Traveling Salesman Problem
Final Report

Fukutoku, Lisa, rfukutok, 54321338
Strand, Michael, mwstrand, 72048580
Van Riper, Brenda, bvanripe, 96124814

COMPSCI 271P Artificial Intelligence

December 8, 2022

Contents

1 Traveling Salesman Problem Introduction

For our project we are solving the traveling salesman problem (TSP) using the application of depth first branch and bound search and stochastic local search. The traveling salesman project can be summarized as - we must visit all of N cities, without repeats, in a circuit such that the traveling distance is minimal. We also assume each city can be visited from any city other than itself. The objective function for this application is simply the cost, or distance, of the full route (state).

The traveling salesman problem has been around since the 1920s and has evolved much since then with the inclusion of heuristic methods and technology [4]. Although the TSP has been around for decades and a plethora of computer scientists have worked on it, there is no proven tractable solution [2].

Our approach to the traveling salesman problem was to use two different search algorithms - depth first branch and bound and stochastic local search. There are many variations of stochastic local search, we chose to go with simulated annealing (SA). For the TSP we will be using python as our programming language for both algorithms. Each input for TSP is a symmetric distance matrix with N *cities, mean, and standard deviation.*

2 Depth First Branch and Bound Search

2.1 Approach

Depth first branch and bound search uses a last in, first out (LIFO) queue using backtracking search. Depth first search starts at a root node and will explore as far as possible down the graph till it reaches a goal state or, if it does not, then it will backtrack and follow down another path. For each new node we expand, all of the neighboring nodes are added onto the frontier.

Our technique to create an algorithm and heuristic for the TSP began with creating the basic depth first branch and bound algorithm focusing on the frontier. The LIFO queue is a list of lists that are composed of partial paths or nodes. Initially, we struggled with coming up with a better solution for the frontier instead of a list of lists because that used up more space and seemed cumbersome. We wanted to try to implement a queue package for the frontier, unfortunately we could not get it to work in the allotted time frame, so we persevered with the list of lists as the frontier.

Along with depth first branch and bound we implemented a heuristic as part of the algorithm. A lower and upper bound heuristic will determine how we prune out paths from the graph structure. At each new node we will compute a heuristic of the lower bound made $L(n) = g(n) + h(n)$ where $h(n)$ is the heuristic (admissible estimated distance to solution) through the node n and $g(n)$ is the cost of the current path from the root to node n in the graph. Upper bound, U, is the true cost of our current best route – if $L(n) \leq U$ we will prune off that path, so we avoid expanding paths that are costly and focus on paths that are promising. If a node is not pruned then its neighbors will be added to the frontier stack and it is popped from the queue. A last in, first out (LIFO) frontier is essential to a depth first branch and bound algorithm, we used lists as our data structure. Every iteration of the algorithm we track the paths (current and best) and costs and overall queue. The LIFO queue is a list of lists from the frontier. For example, if there are 4 cities, the queue may look like

$$[[0,1], \quad [0,2], \quad [0,3], \quad [0,3,1], \quad [0,3,2]]$$

which indicates the order of partial paths to try and expand next. We consider a partial-path list such as $[0, 2]$ to be a node such that the objective function costs (distance + heuristic) are calculated against these partial paths to determine pruning actions.

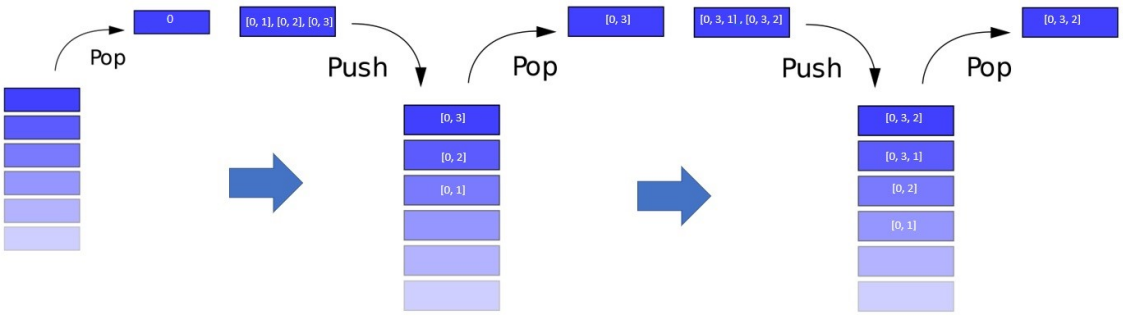

Figure 1: LIFO Python queue informed by the frontier. Initial frontier starts with starting city - After GEN_FRONT (*current, frontier*) gets called where current = *start_city*. Then neighbor cities (1, 2, 3) get added to the frontier. We then pop out the last list from the frontier [0, 3]. After [0, 3] gets popped off the frontier, the path continues through the algorithm since we have not visited all 4 cities yet. Again, GEN_FRONT(current, frontier) gets called where current = *city_3*. Then neighbor cities (1, 2) get added to the frontier.

We researched many heuristics like Held Karp [9], Lin-Kernighan [5], Christofides, and 3-opt [7] to name a few. All of these heuristics were more complex than we wanted to start with; so instead we started with the most painless heuristic - nearest neighbor.

Our first heuristic implemented was nearest neighbor (NN), a greedy algorithm that chooses the cheapest path at every step. This is a simple heuristic only applicable in problems like ours where the distance matrix is symmetric. It is useful for putting an admissible lower bound on the estimated distance remaining from a current state. It is as follows:

1. From the current city in the current path, find the shortest distance to the next unvisited city
2. Repeat while there remain unvisited cities

This estimated cost is used in addition to the real node cost to give an estimate of the full path cost when deciding which neighbors to prune. The use of this heuristic makes it so we are much less likely to visit the global minimum, but the algorithm can complete in a realistic amount of time. This heuristic is good for small Ns, but does not scale well with a large value of N.

