Artificial Intelligence:
The Traveling Salesman Problem

Marián Kurčina

ZS 2024

xkurcinam@stuba.sk
marian.kurcina2003@gmail.com

Table of Contents

Tabu search

Task

The Traveling Salesman must visit multiple cities, passing through each city exactly once and ending at the city where they started. The goal is to find the shortest route. There are 30 to 40 cities, each with randomly generated coordinates. The map dimensions are 500x500. The cost of traveling between two cities is given by the Euclidean distance, calculated using the Pythagorean theorem. The objective is to find the order that results in the shortest total route. The output should be the order of the cities and the length of the corresponding path. This problem should be solved using Tabu search.

Solution description

The route will be stored as a list representing the order of cities I will visit, with each city appearing only once in the list (the last step back to the starting city will not be described in the list). During the search, I will generate successors of the current path and select the shortest one that is not in the tabu list. This path will become my new route. If this path is longer than the current one, I will add the current path to the tabu list to escape the local extreme. After several iterations with the same best path, I will print and plot the best route. For testing the program, I will use the same city layout each time; the layout used for testing is commented in the code.

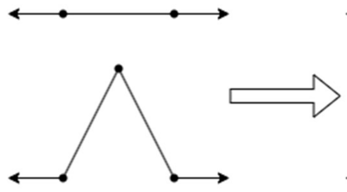
Code description

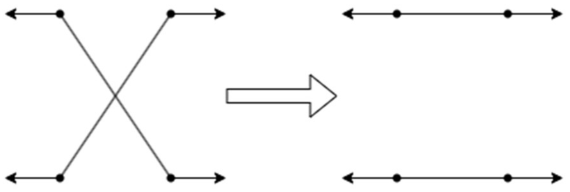
At the beginning of the program, I have stored data about the cities, how many there should be, and their coordinates. The coordinates are stored in two arrays. At the start, the program asks whether the user wants to test the program or just find the best route. If the user wants to test, the program will ask for the length of the tabu list, after how many identical iterations it should stop (iterations where the route does not improve), and how many times the calculation should be repeated. If the user does not want to test, only one calculation will be performed with pre-set parameters.

The program includes the functions draw, fit, generate1, generate2, generate3, generate4, generate5, and search.

- The draw function takes an array of the path and visualizes it.
- The fit function calculates the fitness function for a given path, where the value of the path is expressed in terms of the total distance. The lower this value, the better. The distance between individual cities is calculated using the Pythagorean theorem.
- The functions generate1, generate2, generate3, generate4, and generate5 are different methods of generating all the successors of the path.
    - Generate1 generates successors by swapping the order of two neighboring cities.
    - Generate2 generates successors by swapping the order of two random cities.

- Generate3 generates successors by moving a city from one position to another. This method is efficient for connecting a city to two closer cities. (Image 1)
- Generate4 generates successors by swapping the order of cities between two random cities. This method eliminates "intersections," meaning paths that overlap. (Image 2)
- Generate5 is the most computationally demanding because it is a combination of generate3 and generate4.


Obrázok 1


Obrázok 2

I compared the efficiency of these four methods of generating successors. For each method, I calculated the number of successors created and determined the average route found for 30 and 40 cities (when determining the average route, the tabu list length was 10, the cities always had the same layout, and the average was calculated from 100 trials). The comparison result is shown in the table:

	generate1	generate2	generate3	generate4	generate5
The number of successors created (30 cities)	29	870	870	870	1740
Average found route (30 cities)	5526.389	2802.459	2535.322	2298.810	2288.259
The number of successors created (40 cities)	39	1560	1560	1560	3120
Average found route (40 cities)	7341.443	3515.668	2989.976	2608.948	2577.468

As we can see, the functions generate4 and generate5 find comparably long average routes, but generate4 does so with fewer successors and, therefore, fewer steps. For this reason, I will continue generating successors only with the generate4 function, but I will leave the code for the other functions in the program.

