Lab 9: Local Search Metaheuristics for TSP

Overview

This lab is a guided presentation of four local-search-based optimization algorithms:

- classical local search;
- variable neighborhood search;
- guided local search;
- tabu search.

All four algorithms are demonstrated on the same problem: the Traveling Salesman Problem (TSP). The goal is to find a short cyclic route through a fixed set of cities.

Python Source Files

The Python source files used in this lab assignment are:

File	Algorithm
`local_search.py`	Classical local search
`vns.py`	Variable neighborhood search (VNS)
`gls.py`	Guided local search (GLS)
`tabu_search.py`	Tabu search

To run a file, use for example:

```
python3 local_search.py
```

The scripts use:

- `numpy`;
- `matplotlib`;
- `scipy`.

The Common Problem: TSP

The Traveling Salesman Problem is:

> Given a set of cities, find a short cyclic route that visits each city exactly once and returns to the starting city.

In this lab:

- a city is represented by coordinates (x, y);
- a route is represented as an ordering of city indices;
- the objective value is the total length of the cyclic route;
- a smaller objective value means a better route.

For example, the route:

$$[0, 1, 2, 3]$$

means:

$$0 \to 1 \to 2 \to 3 \to 0.$$

Common Language

For all four algorithms, use the same vocabulary:

Concept	Meaning in this lab
State	One candidate route.
Objective function	Total route length.
Neighbor	A route obtained by modifying the current route.
Move	The modification used to create a neighbor.
Local optimum	A route that cannot be improved by the allowed local moves.

Common Code Pattern

All four algorithms use the same underlying representation:

- a route is a list of city indices;
- the objective function computes the total cyclic route length;
- smaller objective value means a shorter route;
- most moves are based on swapping cities.

The basic objective function is:

```
def objective_function(solution):
    indexes = np.vstack((solution, np.roll(solution, -1))).T
    return np.sum(dist_matrix[indexes[:, 0], indexes[:, 1]])
```

The expression `np.roll(solution, -1)` shifts the route by one position, so each city is paired with the next city. The last city is paired with the first city, which closes the cycle.

1 Algorithm 1: Classical Local Search

Classical local search starts from one route and repeatedly moves to a better neighboring route. In this implementation, neighbors are generated by swapping two cities.

The algorithm stops when no neighboring route is better than the current route.

Classical local search is greedy. At each step it looks at the neighbors of the current route and moves to a better one. In this implementation, the algorithm uses **best descent**: it checks all 2-swap neighbors and chooses the best improving neighbor.

The neighborhood is generated by swapping two positions in the route:

```
def generate_neighbors2(current_solution):
    n = len(current_solution)
    neighbors = []
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = current_solution.copy()
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors
```

The main loop in the actual source code is:

```
for iteration in range(max_iterations):
    neighbors = generate_neighbors2(current_solution)

    best_neighbor = None
    best_neighbor_objective_value = float('inf')

    # Use the best descent strategy and move to the best neighbor
    for neighbor in neighbors:
        neighbor_objective_value = objective_function(neighbor)

        if neighbor_objective_value <
           best_neighbor_objective_value:
            best_neighbor = neighbor
            best_neighbor_objective_value =
               neighbor_objective_value

    # Update the current solution if the best neighbor is better
    if best_neighbor_objective_value < current_objective_value:
        current_solution = best_neighbor
        current_objective_value = best_neighbor_objective_value
    else:
        # If there is no improvement, stop the search
        break
```

So local search improves the route until it reaches a local optimum: a route where no 2-city swap gives a shorter route.

Code to Run

```
python3 local_search.py
```

Questions

After running the code, answer:

a) What is the initial route?

b) What is the objective value of the initial route?

c) What does the function `generate_neighbors2` do?

d) What does the function `objective_function` compute?

e) What route and objective value are found after local optimization?

f) Why can local search get stuck?

Key thing to notice: classical local search is simple and fast, but it can get trapped in local minima. It stops when no one-swap neighbor improves the current route, even though a better route may exist farther away in the search space.

2 Algorithm 2: Variable Neighborhood Search

Variable Neighborhood Search (VNS) tries to improve ordinary local search by using several different neighborhoods.

If the algorithm gets stuck using small changes, it tries larger changes. In this code:

- neighborhood $k = 1$ uses 2-city swaps;
- neighborhood $k = 2$ uses changes involving 3 cities;
- larger k uses larger permutations.

The key idea is:

> If one neighborhood cannot escape a local optimum, another neighborhood might.

VNS adds a new idea to local search: instead of using only one neighborhood, it uses several. The algorithm first makes a random jump in neighborhood k, then applies ordinary local search from that new route.

The random jump is called **shaking**:

```
def shaking(solution, k):
    if k == 1:
        generated_neigs = myls.generate_neighbors2(solution)
    elif k == 2:
        generated_neigs = generate_neighbors3(solution)
    else:
        generated_neigs = generate_neighborsN(solution, k)
    return choice(generated_neigs)
```

The central VNS idea is:

```
shaken_x = shaking(current_solution, k)
ls_result = myls.local_search(shaken_x)
```

If the improved solution is better, VNS accepts it and resets k to 1:

```
if improved_objective_value < current_objective_value:
    current_solution = improved_x
    current_objective_value = improved_objective_value
    k = 1
else:
    k += 1
```

Resetting k means: after finding a better route, try small changes again before moving to larger neighborhoods.