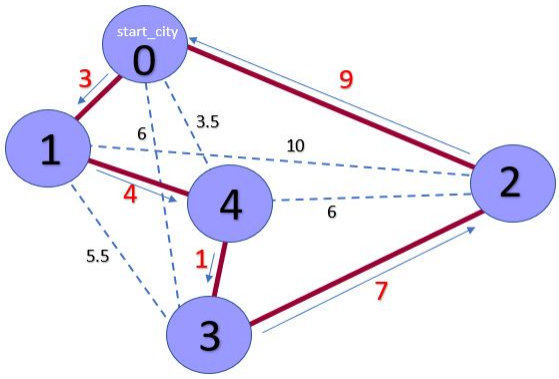

Figure 2: Nearest neighbor path-finding on small example. Nearest neighbor looks for the nearest (least cost) city from the current city, ignoring other paths that are further (more costly). The path (in red) would follow

$0 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 2$
$3 + 4 + 1 + 7 + 9 =$
total distance of 24.

2.2 Algorithm and Heuristics

For our final depth first branch and bound algorithm we implemented clustering along with a 2-opt heuristic aside from our nearest neighbor heuristic - we can choose which heuristic to use when running our program.

- **Clustering:** Clustering is beneficial [6] because it splits one big problem into smaller more manageable problems. $k-$Means clustering is when you partition n datapoints into k clusters based on the nearest mean as a center. For the TSP we use clustering on our unexpanded nodes, using the rows of the distance matrix as coordinate points in $N-$dimensional space where nodes are clustered by relative distance, creating petite subproblems; from there we solve each cluster based on a heuristic, then combine back all k clusters to create one full tour.

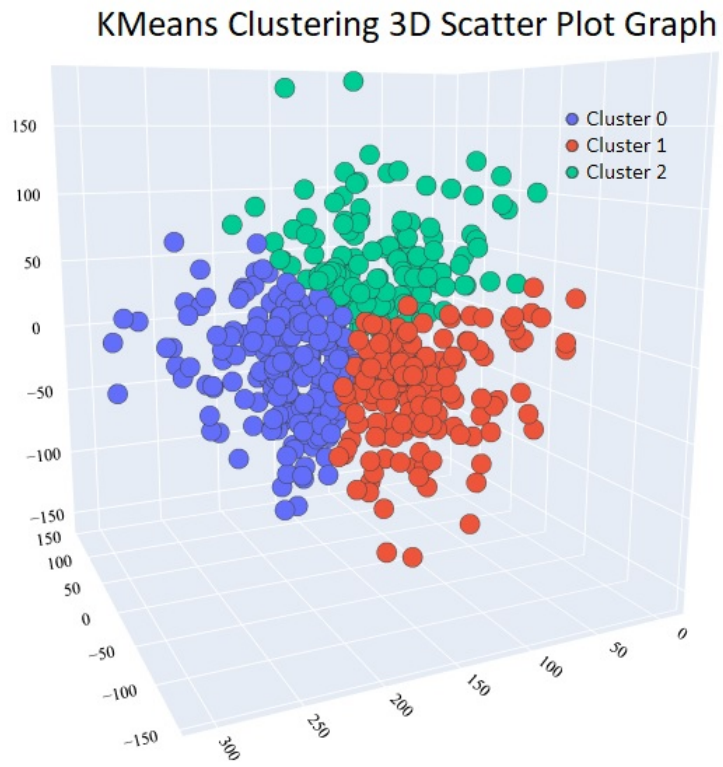

Figure 3: The 3D scatter plot graph shows clustering for a large group of datapoints, split into three different clusters using the k-Means algorithm.

- **$2-$opt :** The leading heuristic that we implemented was 2-opt, a simple but powerful local search algorithm with the main idea to compare all possible combinations of deleting and reforming connections between paths, 2 edges at a time. Many more powerful extensions of this 2-opt approach exist including $k-$opt [7] for varying switches and Lin-Kernighan [5] which stores moves, but for simplicity we opted for the most basic implementation. Although it is slower having time complexity $O(N^2)$, this heuristic provides good approximations and serves to prune effectively in our branch and bound. We may control the number of iterates (check all exchanges) to further refine the heuristic at the expense of time.

  We hypothesized that 2-opt would benefit greatly from clustering as the effective problem size is much smaller per heuristic. However, we did not notice such a benefit in our experimentation.

- **Nearest-neighbor:** Our secondary heuristic is nearest-neighbors which greedily chooses whichever path is the next cheapest until the end. A quicker, messier approximation compared to 2-opt.

