ROB537 – Homework 2 Report

Nathan Butler | butlnath@oregonstate.edu

October 24, 2023

Academic Assertion

Unless otherwise indicated, all work presented in this assignment is my own.

Code Structure

The file "Graph_TSP.py" contains everything needed to run a solver and generate .txt files with specified test parameters. The tkinter package was used to implement a user interface for viewing TSP solutions and networkx libraries were used to display TSP graphs and solutions. Running the program opens a GUI window with several options. Figure 1 shows the ways in which a user may interact with the TSP without editing any code. First, clicking the Init button will generate a graphical representation of the TSP and display it for the user. From here, the user may choose to either immediately solve the TSP and display the shortest path found or run a series of preset tests and store their results in a .txt file.

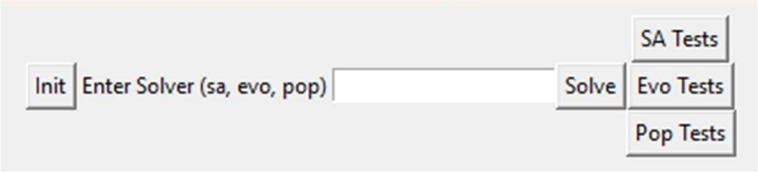

Figure 1. User interface for running solver and tests

To run a solver once, a user must type in to the "Enter Solver" box either "sa", "evo", or "pop" depending on whether they wish to use Simulated Annealing, an Evolutionary Search, or a Population search, respectively. Then, clicking the Solve button will generate and display the results.

Clicking any of the Tests buttons on the right will run tests according the run_tests_(sa, evo, pop) function. These functions allow for parameters to be adjusted such as the number of tests to be run, the number of steps to take for each test, and more. However, changing these values requires modifying the code. The results of each test (Path Distance, Solutions Explored, Final Tour, Performance Over Time) are saved as .txt files labeled to indicate the parameter settings used for each test.

The files "process_results.ipynb" and "plot.py" are used to analyze and display the results generated by running tests. All results produced by these programs are discussed in this report.

Simulated Annealing

The algorithm used for solving the TSP via simulated annealing was modeled after the approach outlined in the Simulated Annealing slide of [1]. The details of the algorithm are as follows.

1. Initialize a random tour using the np.random.shuffle function on an array of cities indices Store the total distance of this tour as current_performance

2. Cycle through the following process for the user-defined number of steps:
    a. Update temperature parameter T by multiplying by linear decay rate alpha
    b. Generate a successor state by swapping two random elements. Store distance as successor_performance.
    c. If successor_performance is less than current_performance (meaning shorter tour), then set current as successor and start next step of loop.
    d. Otherwise, generate a random value between 0 and 1. Set current as successor only if the random values is less than exp(-[successor_performance-current_performance]/T). Then start next step of loop.

Initially, this algorithm was ran using a fast annealing approach as described in [2]. In this approach, T was divided by the loop step in each loop iteration, allowing initially for some poor solutions to be selected but relatively quickly converging to values that could only be changed if a better solution was found. Figure 2 shows the values of the shortest path found over time for each of ten tests using fast annealing and varying initial temperatures. These results show the algorithm rapidly converging on relatively suboptimal solutions.

