A Review of Algorithms for the Traveling Salesman Problem

Shichao Liang
mliang@gatech.edu
Georgia Institute of Technology

REDACTED
REDACTED@gatech.edu
Georgia Institute of Technology

REDACTED
REDACTED@gatech.edu
Georgia Institute of Technology

REDACTED
REDACTED@gatech.edu
Georgia Institute of Technology

ABSTRACT

In this work, we implement four different solvers for the Travelling Salesman Problem (TSP) and apply them to a set of instances of varying structure and size. The first solver implements a branch-and-bound algorithm; the second solver implements a MST-based approximation algorithm; and the third and fourth solvers implement local search approaches, namely simulated annealing and genetic algorithm. We first present each solver in detail, describing their implementations and analyzing their space and time complexities, to theoretically assess their potential strengths and weaknesses. We then compare their performance empirically.

KEYWORDS

traveling salesman problem, genetic algorithm, branch and bound, approximation, simulated annealing

ACM Reference Format:
Shichao Liang, REDACTED, REDACTED, and REDACTED. 2018. A Review of Algorithms for the Traveling Salesman Problem. In *CSE 6140: Computational Science and Engineering Algorithms, Fall 2019, Georgia Institute of Technology*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The Traveling Salesman Problem (TSP) is a renowned and well studied NP-hard optimization problem with applications in resource allocation, electronics, astronomy, and DNA sequencing. In short, the problem is about finding a shortest tour that visits every city and returns to the starting point given a set of cities and a pairwise distance function. There are no polynomial solutions to the TSP problem, the only known exact solutions are either exponential or factorial in the number of cities. However, many heuristics, approximations, and iterative improvement algorithms have been developed that provide solutions that either sacrifice quality or speed. In this project, we explore a branch and bound algorithm, minimum spanning tree approximation with guarantee, and two local search algorithms, namely, simulated annealing and a genetic algorithm. As a result, we've found that branch-and-bound performed best for very small n, approximation algorithms exhibited the best time-performance but with the worst fitness results, and genetic algorithm outperforming simulated annealing in terms of fitness performance.

CSE 6140, Fall 2019, Georgia Institute of Technology

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

2 PROBLEM DEFINITION

Courtesy of the project description, we will define the TSP problem as follows:

Given the x-y coordinates of N points in the plane (i.e., vertices), and a cost function c(u, v) defined for every pair of points (i.e., edge), find the shortest simple cycle that visits all N points. This version of the TSP problem is metric, i.e. all edge costs are symmetric and satisfy the triangle inequality.

3 RELATED WORK

Over the past 50 years, many algorithms were proposed to solve (or to approximate the solution of) the TSP. In this Section, we present a summary of contributions closely related to the algorithms benchmarked in this paper.

3.1 Branch-and-Bound

Held and Karp published two papers [12] [13], between 1970 and 1971, exploring the relationship between the symmetric TSP and 1-trees to derive a sharp lower bound on the cost of an optimum tour based on minimum spanning trees (1-tree relaxation method). In 1982, Volgenant and Jonker leveraged the work by Held and Karp to create a new branch-and-bound algorithm, with new branching scheme and bounding method [24], reporting an average 60% decrease in the number of trees. In 1983, Balas and Toth [5] published a technical report surveying the state-of-the-art in enumerative solution methods for the TSP. The seminal work of Held and Karp would continue to influence later developments, with Shmoys and Williamson analyzing the monotonicity property of Held-Karp 1-trees 20 years after its publication [21]. Also, Tschoke et al. parallelized the 1-tree relaxation, in 1995 [23], to claim an almost linear speed-up on hard problems and the efficient solution of instances comprising up to 318 cities on 1024 CPUs.

3.2 Approximation

The Euclidean restriction of TSP has been subject of an extensive number of approximation strategies. In particular, Arora [4] and Mitchell [16] showed, in 1998 and 1999 respectively, that the Euclidean TSP admits a polynomial-time algorithm that provides a $(1 + \epsilon)$-approximation for every fixed $\epsilon > 0$. Rao and Smith [19]

subsequently improved this result, obtaining runtime of $O(n \log n)$. In 2013, Bartal and Gottlieb presented a linear time approximation scheme for the Euclidean TSP [6].