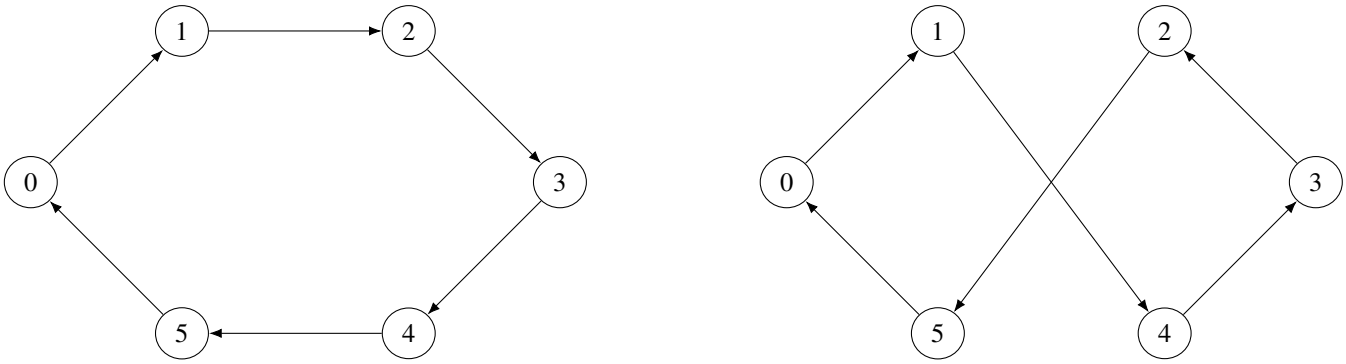

Figure 4: Application of 2-opt on a path [0, 1, 2, 3, 4, 5, 0] *(left)* resulting in [0, 1, 4, 3, 2, 5, 0] *(right)*.

We also have many parameters that we use to control how the depth first branch and bound algorithm runs. We can run with the nearest neighbor or 2-opt heuristic and we can turn clustering on or off. Along with these core arguments for our search algorithm we also have some parameters we can use.

- **Parameters:** We have included various parameters in our depth first branch and bound algorithm: heuristic frequency, opt iterations, and max bottoms. These parameters have been included to make the algorithm slower but more accurate, or faster but not as precise.
  - **Heuristic Frequency:** In the interest of time and computing resources, we can choose to skip a heuristic calculation and randomly decide whether to prune or not. This action is controlled by the heuristic frequency parameter which skips every kth heuristic calculation for $k = 2, 3, ...$
  - **Opt Iterations:** Controls the number of times we check through every possible edge exchange which is expensive to implement, but adds heuristic accuracy.
  - **Max Bottoms:** This is the integer that represents how many times we reach the bottom before terminating. Reaching the bottom here means encountering a path in the frontier of length N.
  - **Number of clusters:** Lets us choose how many clusters to partition our heuristic into. The more we cluster the faster larger problems can be solved but at the expensive of solution quality, as we need to use a sub-optimal route to re-link these disparate paths.

The ideal settings for these parameters are dependent on the size of N and what we are trying to accomplish with the algorithm - speed, accuracy, or some combination of the two.

2.3 Analysis and Properties

- **Time complexity:** Time complexity for branch and bound depth first search is exponential, $O(b^m)$, where b is the branching factor and m is the length of the optimal path [1].
- **Space complexity:** Space complexity of depth first branch and bound is linear, $O(N)$, since it follows down one path and our heuristic will prune off paths as the algorithm runs [11].
- **Optimally and Completeness:** Depth first branch and bound search can be considered complete and optimal if run to completion with proper bounds; it can also be adapted to run as an approximate solution. Proper bounds for optimal solutions are very tricky and the most challenging aspect of the algorithm [11].

2.4 Observation and Assessment

Branch and bound depth first search is a powerful algorithm in some scenarios, but its design suffers from fatal flaws in others. The depth-first and pruning aspects are very beneficial for quickly scoping out good search spaces to probe and deciding how long to explore them, but these features are highly sensitive to the problem at hand. Mean distance, variance, and most especially number of nodes all require special attention, much more so than in simulated annealing as we will see.

The most important factor in the performance of BnB is the heuristic, as it controls the ability to learn which paths to prune. Without the right heuristic, we can prune away far too much of the search space or be too indecisive and unable to avoid checking too many paths. Considering this, we need a powerful algorithm like $k-$opt, which also becomes incredibly expensive for larger problems. Due to these competing factors, we find a little of parameter fine-tuning is required to experience modest performance, and only in smaller systems ($N < 100$). At or above $N = 100$ we see extremely long run times, even under permissive parameter settings.

We employ several tunable features like clustering, different heuristics, heuristic iterations, cutoffs, and heuristic skips that allow us to be a little flexible. Despite all our implementations features our algorithm still struggles often to make the right pruning decisions. We observe that the heuristic will often underestimate early on and then overestimate later in the algorithm resulting in only a few full paths being explored from new starting cities. It seems like an obvious area for improvement would be to add in a more intelligent heuristic like L-K [5] to better steer the algorithm.

For smaller problems with $N < 100$, we find relatively accurate answers in a fair amount of time given the right parameters. For example, there is a trade off between run time and accuracy which we can control by balancing heuristic frequency, number of clusters, heuristic iterations, and number of bottoms before termination. Increasing any of these will create longer run time. In our experience, the heuristic frequency is the most important parameter since the more often we skip the heuristics the faster we get our answer, but worst case we unnecessarily prune and miss a desirable path.

Since we are without a smarter heuristic, we have seen that our DFS has trouble picking out hidden favorable paths under situations of high variance. We originally expected branch and bound to be particularly well-suited for high-variance problems, but in it's current state we still struggle here.