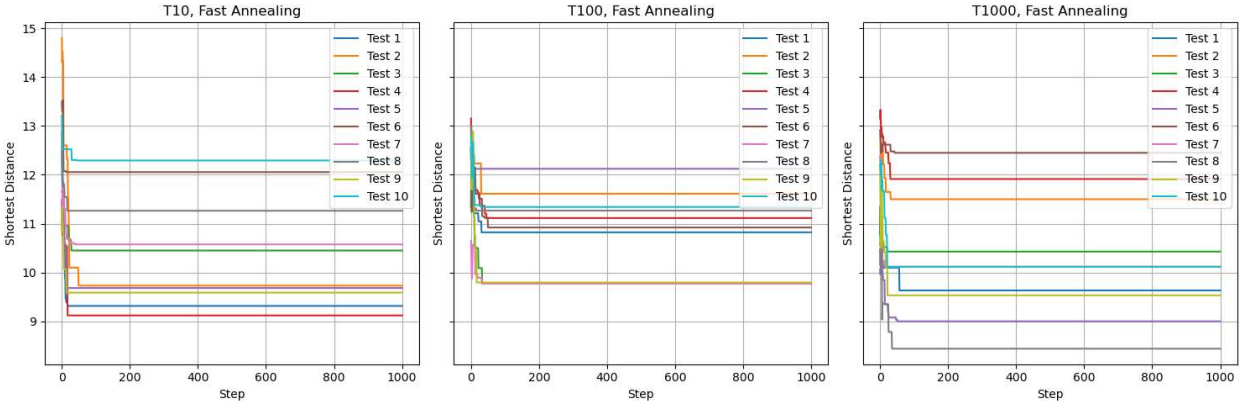

Figure 2. Performances of simulated annealing over 1000 steps using fast annealing approach at varying initial temperatures

An alternative method was found in [3], which uses a linear decay value alpha to decrease T at a slower rate. This change enabled the algorithm to explore a greater portion of “poor” solutions with the potential of discovering new avenues to better solutions over time. This solution was tested using fixed alphas and varying initial temperatures (Figure 3) and a fixed initial temperature with varying alphas (Figure 4). Many of these results also converged quickly, with an alpha of 0.99 enabling the longest period of accepting poor solutions for a starting temperature of 100.

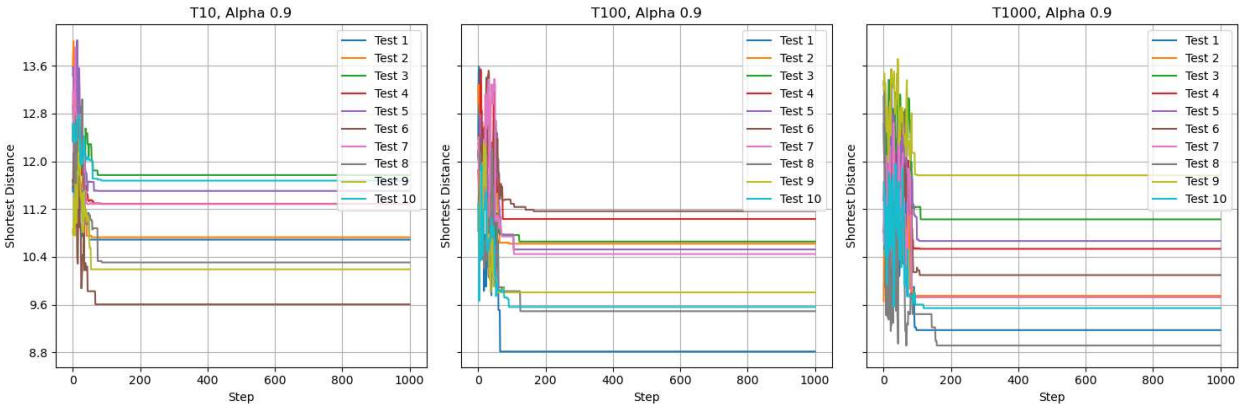

Figure 3. Performances of simulated annealing over 1000 steps using alpha approach at varying initial temperatures

