Documentation for the Tabu Search Algorithm Solving the Traveling Salesman Problem

Aliaksei Zimnitski
FIIT STU

October 9, 2024

Introduction

This documentation describes the implementation of the Tabu Search algorithm to solve the Traveling Salesman Problem. The main objective is to find the shortest path that passes through a given number of cities, while avoiding cycling through previously visited solutions.

Key Program Variables

- **number_of_cities** — the number of cities that need to be visited.
- **tabu_list** — a list of forbidden paths (routes) that have been visited, to prevent cycling.
- **tabu_list_size** — the size of the Tabu list, which dynamically adjusts based on the number of identical best results found consecutively.
- **max_tabu_list_size** — set to 50. This value controls the maximum number of forbidden paths stored in the Tabu list, preventing the algorithm from revisiting them. After multiple tests, this size proved to be effective in maintaining diversity in solutions while avoiding stagnation.
- **number_of_identical_best_to_stop** — set to 60. This value determines the number of consecutive iterations without improvement before the algorithm terminates. Testing showed that this value strikes a good balance between giving the algorithm enough time to explore solutions and preventing excessive runtime when no better solutions are being found.
- **current_path** — the current path being evaluated. This is a tuple consisting of the total distance and the path itself.
- **best_result** — the current best path found by the algorithm.
- **roads_weight** — a 2D array representing the distances between every pair of cities.
- **number_of_identical_best** — the number of consecutive iterations where the best result has not improved.
- **random_path** — the initial random path generated at the start of the algorithm.

- **neighborhood** — the set of neighboring solutions generated from the current path.
- **min_neighbor** — the best (shortest) path found among the generated neighbors.
- **number_of_iterations** — the total number of iterations completed by the algorithm.

Key Functions of the Program

Distance Calculation Functions

- **euclidean_distance(x1, y1, x2, y2)** — calculates the Euclidean distance between two cities. This function is used to construct the distance matrix between cities.
- **path(roads_weight, cities)** — calculates the total distance of a given path through all cities by summing the distances between consecutive cities, including the return to the starting city.

Neighborhood Generation Functions

- **make_neighbors(current_path, roads_weight)** — generates neighboring paths by performing three types of operations on the current path:
  - Swap — exchanges two adjacent cities.
  - Reverse — reverses a segment of the path.
  - Shuffle — randomly shuffles a segment of the path.

  These operations ensure a diverse exploration of neighboring solutions and increase the likelihood of finding a better route.

Visualization Function

- **plot_path(cities_coordinates, best_path)** — visualizes the best-found path by plotting the coordinates of the cities and the connecting lines between them. **This function was generated by ChatGPT and not written by me.**

How the Program Works

Initialization

The program begins by taking user input for the number of cities and their coordinates. It then constructs a distance matrix (`roads_weight`) based on the Euclidean distances between each pair of cities. A random initial path (`random_path`) is generated, and the algorithm starts by setting this as the `current_path`.

Running the Tabu Search Algorithm

In each iteration, the algorithm performs the following steps:

1. The function **make_neighbors** generates a set of neighboring solutions based on the current path.
2. Paths that are present in the `tabu_list` (forbidden paths) are removed from the set of neighbors.
3. If no valid neighbors remain, the `current_path` is reshuffled randomly, and the process restarts with the best result saved.
4. The best neighbor (shortest path) is chosen and set as the new `current_path`.
5. If the new `current_path` is shorter than the best result, it becomes the new `best_result`, and the counter for identical best results is reset. Otherwise, the counter is incremented.
6. The current path is added to the `tabu_list`, and the size of the list is dynamically adjusted based on the number of identical best results. If the list exceeds its maximum size, the oldest element is removed.
7. The algorithm continues iterating until the number of consecutive identical best results exceeds `number_of_identical_best_to_stop`. Once this condition is met, the algorithm terminates and outputs the best path found.

Improvements

- **Dynamic Tabu List Size** — adjusts the size of the `tabu_list` to avoid getting stuck in local optima. The size increases as the number of identical best results grows, helping to diversify the search.

```
tabu_list_size = min(tabu_list_size + (
    number_of_identical_best * 100 /
    number_of_identical_best_to_stop) * max_tabu_list_size
     * 0.01, max_tabu_list_size)
```

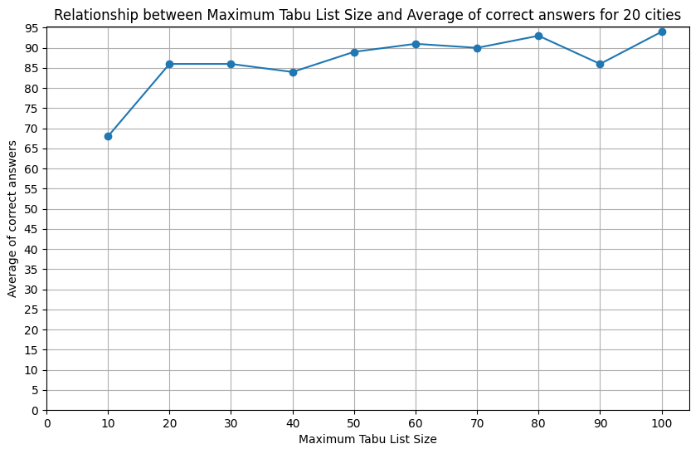
Testing and Algorithm Decisions

To find the optimal value for `max_tabu_list_size`, a series of tests were conducted, and graphs were generated for scenarios with 20 and 30 cities. The graphs show the relationship between the `max_tabu_list_size` and the average number of correct answers. Based on these graphs, it was determined that the optimal value for `max_tabu_list_size` is 50, as it consistently provided the best results across different scenarios.

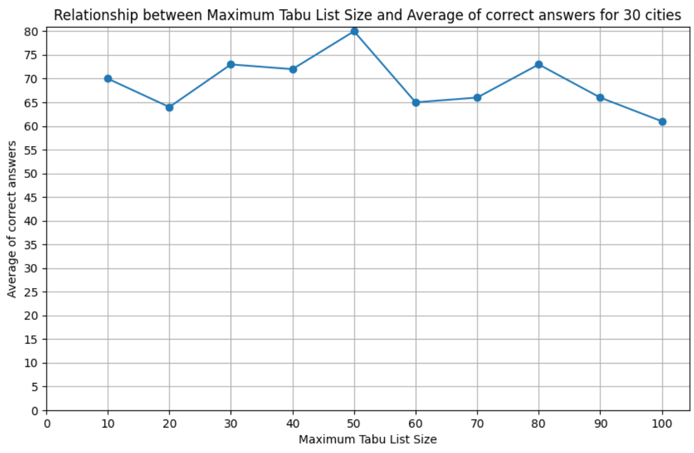
All tests were conducted using the following test cases:

```
20
60 200
180 200
100 180
140 180
20 160
80 160
200 160
140 140
40 120
120 120
180 100
60 80
100 80
180 60
20 40
100 40
200 40
20 20
60 20
160 20
```

```
30
63 23
5 191
129 193
141 25
102 161
115 135
141 187
90 121
142 100
14 82
82 1
172 32
157 199
175 75
168 85
121 191
166 0
137 104
100 114
168 12
23 160
197 117
127 164
29 101
8 129
112 180
69 51
46 145
23 179
107 23
```



For 20 Cities



For 30 Cities