The Traveling Salesman Problem: The Space Filling Curve Approach Using Variations on the Hilbert Curve

Dallan Adamson
dallanadamson@gmail.com

Dallin Seyfried
dseyfr99@gmail.com

Conner Hammond
reuelerman@gmail.com

Spencer Fairchild
fairchildspencer@gmail.com

April 7, 2022
Professor Tony Martinez
BYU, Computer Science

Abstract

The traveling salesman problem is one of the most intensely studied topics in the computer science world. Finding a polynomial solution to this problem would result in a domino effect of breakthroughs in all kinds of other similar problems. Our team tried to find a unique solution to this problem through the use of space filling curves as an estimate of the optimal order of traversal. We mapped all of the cities to a collection of four interconnected Hilbert Curves (Wikimedia) and traversed the cities in order of where they fell on the curve. The result is an algorithm that can solve TSP problems with tightly grouped cities extremely fast with a very good approximate solution. Even with problems that were not tightly grouped our algorithm consistently outperformed the basic greedy algorithm in terms of finding path length and did it in a fraction of the time.

1 Introduction

This report centers on the algorithm created by the authors which solves the Traveling Salesman problem. Different aspects of the authors' algorithm will be analyzed including its functionality, adaptations from other designs, flexible parameters that influence outcomes, design and implementation processes leading to the current algorithm structure, and the effectiveness of the algorithm as compared to a greedy, random, and branch & bound approach. Information will also be given on the usefulness of the author's algorithm as well as improvements and possible future iterations.

2 The Greedy Algorithm

2.1 Implementation

Finding a path for TSP using the Greedy Algorithm involves starting from an arbitrary city (when we ran our tests we just started with the first generated city) and then takes the shortest path leading out from it to the next city. This process is repeated from city to city until all cities have been visited and a path generated.

The source code for the Greedy Approach can be found in the attached files of this report.

2.2 Purpose as Benchmark

The simple implementation and complexity class of the Greedy Algorithm provides a great benchmark for other TSP Algorithms. Because of this it is able to serve as a good average between speed and accuracy.

2.3 Complexity Analysis

The Greedy Approach for TSP consists of the following steps that it goes through during runtime given a list of cities:

- Cycle through every city starting at the first city - O(n)
- Compare each city to every other city to find the shortest untraveled path - O(n)
    - The two points above form a double nested loop that takes O(n^2) time

    - Calls to a costTo function which returns the cost of a edge in constant O(1) time
    - Append shortest edge's city to route - O(1)
- Return the route O(1)

Therefore, the Big-O space complexity of the Greedy Algorithm is O(n) and the time complexity is O(n^2).

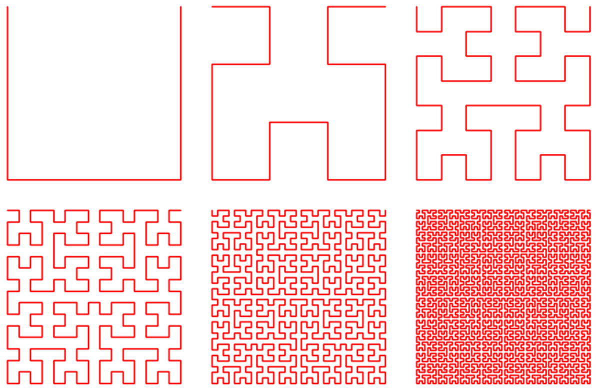
3 Space Filling Curve

3.1 Explanation of Why and How the Algorithm Works

The idea behind the space filling curve heuristic to solve the TSP problem is mapping all of the cities onto the closest point on a space filling curve and then traversing the cities in the order they fall on the curve (or as close as possible to the order they fall). The curve that we chose to use in our algorithm is the Hilbert Curve. The Hilbert curve is a continuous fractal space filling curve that increases in length by

$$2^n - \frac{1}{2^n}$$

for the nth curve while still remaining inside a square of the set area. You can see the first six iterations of the Hilbert curve demonstrated in fig. 1.