2.5 Experimentation Results

Branch and Bound - Analytics

# Nodes (N)	Best distance	CPU time (s)	$\sim 95\%$ optimal distance	Error
10	974.0	0.65	974.0	0.0
	946.0	7.5	946.0	0.0
25	2366.0	334.3	2278.0	88.0
	2389.0	385.8	2316.0	73.0
100	9817.0	600.0	8965.0	852.0
	9813.0	533.578125	8931.0	882.0
500	-	-	-	-
	-	-	-	-
1000	-	-	-	-
	-	-	-	-

Table 1: Error and CPU time for 10^6 iterations. Matrix: mean = 100, s.d. = 5

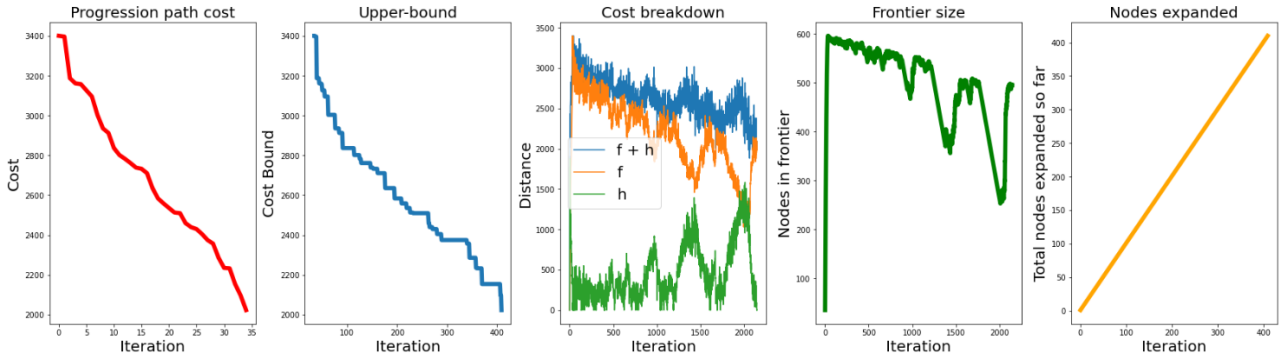

Figure 5: Behaviour of branch-and-bound depth-first search and its sub-processes on $N = 50$ having mean = 100 and s.d. = 5. We skip every 100th heuristic and explore 100 bottoms before termination. We see the effect of pruning on the progressive frontier size.

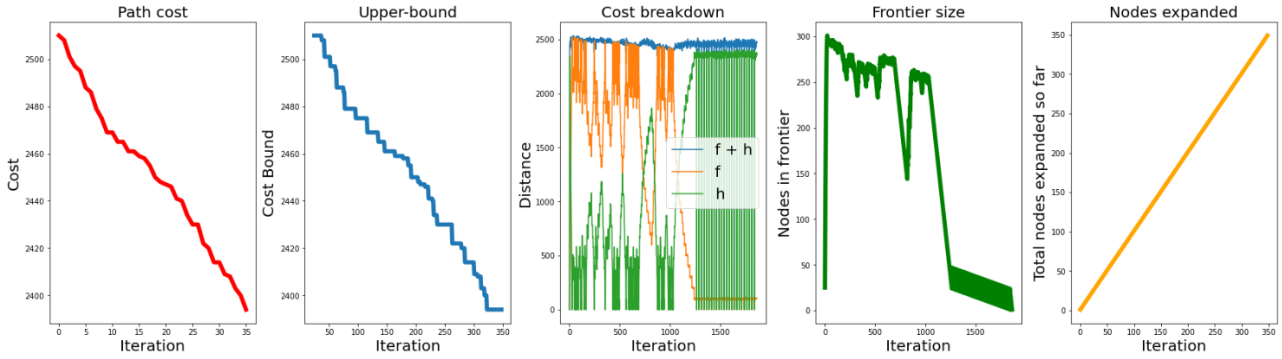

Figure 6: Behaviour of branch-and-bound depth-first search and its sub-processes on $N = 25$ having mean = 100 and s.d. = 5. We skip every 100th heuristic and explore 75 bottoms before termination. Here we use 2-opt without clustering due to the low N.

3 Stochastic Local Search - Simulated Annealing

3.1 Approach

Simulated annealing (SA) is a type of stochastic local search useful for approximating the global minimum of a function. Each iteration, a new state is generated from the previous and if the cost of this new state is less then we store this new state and proceed. However, this algorithm can avoid becoming trapped in local minima by sometimes allowing worse state transitions to be accepted with some variable probability. This probability is primarily dependent on the algorithm's current temperature; the higher the temperature the more likely we are to accept 'poor' choices. The performance of SA depends largely on the choice of a temperature function (how temperature will decrease every iteration), the initial temperature, and the kinds of state transitions we implement. SA iterations are full tours and each iterate represents a path through all N cities. In this setting, we define the state space as the set of all possible full routes (states), of which there are $\frac{(N-1)!}{2}$. The states of SA are in the form of a Python list, which represents a full tour.

We will use four different actions to randomly determine a new state: swapping two cities, moving a city to a new random position, inverting a sub route, and swapping the position of a sub route. We then consider the 'neighbors' of a state to be any new state that results from one of these simple transitions. Originally we used equal probabilities of 0.25 for each of these four actions to happen, the algorithm would randomly pick one. We weighted the probabilities of these four actions in numerous different ways to see if the change would allow our algorithm to get closer to the optimal path - unfortunately it did not make much of a noteworthy difference.

3.2 Algorithm Key Features

The defining feature of SA is the probability to randomly accept tours which have a higher cost function than the current tour. We calculate the probability of accepting a bad transition p as

$$p = e^{\frac{E}{T\alpha(t)}}$$

where E is the difference in distance between the current path and newly generated path and T is the initial temperature. If $E < 0$ then the new path is longer than the current. This means that as $t \to$ max_it, $T\alpha(t) \to 0$ and thus $p \to 0$. The result is a high probability of jumping out of local minimums early on, but slows to 0 towards the end of run time.

Although simulated annealing may seem unsophisticated in it's random-selection approach, there are many unique categories of techniques, from simple to advanced, we can employ to give the algorithm an edge. The keys to fine-tuning this algorithm are the **initial temperature**, **temperature function**, and **stagnation**.