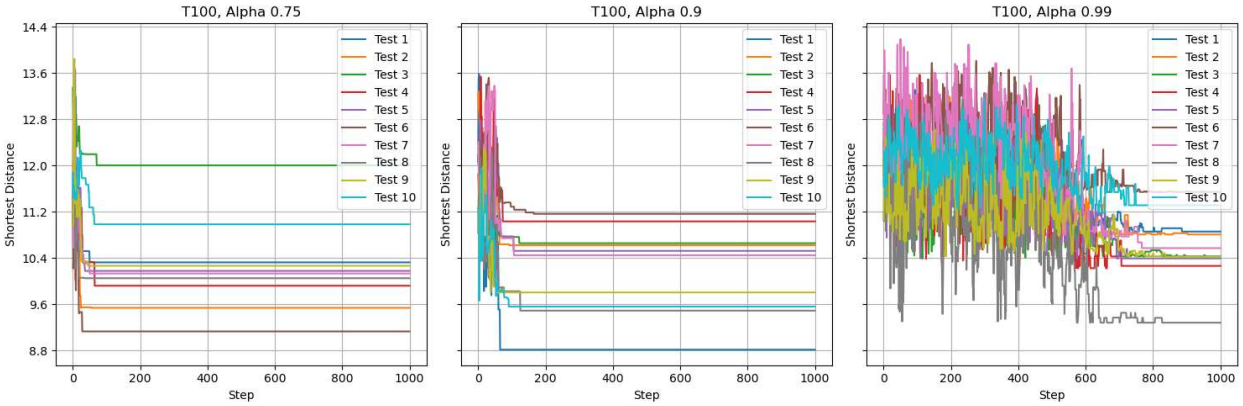

Figure 4. Performances of simulated annealing over 1000 steps at varying alpha values

The run time, solution quality, and repeatability of each simulated annealing method are recorded in Table 1. Each method was used to solve the TSP ten times and the results were analyzed in plot.py. Run time is an average of the time taken to run each of the ten tests. Quality is an average of the lengths of the final tours found for each test. Finally, repeatability is reported as the standard deviation between the final path distances found by each test.

Table 1. Results for simulated annealing tests

Method	Run Time (Average) [s]	Quality (Average Final Distance)	Repeatability (SD)
T10, FA	0.03632	10.40725	1.13194
T100, FA	0.03802	10.85409	0.82517
T1000, FA	0.03599	10.31333	1.28772
T10, a 0.9	0.03701	10.90448	0.71678
T100, a 0.9	0.03413	10.21230	0.75711
T1000, a 0.9	0.04950	10.11968	0.88027
T100, a 0.75	0.03444	10.25346	0.78452
T100, a 0.99	0.03449	10.58894	0.61963

The most significant difference observed between temperature update methods is the repeatability observed in test results. Almost all linear decay-based tests produced results that eventually converged around similar values. This result was likely due to the alpha values allowing for greater exploration of poor

solutions than fast annealing. However, none of these methods produced results that were particularly optimal. This is likely due to the method used for generating successor states. It is likely that there are better means for accomplishing this than by simply swapping two elements.

Additionally, the number of tours searched using simulated annealing is limited by the number of steps allowed by the solver. For the tests discussed above, only 1000 solutions were generated, which is less than 0.0% of the total number of possible solutions. In fact, a TSP with 25 cities has 3.102242e+23 possible solutions according to the calculation method described in [7]. It should be noted now that no method discussed in this report observed more than 0.0% of the overall solution space. Simulated annealing in particular falls victim to this vast space, as the limited number of solutions that it generates makes it difficult for the algorithm to break out of local minima. To its credit, the algorithm is the fastest of those explored in this report, so for smaller solutions spaces it may be the most applicable.

Figure 5 shows a solution to this TSP using one of the better configurations found for simulated annealing, a starting temperature of 100 and an alpha value 0.99.