The search function performs the actual search for the shortest route. It starts by creating a random order of cities and then improves this initial route using forbidden search. The main parameters in this function are the length of the tabu list and the number of iterations with the same longest route before the function ends. To find the optimal value for these two variables, I tested various values for them until I found the most optimal one. The average is calculated from 100 trials. My findings are shown in the table:

30 cities

Length of the tabu list	Number of repetitions	Average length
0	10	2323.913
10	10	2303.115
15	10	2295.424
20	10	2309.027
20	**20**	2293.825
20	**30**	2295.003
25	**20**	2297.969
30	**20**	2292.631
25	40	2300.249
20	100	2295.571
100	100	2284.800

40 cities

Length of the tabu list	Number of repetitions	Average length
0	10	2633.348
10	10	2631.283
15	10	2627.93
20	10	2618.376
20	20	2619.612
25	**20**	2611.878
25	**40**	2608.411
30	40	2611.068
25	60	2605.695
25	100	2613.241
100	100	2601.680

For 30 cities, the best choice is a tabu list length of 20, as increasing it further does not result in significant improvement, while the computation takes longer. For the number of repetitions, I chose the number 20. Increasing it does not show a significant difference in improvement.

For 40 cities, the best choice is a tabu list length of 25, as increasing it further results in a low improvement. For the number of repetitions, I used the number 20.

Since I am looking for one value for the tabu list length and one for the number of repetitions, independent of the number of cities, I will use the higher value for both parameters. Therefore, I will set the tabu list length to 25 and the number of repetitions to 20 in the program.

For 1000 shortest path calculations, the average path length using tabu search for 30 cities is 2297.296, and for 40 cities, it is 2617.100. Without using tabu search, the average path for 30 cities is 2322.106, and for 40 cities, it is 2629.235.

Conclusion

As we can see, the implementation of the tabu list can improve the quality of the result, even if only by approximately 1.5%. For comparison, I prepared a comparison of the improvement in finding the shortest path for the other successor generation functions. Since these functions are less efficient, the difference between using and not using the tabu list is more noticeable:

		30 cities			40 cities		
Length of the tabu list	Number of repetitions	generate2	generate3	generate4	generate2	generate3	generate4
0	10	2953.840	2533.053	2323.913	3471.023	3000.675	2633.348
10	10	2848.950	2496.793	2303.115	3469.831	2945.603	2631.283
20	10	2843.556	2521.327	2309.027	3480.812	2994.746	2618.376
20	20	2808.444	2517.808	2293.825	3425.156	2944.229	2619.612
25	20	2816.923	2503.434	2297.969	3453.272	2954.565	2611.878
25	40	2762.053	2501.691	2300.249	3394.160	2947.775	2608.411

Simulated annealing

Task

The traveling salesman needs to visit several cities, where he wants to pass through each city exactly once and finish at the city where he started. He aims to find the shortest route.

There are 30 and 40 cities, each with randomly generated coordinates. The map size is 500x500. The cost of the path between two cities corresponds to the Euclidean distance, calculated using the Pythagorean theorem. The goal is to find such an order of cities that results in the shortest total path. The output is the order of the cities and the length of the corresponding path. The problem must be solved using simulated annealing.

Solution description

The path will be stored in the form of a list representing the order of cities I will visit, with each city appearing only once in the list (the final step back to the starting city will not be described in the list). At the beginning of the search, the environment will have a certain temperature, and at each iteration, the temperature will decrease. When the temperature becomes lower than the minimum possible temperature or when there are no more successors, the search ends, and the current path is the solution.
During the search, I will generate successors for the current path and choose one of them. I will move to this new path if it is better, or if a randomly generated number is smaller than the result of the expression exp((current_distance-pos_distance)/temp). If this attempt is successful, the new path becomes my current solution. If it is not successful, I will select another successor from the remaining ones. For testing the program, I will use the same configuration of cities each time. The configuration used for testing is commented in the code.

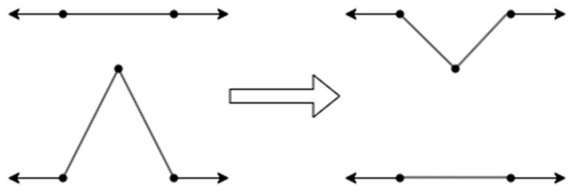
Code description

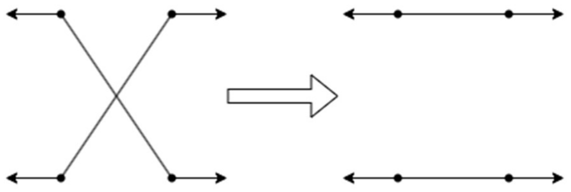
At the beginning of the program, data about the cities are stored, including how many cities there are and their coordinates. The coordinates are stored in two arrays. The program will ask whether the user wants to test the program or just find the best route. If the user wants to test, the program will ask for the initial temperature, the rate of temperature decline, the minimum temperature, and how many times the calculation should be repeated. If the user doesn't want to test, a single calculation will be performed with pre-defined parameters.

