Simulated Annealing

Anderson, Colin Bess, Brady Kartchner, Shaylee

Abstract

The Traveling Salesperson Problem (TSP) is a highly studied and well-known problem in computational complexity theory and computer science. It asks the question "What is the shortest path through a group of cities where each city will be visited once before returning to the original city?" Although it is an NP-hard problem, many algorithms and solution patterns have been created to solve the TSP. This report describes two of these algorithms, a greedy algorithm and simulated annealing, and analyzes their effectiveness in finding an optimal and computationally efficient solution. Through implementation and testing of the algorithms against a variety of problem sizes, it was discovered that …

1 Introduction

This report will showcase two algorithms for solving TSP problems. First a greedy approach will be introduced and discussed. In general, greedy algorithms can be applied to different optimization problems involving multiple steps or iterations, and characteristically they seek solutions by choosing the most appealing local value during each iteration, neglecting to look ahead, which could allow for increased optimization. The strategy and the time complexity of a greedy implementation will be explained.

Afterwards, a simulated annealing approach will be introduced. With the metallurgical heat treating and cooling process of annealing as its computational model, simulated annealing demonstrates a search that becomes more refined and favorable as our problem set is "cooled" over time. Probability and a simulated "temperature" factor into the locality explored in the global search space, and during high temperature iterations there is a greater chance of switching localities by choosing a less optimal neighboring path. This is done in hopes of escaping local optimizations that may appear to the algorithm as global optimizations. An explanation of the time complexity of simulated annealing as well as an analysis of empirical study of simulated annealing to solve TSP problems will be presented.

2 Greedy Algorithm

2.1 Description

The greedy approach to the TSP begins at each city and chooses to visit the next closest unvisited city. It will continue to visit closest unvisited cities until all cities have been visited, where it will then return to the city where it began. In this approach long travels to far away cities are meant to be eliminated, assuring our cost stays as low as possible.

Running the greedy approach once for every city as a starting point gives a great baseline minimum tour, and in cases where the city-tree structure is a fully connected (complete) graph, is a sort of exhaustive brute force way to find the optimal tour with high probability. However in many problem cases it is not guaranteed every city will be connected to every other city. In these cases the algorithm will hit a dead end in its greedy search and return an invalid path, making finding optimal path length in these situations highly unlikely. Still, the greedy algorithm will return better results than a randomly created tour.

2.2 Complexity Analysis

2.2.1 Time

This approach takes $O(n^3)$ time. The algorithm starts with a city, and picks the next shortest path. To find the shortest path, it has to loop through every available path, which is $O(n)$. It has to do this at every step, so that gives us $O(n^2)$. This could potentially find a valid solution, but there is no guarantee it will find a valid path with the initial starting city, or that the path it finds will be the best greedy solution. Because of this, we chose to run the algorithm starting with every city, ensuring we find the best possible greedy solution. This means the algorithm repeats the $O(n^2)$ process with every city, so we end up with $O(n^3)$ time complexity.

2.2.2 Space

The space complexity, however, only costs $O(n^2)$. To calculate the best path, a two-dimensional cost matrix is created, which costs $O(n^2)$ space. Each route takes up $O(n)$ space, which is written over with each iteration, saving only the best path.

3 Simulated Annealing

3.1 Description

Simulated Annealing is an approach to finding and optimal solution without getting stuck in a local minima. The basic concept is that you find a random solution. Then find a random neighbor to that solution. If that neighbor is a better solution, then keep it. If it isn't, then there is a chance that you will take it anyways because it might lead to a better solution.

This makes more sense when looked at in the context of annealing. Annealing is a heat treatment applied to metal in order to reduce hardness and increase flexibility. Flaws in the metal make it rigid, but when you heat metal up these flaws can shift and move through the metal. As it cools these flaws are locked in place. The principle that carries over to simulated annealing is that the flaws can move freely at high temperatures, but as the metal cools it becomes increasingly hard for them to move.