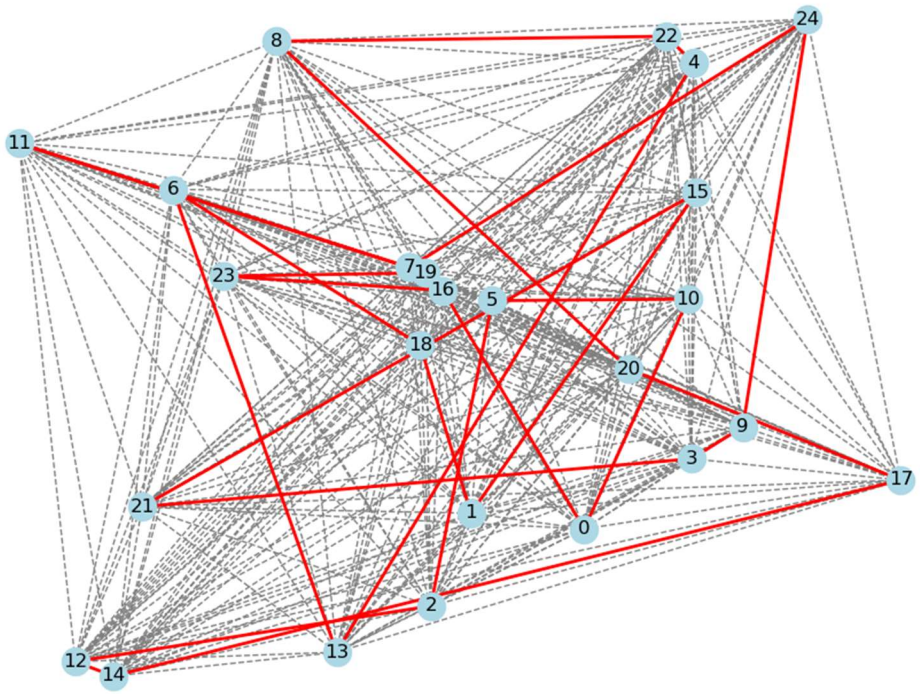

Figure 5. Solution found using simulated annealing with T=100 and alpha=0.99

Evolutionary Algorithm

The algorithm used for solving the TSP via evolutionary search was modeled after the approach outlined in the Evolutionary Search slides in [1]. The details of the algorithm are as follows.

1. Initialize k random tours using the np.random.shuffle function on arrays of cities indices.
2. Cycle through the following process for the user-defined number of steps:

a. Generate k new tours by iterating through the existing list of tours and performing a user-defined number of element swaps on each tour and adding the output to the tour list. Will result in a total population of 2k tours.
b. Using the roulette wheel selection method as described in [4] and [5], generate selection probabilities for maintaining each tour. Shorter tour lengths are assigned higher probabilities. An additional user-defined value alpha is used to exaggerate the probabilities of better solutions by raising the value of normalized tour distances to the power of alpha before once again normalizing these results between 0 and 1.
c. Finally, random numbers are generated to select the k tours to maintain in the population.

The first round of testing conducted on this method evaluated the effects of population size. In general, larger populations were found to allow the algorithm to search through more solutions, but also diluted the probabilities assigned to each element at the selection step, resulting in the loss of good solutions. To counter this, the exaggeration value alpha was introduced. By raising the probabilities to this value and then normalizing again, shorter path lengths could be assigned higher selection probabilities. This change provided additional encouragement for better solutions to be maintained while still allowing for new elements to be maintained in the population. Figure 6 shows testing results for different alphas with population sizes of 8 and 2-swap mutations. These results indicate that greater probabilities for selecting "good" tours encourages stability but seems to limit the search space for each test, resulting in increased favor toward local optimal solutions.

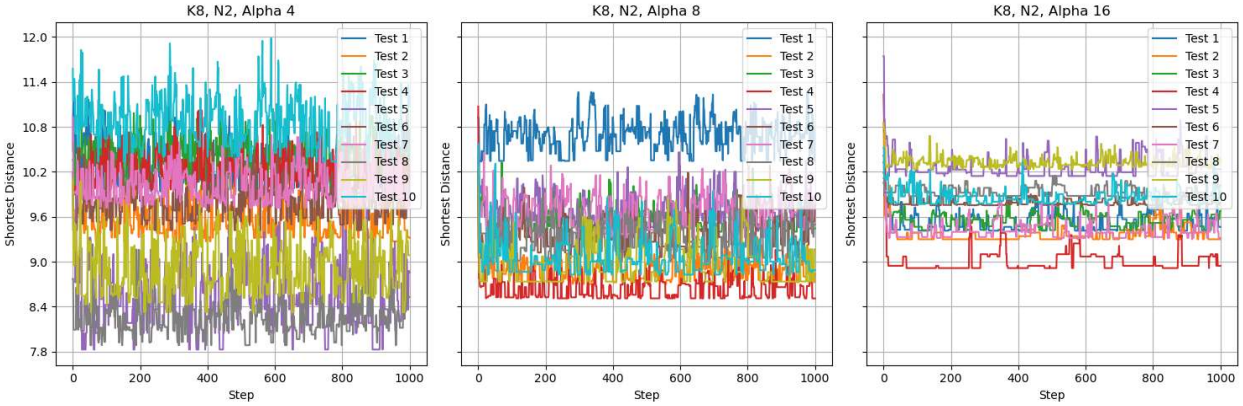

Figure 6. Performance of evolutionary algorithm for population size of 8, mutation of 2 swaps, and varying probability exaggeration values