- **Initial temperature:** In order for this algorithm to be robust against a range of differently distributed distance matrices we must use intention when choosing the initial temperature if we want to ensure a smooth cooling process. Since p depends on E, the difference in tour costs, it will certainly be the case that two different problems will have two different magnitudes for E. For this reason, we implemented a sampling process, as described in [3], which calculates an initial temperature capable of providing a reasonable temperature decline tailored to specific distance matrices. To find this temperature, we coded a simulation process where we generate a sample of poor transitions (new distance longer than old distance) and record the difference in costs before and after. We then recursively find

our initial temperature T by scaling down according to the log ratio of current acceptance probability X_T under T and desired probability X_0. The further off X_T is with the current T the more we adjust until we reach a specified margin of error. We typically choose a sample size such that the first 10% of the runtime has roughly an acceptance probability of X_0.

- **Stagnation and reheating:** The behavior of SA is erratic (willing to accept most tours) at first while the temperature is 'hot' and greedy later once it has 'cooled'. This heat is controlled primarily by the temperature function, but in certain search-spaces it can be desirable to intermittently re-heat after a 'stagnant' period, or a period of time in which no bad transitions have occurred. By re-heating, we allow the algorithm to unsettle from an area local to a small subset of tours and instead become 'hot' again and is volatile just like at the start.

  It is simple to implement this and we can control this stagnation procedure by defining the time until a reheating event, how hot it can be after reheating and subsequent reheats occurs, and how quickly we cool down again. With these mechanisms, we can create a more versatile version of SA which goes back and forth between local and global searches. In our implementation we penalize this reheating procedure to still capture the intentions of SA to slowly sink deeper into different minimums.

- **Temperature function:** The most important aspect of SA is it's ability to jump out of local minimums by accepting worse tours occasionally according to some probability. To ensure the probability lowers as time goes on and for lower quality tours we need a carefully considered temperature function. There are many well-established options and we choose the geometric scalar, defined as -

$$\alpha(t) = \alpha_0^t = \left(1 - \frac{1}{\text{max\_it}^{X_0}}\right)^t$$

  where max_it is the maximum iterations before the algorithm terminates, X_0 is the initial acceptance probability and t is the current heat.

3.3 Path Changing Actions

The neighbors of a state are considered to be any new state which is possible to achieve by permuting the current state according to 1 of the following 4 actions. Each iteration of SA, new states are generated by selecting one of these actions, uniformly at random, and then applying it.

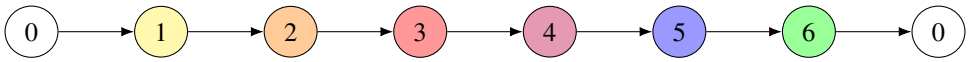

Figure 7: Initial Path

Figure 8: Switch two cities (2 & 6)

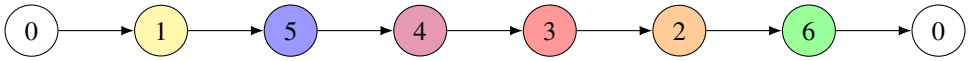

Figure 9: Invert path between two cities (2 & 6)

Figure 10: Insert city into random position (2 & 5)

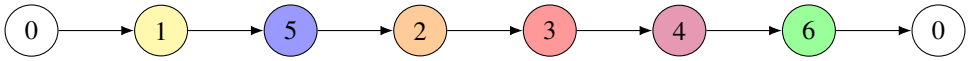

Figure 11: Swap sub route to random position (route 2-4 to 4th position)

Analysis and Properties

- **Time complexity:** Best case is $O(1)$ should the first tour selected be the optimal one and $O(N!)$ if we have to search the entire space. The average time complexity is proposed to be $O(N^4)$ by [8].
- **Space complexity** Space complexity is constant $O(1)$ since we only need to store a fixed amount of information: the current, new, and best tours, the distance matrix, and some constants.
- **Completeness:** Simulated annealing is not complete since there is no guarantee we ever land on the optimal solution due to the randomness and potential to become trapped eventually once the process has cooled.
- **Optimally:** This algorithm may be cost-optimal due to randomization property which allows one to potentially find global minimums.

3.4 Observation and Assessment

- **Problem types:** Due to of the the element of randomness and the simple, unintelligent state transitions, SA is not an intelligent algorithm and cannot effectively seek out optimal paths. The power of SA is it's ability to very quickly test many very large tours and avoid local minimums. For these reasons, we find SA better suited for large ($N > 100$) homogeneous (low variance) problems. When variance is low it can be harder for more intelligent algorithms like BnB to identify the right next step to take. Problems which may be too large and/or difficult for BnB can get a better guess from SA since we can terminate at any time and still have a fair approximation. Since the state space contains $\frac{(N-1)!}{2}$ full tours, SA is extremely effective for very small tours on the order of $N < 15$ as it can explore $1,000,000$ tours in about 120 seconds (CPU time). After around $N = 15$ though, the state space grows at such a rapid rate, it becomes increasingly rarer for SA to find really good solutions.

- **Parameters:** Our main areas of control are in the maximum number of iterations, reheating threshold (in iterations), and most importantly, X_0.

  - **Reheat threshold:** We can specify exactly how many times to reheat during the runtime. The more often we reheat, the more likely we are to jump out of minimums. However, this also makes us more likely to abandon potentially good tours. We find reheating is most useful for systems of low variance and more often than not finds better tours than runs without reheating (with the right penalties).

  - **Max iterations:** This parameter is self-explanatory, but it is also an important scalar used to find our initial temperature function α_0. The runtime is totally dependent on how this value is set and is the sole stopping mechanism. The larger we make max iterations the more accurate we expect our results. However, the difference between the rate at which the state space grows (factorial) and how much we can feasibly increase max iterations makes extremely large problems ($N > 1000$) much less approachable.