The pseudo-code for a generic simulated annealing algorithm is as follows this pseudo-code comes from wikipedia [2]:

Figure 1: Simulated Annealing Pseudocode

1: $s \leftarrow s_0$
2: **for** $k = 0$ through k_{max} **do**
3: $T \leftarrow$ temperature$((k+1)/k_{max})$
4: $s_{new} \leftarrow$randomNeighbor(s)
5: **if** $cost(s_n) \leq cost(s)$ **then**
6: $s \leftarrow s_n$.
7: **else if** $P(cost(s), cost(s_n), T) \geq$ random$(0, 1)$ **then**
8: $s \leftarrow s_n$.
9: **end if**
10: **end for**
11: **return** s

The procedure begins with selecting an initial solution s_0 and storing it as s. Then a temperature T is found. The value of T starts high and reaches 1 as $k \rightarrow k_{max}$. Then a random "neighbor" of s is selected. The cost of s_n is compared to the cost of s. If s_n is cheaper, then it replaces s. However, if it is not, then the cost of each solution, as well as the temperature, are passed into a probability function `P()`. When T is large, there is a high chance that `P` will return a number that is larger than `random(0,1)`. This means that at high temperatures (small values of k) the algorithm will likely select solutions that are more expensive than the previous one. This allows the algorithm to "jump" out of local minima.

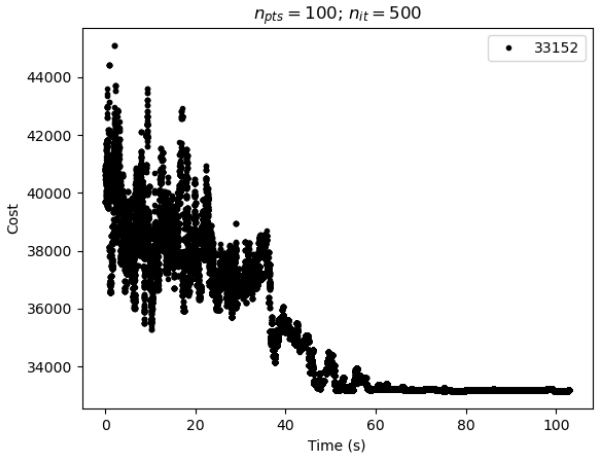


Figure 2: Example solution pattern

Figure 2 shows an example of the algorithm converging to a solution. Early in the algorithm, the cost of the selected solution often increases, but as the temperature decreases the algorithm selects more expensive solutions less and less. Eventually T reaches 1 and the algorithm returns the selected solution.

When implementing a simulated annealing algorithm there are a few design questions that have a large impact on the algorithm's effectiveness. First, you need to figure out what defines a given solution's "neighborhood". The "neighborhood" consists of other solutions that are similar to the current solution. Once the neighborhood has been defined the following items need to be developed:

- Annealing schedule
- Number of Iterations
- Starting temperature
- Probability Function

To answer these questions we did some research and found a website which implemented an annealing algorithm to solve a TSP problem (written in R and used on fully connected TSP problems) [1]. We used that code base as our source of many equations and relationships at the heart of our code.

The annealing schedule refers to the `temperature` function on line 2 of Figure 1. This function dictates the cooling curve for the algorithm. An aggressive function, such as exponential decay, can cause the algorithm to freeze up early and converge to a local solution. A sluggish function, such as linear decay, and cause the algorithm to keep "bouncing" out of solutions all the way to it's completion. A function that balances the two should be used. We used the following function for our annealing schedule:

$$T = \frac{2T_{max}}{1 + \exp(x/100)} \tag{1}$$

This equation came from the source code we obtained from Todd Schneider's website [1].

The number of iterations doesn't affect the difference between T_{max} and T_{min}, but it does dictate how quickly a solution is reached. More iterations will slow the algorithm down and give it time to move towards the optimal solution. We decided to make the number of iterations dependent on the problem size.

$$k_{max} = 100 \cdot n_{cities} \tag{2}$$

We decided on this relationship after experimenting with the iteration count. Further discussion on this can be found in Section 4.

The starting temperature T_{max} impacts how easy it is for the algorithm to move from a solution to a less-optimal neighbor. Section 4 will contain more of our analysis of this. As we did with k_{max}, we made the starting temperature dependent on the problem size. The justification behind this was that larger problems require the algorithm to "jump around" more to try and find the optimal solution. The following equation related the problem size to T_{max}:

$$T_{max} = 5 \cdot k_{max} \tag{3}$$

The probability function refers to `P()` on line 7 of Figure refanneal. This function relates the difference in cost between s and s_n with the current temperature to randomly determine if the algorithm will move to s_n or not. This has a similar impact on the algorithm as the annealing schedule. If it is easy to move to neighbors then it is easier to move towards the optimal solution. But if it is too easy then the algorithm can move past the optimal solution and settle on a local minima as it cools. We decided on the following probability function (again from Todd Schnieder's website [1]) :

