Eddie Christopher Fox

June 9, 2016

CS 325: Analysis of Algorithms

Project 4 Report

Prelude:

While researching the Traveling Salesman Problem (hereafter referred to as TSP), I had several goals. First I wanted to look at educational resources to find algorithmic implementations of TSP that were written in C++. It was the first programming language I had learned, and the one I was the most comfortable with. Although I was becoming more comfortable with python, in a project this large, working by myself, I wanted to use the most familiar language I had.

I briefly used google scholar and old textbooks and pdfs from prior coursework. I initially used Marsland's Machine Learning, and then used Wikipedia and the algorithms textbook to learn more about promising algorithms.

In the end, I decided to go with a combination of nearest neighbor to construct the initial path, and Opt-2 to optimize the results afterwards. Other algorithms I researched included minimum spanning trees, exhaustive, hill climbing, and simulated annealing.

Research:

Exhaustive Algorithm:

While an exhaustive algorithm might seem like a bad choice, it does have a big advantage in that it is guaranteed to arrive at a globally optimal solution. Approximation algorithms only approximate an optimal. It was also easy to implement. It might be worth it to use an exhaustive algorithm in cases where precision is very important. In our case, however, we were constrained by a 3 minute limit on competition instances and only needed to be within 25%, which certainly other algorithms could do. Therefore I deemed this unsuitable for the project.

In the context of TSP, this algorithm works by calculating the distance between every city and every city, and summing up the distances between the cities in every valid combination. A running total of the smallest distance is kept, and with each combination tested, the resulting sum distance is compared to the running smallest distance, replacing it if smaller. In the end, the optimal tour path and distance is returned.

Continued on next page:

Hill climbing:

Hill Climbing is an optimization approximation algorithm that, while not as precise as Exhaustive, provides a reasonably good result in a reasonable amount of time. Hill Climbing begins with an arbitrary solution with a completely random tour path of cities. This bad solution is gradually improved as the algorithm performs a local search. Small elements of the solutions are better, and if the overall tour is better, it is kept. While a relatively short tour path will likely be found, TSP is not a convex problem, meaning that the weakness of Hill Climbing applies. This is that it is possible for Hill Climbing to not arrive at a solution sufficiently close to the global optimum, as it might get stuck around a local optimal due to poor early choices that only benefit in the short term. Hill climbing can easily stagnate into a configuration where it can no longer be improved. While it was not likely to take long, I did not want to risk it falling outside the 25% margin.

Simmulated Annealing:

This was a promising algorithm. Simulated Annealing is similar to Hill Climbing, but modified in a way that can avoid the downfalls of Hill Climbing being stuck in local minimums / maximums. The name for Simulated Annealing comes from annealing in metallurgy, which applied controlled cooling to crystals for better results. Similarly, this algorithm applies a metaphorical form of annealing for better results.

Like Hill Climbing, Simulated Annealing begins with an arbitrary solution and relies on randomized swapping to get better results. If a solution is found with a shorter tour length, it is accepted. However, in the case of a solution that does not improve (shorten) the tour length, or actually increases it, it is probabilistically accepted and denied based on the global "temperature" at the moment. Temperature begins at a high value. You can imagine a heated and energetic environment. When the temperature is high, bad solutions that are clearly not better than the current best are accepted with greater probability, in an attempt to avoid being stuck at a local minimum / maximum due to configurations formed early in the operation of the algorithm. Gradually, the algorithm "cools" down, and the probability of bad solutions being accepted dwindles proportionally. As the algorithm cools, it becomes increasingly likely that onlybetter solutions are accepted. The entire system naturally transitions to lower energy states asbetter and better solutions are achieved.

Eventually, the algorithm does stop when the temperature reaches 0 or a preset computational budget has been expended. Generally, the results become more accurate (closer to the global optimum) as the annealing schedule gets slower. The feature of the algorithm supporting a computational budget was actually another thing which made the algorithm desirable, as we could guarantee the algorithm finished within 3 minutes for the competition solutions.

The problem with this algorithm that while it is more likely to be accurate, repeatedly annealing the schedule can be slow, especially if the function is expensive to compute. Simulated annealing can be overkill in certain cases, and there is no way for the algorithm to actually tell if it has found an optimal solution, it just very likely avoids the possibility. Due to slight accuracy and large time concerns, I looked for something else.

2 opt and Nearest Neighbor:

2-opt is a local search heuristic, but it isn't too good at constructing an initial path, so that was what Nearest neighbor was for. Nearest neighbor is a good approximate construction algorithm. The randomized form starts at a random city and goes to the nearest neighbor every time until all cities are visited. I used a seeded version where it began at the first city and searched for nearest neighbors to the first city. I figured if the city was randomized, it was arbitrary, so why not just start at the first city and save a slight amount of time by not needing to select a random city number to begin with? It was only a slight speed improvement, but still worth it, especially since I didn't need to write any code extra. In fact, it saved me the time of needing to write randomization code.

