EC504 Project Report: Traveling Salesman Problem

Authors:

Songlan Wang	BU ID: U96545272	SSC Username: slanwang
Ye Zhou	BU ID: U76858097	SCC Username: ninazhou
Nicholas Sacco	BU ID: U63377563	SCC Username: nsacco
Wenjun Zhang	BU ID: U53328342	SCC Username: wjz
Tianhao Yao	BU ID: U85248058	SCC Username: yth0922

Abstract

The *Traveling Salesman Problem* (TSP) is a classic combinatorial optimization problem with a simple description, but no known efficient solution. Assume there are N cities, each accessible from every other city. Let $d(i, j)$ be the distance between cities i and j. For simplicity, we assume that the distance metric is the 2D-Euclidean distance, defined as:

$$d(i, j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Where (x_i, y_i) are the coordinates of city i. Since every city is accessible to every other city, we have that $d(i, j) < \infty \, \forall j \neq i$. Our objective is to find the **shortest path that visits every city exactly once, and returns back to the starting city.** We can reframe this problem using the language of graph theory - given a complete graph with N nodes, $O(N^2)$ edges, and edge weight $d(i, j)$, what is the shortest **Hamiltonian cycle** that can be found in the graph?

Despite this simple problem construction, there is **no known efficient (i.e., polynomial time) algorithm** that solves the TSP. It belongs to the **NP-complete** complexity class, a class of problems considered to be the most difficult problems to solve. An interesting property of these problems is the fact that efficient, polynomial time reductions exist that transform problems in this complexity class into a wide variety of similarly difficult problems. Thus, if we could find a polynomial time algorithm to any one of the NP-complete problems, we could efficiently solve **any other** hard problem. (In this case, the hardest problem one might still have to solve is the question of what to do with all of the prize money).

In spite of the difficulty of exactly solving the TSP, we can actually obtain very good solutions that are close to the optimal solution (i.e., close to the shortest path length) through the use of efficient *approximate algorithms*. In this project, we consider five main algorithms (1 exact, 4 approximate) for approximating the solution to the Traveling Salesman Problem:

- Dynamic Programming
- Simulated Annealing
- k-nearest neighbor algorithm
- Lin-Kernighan Algorithm
- Christofides Algorithm

We test each algorithm at different system sizes, ranging from 15 cities to ~71,000 cities. We use randomly generated city data as well as the TSPLIB datasets for benchmarking our solutions against known results. We collect various statistics, including run-times, estimated path length and associated errors. We include various visualizations of the computed paths to better illustrate the complexity of this problem for large system sizes.

Instructions

1. Navigate to the GitHub repository (link) and clone.

2. There are three main directories: the **include** directory, **src** directory, and **test_data** directory. The **include** and **src** directories contain the application code implementing the proposed algorithms. The **test_data** directory contains sample city configurations in the `.tsp` format. Datasets specifically designed for the Traveling Salesman Problem can be found online, such as (link). Some small custom datasets can be used, but they must be stored in the `.tsp` format to be properly parsed by the application.
3. Inside the **src** directory is the **main.cpp** file; some algorithms require setting parameters, which can be modified within the **main.cpp** file. Using the default parameters requires no modification to the **main.cpp** file and is recommended for the first use.
4. To compile the code, we recommend navigating to the top level of the hierarchy and using the following command:

   ```
   g++ ./src/*.cpp -o run -std=c++11
   ```

5. Once the code has been compiled, execute the code by running the command:

   ```
   ./run ./test_data/<file_name>.tsp
   ```
6. As an example, to run the code on the `ei8246.tsp` dataset, run the command:

   ```
   ./run ./test_data/ei8246.tsp
   ```
7. The code will automatically reject datasets that are too large for certain algorithms; for example, the dynamic programming algorithm cannot be used for datasets much larger than 100 cities; thus, for thorough (and quick!) testing of the software, we recommend using smaller datasets. The simulated annealing algorithm can run using the larger datasets, if desired. Using the ei15.tsp file to test dynamic programming is suitable.
8. Several visualization scripts were provided. To run the MATLAB code, just open up the script and run. Two scripts were provided for visualization of the simulated annealing algorithm: `tsp_dynamics.m` and `tsp_plot.m`. The `tsp_dynamics` script will plot the evolution of a path over the course of the algorithm, while the `tsp_plot.m` will provide summary statistics for tests completed using the simulated annealing routine.

Instruction for plot.py/plot_head.py:
Both plot.py and plot_head.py can be used with *python plot.py/plot_head.py inputfile outputfile* like python plot_head.py ../test_data/ei8246.tsp LkResult.tsp

If the input tsp file have first 7 rows such as:
NAME : ei8246
COMMENT : 8246 locations in Ireland
COMMENT : Derived from National Imagery and Mapping Agency data
TYPE : TSP
DIMENSION : 8246
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
we use plot_head,py

Else we use plot.py to plot tsp files.