Testing proceeded using alpha 8 for mutation magnitude tests and population size tests. Figure 7 shows the results for performing 1, 2, and 4-swap mutations. Using the swapping approach, greater numbers of swaps are shown to yield a possibility for the best results, but with low repeatability. Less extreme mutations yielded more repeatable results, but could only produce limited results due to the restricted changes made in each mutation. Figure 8 shows results for 2-swap mutations with population sizes of 4, 8, and 16 tours. Increased population sizes are shown to improve stability and repeatability in solution performance. In many tests, it can be observed that after an initial improvement period performance seems to oscillate around a steady-state. This may be due to local minima that cannot be escaped using the basic swapping mutation.

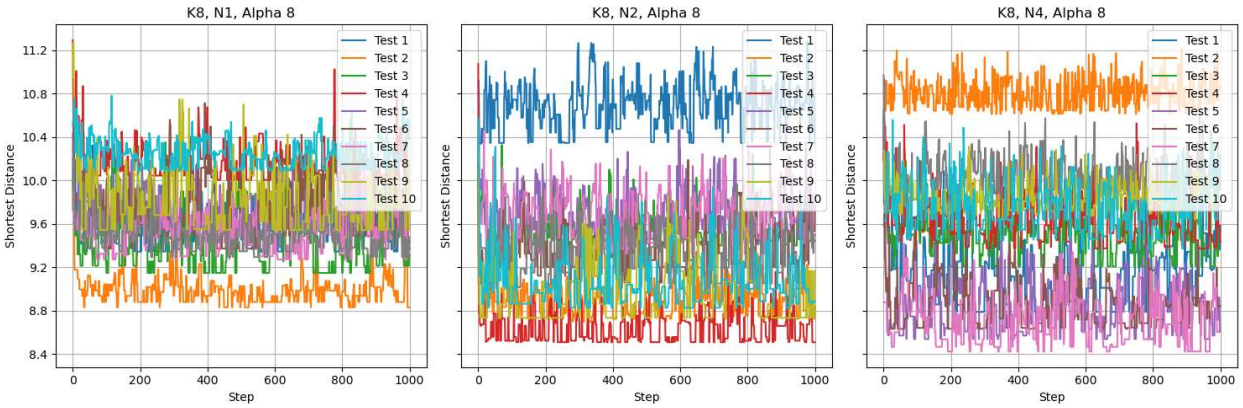

Figure 7. Performance of evolutionary algorithm for population size of 8, probability exaggeration of 8, and varying mutation swaps