(fig. 1)

The way our algorithm utilizes this curve is by choosing a high enough order curve (i.e. setting n to a large enough number) and then translating 4 of these curves so that they are all interconnected and begin where they end. This allowed us to easily map all of the 3D city locations into 1D space and traverse them in order. This method results in the traversal of cities close together in space being a priority. If the cities are originally grouped close together this method approximates the optimal solution much better than if they are far apart. Because the mapping from 3D space to 1D space is extremely fast, this algorithm approximates a solution extremely quickly. Although this algorithm is unlikely to give you the optimal solution it does approximate it decently well in a fraction of the time of other algorithms.

3.2 Implementation

Our implementation of this algorithm was fairly simple. We started off by traversing all of the cities and translating their x/y coordinates based off of which quadrant they fell in. We translated the coordinates so that they would work with the rotated hilbert curves. Essentially all of the Hilbert curves had to be rotated so that their start and end points lined up. Instead of actually building these curves this way, we just translated the points so that they would match up correctly if we were to build those curves. We then passed these points into a function that would convert an x/y coordinate into a Hilbert curve index (the code we used to do this was adapted from the C code found here) (Wikipedia). Since we had already translated the coordinates we could use the same function no matter what quadrant the city fell in and still get a correct index on the curve. After the cities were assigned an index on the curve we traversed them in the order of their new indices. Because we would basically smash all of the coordinates down to 2D (ignoring elevation), this led us to a problem with some cities being unreachable from other cities. The way we handled this was with a queue of cities that had not yet been visited but were a priority. Every time we could not visit the next city in order, we just added it to the queue and moved on to the city after that. Before trying to visit any new cities we would always try to visit any cities left on the queue. This implementation resulted in the possibility of a solution not being found if the final city did not connect to the initial city. To solve this problem we ran the algorithm for building the path based off of the Hilbert indices 4 times, starting with different quadrants. This resulted in an extremely good likelihood that a solution would be found while still resulting in a very fast algorithm.

The source code for our implementation of the Space-Filling Curve Approach can be found in the attached files of this report.

3.3 Complexity

In general, the time complexity of our algorithm is quite fast at O(n log(n)), but there are some exceptions to that which will be discussed. For strongly connected graphs, where every city is connected to every other city, our algorithm is quite efficient in that it only has to read through every city once and map it to the order of the curve. The mapping function we use is O(log(n)) time because it searches distances on the curve by cutting it in half each time. This mapping is done for every city once, giving a time complexity for strongly connected graphs of O(n log(n)). The other pieces of the algorithm fall under O(n log(n)) time as they are linear searches through structures we've already created such as the already mapped cities, or finding the borders of the graph which the curve needs to fill..

Further complexity is brought to the problem when graphs are weakly connected. If a city cannot be visited by the previous city on the curve because it's not connected, it's added to a queue which is then checked at each of the cities down the line as the curve continues. In the worst case the cities could be ordered and connected in such a way that they could only be traversed exactly backward to the direction of starting the curve, leading us to add every city to the queue, and search the whole queue at each city. That would be O(n) cities searched at each of the n cities, leaving O(n^2) as the time complexity.

The space complexity of the algorithm is quite simple since we are storing different orderings of the cities in lists. Since we never store an n multiple of the cities, our overall space complexity will simply be a scalar of O(n). The structures needed and used in our algorithm include a list of cities, a growing list of cities in the order the curve visits them, and a queue of cities that have been unable to be visited (which will have a maximum of n cities). The other pieces of the algorithm are simple formulas which determine where on the curve cities lay, and don't bear relevance to the space complexity.

