(parents, num_offspring)
            mutation(offspring, mutation_rate)
            population = parents + offspring

        evaluate_population(population, distance_matrix)
        best_individual = min(population, key=lambda x: x.distance)

        return best_individual

    # Example usage
    distance_matrix = np.array([
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ])

    num_cities = len(distance_matrix)
    num_individuals = 100
    num_generations = 500
    num_parents = 50
    num_offspring = 50
    mutation_rate = 0.01

    best_solution = tsp_genetic_algorithm(distance_matrix, num_cities, num_individuals, num_generations, num_parents, num_offspring, mutation_rat

    print("Best Solution:")
    print("-------------")
    print("Path:", best_solution.chromosome)
    print("Total Distance:", best_solution.distance)
```

```
Best Solution:
-------------
Path: [2, 3, 1, 0]
Total Distance: 80
```

## Notes

- Code Flow: The genetic algorithm approach for solving the Traveling Salesperson Problem (TSP) follows these steps:

1. Create an initial population of individuals, where each individual represents a possible tour.
2. Evaluate the fitness of each individual by calculating the total distance of the tour.
3. Select parents from the population based on their fitness.
4. Perform crossover to create offspring by combining the genetic material of the parents.
5. Apply mutation to introduce small random changes in the offspring.
6. Update the population by replacing some individuals with the offspring.
7. Repeat steps 2-6 for a specified number of generations.
8. Select the best individual (solution) from the final population.

- Time Complexity Analysis: The time complexity of the genetic algorithm for TSP depends on several factors, including the population size, the number of generations, and the complexity of fitness evaluation. Generally, the algorithm requires O(num_individuals * num_cities^2) time for each evaluation of the fitness function, and the total time complexity is approximately O(num_generations * num_individuals * num_cities^2).
- Important Considerations:
  1. Population Size: Choosing an appropriate population size is crucial. A larger population allows for more exploration of the search space but increases computation time.
  2. Crossover and Mutation: The choice of crossover and mutation operators can influence the exploration and exploitation balance. Different strategies, such as uniform crossover or swap mutation, can be experimented with.
  3. Parameter Tuning: The performance of the genetic algorithm is sensitive to parameter values, including the number of parents and offspring, mutation rate, and number of generations. Parameter tuning is often necessary to achieve good results.
  4. Initialization: The initial population should be diverse to avoid premature convergence to suboptimal solutions. Random initialization or other techniques like random shuffling can help achieve diversity.
  5. Convergence Criteria: Defining appropriate stopping criteria, such as a maximum number of generations or a threshold for improvement, is essential to terminate the algorithm effectively.
- Solution Quality: The genetic algorithm is a heuristic approach that provides approximate solutions to the TSP. The quality of the solution depends on various factors, including the parameters, the characteristics of the problem instance (e.g., number of cities, geometry), and the randomness in the algorithm. It may find good solutions but does not guarantee finding the optimal solution.
- Scalability: The genetic algorithm is suitable for larger TSP instances compared to brute-force methods. However, as the number of cities increases, the time required for fitness evaluation and the search space grow, making it challenging to find high-quality solutions within a reasonable time for very large-scale problems.

Overall, the genetic algorithm is a flexible and widely used approach for tackling the TSP. It strikes a balance between exploration and exploitation, providing approximate solutions to TSP instances with reasonable computational resources. Effective parameter tuning and implementation considerations can significantly impact the performance and quality of the solutions obtained.

## Held-Karp Algorithm for TSP:

```
import numpy as np

def tsp_held_karp(distance_matrix):
    num_cities = len(distance_matrix)
    memo = {}

    def tsp_dp(mask, current_city):
        if mask == (1 << num_cities) - 1:
            return distance_matrix[current_city][0], [0]

        key = (mask, current_city)
        if key in memo:
            return memo[key]

        min_distance = float('inf')
        min_path = None
        for next_city in range(num_cities):
            if (mask >> next_city) & 1 == 0:
                new_mask = mask | (1 << next_city)
                distance, path = tsp_dp(new_mask, next_city)
                total_distance = distance + distance_matrix[current_city][next_city]
                if total_distance < min_distance:
                    min_distance = total_distance
                    min_path = [current_city] + path
        memo[key] = min_distance, min_path
        return min_distance, min_path

    initial_mask = 1  # Starting with the first city visited
    min_distance, min_path = tsp_dp(initial_mask, 0)
    return min_distance, min_path