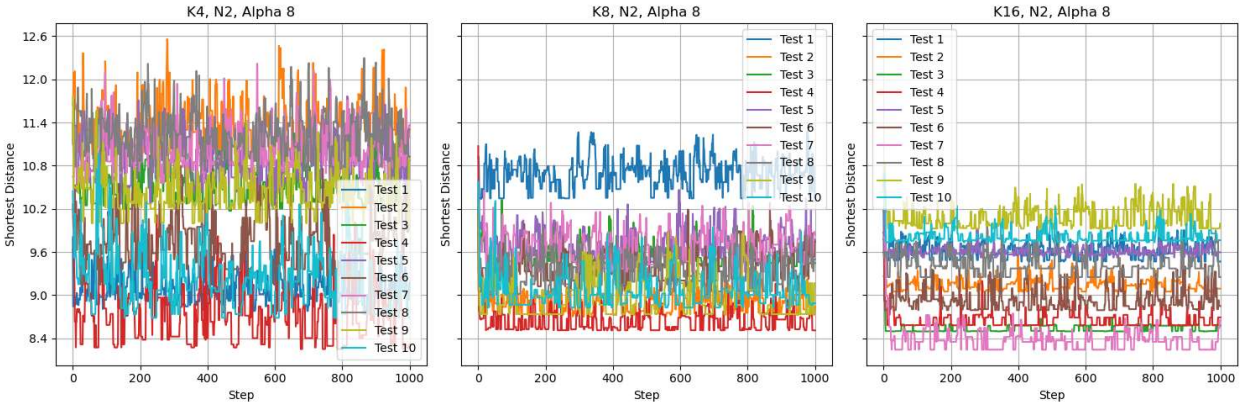

Figure 8. Performance of evolutionary algorithm for mutation of 2 swaps, and probability exaggeration of 8, and population sizes

The run time, solution quality, and repeatability of each search configuration are recorded in Table 2 and have been calculated using the methods discussed in the simulated annealing section.

Table 2. Results for evolutionary search tests

Method	Run Time (Average) [s]	Quality (Average Final Distance)	Repeatability (SD)
K8, N2, a4	0.23247	9.80680	0.83381
K8, N2, a8	0.22404	9.40497	0.55641
K8, N2, a16	0.24540	9.69986	0.44366
K4, N2, a8	0.15339	10.21826	1.14016
K16, N2, a8	0.43012	9.19286	0.50158
K8, N1, a8	0.21733	9.62777	0.49490
K8, N4, a8	0.23674	9.55162	0.72948

The number of solutions explored using this method depend on the size of the population. 4000 solutions were generated for K4, 8000 for K8, and 16000 for K16. Again this is a tiny fraction of the overall solution space. Overall, this method performed better than simulated annealing, but still did not converge on overly optimal solutions when compared to the population search discussed in the next section.

Figure 9 shows a resulting tour found using this evolutionary algorithm K16, N2, a6 settings.