3.3 Local Optimization

Skiscim and Golden [22] first applied the simulated annealing strategy, proposed by Kirkpatrick et al. [15], to the TSP problem in 1983. In 1988, Aarts et al. [3] presented a quantitative analysis of the results obtained by the simulated annealing strategy, modeling the statistical properties of obtained results as a function of the parameters of the cooling schedule. More recently, Geng et al. proposed the use of a greedy search heuristic to speed up the convergence rate [9], in 2011.

In 1985, Grefenstette et al. [11] presented a representative set of approaches to design genetic algorithms for combinatorial optimization problems, using the TSP as an illustrative case. In 1990, Braun [7] reported an improvement of the results obtained by Muhlenbein et al. in 1988 [17], claiming to have obtained very good (yet not optimal) results for instances comprising up to 531 cities. More recently, Potvin [18] surveyed the various extensions of genetic algorithms to approximate the solution of TSP, in 1996.

4 ALGORITHMS

Four algorithms were investigated for the `TravelingSalesman` problem. A brief description of each algorithm is provided below.

4.1 Branch and Bound

4.1.1 High-Level Description. The Branch-and-Bound algorithm maintains a set of partial solutions called *frontier*, where a partial solution is simply a path in the graph. Initially, this *frontier* set has a single path containing only vertex 1. A lower bound $lb(p)$ is calculated for each partial solution p in the *frontier* set: no solution derived from p can cost less than $lb(p)$. Specifically, we have used the *MST-based* approach to calculate the lower bounds in our experiments: the lower bound of p is the sum of the cost of the path p itself, the cost of the minimum spanning tree covering the vertices not in p, and the cost of connecting p to this minimum spanning tree. Furthermore, a global *upper_bound* is maintained with the best solution (i.e. the route with the lowest cost) found so far. Initially, this global *upper_bound* is set with the cost of a trivial route visiting vertices in the following order: 1-2-3-...-N-1.

Until the *frontier* set is empty, the most promising partial solution p' in the *frontier* set is selected and removed from the set. Typically, this is the one with the lowest lower bound (in Section 5, we detail an experiment in which the most promising partial solution is the one with the lowest ratio $lb(p')/len(p')$, where $len(p')$ is the length of path p' itself). The most promising partial solution p' is then expanded, with an unvisited vertex appended to its end, generating a new candidate path c that is only added to the *frontier* set if $lp(c) < upper\_bound$. This is exactly the pruning phase of the algorithm: a branch is not explored if no possibly derived solutions can cost less than what was already found.

4.1.2 Data Structures and Implementation Details. The graph is implemented with an adjacency matrix (a 2-dimensional *numpy* array), and paths are plain Python lists. The *frontier* set is a priority queue, specifically, a min-heap. We used the Prim's algorithm to build the minimum spanning tree needed to calculate the lower bound of each partial solution, and this algorithm has a priority queue, specifically, a min-heap as well. To implement these min-heaps, we used the methods *heappush* and *heappop* of Python's *heapq* module.

4.1.3 Time and Space Complexities. In the worst-case, we still have to evaluate all possible routes. For each route, we augment the path one vertex at a time, calculating its lower bound with the Prim's Algorithm (time complexity in $O((V+E)logV)$, where $E = V^2$, with the min-heap implementation) and two linear searches to join the ends of the path to the resulting spanning tree. Therefore, this algorithm has time complexity in $O(V!V^2logV)$. Since each configuration stores its path only (the graph data structure is shared among them), the space complexity is in $O(V!V)$.

```
1  Procedure: BranchAndBoundLowerBound
   Input: Graph G = (V, E), Path p =< p_0, p_1, ..., p_n >
   Output: Lower bound of all solutions possibly derived from
           p
2  V_p ← Set of vertices in p
3  p_0 ← First endpoint of p
4  p_n ← Second endpoint of p
5  MST ← Minimum spanning tree of V − V_p
6  e_0 ← Shortest edge connecting p_0 to V − V_p
7  e_n ← Shortest edge connecting p_n to V − V_p
8  return Cost of p + Cost of MST + weight(e_0) + weight(e_n)
```

```
1  Procedure: BranchAndBound
   Input: Graph G = (V, E)
   Output: Exact solution of the TSP problem
2  upper_bound ← cost of the trivial route 1 − 2 − ... − N − 1
3  frontier ← {[1]}
4  while frontier is not empty do
5      p ← path with the lowest lower bound in frontier
6      Remove p from frontier
7      for v in V do
8          if v not in p then
9              new_p ← append v to p
10             if new_p is a solution then
11                 upper_bound ←
                    min(upper_bound, cost of new_p)
12             end
13             else
14                 if lb(new_p) < upper_bound then
15                     Add new_p to frontier
16                 end
17             end
18         end
19     end
20 end
21 return upper_bound
```