$$\exp\left(\frac{-(s_n - s)}{T}\right) \geq \text{random}(0, 1) \tag{4}$$

3.2 Complexity Analysis

3.2.1 Time

The time complexity for our simulated annealing approach is $O(n^3)$. The algorithm first generates a valid path by using the greedy approach described above, which was determined to cost $O(n^3)$. This initialization is only run once. After this, the algorithm finds a neighbor to the initial path. This involves randomly picking two indices in the initial path, and greedily finding a new way between the two. This process is only repeated once; even if a valid path is not found, the algorithm simply continues and generates a new neighbor. This means that generating the neighbor only costs $O(n^2)$, since we omit the step in the initial greedy approach of checking every starting node.

After this, the algorithm compares the costs between the initial path and the cost of the newly generated neighbor. If the new path is shorter, it is chosen to continue to the next iteration. If not, it may randomly be chosen in the simulated annealing process. All of this comparison costs $O(n)$.

This process of generating a new neighbor and comparing the two path costs is repeated $n*p$, where p is a number chosen to increase the iterations. (does this make sense?) Simplified, the process is repeated $O(n)$ times, which gives us a total time complexity of $O(n^3)$.

3.2.2 Space

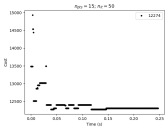
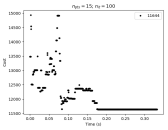
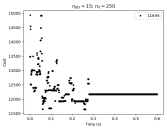
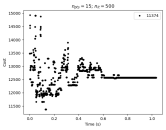
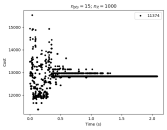
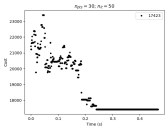
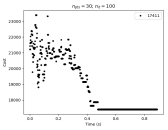
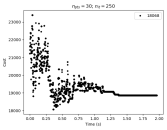
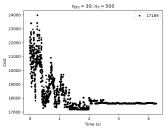
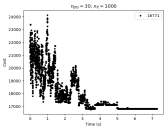
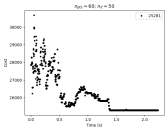
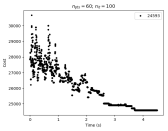
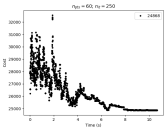
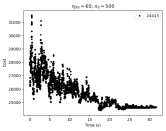
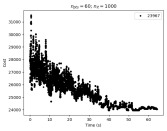
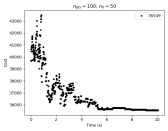
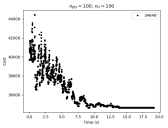
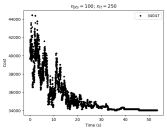
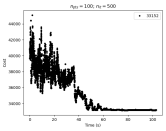
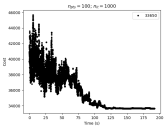
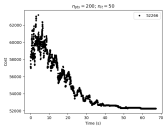
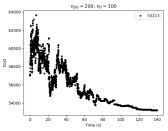
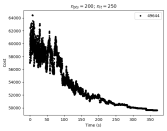
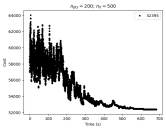
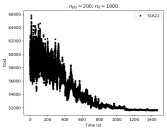
In terms of space complexity, the function that generates a new neighbor creates a cost matrix to determine valid paths, which gives the function a space complexity of $O(n^2)$. The only other space needed in the algorithm is that of the current path and neighbor path, which each cost $O(n)$. Overall, then, the space complexity for this simulated annealing algorithm is $O(n^2)$.

4 Results and Analysis

4.1 Empirical Results

The first results that we have are related to the number of iterations in a given solution. We were unsure how many iterations to do, so we ran our algorithm at various iteration counts. The results of our experimetns can be found in Table 1.

Table 1: Tour Distance Over Time at Various Iteration Counts

Number of cities	Number of Iterations				
	50	100	250	500	1000
15					
30					
60					
100					
200					

References

[1] Todd W. Schneider. *The Traveling Salesman with Simulated Annealing, R, and Shiny.* URL: https://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/.

[2] Wikipedia. *Simulated Annealing.* URL: https://en.wikipedia.org/wiki/Simulated_annealing.