Metaheuristics and Computational Intelligence

Approximation Methods for the Traveling Salesman Problem

A Comprehensive Study on Heuristic and Metaheuristic Approaches

Réalisé par :
Aymane El Fahsi
Yassine Belaabed

Encadré par :
Prof. El Ghazali Talbi

Academic Year 2024-2025

Table des matières

1 Introduction and Problem Statement

The **Traveling Salesman Problem (TSP)** asks :

Given N cities and the distances between every pair of cities, find the shortest possible tour that visits every city exactly once and returns to the starting city.

In our experiments, the Euclidean distance is used :

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

We implement several approaches :

- Constructive (Nearest Neighbor) Method
- Local Search (City-Swap and Two-opt)
- Simulated Annealing
- Genetic Algorithms
- Hybrid (Memetic) Approach

Key Questions Answered :

- **Q2.** Quality is measured by the total distance traveled.
- **Q6.** The search space size for a symmetric TSP with N cities is

$$\frac{(N-1)!}{2}.$$

- **Q7.** A solution is represented as an ordered list of city indices (with the starting city repeated at the end).
- **Q8.** Although a cyclic tour's cost is invariant, the starting city matters in greedy constructive methods.
- **Q9.** Our objective is to minimize the total tour distance.

Note Importante

Ce rapport se concentre sur l'exposé des notions essentielles relatives aux méthodes d'approximation du TSP. Pour une compréhension approfondie et interactive de notre démarche, nous vous invitons monsieur à consulter le notebook `Metaheuristics_assignement_ELFAHSIAymane_BELA`
Ce dernier contient de nombreux commentaires, figures, animations et comparaisons détaillées qui illustrent l'évolution des solutions au fur et à mesure de l'exécution des différentes méthodes.

2 Data Loading and Preprocessing

We load TSPLIB files (e.g., `berlin52.tsp` and `eil101.tsp`) and extract city coordinates. The function `read_tsplib` processes the file and returns a dictionary with the instance name, dimension, and coordinates.

Listing 1 – Reading a TSPLIB file

```
def read_tsplib(file_path):
    """Reads a TSPLIB file and returns a dictionary with 'name', 'dimension', and 'coords'."""
    coords = {}
    name, dimension = None, None
    reading_coords = False
    with open(file_path, 'r') as f:
        for line in f:
            line = line.strip()
            if not line:
                continue
            if "NAME" in line:
                name = line.split(":")[-1].strip()
            elif "DIMENSION" in line:
                dimension = int(line.split(":")[-1].strip())
            elif line.startswith("NODE_COORD_SECTION"):
                reading_coords = True
                continue
            elif reading_coords:
                if line.startswith("EOF"):
                    break
                parts = line.split()
                if len(parts) >= 3:
                    idx = int(parts[0])
                    x = float(parts[1])
                    y = float(parts[2])
                    coords[idx] = (x, y)
    return {"name": name, "dimension": dimension, "coords": coords}
```

3 Visualization Functions

Visualization is key to understanding our solutions. The `initialize_canvas` function maps city coordinates onto an image, drawing each city as a blue circle. This image is later updated to show tour progress.

Listing 2 – Initializing the visualization canvas

```
def initialize_canvas(coords, img_size=800, margin=50):
    xs = [coords[k][0] for k in coords]
    ys = [coords[k][1] for k in coords]
    min_x, max_x = min(xs), max(xs)
    min_y, max_y = min(ys), max(ys)
    scale_x = (img_size - 2*margin) / (max_x - min_x) if (max_x - min_x) > 0 else 1
    scale_y = (img_size - 2*margin) / (max_y - min_y) if (max_y - min_y) > 0 else 1
    scale = min(scale_x, scale_y)
    img = np.ones((img_size, img_size, 3), dtype=np.uint8)*255
    pixel_coords = {}
    for k in coords:
        x,y = coords[k]
        px = int(margin + (x - min_x)*scale)
        py = img_size - int(margin + (y - min_y)*scale)
```

```
        pixel_coords[k] = (px,py)
        cv2.circle(img, (px,py), radius=5, color=(255,0,0), thickness=-1)
    return img, pixel_coords
```

4 Constructive Methods : Nearest Neighbor

The Nearest Neighbor heuristic constructs a tour by always visiting the nearest unvisited city.

Discussion :

- **Q1.** It starts from an arbitrary city and repeatedly selects the nearest neighbor.
- **Q3.** The starting city can affect the final tour since the choices are locally greedy.
- **Q4.** Advantages include simplicity and speed, while disadvantages include potential suboptimality.
- **Q5.** Testing on datasets (e.g., berlin52 and eil101) yields tours with total distances around 9575 and 815, respectively.