4.2 MST Approximation

4.2.1 High-Level Description. The Minimum Spanning Tree (MST) is a 2-approximate algorithm whose approximation guarantee hinges on the Triangle Inequality property of the TSP problem instance. The problem is represented as a weighted undirected graph, $G = (V, E)$, with cities as nodes and paths between cities as edges. Cost of an edge (u,v) is defined as Euclidean distance from city u to city v. This algorithm chooses the city with index 0, c_0, to be the starting point of the cycle. Using c_0 as a root, a Minimum Spanning Tree is constructed using Prim's Algorithm. We greedily add the minimum edge cost, and update all minimum distances as needed until all nodes are in set T. Using the parent node data, we backtrack the children of each node starting with the root node to construct an adjacency list. We then perform Depth First Search (DFS) to create a euler tour, a tree traversal where nodes are added as they are reached in the DFS algorithm. Duplicates of nodes are removed, and c_0 is appended to the end to find the hamiltonian tour solution.

```
Procedure: MST
Input: coordinates dict C = {'1' : (x, y),' 2' : (x2, y2)...}, N,
       root
Output: Approximate solution of the TSP problem
parent_nodes = prim(coordinates)
adj_list = backtrace adj_list from parent_nodes
euler_walk = []
euler_walk = DFS(euler_walk, adj_list, root)
solution = remove duplicates from euler_walk
solution.append(root)
return solution
```

4.2.2 Data Structures and Implementation Details. Our implementation uses two dictionaries to store minimum cost of adding each node to the tree, and track the parent node of each node. We also maintain a set of nodes, T that are already part of the tree. A priority queue is used to access the the minimum edge cost that connects set T to $V - T$. In this implementation, the parent node dictionary is converted into a dictionary that stores child nodes of each node, or essentially an adjacency list.

4.2.3 Time and Space Complexities. This MST implementation uses both the Prim's (time complexity in $O((V + E)logV)$, where $E = V^2$, with the min-heap implementation) algorithm, and the Depth First Search algorithm (time complexity in $O((V + E))$. Therefore, in the worst case, the MST time complexity is $O((V + E)logV)$. The space complexity is $O(V)$ since the data structures used in the algorithm store information about each node, namely, minimum edge cost, parent nodes, or being of the tree.

4.2.4 Approximation Guarantees.
Claim: Our MST algorithm has an approximation ratio of 2.

Proof. We will prove that the greedy algorithm has an approximation ratio of 2 by showing that the greedy solution is within a factor of 2 from the optimal solution.

Let $G = (V, E)$ be the weighted, undirected graph represented by the TSP problem where V is the set of cities, and E is the set of paths between cities. Cost of an edge (u,v) is defined as Euclidean distance from city u to city v. Define TSP as the total cost of the tour produced in the greedy solution and define TSP^* as the total cost of the tour produced in the in the optimal solution. Lastly, we will define m as the total cost of G's minimum spanning tree (MST).

By the definition of a Minimum Spanning Tree as the lowest cost tree that contains every node, we can construct the following lower bound on TSP^*:

$$TSP^* \geq m$$

By our algorithm, we know that every edge in the MST is visited at most twice in the creation of the euler tour using the DFS algorithm. This is valid since DFS traverses from parent node to child node, only returning to the parent nodes once all routes from the child node have been exhausted. Moreover, we remove duplicate nodes in our implementation. By the triangle inequality principle, the resulting total cost must be less after nodes have been removed.

```
Procedure: Prim
Input: coordinates dict C = {'1' : (x, y),' 2' : (x2, y2)...} ,
       root
Output: Predecessor nodes for each node
Init Q = [], cost = {}, pred = {}
pred[root] = None
for v in nodes other than root: do
    cost[v] = distance(root,v)
    heappush(Q, (distance(root,v), v))
    pred[v] = root
end
tree = set([root])
while Q is not empty: do
    (weight, v) = heappop(Q)
    tree.add(v)
    for u in nodes other than root: do
        if u not in tree and cost[u] > distance(v,u): then
            cost[u] = distance(v,u)
            heappush(Q, (distance(v,u), u))
            pred[u] = v
        end
    end
end
return pred
```

```
Procedure: DFS
Input: visited, child_nodes, root
Output: Euler Walk, tour with duplicates
if root not in visited: then
    visited.append(root)
    for child in child_nodes[root]: do
        self.DFS(visited, child_nodes, child)
    end
end
return visited
```