<u>Results and Discussion</u>

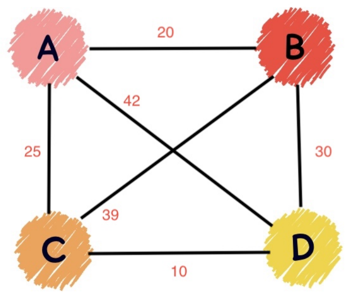
Brute force:

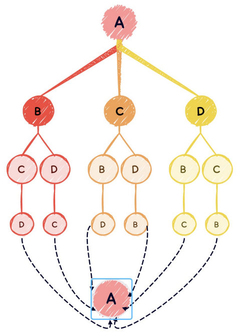
Brute force is the most straightforward way to get the exact solution of the traveling salesman problem. It basically just tries every unvisited node and compute the cost of the whole visiting to decide which one is the shortest path.

For example, suppose we have an undirected complete graph like the following and we will start from the city A to travel to every other cities exactly once and finally come back to the starting node, which is A in this example.

To solve this question, we can draw the figure of every possible paths and each brach denotes one possible path in the following graph. For city A, the salesman can go to B or C or D. And for city B, C, and D, they all can go to other two unvisited cities and then the last unvisited city and finally come back to A. So the whole process will be counting every path's cost (including A—>B—>C—>D—>A, A—>B—>D—>C—>A, A—>C—>B—>D—>A, A—>C—>D—>B—>A, A—>D—>B—>C—>A, A—>D—>C—>B—>A) and pick the shortest one.

The time complexity for the brute force method will be O(n!).





Dynamic programming:

Dynamic programming method is also aimed to find the exact solution. What makes it different comparing to the brute force method is that dynamic programming method can
store useful data in the dynamic programming table so that we will look up the table first to check if the data we want has already been calculated, if yes then we can avoid doing the calculation again.

Taking the above picture as an example,

- Divide TSP into subproblems
- Solve the base case: if (all_visited) return A
- Get the recurrence function:

g(A,{B,C,D}) = Min { dist[A][B] + g(B, {C,D}), dist[A][C] + g(C, {B,D}), dist[A][D] + g(D, {B,C}) }

The most important thing is that if we already have the minimum cost from B to {C, D} and then return to A is the path B—>C—>D—>A, then when we compute the minimum cost from A to {B, C, D}, it's reasonable to discard the route A->B->D->C->A, which will save us time.

Another trick to improve efficiency is using bit operation. We can use bit mask to represent the cities, especially the city state like if city A has been visited. For instance, the ith bit '0' represents the ith city has not been visited while the ith bit '1' represents the ith city has been visited (checking from right to left and beginning with city 0), such as '0010' denoting the city

1 has been visited and ‘0101’ denoting the city 0 and city 2 has been visited. In this way, we can easily know if the ith city is visited or not.

The time complexity for the dynamic programming method will be O(n^2 * 2^n)

How to execute the dynamic programming part code?
g++ -o test main.cpp
./test ../test_data/ei15.tsp >res_15.tsp
Due to the huge time complexity of dp method, if you test to run the code on a big dataset like ei131.tsp or ei8246.tsp, it would be skipped. So please test dp method with ei15.tsp.

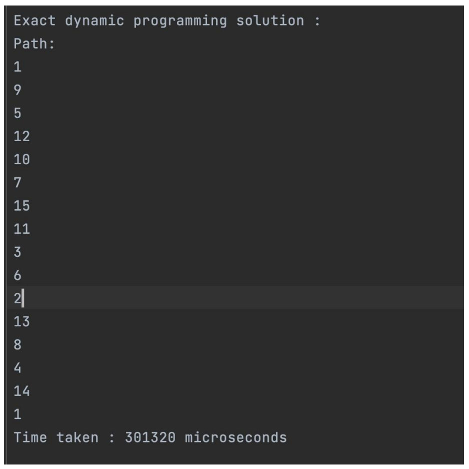
Results:
res_15.tsp (time for running the function, shortest path for 15 cities)
dp_result.tsp (data for plotting)

It’s supposed to have 2 .tsp files which the first one(res_15.tsp) concludes all information about the shortest path for 15 cities, time for running the dynamic programming method and so on; the second one(dp_result.tsp) concludes only the data for plotting.