Key Code Segment :

Listing 3 – Selecting the nearest unvisited city using a lambda function

```
next_city = min(unvisited, key=lambda city: math.hypot(
    coords[current][0] - coords[city][0],
    coords[current][1] - coords[city][1]
))
```

This succinct lambda computes the Euclidean distance and determines the next city.

5 Local Search Methods : City-Swap and Two-opt

5.1 City-Swap Neighborhood

Q10. For a tour with N cities, swapping any two cities produces

$$\frac{N(N-1)}{2}$$

neighbors (a quadratic neighborhood).

Q11. Two strategies :

- **First Improvement :** Accept the first swap that reduces cost.
- **Best Improvement :** Evaluate all swaps and pick the one with the greatest improvement.

Q12. All neighbors can be generated by iterating over all index pairs (i, j) and storing each new tour in a list.

Q13. Instead of recalculating the full tour cost (which is $O(N)$), we update only the edges affected by the swap :

$$\Delta = [d(v_{i-1}, v_j) + d(v_i, v_{j+1})] - [d(v_{i-1}, v_i) + d(v_j, v_{j+1})]\,.$$

If $\Delta < 0$, the swap is accepted.

Q14. We choose the Nearest Neighbor heuristic for our initial solution because it typically produces a reasonable tour.

Listing 4 – Swapping two cities in a tour

```
def swap_cities(tour, i, j):
    """Swap two cities in the tour."""
    new_tour = tour.copy()
    new_tour[i], new_tour[j] = new_tour[j], new_tour[i]
    return new_tour
```

5.2 Two-opt Neighborhood

Q16. The two-opt move excludes consecutive pairs, leading to a neighborhood size of

$$\frac{N(N-3)}{2}.$$

Q17. A two-opt swap is defined as :

$$\texttt{new\_tour} = \texttt{tour[:i]} + \texttt{tour[i:k+1][::-1]} + \texttt{tour[k+1:]}.$$

Q18. To calculate the quality change intelligently, we update only the affected edges :

Listing 5 – Two-opt incremental cost update

```
Delta = [d(v_{i-1}, v_k) + d(v_i, v_{k+1})] - [d(v_{i-1}, v_i) + d(v_k, v_{k
    +1})]
```

Key Function : `compute_delta_cost_city_swap`

This function is essential because it evaluates the cost change due to swapping two cities without recomputing the entire tour cost.

Listing 6 – Incremental cost evaluation for city-swap

```
def compute_delta_cost_city_swap(coords, tour, i, j):
    """
    Computes the cost change (delta) from swapping two cities at positions i
        and j.
    Handles both adjacent (including cyclic) and non-adjacent cases.
    """
    n = len(tour)
    if i > j:
        i, j = j, i
    %--- Adjacent or cyclic adjacent swap ---
    if j == i + 1 or (i == 0 and j == n - 1):
        if j == i + 1:
            A = tour[i-1] if i-1 >= 0 else tour[-1]
            B = tour[i]
            Y = tour[j]
            F = tour[(j+1) % n]
            old_cost = distance(coords[A], coords[B]) + distance(coords[B],
                coords[Y]) + distance(coords[Y], coords[F])
            new_cost = distance(coords[A], coords[Y]) + distance(coords[Y],
                coords[B]) + distance(coords[B], coords[F])
```

```
        else:
            B = tour[0]
            Y = tour[n-1]
            A = tour[n-2]
            F = tour[1]
            old_cost = distance(coords[A], coords[Y]) + distance(coords[Y],
                coords[B]) + distance(coords[B], coords[F])
            new_cost = distance(coords[A], coords[B]) + distance(coords[B],
                coords[Y]) + distance(coords[Y], coords[F])
        return new_cost - old_cost
    else:
        A = tour[i-1] if i-1 >= 0 else tour[-1]
        B = tour[i]
        C = tour[(i+1) % n]
        D = tour[j-1] if j-1 >= 0 else tour[-1]
        E = tour[j]
        F = tour[(j+1) % n]
        old_cost = distance(coords[A], coords[B]) + distance(coords[B],
            coords[C]) + distance(coords[D], coords[E]) + distance(coords[E],
            coords[F])
        new_cost = distance(coords[A], coords[E]) + distance(coords[E],
            coords[C]) + distance(coords[D], coords[B]) + distance(coords[B],
            coords[F])
        return new_cost - old_cost
```

This function is highly efficient (with $O(1)$ complexity) and is used extensively in our local search routines.

6 Simulated Annealing

Simulated Annealing (SA) helps escape local optima by probabilistically accepting moves that worsen the solution. The acceptance probability is given by

$$\exp\left(-\frac{\Delta}{T}\right),$$

where T is the temperature. At high T, the algorithm is more exploratory ; at low T, it becomes more selective.

Q21. SA can escape local optima by allowing worse moves early on. **Q22.** Parameter tuning (initial temperature T_{init}, cooling rate α, iterations per temperature) is crucial.

Key Code Segment :

Listing 7 – SA acceptance condition snippet

```
if delta < 0 or random.random() < math.exp(-delta / T):
    % Accept the move
```

This condition is the heart of SA, allowing it to accept suboptimal moves based on temperature.

7 Genetic Algorithm

The Genetic Algorithm (GA) evolves a population of solutions through selection, crossover, and mutation.

Population Initialization (Q23 & Q24)

We initialize the population using the Nearest Neighbor heuristic. Each individual is a dictionary containing the tour and its cost.

Roulette Selection (Q26 & Q27)

Roulette selection is used to select individuals probabilistically based on fitness (inverse of cost).

Listing 8 – Roulette selection snippet

```
def roulette_selection(population, num_parents):
    fitness = [1.0 / (ind["cost"] + 1e-6) for ind in population]
    total_fitness = sum(fitness)
    probabilities = [f / total_fitness for f in fitness]
    parents = []
    for _ in range(num_parents):
        r = random.random()
        S = 0.0
        for ind, p in zip(population, probabilities):
            S += p
            if S >= r:
                parents.append(ind)
                break
    return parents
```

One-Point Crossover (Q28 & Q29)

The one-point crossover operator is defined as :

Listing 9 – One-point crossover snippet

```
def one_point_crossover(parent1, parent2):
    size = len(parent1)
    cut = random.randint(1, size - 2)
    def create_child(p1, p2):
        child = p1[:cut]
        for city in p2:
            if city not in child:
                child.append(city)
        return child
    child1 = create_child(parent1, parent2)
    child2 = create_child(parent2, parent1)
    return child1, child2
```

Elitism and Mutation (Q30–Q32)

Elitism merges the current population with offspring and selects the best individuals :

Listing 10 – New population generation (elitism)

```
def create_new_population(population, offspring, population_size):
    combined = population + offspring
```

```
    combined.sort(key=lambda ind: ind["cost"])
    return combined[:population_size]
```

Mutation is performed using a simple city-swap :

Listing 11 – Mutation using city-swap

```
def mutate(individual, mutation_rate=0.1, coords=None):
    tour = individual["tour"].copy()
    if random.random() < mutation_rate:
        i, j = random.sample(range(len(tour)), 2)
        tour[i], tour[j] = tour[j], tour[i]
    cost = compute_total_distance(coords, tour) if coords else
       compute_total_distance(berlin52["coords"], tour)
    return {"tour": tour, "cost": cost}
```

8 Hybrid Approach (Memetic Algorithm)

Q36. In our Hybrid GA, we apply a Two-opt local search to each offspring generated by the GA. This approach combines :

— The global exploration capability of GA.

— The fine-tuning strength of local search.

Key Integration :

Listing 12 – Hybrid GA : Applying Two-opt to offspring

```
child1, child2 = one_point_crossover(parent1, parent2)
child1 = two_opt_local_search(child1, coords, visualize=False)
child2 = two_opt_local_search(child2, coords, visualize=False)
```

This hybridization leads to faster convergence and high-quality solutions.

9 Error Analysis and Conclusion

We compare our best-found solutions against known optimal values. The relative error is calculated by :

$$\text{Relative Error}(\%) = \left(\frac{\text{Found Cost} - \text{Optimal Cost}}{\text{Optimal Cost}} \right) \times 100.$$

Dataset	Optimal Cost	Best Found Cost	Relative Error (%)
berlin52	7542	7544.37	0.03%
eil101	629	640.42	1.82%

Key Takeaways :

— The Nearest Neighbor heuristic provides a fast initial tour, although it is sensitive to the starting city.

— Local search methods (City-Swap and Two-opt) yield significant improvements.

— Simulated Annealing offers a mechanism to escape local optima.

— The Genetic Algorithm, particularly when enhanced via local search (Hybrid GA), produces near-optimal results.

Final Verdict :
The **Hybrid Genetic Algorithm (Memetic Algorithm)** is a powerful and scalable approach for solving the TSP on medium-sized datasets. While the solution for `berlin52` is nearly optimal (with an error of only 0.03%), further tuning may be needed for larger instances like `eil101`.

Future Work : Future improvements might include adaptive cooling in SA, more sophisticated elitist strategies in GA, or hybridization with methods like Ant Colony Optimization (ACO).