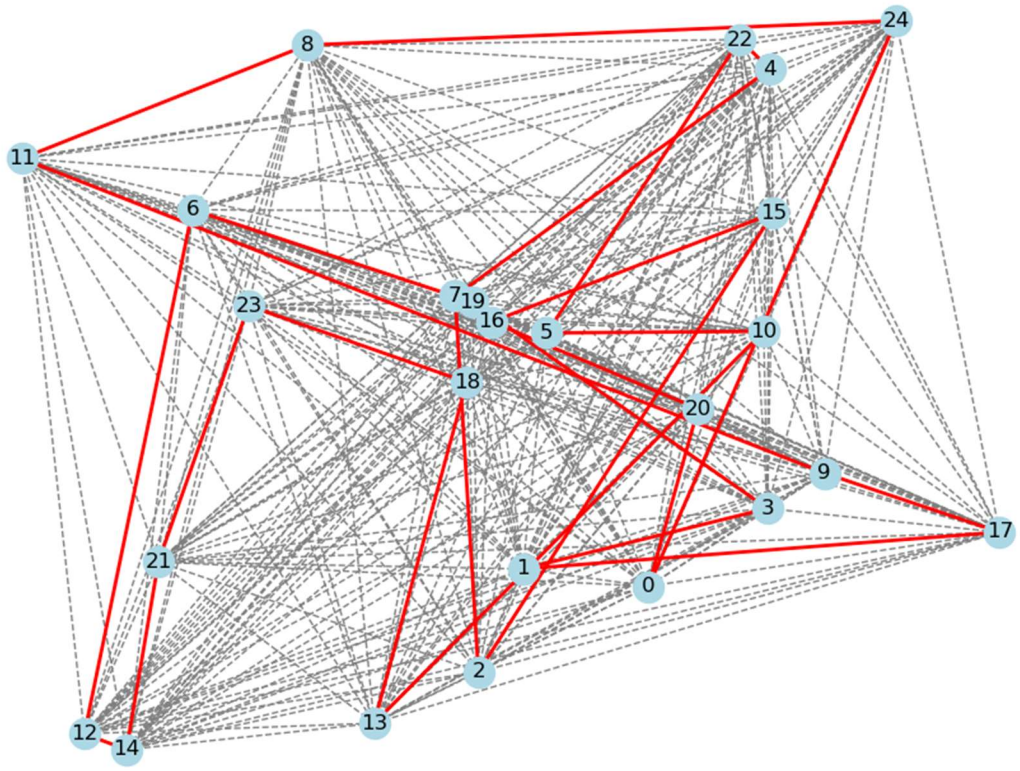

Figure 9. Solution found using evolutionary algorithm with K16, N2, a8 settings

Population Search

The algorithm used for solving the TSP via population search was modeled after the approach outlined in the Beam Search slides in [1]. The details of the algorithm are as follows.

1. Initialize k random tours using the np.random.shuffle function on arrays of cities indices.
2. Cycle through the following process for the user-defined number of steps:
    a. Generate successor states for all k current states. In this implementation, a successor state is reached by swapping any 2 elements. So, for all elements i and j in k states, new states are added in which i and j are swapped.
    b. Determine the k shortest tours from the pool of states using method similar to [6]. Set these as current states and discard all other generated states.

The testing performed on this algorithm was conducted to determine better beam widths (k). Figure 10 shows the results for widths 2, 4, and 8. Interestingly, though doubling width k doubles the search space there is not a large difference in performance beyond computation time. Each is shown to rapidly converge on a solution and remain at what is likely a locally optimal solution for the remainder of runtime.

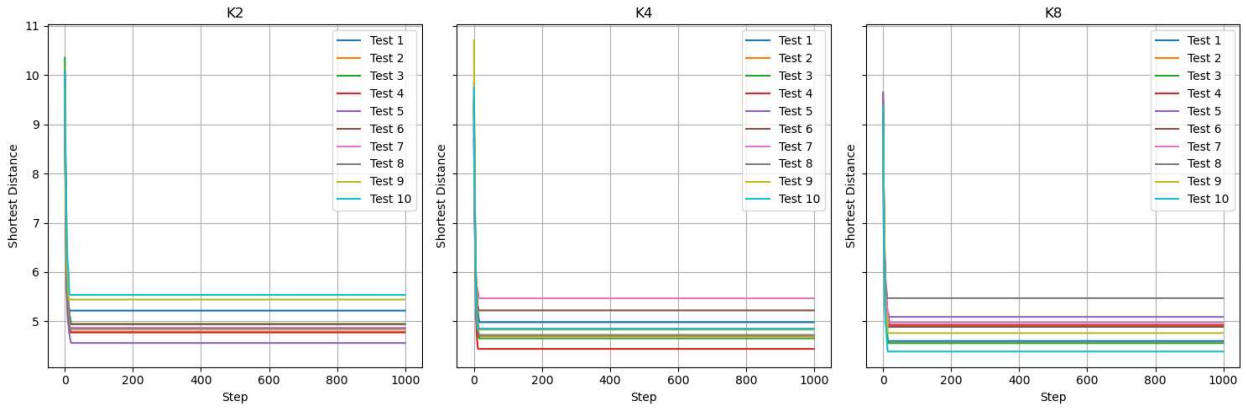

Figure 10. Performance of beam search for beam widths 2, 4, and 8

The run time, solution quality, and repeatability of each method are recorded in Table 2 and have been calculated using the methods discussed in the simulated annealing section.

Table 2. Results for evolutionary search tests

Method	Run Time (Average) [s]	Quality (Average Final Distance)	Repeatability (SD)
K2	4.96894	4.99146	0.31051
K4	10.27700	4.87076	0.29406
K8	21.01143	4.85309	0.30486

Compared to the other algorithms this much more exhaustive method excels in quality and repeatability. A significantly more thorough search of the state space is conducted, with K=2 exploring 552002 solutions, K=4 exploring 1104004 solutions, and K=8 exploring 2208008 solutions. This performance does however come at the cost of run time, which is significantly worse than the other methods. And it should also be noted that even 2208008 solutions is less than 0.0% of all solutions.