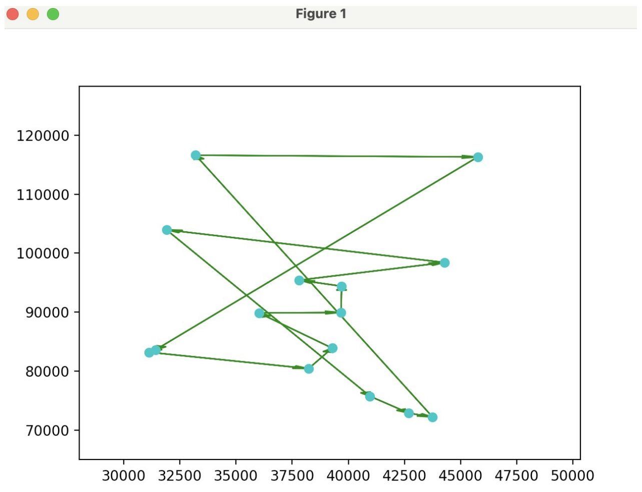
How to plot?
Python plot.py [input file] [output file]
Plot.py is a python script for plotting the path of traveling salesman problem.

Example: python plot.py /Users/zhouye/Documents/GitHub/EC504_TSP/test_data/ei15.tsp /Users/zhouye/Documents/GitHub/EC504_TSP/src/dp_result.tsp

Dynamic Programming results:
Time taken for a dataset with 15 cities: about 0.301320 seconds.
Time taken for a dataset with 25 cities: about 639.218269 seconds.



```
Exact dynamic programming solution :
Path:
1
9
5
12
10
7
15
11
3
6
2
13
8
4
14
1
Time taken : 301320 microseconds
```

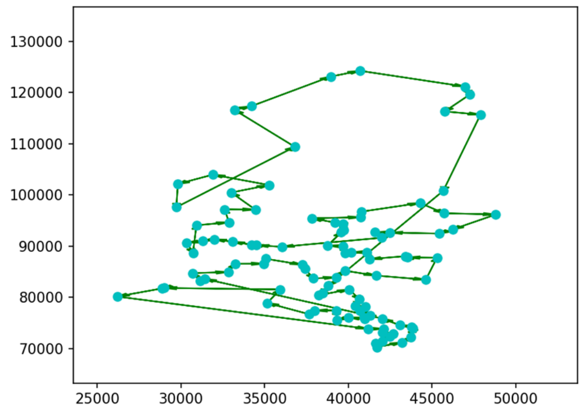


KNN

The nearest neighbor algorithm is a kind of greedy algorithm. It lets the salesman choose the nearest unvisited city as his next move. This algorithm quickly yields an effectively short route. The pseudo code is shown as follow.

1. Initialize all vertices as unvisited.
2. Select an arbitrary vertex, set it as the current vertex u. Mark u as visited.
3. Find out the shortest edge connecting the current vertex u and an unvisited vertex v.
4. Set v as the current vertex u. Mark v as visited.
5. If all the vertices in the domain are visited, then terminate. Else, go to step 3.

Since we need to iterate all the vertices and scan each vertice during each iteration, the time complexity is clearly O(n^2).



Graph for 100 cities

The KNN algorithm is easy and quick, but it can sometimes miss shorter routes due to its "greedy" nature. Generally, if the last few stages of the tour are comparable in length to the first stages, then the tour is reasonable; if they are much greater, then it is likely that much better tours exist. In the result for 100 cities, the last few tours are much longer than the first stages, so it might not be a perfect result.

Lin-Kernighan algorithm:

The core idea of Lin-Kernighan algorithm:

Derive from graph partitioning problem and the basic approach:

1. Generate a pseudorandom feasible solution, that is, a set T that satisfies it is a tour.
2. Attempt to find an improved feasible solution T' by some transformation of T.
3. If an improved solution is found, i.e., f(T') <f(T), then replace T by T' and repeat from Step 2.
4. If no improved solution can be found, T is a locally optimum solution. Repeat from Step 1 until computation time runs out, or the answers are satisfactory.

Details of step2:

Suppose that g is the profit or 'gain' associated with the exchange of x and y. Each exchange with x and y, we compute the sum of g=|x|-|y|.

When the sum of g is maximum, which means for such a starting point, no further improvement can be achieved.

After this, iterate from the new starting point until no further reduction.

Cease at $G = \sum_{1}^{i} g < 0$

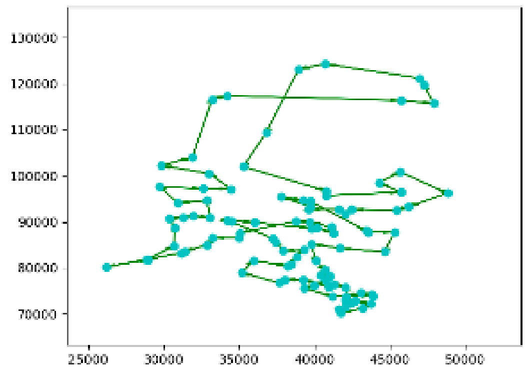
Lin-kernighan results:

Test with 100 cities tsp file and use matlibplot pyplot to generate the path.