  - $\boldsymbol{X_0}$ **:** This parameter accepts values from 0 to 1 and defines the initial acceptance probability for bad transitions. This means, for about the first 10% of runtime, bad transitions will be accepted anyway with about a probability of X_0. We set this to between 0.75 and 0.85 such that about $1/3$ of total runtime there is a modest chance to accept a bad transition. Higher X_0 will lead to totally random behavior and lower becomes impractically greedy.

- **Improvements:** There is little room for improvement since simulated annealing is naturally limited, but one an obvious area to explore in the future would be trying different/more adaptable temperature functions. Our geometric scalar value is only one possible way to reduce temperature and others such as logarithms [10] and rational functions have been used with success before. Our current temperature function cools at an exponential rate and we need to make up for this with reheating, but other behaviors are possible. It would be worthy to explore whether different temperature functions solve differently distributed systems better.

3.5 Experimentation Results

Simulated Annealing - Analytics

# Nodes (N)	Best distance	CPU time (s)	$\sim 95\%$ optimal distance	Error
10	941.0	48.5	941.0	0.0
	947.0	46.4	947.0	0.0
25	2377.3	47.9	2317.5	59.8
	2376.5	46.3	2293.9	82.6
100	9732.3	121.5	9015.8	716.5
	9742.6	136.7	8973.0	769.5
500	49437.7	562.5	43432.0	6005.7
	49451.8	603.8	43475.2	5976.6
1000	99161.4	1342.6	85989.1	13172.2
	99198.2	1943.0	85991.5	13206.7

Table 2: Error and CPU time for 10^6 iterations. Matrix: mean = 100, s.d. = 5

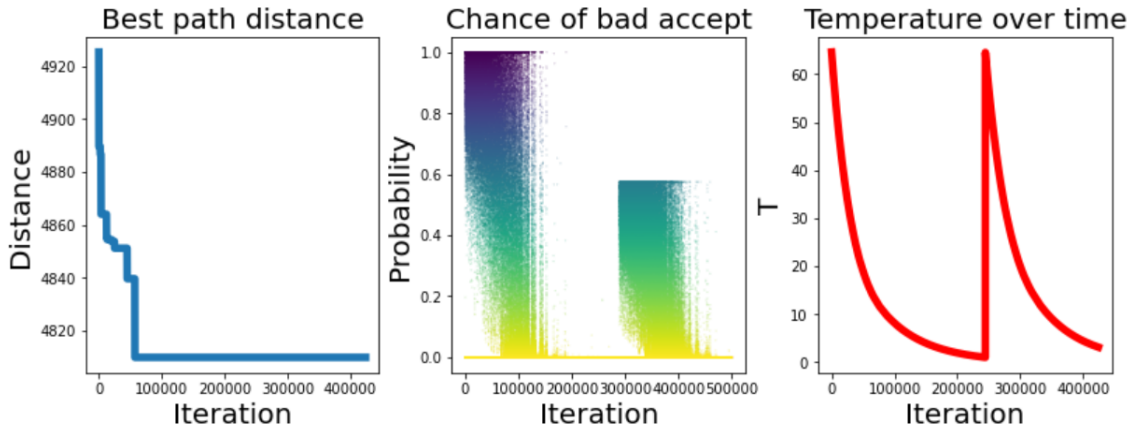

Figure 12: Behaviour of simulated annealing and it's sub-processes on a random $N = 50$ having mean = 100 and s.d. = 5. We take $X_0 = 0.85$ and $50,000$ iterations. Here we see reheating applied once over the runtime however, this reheating did not reveal a better search space as it sometimes does.

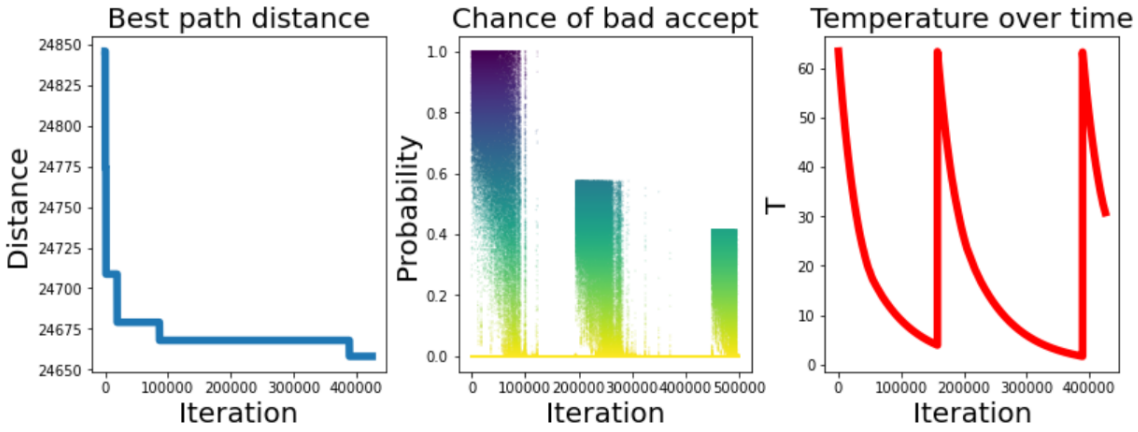

Figure 13: Behavior of SA with random $N = 250$ having mean = 100 and s.d. = 5. We take $X_0 = 0.85$ and $50,000$ iterations. Here it appears reheating brought us to a space where a better solution resided.