Finally, Figure 11 shows the resulting graph found by conducting a beam search with population size K=2.

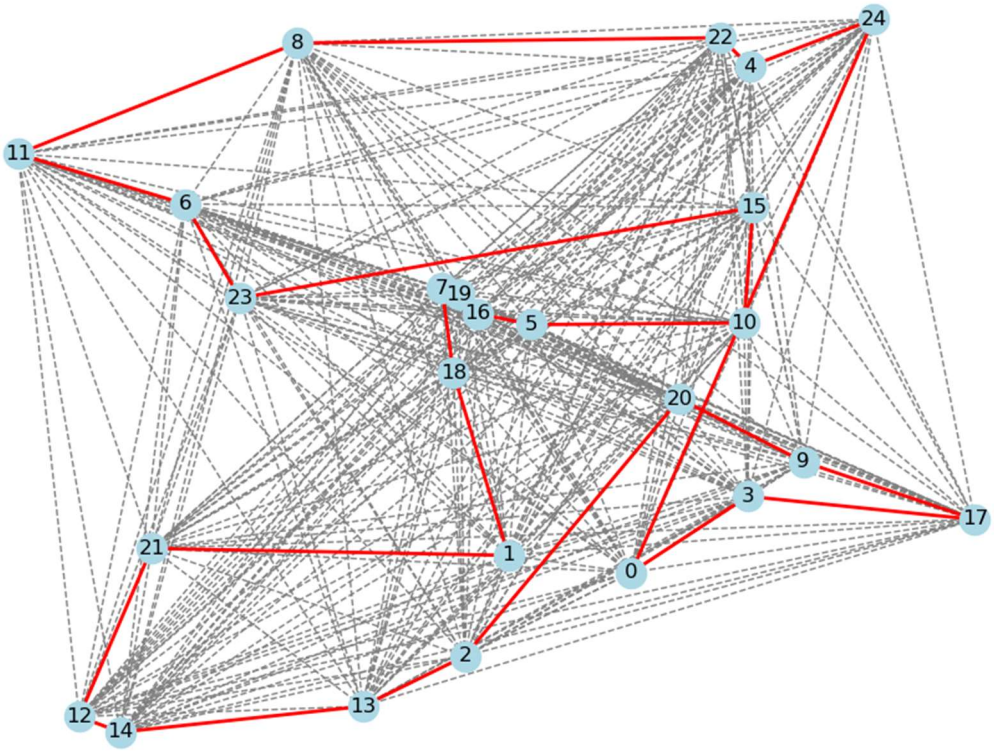

Figure 11. Solution found using population search algorithm with beam width 2

References

[1] "*ROB537 2.1 Search Notes.*" ROB 537 Learning Based Control. Oregon State University

[2] "*Simulated Annealing from Scratch in Python.*" Machine Learning Mastery. https://machinelearningmastery.com/simulated-annealing-from-scratch-in-python/

[3] "*Simulated Annealing.*" GeeksforGeeks, https://www.geeksforgeeks.org/simulated-annealing/.

[4] *"Selection (genetic algorithm)."* Wikipedia, https://en.wikipedia.org/wiki/Selection_(genetic_algorithm)#:~:text=Selection%20is%20the%20stage%20of,individuals)%20for%20the%20next%20generation

[5] "*What is Roulette Wheel Selection?*" Educative, https://www.educative.io/answers/what-is-roulette-wheel-selection

[6] "*How to find the index of N largest elements in a list or NumPy array?*" StackOverflow, https://stackoverflow.com/questions/16878715/how-to-find-the-index-of-n-largest-elements-in-a-list-or-np-array-python

[7] *6.4.5 Node Covering: The Traveling Salesman Problem*. web.mit.edu, https://web.mit.edu/urban_or_book/www/book/chapter6/6.4.5.html#:~:text=different%20orderings%20of%20the%20points,about%201.2%20x%201018.

Appendix

Access all files in hw2 folder at: https://github.com/natbut/rob537_hw/tree/main