The program contains functions such as draw, fit, generate1, generate2, generate3, generate4, generate5, and search.

- The draw function takes an array of the path and draws it.
- The fit function calculates the fitness for the given path, where the path value is expressed as the total distance. The lower this value is, the better. The distance between individual cities is calculated using the Pythagorean theorem.

- Functions generate1, generate2, generate3, generate4, and generate5 are different methods of generating all the successors of the path.
    - Generate1 generates successors by swapping the order of two neighboring cities.
    - Generate2 generates successors by swapping the order of two random cities.
    - Generate3 generates successors by moving a city from one position to another. This method is efficient for connecting a city to two closer cities. (Image 1)
    - Generate4 generates successors by swapping the order of cities between two random cities. This method eliminates "intersections," meaning overlapping paths. (Image 2)
    - Generate5 is the most computationally expensive, as it combines generate3 and generate4.


Obrázok 1


Obrázok 2

I compared the efficiency of these four methods for generating successors. For each method, I calculated the number of created successors and determined the average path length for 30 and 40 cities (during the calculation of the average path, the initial temperature was 50, the cooling rate was 0.99, and the minimum temperature was 1, with the cities always having the same arrangement, and the average was calculated from 100 attempts). The result of the comparison is shown in the table:

	generate1	generate2	generate3	generate4	generate5
Number of successors created (30 cities)	29	870	870	870	1740
Average found path (30 cities)	5463.709	2663.060	2362.604	2318.053	2283.089
Number of successors created (40 cities)	39	1560	1560	1560	3120
Average found path (40 cities)	7238.903	3291.386	2750.340	2672.184	2605.113

As we can see, the function generate5 finds better average paths, but generate4 does so with fewer offspring, and therefore fewer steps. Therefore, I will continue generating offspring only with the generate4 function, but I will keep the code of the other functions in the program.

The search function performs the actual search for the shortest path. It begins by creating a random order of cities, and then improves this initial path using simulated annealing. The key parameters in this function are the initial temperature, cooling rate, and minimum possible temperature. To find the optimal values for these three variables, I tried different values until I found the most optimal ones. My goal was to keep the number of iterations for each variant below 500, while setting a minimum temperature and cooling rate. The initial temperature was chosen such that the number of iterations is as close as possible to 500 using the equation x*Power[0.99,500]<1. The average is calculated from 100 trials. My findings are shown in the table:

30 cities

Initial temperature	Cooling rate	Minimum temp.	Average length
152	0.99	1	2311.201
12.25	0.995	1	2309.705
4.49	0.997	1	2326.961
7.41	0.996	1	2337.068
33.52	0.993	1	2309.620
55.48	**0.992**	**1**	2300.361
91.86	0.991	1	2314.221
15.21	0.99	0.1	2334.671
2437.59	0.98	0.1	2335.716
1521.95	0.99	10	2497.713
76.09	0.99	0.5	2305.456
6.12	0.995	0.5	2326.503
45.93	0.991	0.5	2306.432
47.69	0.9923	1	2305.517

40 cities

Initial temperature	Cooling rate	Minimum temp.	Average length
152	0.99	1	2659.870
12.25	0.995	1	2650.666
4.49	0.997	1	2677.642
7.41	0.996	1	2677.509
33.52	0.993	1	2640.848
55.48	0.992	1	2646.923
43.12	0.9925	1	2641.359
38.99	0.9927	1	2643.400
47.69	**0.9923**	**1**	2638.450
15.21	0.99	0.1	2688.258
191.37	0.985	0.1	2669.466
41.83	0.988	0.1	2664.186
53.88	0.9875	0.1	2656.075
56.68	0.9874	0.1	2663.730

Conclusion

I found that the best initial temperature is around 50, the cooling rate around 0.992, and the lowest possible temperature 1. With this combination, the average path lengths are the shortest. For my program, I chose the initial temperature = 55.48, cooling rate = 0.992, and the lowest possible temperature = 1. After 1000 shortest path calculations, the average shortest path for 30 cities was 2305.661, and for 40 cities, it was 2645.046.