This allows us to construct the following upper bound on TSP:

$$TSP \leq 2m$$

Therefore, we can write:

$$2TSP^* \geq 2m$$

$$TSP \leq 2m \leq 2TSP^*$$

$$TSP \leq 2TSP^*$$

Since the total cost of the tour produced in the greedy solution is within a factor of 2 from the optimal solution, there is an approximation ratio of 2 for this algorithm. ■

4.3 Simulated Annealing

4.3.1 High-Level Description. Simulated annealing (SA) is the first of our local searches. In general, it is a technique for approximating the global minimum of a given function. Drawing inspiration from the physical process of annealing in metallurgy, SA slowly decreases the probability of accepting worse solutions during search. As a result, SA can move out of local minima while the temperature is still high, but will eventually settle as the temperature cools according to a cooling schedule. Specifically, SA probabilistically decides to move from a state s to candidate neighbor state s' in order to minimize the thermodynamic energy of the system. This iterative process stops once an acceptable solution is found, or until a computational budget is exceeded. In general, the probability that simulated annealing terminates with the global minima converges to 1 as the cooling schedule is lengthened[10]. Practically, the time required for such a cooling schedule will exceed that required of deterministic solutions such as Branch-and-Bound.

In particular, for the TSP problem, each candidate solution is a valid tour with covering all vertices and returning to the origin. In addition, we define the fitness of a candidate solution as the tour length. At the start of the annealing process, we:

- **Generate an initial candidate tour:** For our implementation, we generate an initial tour using the Nearest Neighbors Heuristic[20].
- **Determine a high initial temperature:** Typically, this is 10^10 to 10^30.
- **Determine a stopping temperature:** We used values between 10^{-2} and 10^{-8}.
- **Determine a cooling schedule:** For our implementation, we used a geometric series with factor $\alpha = 0.999$.

During the annealing step, we iteratively generate new candidate solutions by selecting two random cities on the tour (with the exception of the starting city), and then reverse the sub-tour between these two cities[8]. The goal of this candidate generation is to hopefully reduce any local crossovers in tour paths. We then evaluate this new candidate solution for fitness, accepting it if it is strictly better than the currently-accepted solution, or if it isn't accepting it with probability:

$$p = e^{\frac{f(S_{current}) - f(S_{candidate})}{T}}$$

where $f(S)$ is the fitness function on a solution tour S. As we are considering candidate solutions, we are also keeping track of the best solution seen thus far, updating it if ever our candidate is better. At the end of the annealing step, we reduce the temperature according to the cooling schedule, and repeat the process until we reach the stopping temperature.

However, one iteration of the annealing process is not guaranteed to converge to the global optimum, since we use a finite cooling schedule. In addition, as mentioned above, we cannot lengthen the cooling schedule to guarantee convergence, since we cannot guarantee our implementation will do better than deterministic algorithms. Therefore, we implemented restarts whenever our temperature fell to the stopping temperature thresholds. For each restart, we lengthened the cooling schedule, and then restarted the annealing algorithm. However, one notable change is that for the restart, we will use the previously generated best tour as the initial tour for the new annealing process.

```
1  Procedure: SimulatedAnnealing
   Input: Graph G = (V, E), Initial Solution S_0, Initial
          Temperature T_0, Stopping Temperature T_s,
          Geometric Ratio α
   Output: Best solution found for the TSP problem
2  S ← S_0
3  T ← T_0
4  S_best ← S_0
5  while T > T_s do
6  |  S_candidate ← GetNeighbor(S_0)
7  |  if f(S_candidate) < f(S) or
   |     e^((f(S_current) − f(S_candidate))/T) > rand(0, 1) then
8  |  |  S ← S_candidate
9  |  end
10 |  if f(S) < f(S_best then
11 |  |  S_best = S
12 |  end
13 |  T ← αT
14 end
15 return S_best
```

```
1  Procedure: GetNeighbor
   Input: Tour Solution S = {s_1, s_2, ..., s_n}
   Output: Neighbor of input tour solution S
2  i' = rand(2, n)
3  j' = rand(1, n)
4  i = min(i', j')
5  j = max(i', j')
6  S'_segment ← {s_i, s_{i+1}, ..., s_j}
7  S'_segment = reverse(S'_segment)
8  S' ← {s_1, s_2, ..., s_{i−1}} appends S'_segment
9  S' ← S' appends {s_{j+1}, s_{j+2}, ..., s_n}
10 return S'
```

4.3.2 Data Structures and Implementation Details. As with the other algorithms, the graph is represented as an adjacency matrix (or distance matrix), and paths (candidate, best, and current) are plain Python lists. The benefit of simulated annealing is that more complex data structures are not required.