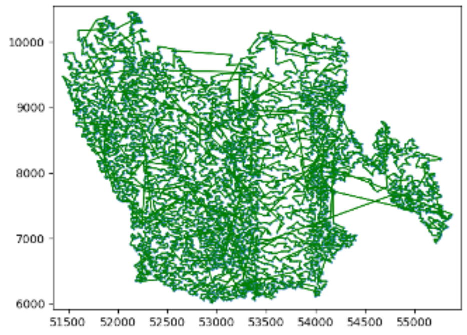
And the time of processing 100 cites using Lin Kernighan is

0.016 seconds.When it comes to 1000 cities,it will cost about 1.6 seconds. The result confirms ***<u>O(n^2.2)</u>***

Generated graph for 100 cities using Lin-Kernighan



Generated graph for Ireland 8246 cities using Lin-Kernighan



Christofield's algorithm:

The core idea of Christofield's algorithm:

1. Build a Minimum spanning tree based on the original graph.
2. Find all the nodes within an odd number of edges.(*Double Tree algorithm)
3. Find the best matching that can connect those odd-degree nodes into even-degree nodes.(*Double Tree algorithm)
4. Find a Eulerian paths according after the best-matching
5. Find a Hamlitonian paths

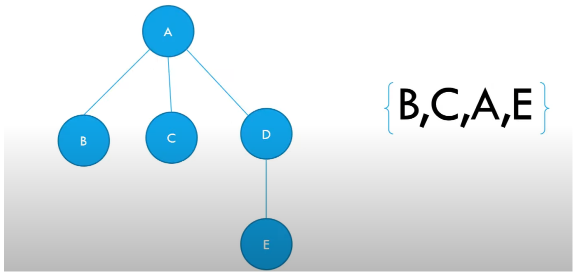


Figure.1

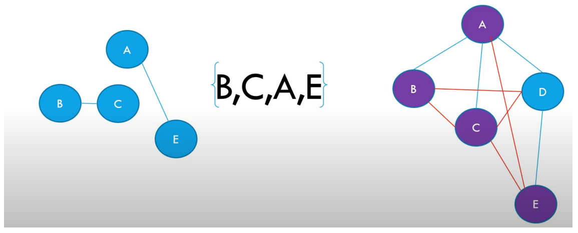


Figure.2

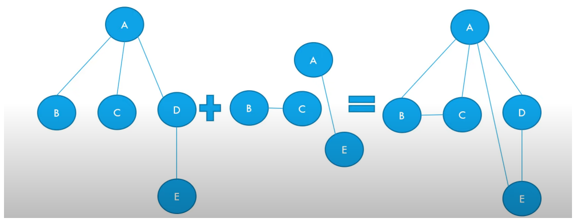


Figure.3

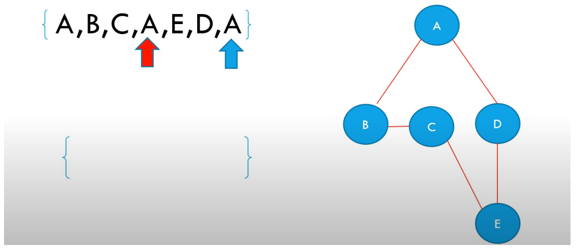


Figure.4

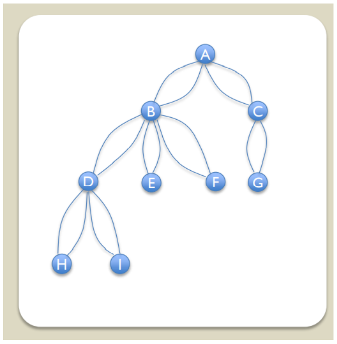


Figure.5

Discussion:

For the first step, we can easily find the MST by using Prim's algorithm or Kruskal’s algorithm. For this project, I am using Kruskal’s algorithm as my MST algorithm.

For the second and third steps, I do not include them in my first solution(I will talk about this in the following section), I use the Double Tree algorithm to replace them. The basic idea is to treat all the edges as directed edges, and add the opposite direction edges to the same two nodes. To realize this I use stack to make it happen.

For the fourth step, there are two ways to travel the graph in my two solutions. In my first solution because it’s stack, so first in, last out. In the figure 5, the Eulerian path will be different with solution two: “ACGCABDHDBEBFBA”. Instead, in solution two. Because we are using the real christofield’s algorithm, then it will be formed to the figure.3, which is “ABCAEDA”

For the last step, we already have the Eulerian path, so what we need to do is to erase those duplicate traveled nodes and we will have the Hamiltonian path. In my solution one, the Hamiltonian path will be “ACGBDHEFA”, and for my second solution this graph will be simply just :”ABCEDA“.