A Competition Results

Note: all problems were solved using simulated annealing and time is measure in CPU time

	SLS	Distance	Time (seconds)	Iterations
	tsp-problem-**n**-**k**-**u**-**v**-**i**			
1	tsp-problem-25-6-100-5-1.txt	2496.8039	132.8906	10^7
2	tsp-problem-25-6-100-25-1.txt	1737.3368	126.2344	
3	tsp-problem-25-31-100-5-1.txt	2367.7494	128.0781	
4	tsp-problem-25-31-100-25-1.txt	1740.0173	129.2031	
5	tsp-problem-25-62-100-5-1.txt	2390.8067	185.2969	
6	tsp-problem-25-62-100-25-1.txt	1897.0449	126.6719	
7	tsp-problem-25-125-100-5-1.txt	2365.9352	128.6719	
8	tsp-problem-25-125-100-25-1.txt	1833.4917	131.9531	
9	tsp-problem-25-250-100-5-1.txt	2370.2767	127.0938	
10	tsp-problem-25-250-100-25-1.txt	1868.3504	128.6875	
11	tsp-problem-50-25-100-5-1.txt	4881.4725	186.7188	10^7
12	tsp-problem-50-25-100-25-1.txt	3932.3222	196.7969	
13	tsp-problem-50-125-100-5-1.txt	4754.3795	193.8438	
14	tsp-problem-50-125-100-25-1.txt	4311.9246	189.9688	
15	tsp-problem-50-250-100-5-1.txt	4797.8563	204.6250	
16	tsp-problem-50-250-100-25-1.txt	4090.0969	199.2813	
17	tsp-problem-50-500-100-5-1.txt	4777.6277	200.1094	
18	tsp-problem-50-500-100-25-1.txt	4074.0209	199.1250	
19	tsp-problem-50-1000-100-5-1.txt	4836.0265	244.2188	
20	tsp-problem-50-1000-100-25-1.txt	4054.9187	246.4844	

Figure 14: Competition results: N = 25 - 50

	tsp-problem-**n-k-u-v-i**	**Distance**	**Time (seconds)**	**Iterations**
21	tsp-problem-75-56-100-5-1.txt			
22	tsp-problem-75-56-100-25-1.txt	6533.7700	332.3281	
23	tsp-problem-75-281-100-5-1.txt	7247.1746	337.0781	
24	tsp-problem-75-281-100-25-1.txt	6158.3859	336.4531	
25	tsp-problem-75-562-100-5-1.txt	7317.8407	337.7188	10^7
26	tsp-problem-75-562-100-25-1.txt	6400.4855	338.6094	
27	tsp-problem-75-1125-100-5-1.txt	7253.1496	337.2344	
28	tsp-problem-75-1125-100-25-1.txt	6495.5312	337.2188	
29	tsp-problem-75-2250-100-5-1.txt	7280.8205	333.4531	
30	tsp-problem-75-2250-100-25-1.txt	6104.3635	336.5313	
31	tsp-problem-100-100-100-5-1.txt	9738.3730	250.8594	
32	tsp-problem-100-100-100-25-1.txt	8342.6133	451.1563	
33	tsp-problem-100-500-100-5-1.txt	9780.2278	343.8750	
34	tsp-problem-100-500-100-25-1.txt	8668.5175	438.5938	
35	tsp-problem-100-1000-100-5-1.txt	9693.0207	304.6875	
36	tsp-problem-100-1000-100-25-1	8703.9554	306.2656	$5 * 10^6$
37	tsp-problem-100-2000-100-5-1.txt	9754.5139	306.3750	
38	tsp-problem-100-2000-100-25-1.txt	8753.9190	309.7969	
39	tsp-problem-100-4000-100-5-1.txt	9727.4995	306.7813	
40	tsp-problem-100-4000-100-25-1.txt	8471.2648	442.5156	

Figure 15: Competition results: N = 75 - 100

	tsp-problem-**n-k-u-v-i**	**Distance**	**Time (seconds)**	**Iterations**
41	tsp-problem-200-400-100-5-1.txt	19672.6741	452.5000	
42	tsp-problem-200-400-100-25-1.txt	17967.7110	436.5938	
43	tsp-problem-200-2000-100-5-1.txt	19583.7369	454.0156	
44	tsp-problem-200-2000-100-25-1.txt	18416.1816	434.6406	
45	tsp-problem-200-4000-100-5-1.txt	19609.4007	433.9375	
46	tsp-problem-200-4000-100-25-1.txt	17989.6143	435.3281	$2 * 10^6$
47	tsp-problem-200-8000-100-5-1.txt	19602.5575	433.3438	
48	tsp-problem-200-8000-100-25-1.txt	18269.6008	600.0000	
49	tsp-problem-200-16000-100-5-1.txt	19604.3542	600.0000	
50	tsp-problem-200-16000-100-25-1.txt	18212.6718	579.0156	
51	tsp-problem-300-900-100-5-1.txt	29555.0068	491.4688	
52	tsp-problem-300-900-100-25-1.txt	28067.2507	495.3125	
53	tsp-problem-300-4500-100-5-1.txt	29572.6640	493.5000	
54	tsp-problem-300-4500-100-25-1.txt	27469.1692	517.5000	
55	tsp-problem-300-9000-100-5-1.txt	29606.5847	515.7031	
56	tsp-problem-300-9000-100-25-1.txt	27841.7921	496.7969	10^6
57	tsp-problem-300-18000-100-5-1.txt	29555.7541	575.0938	
58	tsp-problem-300-18000-100-25-1.txt	27822.6785	517.9219	
59	tsp-problem-300-36000-100-5-1.txt	29553.3514	514.0938	
60	tsp-problem-300-36000-100-25-1.txt	27756.9506	600.0000	