Procedure: `NearestNeighborTour`
Input: Graph $G = (V, E)$
Output: Best solution found for the TSP problem using the heuristic
$S \leftarrow \{\}$
$V' \leftarrow V$
$node \leftarrow$ remove random $v \in V'$ from V'
S appends $node$
while $|V'| > 0$ **do**
  $next \leftarrow$ remove $argmin_v(distance(v, node), v \in V')$ from V'
  $node \leftarrow next$
  S appends $node$
end
return S

4.3.3 Time and Space Complexity. Let n be the number of cities. We will proceed with our analysis by investigating the components of our algorithm:

- **Generate initial tour:** Looking at the implementation of our nearest-neighbor greedy heuristic, which we used to generate an initial tour. Then, our time complexity is $O(n^2)$ and our space complexity is $O(n)$.
- **Getting a neighboring candidate solution:** Picking two random indices and reversing the order of the tour path between them yields time complexity of $O(n)$ and space complexity of $O(n)$.
- **Each anneal step:** Each anneal step acquires a neighbor, but then each other step is constant time, with the exception of the fitness function, which requires time complexity of $O(n)$. Space complexity is still constrained by the acquisition of a candidate solution, with $O(n)$
- **Annealing Loop:** However, the loop of the anneal step is the determining factor, as the number of iterations m of the geometric cooling schedule is determined by T_s, T_0 and α. In fact, using the formula for a geometric series, we find that the number of iterations is:

$$m = \frac{\log T_s - \log T_0}{\log \alpha} + 1$$

  We can conclude that this reduces to a time-complexity of $O(m) = O(\log T_s - \log T_0)$.

Thus, overall, for each iteration of the annealing process, we are in time complexity $O(n)$ and space complexity $O(n)$, but this occurs for $O(m) = O(\log T_s - \log T_0)$ iterations, which means our overall time complexity is $O(n^2 + n(\log T_s - \log T_0))$ for the initial nearest neighbor candidate and the annealing process, respectively. Note, this is complicated by the restart implementation, which is not suitable for time and space complexity analysis. Overall space complexity is still $O(n)$.

4.4 Genetic Algorithms

4.4.1 High-Level Description. Genetic algorithm is the second of our local searches. Genetic algorithm is inspired by The famous 'theory of natural evolution' by Charles Darwin. This states that over a long period of time, and a sufficiently large population, the fit survive and the weak do not. Following this, we initialize a population, either randomly, or using some heuristics, then, in each iteration, we select some of the fittest members of the current population, and 'breed' them, to get new members. We also 'mutate' some of the members with a small probability. Then, of all these new members, and the old members, we select the fittest members, and this forms our new population. We repeat this process, and end up getting a close to optimal path.

Specifically, in TSP, each individual in the population is a valid path covering all cities and returning to origin city. Initially, we use All Nearest Neighbor heuristics [14], as well as random initialization to create a population. Then, we use Order Crossover Operator [2], on our paths to create new offsprings, that hopefully retain the good properties from both their parents. We also use mutation, by randomly swapping 2 cities in the path, to potentially explore new paths that we would otherwise not be in our search space. The fitness function that we use is just 1/(path length), as our objective is just to decrease path length of tour, and that is directly achieved by an increase in fitness.

Procedure: `globalCompetition`
Input: population, newPopulation
Output: finalPopulation containing fittest individuals
$population \leftarrow sort(population)$
$newPopulation \leftarrow sort(newPopulation)$
$j, k \leftarrow 0, 0$
for i *in range(*pop_size*)* **do**
  $finalPopulation[i] \leftarrow max(population[j], newPopulation[k]$ **if** *population[j]>newPopulation[k]* **then**
    $j+ = 1$
  end
  else
    $k+ = 1$
  end
end
return $finalPopulation$

4.4.2 Data Structures and Implementation Details. As with the other algorithms, the graph is represented as an adjacency matrix (or distance matrix), and paths (candidate, best, and current) are plain Python lists. It also has the same benefit as simulated annealing, that more complex data structures are not required.