3.4 Pros and Cons

Pros:

- **Extremely fast**

After running several trials the time complexity of the Hilbert curve algorithm is close to constant. If the amount of points you have is in the thousands our algorithm will run quickly and since the points will be closer together the algorithm will give a closer approximation to the optimal solution. Sometimes it is better to take the less optimal solution because time to find a better solution would be better spent actually traveling the cities ("Some Combinatorial Applications of Spacefilling Curves").

- **Very good for well connected graphs**
  Our algorithm excels at finding points near each other. If there are more connections between cities then the probability that our algorithm will find an optimal solution increases.
- **The more dense the cities, the closer to optimal**
  The idea behind the Hilbert curve is that the curve tries to visit points near each other first before points far away from it. Since our hilbert curve visits nearby cities first, the algorithm finds a solution close to the optimal when cities are densely packed.

Cons:

- **Unlikely it finds the optimal solution**
  Hilbert struggles to find the optimal solution when there aren't dense groups of cities. The hilbert curve also struggles with points along the axis since they are in the middle between two curves. This can lead to inefficient results as the curve assigns those cities to a spot on the curve away from potential neighbors in another curve.
- **If cities are not connected to other cities close by it becomes less optimal**
  Since our algorithm tries to match cities with others close to them, it increases the time if multiple cities near each other are not connected. One of the cities will have to be set in a queue to try and find the next closest neighbor with a path to that city. The worst case scenario is if every city was only connected to the point farthest away from it. This would lead the algorithm to have to fit every point in the wait queue and would be a time complexity of N squared.
- **Does not handle weighted/negative edges - purely a Euclidean algorithm**
  The space curve doesn't look at weight to determine which path to use; it strictly uses where a point is in space, therefore if there were negative or weighted edges it could not accurately assess the problem and give a good approximate solution

3.5 Future Work

Future work for this algorithm would include exploring ways to make the space-filling heuristic work for a graph with weighted edges if such a modification were possible.

Another interesting proposition (Platzman 724) is that our algorithm is bounded by a Big-O time complexity of O(nlog(n)) arithmetic operations, but if we were able to find a way to draw the fractal space-filling curve from start to end and implement a binsort by approximating each city's coordinates to a vertex on the curve then we could effectively sort them from there and generate the solution in O(n) time.

It would also be intriguing to see if we could use the space filling curve to enable an implementation of Branch and Bound/Beam search for when it reaches a number of cities higher than what it currently can compute before timing out. Using the space filling curve to find the initial best solution would enable an actual solution to be returned quickly and thus begin effective pruning earlier on than what it would be able to do with starting with Greedy or Random where both time out at a high number of cities where it is a hard problem set.

4 Empirical Analysis of Algorithms

4.1 Tabular Results (See end of section for combined table)

Random

# of Cities	Speed	Path
15	.0032	20105
30	.038	38888.2
60	40.95	76459
100	TB	TB
200	TB	TB

Greedy

# of Cities	Speed	Path	% of Random
15	.00037	11344.4	.5642408085
30	.0017	19067	.4903029711
60	.0048	26638.6	.3484037196
100	.012	37481.6	-
200	.077	48888	-
1000	6.2412794	160938.2	-
2000	46.52	246089	-

10000	TB	TB	-

Branch and Bound

# of Cities	Speed	Path	% of Greedy
15	9.30	9630.4	.8489122386
30	TB	TB	
60	TB	TB	-
100	TB	TB	-
200	TB	TB	-

Group TSP

# of Cities	Speed	Path	% of Greedy
15	.00044	14802.6	1.304837629
30	.0010	26660	1.398227304
60	.0016	44855.6	1.68385726
100	.0023	73346.8	1.95687484
200	.0043	89727	1.83535837
1000	.023	680990.6	4.231379498
2000	.061	1381137	5.61247565
10000	.57	6831362	-
50000	1.74	34140552	-