Theorem vs. Result:

For Christofield’s algorithm, there are few theorems. It can guarantee 3/2 length of the optimal paths if its best matching part is implemented. However in my result it does not perform well as it describes. I guess this is because my best matching strategy and Eulerian Travel is bad. Figure.6 is the result of using the Double Tree algorithm to examine the path of XQF131.tsp, The optimal path length is 564, for my first solution I have 1.1e+03. Figure.7 is using normal Christofield’s algorithm to examine the path of XQF131.tsp. For my second solution, I have 9.8e+02, which is way better than the Double Tree algorithm. Even though, it still does not fulfill 3/2 length of the optimal theorem.

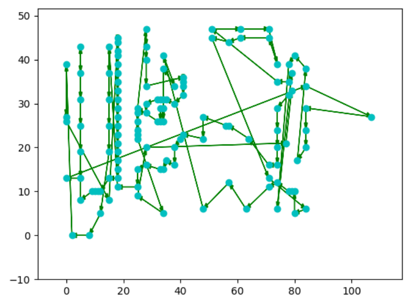


Figure.6

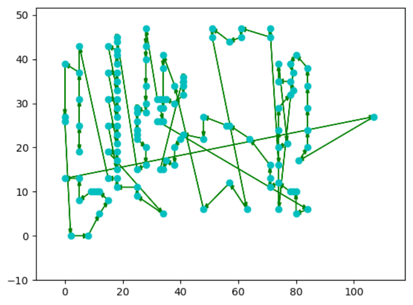
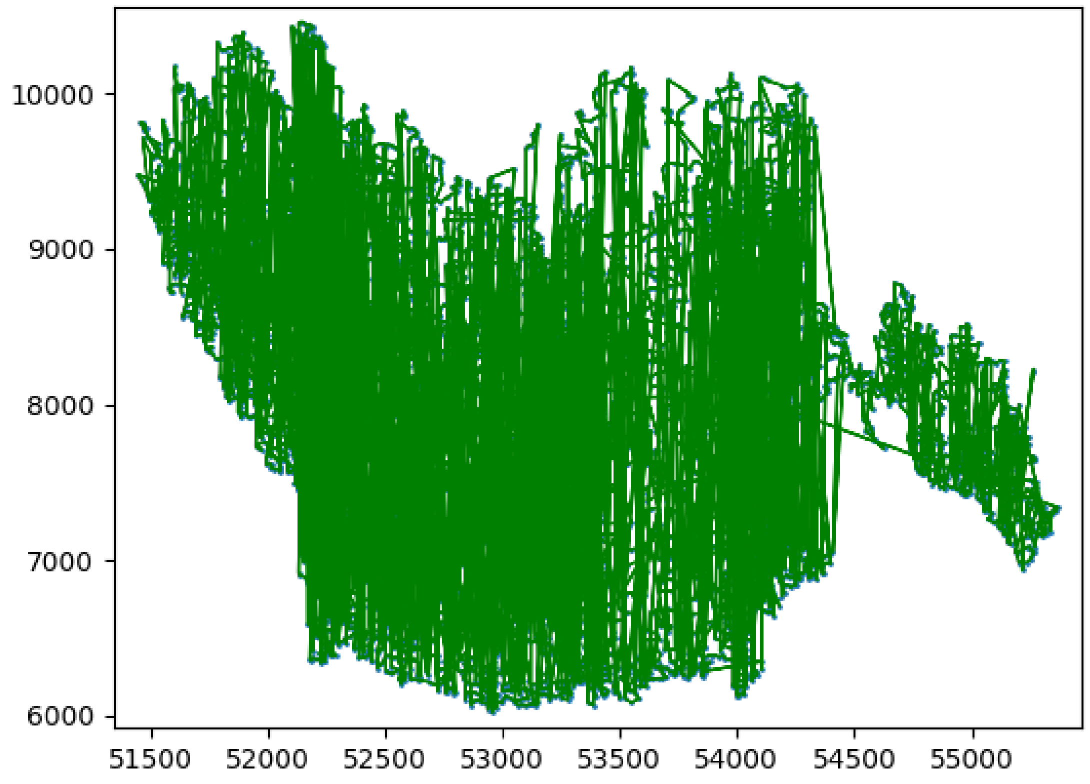