4.4.3 Time and Space Complexities. Let n be the number of cities, p be the population size. We will proceed with our analysis by investigating the components of our algorithm:

- **Generate initial tour:** Looking at the implementation of our nearest-neighbor greedy heuristic, which we used to generate one initial tour. Then, our time complexity is $O(n^2)$ and our space complexity is $O(n * p)$. We do this for all nodes as a starting node, so the overall complexity is $O(n^3)$.
- **Getting a mutation/crossover pool** Picking random/fittest individuals and returning them in a list, O(p). Space complexity O(p*n)

```
Procedure: GeneticAlgorithm
Input: Graph G = (V, E)
Output: Best solution found for the TSP problem
population ← allNearestNeighbors(G)
population.append(randomPopulation(G))
while time ≤ cutoffTime do
    mutationPool ← selectIndividualsForMutaion(population)
    newPopulation ← mutate(mutationPool)
    crossoverPool ← selectIndividualsForCrossover(population)
    newPopulation ← crossover(crossoverPool)
    population ← globalCompetition(population, newPopulation)
end
return getBestIndividual(population)
```

- **Each mutate step:** Each mutate step is just random swapping of 2 cities, so constant time.
- **Evaluating fitness:** Evaluating fitness for a single path takes O(n) time. So for all indivuals its n*p
- **Each crossover step:** In each crossover step , we iterate over the 2 individuals we cross, so O(n) time.
- **Genetic Algorithm Loop:** However, the loop of moving from one population to a new, better, or atleast similar population is the determining factor.

Thus, overall, for each iteration of the annealing process, we are in time complexity $O(p + n * p + n * p)$ and space complexity $O(p * n)$, but this occurs for m iterations, which means our overall time complexity is $O(n^3 + m(p + n * p + n * p))$.

5 EMPIRICAL EVALUATION

5.1 Branch-and-Bound

We conducted two experiments to evaluate the performance of the branch-and-bound algorithm. For these experiments, we used Dell PowerEdge 8220 servers, whose specification is detailed in Table 1. The code was implemented in Python, and the interpreter version used was 3.6.9.

Table 1: Dell PowerEdge 8220 server specification

CPU	Two Intel E5-2660 v2 10-core CPUs at 2.20 GHz (Ivy Bridge)
RAM	256 GB ECC Memory (16x16 GB DDR4 1600MT/s dual rank RDIMMs)
Disk	Two 1 TB 7.2K RPM 3G SATA HDDs

The first experiment used a best-first approach, selecting the partial solution with the lowest lower bound to expand at each iteration. When using a cutoff-time of 6 hours, most instances did not even reach a solution, which indicates that this approach is expanding the tree in a more breadth-first (i.e. horizontal) way rather than depth-first (i.e. vertical) way. The results for each instance are reported in Table 2. Instances *Cincinnati* and *UKansasState* have only 10 vertices so the branch-and-bound algorithm quickly found their optimal solutions. Instance *Atlanta* has 20 vertices and its optimal solution was found in less than 4 hours. For all other instances, the best solution found was exactly the initial upper bound (i.e. the cost of a trivial route 1-2-3-...-N).

Table 2: Branch-and-Bound Experiment 1 – Best-first approach – Running Time: 6 Hours

Instance	Time (s)	Optimal	Branch-and-Bound	. Error
Atlanta	13323.69	2003763	2003763	0.0000%
Berlin	21600.00	7542	22205	194.4179%
Boston	21600.00	893536	2405366	169.1963%
Champaign	21600.00	52643	227486	332.1296%
Cincinnati	1.25	277952	277952	0.0000%
Denver	21600.00	100431	600924	498.3451%
NYC	21600.00	1555060	7753368	398.5896%
Philadelphia	21600.00	1395981	4585416	228.4727%
Roanoke	21600.00	655454	6994624	967.1419%
SanFrancisco	21600.00	810196	6053947	647.2201%
Toronto	21600.00	1176151	9812537	734.2923%
UKansasState	0.15	62962	62962	0.0000%
UMissouri	21600.00	132709	692495	421.8146%

To improve the results obtained within a time window, we also tested an approach that expands the tree in a more depth-first (i.e. vertical) way, without completely ignoring the lower bound of each partial solution. This idea was proposed by classmate Conlain D. Kelly, in Piazza [1]. The key idea is to expand, at each iteration, the partial solution with the lowest value of lb/k, where lb is its lower bound and k is the length of its path. By prioritizing longer paths, we keep the frontier set smaller and reach solutions faster, at the cost of having less pruning in the tree. The results for each instance are reported in Table 3. This hybrid approach presented better results than the obtained in the first experiment with a time-window of only 10 minutes.