4.2 Analysis of Results

Analyzing the data gathered we are able to see that our algorithm does run in a variation of O(n log(n)) time with a constant multiple of around 1/80,000 with the machine we chose to do our testing on. In fact it actually appears to run in O(n) time since we hold the arithmetic operations to a constant number (which is what the log(n) in our complexity of O(nlog(n)) is based off of). If we allow the arithmetic operations to expand as the problem size expands then we would start to see more of the O(nlog(n)) theoretical time complexity. When comparing this speed to other algorithms, we are able to see that it is the fastest, though the difference in speed will be most applicable when using our algorithm on datasets with large numbers of cities. 2000 cities for example timed out on the random generator, and greedy took 46 seconds compared to our 6 hundredths of a second. This immediate return of a result can be very valuable with large sets of data, but it does come at the cost of some accuracy.

Branch and Bound can be seen to find the most optimal path if given enough time, but is incredibly slow, not even finishing during our test periods with 30 cities. Our space filling curve algorithm is able to handle those larger datasets, but can also be seen to give a less optimal solution. For sets of data in the range beyond 30 nodes, our space filling curve is mainly competing against greedy, since random is still slower and very unlikely to give anywhere near an optimal result. Greedy does tend to get a more optimal path, but again it is slower than our algorithm. This leaves the user of the algorithms a choice between optimality and speed. In some cases, it may be better to take a less optimal solution and start working on it sooner, rather than to have an optimal solution which took so long to derive that the less optimal solution could already have been found and traveled.

Discussing the variation between a greedy solution and what our algorithm would get is difficult as our algorithm's solution has the capability of being better than greedy, though that won't frequently be the case. Previous research has shown that space filling curves are guaranteed to have a solution within O(log n) of the optimal (Buchin 14). That guarantee coupled with the instantaneous solution our algorithm gives makes a strong case for using our modified Hilbert curve algorithm.

Combined Table of Empirical Results

	Random		**Greedy**			**B&B**			**Space-Filling Curve**		
# of Cities	Time (Sec)	Path Length	Time (sec)	Path Length	% of Random	Time (sec)	Path Length	% of Greedy	Time (sec)	Path Length	% of Greedy
15	.0032	20105	.00037	11344.4	.5642408085	9.30	9630.4	.8489122386	.00044	14802.6	1.304837629
30	.038	38888.2	.0017	19067	.4903029711	TB	TB		.0010	26660	1.398227304
60	40.95	76459	.0048	26638.6	.3484037196	TB	TB	-	.0016	44855.6	1.68385726
100	TB	TB	.012	37481.6	-	TB	TB	-	.0023	73346.8	1.95687484
200	TB	TB	.077	48888	-	TB	TB	-	.0043	89727	1.83535837
1000	TB	TB	6.2412794	160938.2	-	TB	TB	-	.023	680990.6	4.231379498
2000	TB	TB	46.52	246089	-	TB	TB	-	.061	1381137	5.61247565
10000	TB	TB	TB	TB	-	TB	TB	-	.57	6831362	-
50000	TB	TB	TB	TB	-	TB	TB	-	1.74	34140552	-

References

Buchin, K. “Space-Filling Curves.” *Refubium*, 2008, https://refubium.fu-berlin.de/.

Grigni, M., & Bertsimas, D. (1988). (tech.). *On the Spacefilling Curve Heuristic for the Euclidean Traveling Salesman Problem* (pp. 1–6). Cambridge, MA: MIT.

Platzman, Loren K., and John J. Bartholdi. “Spacefilling Curves and the Planar Travelling Salesman Problem.” *Journal of the ACM*, vol. 36, no. 4, 1989, pp. 719–737., https://doi.org/10.1145/76359.76361.

“Some Combinatorial Applications of Spacefilling Curves.” *Home Page for Spacefilling Curves and Applications*, 11 Aug. 2012, https://www2.isye.gatech.edu/~jjb/research/mow/mow.html.

Wikimedia Foundation. (2022, April 4). *Hilbert curve*. Wikipedia. Retrieved April 9, 2022, from https://en.wikipedia.org/wiki/Hilbert_curve