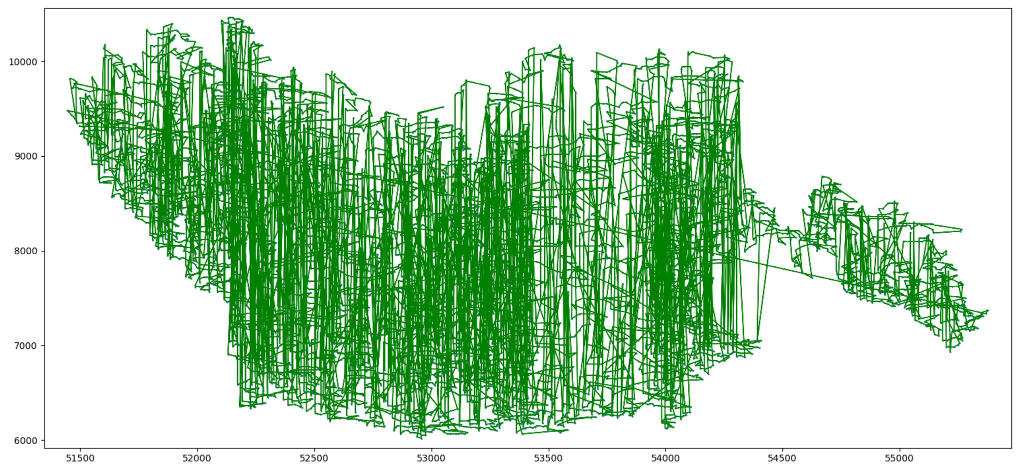
Figure.7

We will always find even numbers of odd-degree nodes to connect each other. The reason is if we make a MST, there must be a leaf that only has one degree of edge. Also adding all the degrees together we will get two times of edge numbers, this means that if there is an odd-degree node, there should be one other node with odd degree of edges. For this part I am good.

Generated graph for Ireland 8246 cities using Christofield's Algorithm:



Generated graph for Ireland 8246 cities using Christofield's Algorithm implemented by using Double Tree Algorithm:



Simulated Annealing

Simulated Annealing is a broad category of optimization algorithms inspired by physical phenomena. We can reframe many optimization problems in terms of energy - the function we wish to minimize describes the "potential energy landscape," possibly with many local and global minima. Finding an optimal solution to the optimization problem requires "descending" through the energy landscape. The simulated annealing algorithm runs for T iterations; morally, at each time step we perform the following actions:

1. Propose a new configuration S_t drawn from the state space according to some principled **update rule**
2. Compute the energy E_t of the proposed configuration S_t
3. Compute the **change in energy** from the current configuration to the proposed configuration: $\Delta E = E_t - E_{min}$
4. If $\Delta E < 0$, then we have proposed a lower-energy configuration - we always **accept** configurations that improve the current state by lowering the energy: $S_{min} = S_t$, $E_{min} = E_t$
5. If $\Delta E > 0$, then we have proposed a higher-energy configuration - we accept the new configuration with some small probability. We are motivated to occasionally accept "bad"

energy configurations if they enable further state exploration, potentially helping the algorithm "bounce" out of a local minimum and continue descending to a better solution. Thus: $S_{min} = S_{t'}$ $E_{min} = E_t$ with probability $p = \exp(-\Delta E/T)$

6. "Cool" the temperature T - we want to reject most changes at low temperatures because we generally believe the current configuration proposed by the algorithm (S_{min}) is close to the actual optimal configuration (S_*), and thus we don't require as much exploration as we do refinement.

The steps described in this algorithm are quite general, which is why simulated annealing is such a powerful algorithm for finding near-optimal solutions to optimization problems. The only problem-specific parts of the algorithm we must design are the **update rule** and **cooling schedule**; otherwise, nearly every single problem that can be reframed as an "energy" problem can be run using some variation of the above steps!

Some simulations results are shown below:

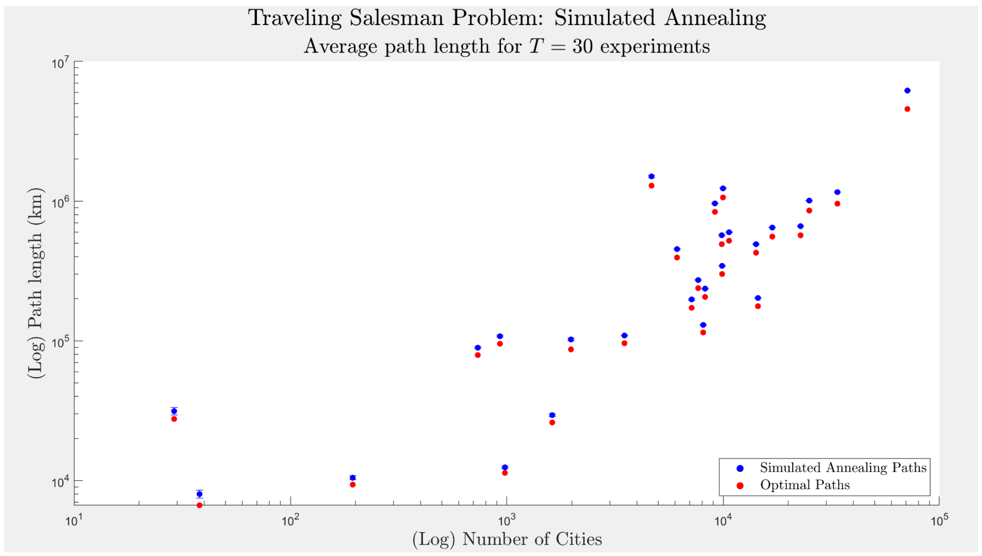


Fig 1: Simulated Annealing Exhaustive Test Suite - Run simulated annealing 30 times on TSPLIB datasets. The paths generated by the algorithm are shown in ***blue****, while the optimal paths are shown in* ***red****. As expected, the simulated annealing results overestimated the optimal path length, but managed to provide a decent solution for datasets up to ~71000 cities.*

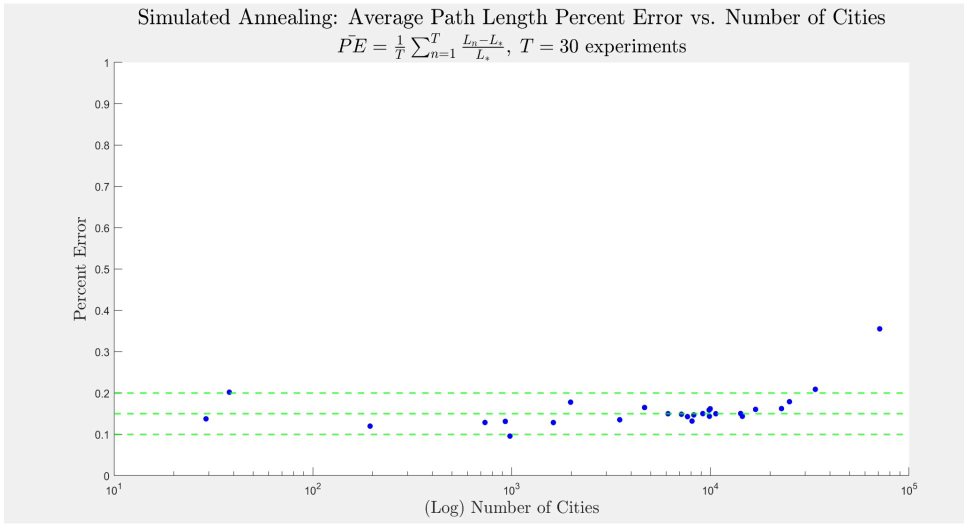


Fig 2: Simulated Annealing Exhaustive Test Suite - Errors for the 30 experiments. For most of the simulations, the paths proposed by the simulated annealing algorithm were about 15% longer than the optimal path. The error tended to increase with the number of cities, which is also expected. An important point to note here is that all datasets were tested using the same parameters proposed in the “Simulated Annealing” paper; to be very thorough, a follow-up to this experiment would propose a “stronger” set of parameters that, while more computationally and time expensive, would result in paths closer to the optimal length. This demonstrates an important part of the simulated annealing class of algorithms - we have control over parameters, so if we determine that a particular solution is not strong enough, we can always keep running with stricter parameters or new update rules to find a better solution!

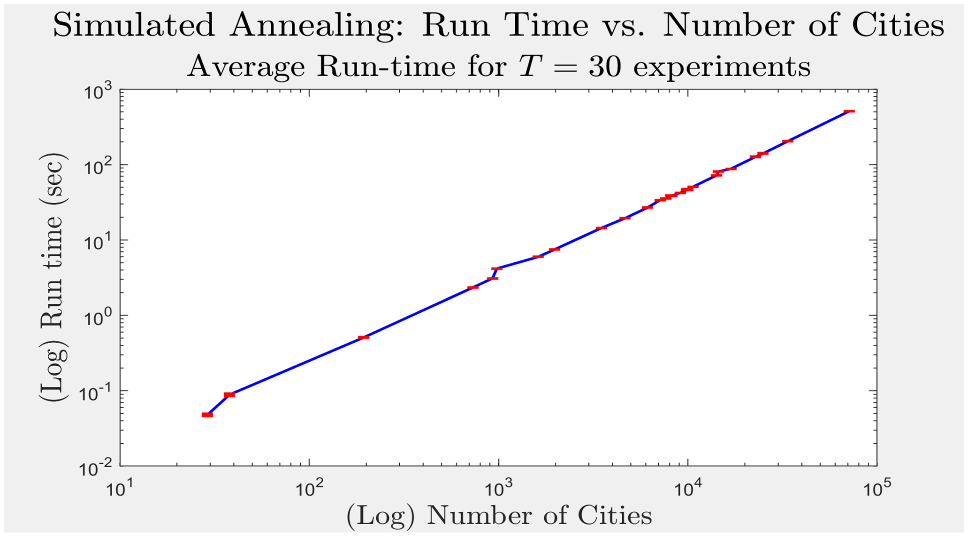


Fig 3: Simulated Annealing Exhaustive Test Suite - Average run-time for the 30 experiments. As expected, the run-time increased quite steadily for large number of cities. The theory suggested the run time behaves as $O(N^2 \log N + N \log N)$ while the empirical results might suggest an exponential-ish run-time. Further testing is required to confirm the empirical asymptotics of the algorithm agree with the theoretical asymptotics. Regardless, the run-times

for each TSPLIB dataset are manageable given current computing hardware; in fact, it is still probably feasible to push the number of cities even higher and still achieve a near-optimal path length in a reasonable amount of time.

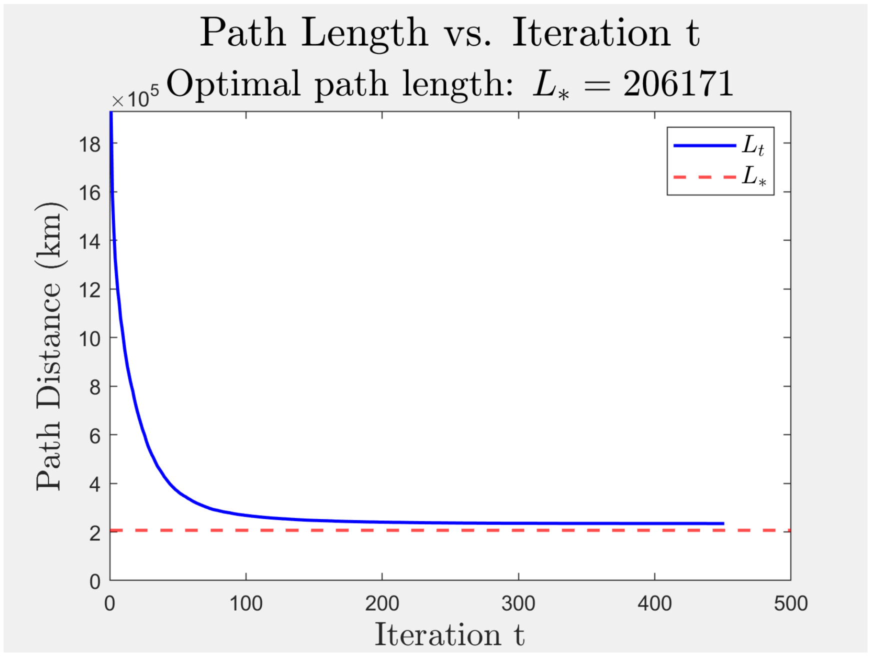


Fig 4: Sample Path Length Evolution for Ireland, N = 8246 cities. As shown above, the simulated annealing algorithm can quickly generate a near-optimal path; most changes are accepted early on in the algorithm to quickly explore the space, while most changes are rejected later on in the algorithm, as evident by the flattening of the path length curve. If we ran the algorithm for more iterations, we would see the path estimate (blue) continue to approach the optimal path (red).

References

[1] https://web.stanford.edu/class/cme334/docs/2012-11-14-Firouz_TSP.pdf

[2]https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm#:~:text=The%20Held%E2%80%93Karp%20algorithm%2C%20also,to%20find%20a%20minimum%2Dlength

[3] https://github.com/search?q=tsp+dynamic+programming

[4] http://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf

[5] https://arthur.maheo.net/implementing-lin-kernighan-in-python/

[6] https://www2.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html

[7] https://github.com/topics/lin-kernighan-heuristic

[8] An Effective Heuristic Algorithm for the Traveling-Salesman Problem Author(s): S. Lin and B. W. Kernighan Source: Operations Research, Vol. 21, No. 2 (Mar. - Apr., 1973), pp. 498-516 Helsgaun, K., 2000. An effective implementation of the Lin–Kernighan traveling salesman heuristic. European journal of operational research, 126(1), pp.106-130.

[9] Hansen, Per Brinch, "Simulated Annealing" (1992). Electrical Engineering and Computer Science - Technical Reports. 170.

[10] https://en.wikipedia.org/wiki/Christofides_algorithm

[11] Traveling Salesman Problem and Approximation Algorithms (bochang.me)