Nearest neighbors steps from Wikipedia source #3:

1. start on an arbitrary vertex as current vertex.
2. find out the shortest edge connecting current vertex and an unvisited vertex V.
3. set current vertex to V.
4. mark V as visited.
5. if all the vertices in domain are visited, then terminate.
6. Go to step 2.

While nearest neighbor can rapidly construct a decent tour, it by no means constructs a perfect one. This is where 2-opt comes in. 2-opt is a local search algorithm proposed by Croes in 1958 for solving the Traveling Salesman Problem. It takes paths that cross over themselves and switch them around so that they do not. Where the paths do not overlap, it saves time.

2 opt swap takes two cities and routes from the first city to the lesser (in number) city minus 1 and adds it to a new route. Then it routes from the lesser city to the greater number in reverse order, adding it to the new route. I don't know why that works, but it does. Finally, it routes from the greater city to the end of the list of cities and adds this to the new route.

The code of the 2-opt is pasted from my assignment. See the actual code for comments. I used two functions: optSearch, and optSwap. optSearch uses optSwap.

```
void optSearch(int **matrix, int *tour, int *tourLength, char *outputFileName, int
numberOfCities)
{
	bool improved = true;
	int oldTourLength = *tourLength;

	while (improved)
	{

		improved = false;
		bool exit = false;
```

```
                for (int i = 1; i < numberOfCities - 1 && !exit; i++)
                {
                        for (int j = i + 1; j < numberOfCities && !exit; j++)
                        {
                                oldTourLength = *tourLength;

                                optSwap(matrix, tour, tourLength, numberOfCities, i, j);



                                if (*tourLength < oldTourLength)
                                {
                                        improved = true;
                                        exit = true;
                                 }
                        }
                }
        }
}

void optSwap(int **matrix, int *tour, int *tourLength,int numberOfCities, int cityA, int cityB)
{


        int minimumCity = MIN(cityA, cityB);
        int maximumCity = MAX(cityA, cityB);
        int newTourIndex = 0;

        int removedPathLength = 0;
        int addedPathLength = 0;

        int newTour[numberOfCities];

        if (maximumCity + 1 < numberOfCities)
        {
                removedPathLength = matrix[tour[minimumCity - 1]][tour[minimumCity]] +
                matrix[tour[maximumCity]][tour[maximumCity + 1]];

                addedPathLength = matrix[tour[minimumCity - 1]][tour[maximumCity]] +
                matrix[tour[minimumCity]][tour[maximumCity + 1]];
        }

        else
        {
                removedPathLength = matrix[tour[minimumCity - 1]][tour[minimumCity]] +
                matrix[tour[maximumCity]][tour[0]];
```

```
            addedPathLength = matrix[tour[minimumCity - 1]][tour[maximumCity]] +
            matrix[tour[minimumCity]][tour[0]];
        }

        int lengthSaved = removedPathLength - addedPathLength;



        if (lengthSaved > 0)
        {

            for (int i = 0; i < minimumCity; i++)
            {
                newTour[newTourIndex] = tour[i];
                newTourIndex++;
            }

            // Route in reverse order from minimum to maximum city.

            for (int i = maximumCity; i >= minimumCity; i--)
            {
                newTour[newTourIndex] = tour[i];
                newTourIndex++;
            }

            // Route from maximum city + 1 to the end in order.

            for (int i = maximumCity + 1; i < numberOfCities; i++)
            {
                newTour[newTourIndex] = tour[i];
                newTourIndex++;
            }



            memcpy(tour, newTour, sizeof(int) * numberOfCities);


            *tourLength -= lengthSaved;
        }
}
```

Algorithm Selected:

It was close between simulated annealing and a combination of 2-opt and nearest neighbor, but I went with the later due to word of mouth about its performance. It would often yield exactly optimal tour solutions, and even in cases where it did not, it got within a few percent, and it did so very fast, so that was all promising.

Tour Solutions and Runtimes: (See attached tour files for exact city ordering)

Example Tour Solutions:

Example 1: 109,067 0.035 seconds.

Example 2: 2,654. 2.23 seconds

Example 3: Unfortunately, my latest implementation of the software had problems with providing an output file unless it terminated manually. The first time I calculated it took over 6 hours, and I did not have time to do it again.

Competition 1: 5,373 0.029 seconds

Competition 2: 7,487 0.1149 seconds

Competition 3: 12,459 4.217 seconds

Competition 4: 17,318 49.989 seconds

Competition 5: 24,085 Hit 3 minutes.

Competition 6: 34,405 Hit 3 minutes

Competition 7: 53,985. Hit 3 minutes

References:

Marslands Machine Learning: An Algorithmic Perspective

Google

Wikipedia:

1. https://en.wikipedia.org/wiki/Travelling_salesman_problem
2. https://en.wikipedia.org/wiki/2-opt
3. https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm