A heuristic solution to the Traveling Salesman Problem

Jonah Groendal
Western Michigan University
CS4310

Abstract

An optimal solution to the traveling salesman problem is one in which the shortest circuit connecting every vertex in a completely connected and weighted graph is found. Since this is an NP-Hard problem [3], current algorithms can only optimally solve this problem in exponential time. Heuristics must be used to solve reasonably sized TSP problems. This paper will discuss my C implementation of a modified version of the nearest neighbor algorithm. The experimental results parallel closely with theoretical in terms of time complexity.

1 Introduction and Target Problem

The traveling salesman problem is a well known optimization problem with many real-world uses. Its name comes from the commonly used analogy of a traveling salesman who must visit every city within a certain area. Understandably, the salesman would like to complete his trip in the shortest manner possible, which leads to the issue of finding the shortest path. To find this path, many comparisons are required. In fact, so many comparisons are needed that an algorithm which would guarantee an optimal solution cannot find this solution in a reasonable amount of time. This is why heuristic algorithms are used. They cannot guarantee an optimal solution, but they can find an approximately optimal one in reasonable (polynomial) time. In practice, these algorithms can often find a solution within 2-5% of optimal [1]. Although my implemented algorithm is not quite this accurate, it was able to find a solution in a reasonable amount of time.

2 Intellectual Merit

For this project, I have implemented the nearest neighbor algorithm to find an approximate solution to the traveling salesman problem. This algorithm starts by selecting a random node in a completely connected graph. It then selects the “closest” node (the node connected by the lowest weight edge) to this previously selected node and continues to do so until a path has been created through all nodes in the graph. A connection is then made to the originally selected node to complete the circuit. Using this process, a solution is often found to be within 25% of optimal [1].

In my implementation, I decided to sort all edges in the graph with respect to their weight (in ascending order) before applying the nearest neighbor algorithm. This allowed me to, when selecting the next closest node, simply select the first unused node in the list that is connected to the current node. This proved useful in simplifying the code without increasing its theoretical time complexity beyond a significant amount.

3 Implementation

I decided to write my implementation of this algorithm in C. I did this mainly to avoid the complications of limited memory space imposed by higher level languages, such as Java and Python. To create the random graphs for my implementation, I used a graph generation program written by Richard Johnsonbaugh and Martin Kalin of DePaul University. The graphs generated are completely connected and weighted, with sizes ranging from 10 to 10,000 nodes. These graphs are represented in a text file of records, with each record representing an edge between two nodes. I used completely connected graphs because in the traveling salesman problem, the salesman is free to move between cities in any order.

Because the graph is formatted as a list of records, I decided to load the graph into memory as an array of structs. Each struct represents an edge, which contains attributes for the two connected nodes as well as the weight between them. After loading the edges into memory, my program sorts them using the qsort() (quicksort) function, which is a part of the C standard library. This is the most costly operation in terms of time. The nearest neighbor algorithm is then performed to find an approximate shortest path between all nodes in the graph. To keep track of the nodes used in the path while the algorithm is running, a separate array of nodes (integers) is created and referenced by the algorithm.

The time and space complexity of this algorithm can each be dissected into two parts:

Time complexity:

1. Sorting the edges:

Complexity of quicksort:	O(k log(k))	k being the number of edges.
Number of edges:	n(n-1)/2	n being the number of nodes.
Total complexity:	O(n^2 log(n))	

2. The algorithm:

Complexity of nearest neighbor: O(n^2)

The overall time complexity is O(n^2 log(n)). This is slightly greater than that of the nearest neighbor algorithm without presorting, which is still O(n^2).

Space complexity:

1. The graph in memory:

Complexity of stored graph:	O(k)	k being the number of edges
Number of edges:	n(n-1)/2	n being the number of nodes.
Total complexity:	O(n^2)	

2. Memory used by algorithm:

Complexity of nearest neighbor O(n)

The overall space complexity is O(n^2). Since the nearest neighbor algorithm only requires space to record the final path, the total complexity is proportional to the number of edges in the graph.

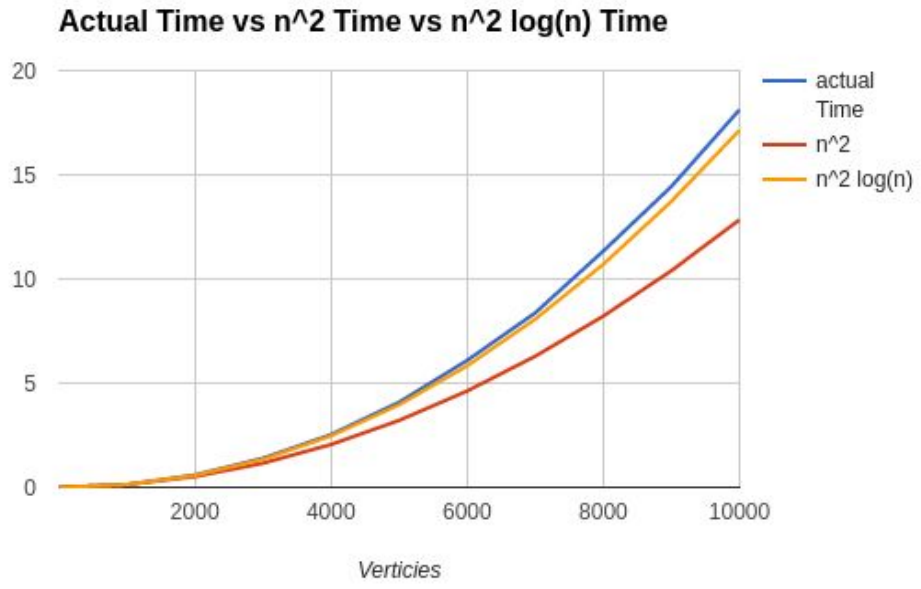
4 Performance

To test the performance of my implementation, random graphs of various sizes were used and multiple iterations of the program were performed on each graph. Time was recorded using system time functions.

Raw data:

Vertices	Average Time (in seconds)	Space (in bytes)
10	0.000005	540
1000	0.128394	5994000
2000	0.590468	23988000
3000	1.391269	53982000
4000	2.530918	95976000
5000	4.083081	149970000
6000	6.090317	215964000
7000	8.372765	293958000
8000	11.341847	383952000
9000	14.439999	485946000
10000	18.123883	599940000
15000	44.41286	1349910000

Graphical representation of actual vs theoretical time (in seconds):



Shown in the graph above, the experimental times are very similar to theoretical, which is O(n^2 log(n)). As expected, the actual time to find the approximate solution is only slightly greater than the theoretical time required.

Two additional big data trials were performed to show a slow down due to thrashing. This occurs in my program when the graph data size exceeds the available memory. Instead of simply increasing the graph size, I decided to allocate 15 additional bytes of unused memory space in each edge struct. This increased the total memory usage in a wasteful manner. I used this technique because any graph large enough to efficiently use this much memory would require an unreasonable amount of time to compute.

Big data trials:

Number of nodes	Time (in seconds)	Space (in bytes)
10000	213.506221	8999100000
10000	215.366478	8999100000

During these trials, roughly 2GB of space was offloaded onto virtual memory. As expected, the time taken to process these graphs was significantly greater than the projected theoretical time.

5 Broader Impacts

There are numerous real world applications for heuristic traveling salesman algorithms. Nearly all logistical services can and do use these algorithms, although this is an obvious application. Uses do, in fact, extend beyond the apparent. With improvements in TSP heuristic algorithms, greater efficiency in networking, drilling circuit boards, and even x-ray crystallography can be attained [2]. These are just a few examples of the many applications of this problem.

6 Proposed Improvement

In my specific case, applying the nearest neighbor algorithm without presorting the graph's edges would have been an improvement to the running time. But looking beyond the scope of my implementation, more accurate algorithms exist to solve this problem. These include the greedy algorithm (O(n^2 log(n)), the nearest insertion algorithm (O(n^2)), and the well known Christofides algorithm (O(n^3)). All of these approaches are more complicated than the nearest neighbour method but usually offer a solution that is closer to optimal. Christofides algorithm can actually guarantee a solution to within 3/2 of optimality.

Tour improvement methods can also be applied to improve the solution. These most commonly use the 2-opt and 3-opt techniques, which remove two or three edges in order to reconnect them if it creates a shorter cycle. Running the 2-opt and 3-opt methods can usually reduce the tour length to 5% and 3% above optimal, respectively.

7 Conclusion

Solving the traveling salesman problem in a reasonable amount time requires the use of heuristics. My implementation of the nearest neighbour algorithm was able to provide an approximation of an optimal result in polynomial time. Presorting edges simplified the code while maintaining the same accuracy, but this was at the expense of a slightly increased time complexity. In any real world implementation, tour improvement methods should be used as they offer a solution that is much closer to optimal.

References

[1] Christian Nilsson. *Heuristics for the Traveling Salesman Problem*. Linköping University

[2] Rajesh Matai, Surya Singh and Murari Lal Mittal (2010). *Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches, Traveling Salesman Problem, Theory and Applications*, Prof. Donald Davendra (Ed.), InTech, DOI: 10.5772/12909.

[3] Leonardo Zambito (2006). *The Traveling Salesman Problem: A Comprehensive Survey*.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#define MAX_BUFF 30
#define FILENAME "cgraph.txt"
#define ITERATIONS 1

int compareweights(const void *a, const void *b);
int isUsed(int vert, int len);
void printEdges(int len);

struct Edge {
	int v1;
	int v2;
	int weight;
};

struct Edge *edges;
int *usedVerticies;

int main()
{
	int nVertices;
	int nEdges;
	int maxIteraton = ITERATIONS;
	int iteration;
	double avgTime = 0;
	for (iteration=0; iteration < maxIteraton; iteration++)
	{
		/*****************************************
		 * READ GRAPH INTO MEMORY
		 *****************************************/
		FILE *fp;
		fp = fopen(FILENAME, "r+");
		char buffer[MAX_BUFF];

		fgets(buffer, MAX_BUFF, fp);
		nVertices = atoi(strtok(buffer, " "));
		nEdges = atoi(strtok(NULL, " "));
		//printf("%d,%d\n", nVertices, nEdges);

		edges = malloc(sizeof(struct Edge) * nEdges);
		usedVerticies = malloc(sizeof(int) * nVertices);

		int i;
```

```
	for(i=0; i<nEdges; i++)
	{
		fgets(buffer, MAX_BUFF, fp);
		edges[i].v1 = atoi(strtok(buffer, " "));
		edges[i].v2 = atoi(strtok(NULL, " "));
		edges[i].weight = atoi(strtok(NULL, " "));
	}
	printf("Graph in memory, Starting Algorithm!\n");

/*********************************************
 * NEAREST NEIGHBOUR ALGORITHM
 *********************************************/

	clock_t begin, end;
	double iTime;
	begin = clock();
qsort(edges, nEdges, sizeof(struct Edge), compareweights);

srand(time(NULL));
usedVerticies[0] = rand() % nVertices +1;

	i = 0;
	int j = 0;
	while(i < nVertices-1)
	{
		if (edges[j].v1 == usedVerticies[i] && isUsed(edges[j].v2, i+1) == 0)
		{
			usedVerticies[i+1] = edges[j].v2;
			i++;
			j=0;
		}
		else if (edges[j].v2 == usedVerticies[i] && isUsed(edges[j].v1, i+1)
==0)
		{
			usedVerticies[i+1] = edges[j].v1;
			i++;
			j=0;
		}
		else
		{
			j++;
		}
	}
	end = clock();
	//printEdges(nVertices);
	//iTime = (double)(end - begin) / CLOCKS_PER_SEC;
	//printf("iTime: %f\n",iTime);
	avgTime = (avgTime*(iteration) + (double)(end - begin) / CLOCKS_PER_SEC) /
(iteration+1);
	}
```

```
        printf("verticies: %d edges: %d\n", nVertices, nEdges);
        int size = sizeof(struct Edge) * nEdges;
        printf("graph size in memory: %d bytes\n", size);
        printf("iterations: %d\n", maxIteraton);
        printf("avg time: %f\n", avgTime);
}

int isUsed(int vert, int len)
{
        int i;
        for (i=0; i<len; i++)
        {
                if (vert == usedVerticies[i])
                        return 1;
        }
        return 0;
}

void printEdges(int len)
{
        printf("Path: ");
        int i;
        for (i=0; i<len; i++)
        {
                printf("%d ", usedVerticies[i]);
        }
        printf("\n");
}

/* Compare method for Edge structs. Used by qsort. */
int compareweights(const void *a, const void *b)
{
        struct Edge *edge_a = (struct Edge *)a;
        struct Edge *edge_b = (struct Edge *)b;

        return (edge_a->weight > edge_b->weight);
}
```