Code to Run

```
python3 vns.py
```

Questions

After running the code, answer:

a) What does the function `shaking` do?

b) What happens when VNS finds a better solution?

c) Why does the algorithm reset k to 1 after an improvement?

d) Compare the result of VNS with the result of classical local search.

e) Why is VNS usually slower than simple local search?

Key thing to notice: VNS keeps trying larger shakes only when smaller shakes fail. If any shake leads to a better local optimum, it accepts it and starts again from small shakes.

3 Algorithm 3: Guided Local Search

Guided Local Search (GLS) modifies the objective function when the search gets stuck. For TSP, the features of a solution are the edges used in the route.

The original objective is:

$$f(x) = \text{route length}.$$

GLS uses a penalized objective:

$$g(x) = f(x) + \lambda \sum_i I_i(x) p_i.$$

Here:

- $I_i(x) = 1$ if feature i is present in solution x;
- p_i is the penalty assigned to feature i;

- λ controls how strongly penalties affect the search.

When local search gets stuck, GLS penalizes selected features of the current route. This makes the current local optimum less attractive and encourages the search to move elsewhere.

GLS keeps two objective functions in mind:

- the real objective: the actual route length;
- the penalized objective: route length plus penalties.

The penalized objective is implemented as:

```
def penalized_objective_function(solution, penalty_matrix,
   lambda_value):
    original_value = myls.objective_function(solution)
    used_edges = identify_edges(solution)
    s = 0
    for e in used_edges:
        s += penalty_matrix[e[0], e[1]]
    penalized_value = original_value + np.sum(s * lambda_value)
    return penalized_value
```

The route features are the edges used in the route:

```
def identify_edges(solution):
    shifted_solution = np.roll(solution, -1)
    identified_edges = []
    for i in range(0, len(shifted_solution)):
        identified_edges.append((solution[i], shifted_solution[i])
           )
    return identified_edges
```

When GLS gets stuck, it chooses edges with high utility and increases their penalties:

```
identified_edges = identify_edges(current_solution)
components_to_penalize = maximum_utility(identified_edges,
   penalty_matrix)
for component in components_to_penalize:
    penalty_matrix[component] += 1
```

This does not change the real route length. It changes what the search is encouraged to avoid next.

Code to Run

```
python3 gls.py
```

Questions

After running the code, answer:

a) What are the features in this TSP implementation?

b) What does the penalty matrix store?

c) What is the difference between the real objective and the penalized objective?

d) Why does GLS penalize some edges?

e) Compare the result of GLS with the result of classical local search.

Key thing to notice: GLS escapes local minima by changing the objective function, not by changing the route directly. When the search gets stuck, it penalizes selected edges of the current route so that future local search is guided toward routes using different edges.

4 Algorithm 4: Tabu Search

Tabu search is local search with memory. It keeps a short list of recently used moves and temporarily forbids them.

In this implementation:

- the neighborhood is still based on swapping two cities;
- the tabu list stores recent swaps;
- a tabu move is normally forbidden;
- the aspiration rule allows a tabu move if it gives the best route seen so far.

The purpose of the tabu list is to prevent cycling, for example:

$$A \to B \to A \to B.$$

Tabu search also uses 2-city swaps, but it remembers recent swaps in a tabu list:

```
tabu_list = deque(maxlen=tabu_tenure)
```

When a candidate route is created, the algorithm records which two city labels were swapped:

```
move = tuple(sorted((current_solution[i], current_solution[j])))
is_tabu = move in tabu_list
```

Normally, tabu moves are skipped:

```
if is_tabu and candidate_value >= best_objective_value:
    continue
```

The exception is the **aspiration criterion**: if a tabu move gives the best route seen so far, it is allowed anyway.

After accepting a move, the move is added to memory:

```
current_solution = best_candidate
tabu_list.append(best_candidate_move)
```

So tabu search can move away from a local optimum, but avoids immediately undoing recent moves.

Code to Run

```
python3 tabu_search.py
```

Questions

After running the code, answer:

a) What is stored in the tabu list?

b) What does `tabu_tenure` mean?

c) Why is tabu search allowed to move to a worse solution?

d) What is the aspiration criterion?

e) Compare the result of tabu search with the results of the other methods.

Key thing to notice: tabu search is local search that is allowed to move to a worse route, but it remembers recent moves so it does not immediately go back.

Final Comparison

At the end of the lab, complete the following comparison table:

Algorithm	Escape mechanism	Main disadvantage
Local search	None	Can get stuck in local optima
VNS	Changes neighborhood size/type	Can be slower
GLS	Penalizes bad features	More abstract; needs penalty design
Tabu search	Forbids recent moves	Needs memory parameters

Main Takeaway

All four algorithms use the same basic idea:

$$\text{current solution} \rightarrow \text{neighbor solution} \rightarrow \text{possibly better solution.}$$

They differ in how they try to avoid getting stuck in local optima.