# Example usage
distance_matrix = np.array([
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
])

min_distance, optimal_path = tsp_held_karp(distance_matrix)
print("Minimum Distance:", min_distance)
print("Optimal Path:", optimal_path)
```

```
Minimum Distance: 80
Optimal Path: [0, 1, 3, 0]
```

## Notes

- Code Flow: The Held-Karp algorithm for solving the Traveling Salesperson Problem (TSP) using dynamic programming follows these steps:
    1. Create a memoization dictionary to store computed distances and paths.
    2. Define a recursive function `tsp_dp` that takes a bitmask representing the visited cities and the current city as inputs.
    3. Check if all cities have been visited (mask is equal to `(1 << num_cities) - 1`), and if so, return the distance from the current city back to the starting city and the path consisting of only the starting city.
    4. If the result for the current bitmask and city is already memoized, return it.
    5. Initialize the minimum distance as infinity and the minimum path as `None`.
    6. Iterate through each unvisited city. For each city, recursively calculate the minimum distance and path from that city to the remaining unvisited cities.
    7. Update the minimum distance and path if a shorter distance is found.
    8. Memoize the result for the current bitmask and city.
    9. Return the minimum distance and path.
    10. Initialize the bitmask with the starting city visited (bit 0 set to 1) and call `tsp_dp` with the initial bitmask and starting city.
    11. Print the minimum distance and optimal path obtained.
- Time Complexity Analysis: The Held-Karp algorithm solves the TSP optimally but has an exponential time complexity. It explores all possible subsets of cities, resulting in 2^n subsets, where n is the number of cities. For each subset, it performs a constant-time operation for each city. Hence, the overall time complexity is O(2^n * n^2).
- Important Considerations:
    1. Memoization: The use of memoization helps avoid redundant computations and significantly improves the algorithm's efficiency.
    2. Subproblem Formulation: The algorithm breaks down the TSP into subproblems by considering all possible subsets of cities and the current city within each subset.
    3. Bitmask Representation: The bitmask is used to represent the visited cities efficiently, with each bit corresponding to a city's presence (1 if visited, 0 if not visited).
    4. Recursive Approach: The algorithm employs a recursive approach, where each recursive call represents a subproblem.
    5. Backtracking and Path Reconstruction: The algorithm keeps track of the optimal path by storing the previous city in each recursive call and reconstructing the path from the calculated optimal path.
- Solution Quality: The Held-Karp algorithm guarantees an optimal solution for the TSP. It finds the minimum distance and the corresponding optimal path that visits all cities exactly once and returns to the starting city. The obtained solution is guaranteed to be the shortest possible tour among all possible tours.
- Scalability: The exponential time complexity of the Held-Karp algorithm makes it impractical for large problem instances. As the number of cities increases, the computation time grows exponentially, limiting its applicability to small and moderate-sized TSP instances.
- Note: While the Held-Karp algorithm is guaranteed to provide the optimal solution, it is not suitable for solving large-scale TSP instances due to its high computational complexity. Approximate algorithms like heuristic or metaheuristic approaches are often used for larger TSP instances to obtain suboptimal but efficient solutions within a reasonable time frame.

## Comparison of the Three Methods for the Traveling Salesperson Problem (TSP):

Brute Force:

- Guarantees optimality by exhaustively searching all permutations.
- Time complexity grows factorially with the number of cities (O(n!)).
- Always finds the optimal solution but becomes impractical for large instances.

Genetic Algorithm:

- Approximate solution approach for TSP.
- Time complexity depends on parameters and population size (O(num_generations * num_individuals * num_cities^2)).
- Solution quality depends on parameters and problem characteristics.
- Can handle larger instances compared to Brute Force, but finding high-quality solutions for very large problems can be challenging.

Held-Karp Algorithm:

- Guarantees optimality using dynamic programming.
- Exponential time complexity (O(2^n * n^2)).
- Provides the optimal solution with minimum distance and path.
- Practical for small to moderate-sized instances due to its time complexity.

Overall:

- Brute Force guarantees optimality but is computationally expensive for large instances.
- Genetic Algorithm provides approximate solutions efficiently but does not guarantee optimality.
- Held-Karp Algorithm guarantees optimality but is limited to small problem sizes due to its exponential time complexity.
- The choice of method depends on problem size, desired solution quality, and available computational resources.

Double-click (or enter) to edit