Figure 16: Competition results: N = 200 - 300

	tsp-problem-**n-k-u-v-i**	Distance	Time (seconds)	Iterations
61	tsp-problem-400-1600-100-5-1.txt	39470.3530	558.8125	$8 * 10^5$
62	tsp-problem-400-1600-100-25-1.txt	37438.7930	560.6094	
63	tsp-problem-400-8000-100-5-1.txt	39464.6243	559.9063	
64	tsp-problem-400-8000-100-25-1.txt	37158.7569	583.8906	
65	tsp-problem-400-16000-100-5-1.txt	39547.1049	600.0000	
66	tsp-problem-400-16000-100-25-1.txt	37303.9662	565.4844	
67	tsp-problem-400-32000-100-5-1.txt	39525.9632	579.3438	
68	tsp-problem-400-32000-100-25-1.txt	37295.6905	582.6719	
69	tsp-problem-400-64000-100-5-1.txt	39503.7394	600.0000	
70	tsp-problem-400-64000-100-25-1.txt	37109.4449	375.1563	
71	tsp-problem-600-3600-100-5-1.txt	59353.6453	469.2656	$8 * 10^5$
72	tsp-problem-600-3600-100-25-1.txt	57521.7023	600.0000	
73	tsp-problem-600-18000-100-5-1.txt	59403.1015	600.0000	
74	tsp-problem-600-18000-100-25-1.txt	56793.5923	600.0000	
75	tsp-problem-600-36000-100-5-1.txt	59333.8748	600.0000	
76	tsp-problem-600-36000-100-25-1.txt	56604.6497	600.0000	
77	tsp-problem-600-72000-100-5-1.txt	59424.6712	472.7813	
78	tsp-problem-600-72000-100-25-1.txt	56867.2952	470.4219	
79	tsp-problem-600-144000-100-5-1.txt	59390.6739	469.7344	
80	tsp-problem-600-144000-100-25-1.txt	56939.6346	472.7188	

Figure 17: Competition results: N = 400 - 600

	tsp-problem-**n-k-u-v-i**	Distance	Time (seconds)	Iterations
81	tsp-problem-800-6400-100-5-1.txt	79241.8465	529.2813	$6 * 10^5$
82	tsp-problem-800-6400-100-25-1.txt	76124.6051	528.0781	
83	tsp-problem-800-32000-100-5-1.txt	79296.0962	527.6406	
84	tsp-problem-800-32000-100-25-1.txt	76118.8115	529.2344	
85	tsp-problem-800-64000-100-5-1.txt	76649.7566	526.8125	
86	tsp-problem-800-64000-100-25-1.txt	76761.0270	454.2344	
87	tsp-problem-800-128000-100-5-1.txt	79385.0235	460.7188	
88	tsp-problem-800-128000-100-25-1.txt	76824.0541	600.0000	
89	tsp-problem-800-256000-100-5-1.txt	79331.1862	459.0000	
90	tsp-problem-800-256000-100-25-1.txt	76722.2504	455.8438	
91	tsp-problem-1000-10000-100-5-1.txt	99113.1904	560.9375	$5 * 10^5$
92	tsp-problem-1000-10000-100-25-1	95878.4036	581.5156	
93	tsp-problem-1000-50000-100-5-1.txt	99264.7532	558.5156	
94	tsp-problem-1000-50000-100-25-1.txt	96588.8032	588.6406	
95	tsp-problem-1000-100000-100-5-1.txt	99202.4145	589.0781	
96	tsp-problem-1000-100000-100-25-1.txt	95901.7557	592.8125	
97	tsp-problem-1000-200000-100-5-1.txt	99083.9386	558.7344	
98	tsp-problem-1000-200000-100-25-1.txt	96152.4282	557.1250	
99	tsp-problem-1000-400000-100-5-1.txt	95855.6734	586.2500	
100	tsp-problem-1000-400000-100-25-1.txt	96175.0237	558.5156	

Figure 18: Competition results: N = 800 - 1000

B Bibliography

References

[1] Branch bound, summary of advanced topics. https://www.cs.ubc.ca/~hutter/teaching/cpsc322/2-Search7.pdf.

[2] Complexity and tractability 12.4. the travelling salesman problem. https://www.csfieldguide.org.nz/en/chapters/complexity-and-tractability/the-travelling-salesman-problem/.

[3] Walid Ben-Ameur. Computing the initial temperature of simulated annealing. *Computational Optimization and Applications*, 29:369–385, 12 2004.

[4] Nigel Cummings. A brief history of the travelling salesman problem. https://www.theorsociety.com/about-or/or-methods/heuristics/a-brief-history-of-the-travelling-salesman-problem/.

[5] Keld Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130, 10 2000.

[6] Majd Latah. Solving multiple tsp problem by k-means and crossover based modified aco algorithm. *International Journal of Engineering Research and Technology*, 5:430–434, 02 2016.

[7] Zhibei Ma, Lantao Liu, and Gaurav S. Sukhatme. An adaptive k-opt method for solving traveling salesman problem. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 6537–6543, 2016.

[8] Galen Hajime Sasaki. *Optimization by Simulated Annealing: A Time-Complexity Analysis*. PhD thesis, USA, 1987. AAI8803190.

[9] Mathematics Science, David Shmoys, and Pmfi Williamson. Analysis of the held-karp heuristic for the traveling salesman problem. 03 2001.

[10] Mohammad-Taghi Vakil-Baghmisheh and Alireza Navarbaf. A modified very fast simulated annealing algorithm. 08 2008.

[11] Weixiong Zhang. Depth-first branch-and-bound versus local search: A case study. *AAAI/IAAI*, 04 2000.