Table 3: Branch-and-Bound Experiment 2 – Hybrid Approach – Running Time: 10 minutes

Instance	Time (s)	Optimal	Branch-and-Bound	. Error
Atlanta	600.00	2003763	2025073	1.0635%
Berlin	600.00	7542	8838	17.1838%
Boston	600.00	893536	971877	8.7675%
Champaign	600.00	52643	57665	9.5397%
Cincinnati	1.88	277952	277952	0.0000%
Denver	600.00	100431	115377	14.8819%
NYC	600.00	1555060	1670828	7.4446%
Philadelphia	600.00	1395981	1447428	3.6854%
Roanoke	600.00	655454	6994624	967.1419%
SanFrancisco	600.00	810196	926744	14.3852%
Toronto	600.00	1176151	1228732	4.4706%
UKansasState	0.12	62962	62962	0.0000%
UMissouri	600.00	132709	166779	25.6727%

5.2 MST-Approximation

We conducted two experiments to evaluate the performance of the MST 2-approximation algorithm. For both experiments, we used a personal computer, whose specification is detailed in Table 4. The code was implemented in Python, and the interpreter version used was Python 3.7.

Table 4: Personal Computer 1

CPU	One Intel CPU Core i5 at 2 GHz
RAM	8 GB Memory (1867 MHz LPDDR3)
Disk	256GB PCIe-based onboard SSD

For the first experiment, we used the first city in instance tsp file as the root node which then became the starting and ending node of the generated tour. The results for each instance are reported in Table 5. This experiment was a proof of concept to show that an arbitrary option of MST root for every instance is able to generate a tour quickly, in this case in less than 0.2 second, with a cost within the approximation guarantee.

Table 5: Single Root MST Approximation Solution vs. Optimal

Instance	Time (s)	Optimal	MST	. Error
Atlanta	0.0007	2003763	2380448	18.80%
Berlin	0.004	7542	10402	37.92%
Boston	0.002	893536	1150963	28.81%
Champaign	0.005	52643	65712	24.83%
Cincinnati	0.0002	277952	301216	8.37%
Denver	0.012	100431	134748	34.17%
NYC	0.009	1555060	2027107	30.36%
Philadelphia	0.002	1395981	1646249	17.93%
Roanoke	0.196	655454	838282	27.89%
SanFrancisco	0.015	810196	1134989	40.09%
Toronto	0.032	1176151	1675105	42.42%
UKansasState	0.0003	62962	68090	8.14%
UMissouri	0.018	132709	178249	34.32%

For the second experiment, we iterated over every node using that node as root of the MST in the 2-approximation algorithm. This root then became the starting and ending node of the generated tour. Although the MST is the same in each case, the tour generated from the MST may differ depending on which root node is chosen. Therefore, we kept track of the best solution found while iterating over the different MST roots. A cutoff-time of 60 seconds was used to ensure that each instance would finish running, trying every MST root option. Most instances ran in less than 1 second. The results for each instance are reported in Table 6. This iterative approach was used to generate the most optimal MST approximation solution. A trivial tour of $1 - 2 - 3 - \ldots - N - 1$ was used as the initial solution.

5.3 Simulated Annealing

We conducted three experiments to evaluate the performance of the simulated annealing algorithm. For these experiments, we used a personal computer, whose specification is detailed in Table 7. The code was implemented in Python, and the interpreter version used was Python 3.7.

Table 6: Iterative MST Approximation Solution vs. Optimal

Instance	Time (s)	Optimal	MST	. Error
Atlanta	0.010	2003763	2270785	13.33%
Berlin	0.224	7542	9550	26.62%
Boston	0.091	893536	1028494	15.10%
Champaign	0.249	52643	62395	18.52%
Cincinnati	0.002	277952	297490	7.03%
Denver	0.795	100431	133065	32.49%
NYC	0.627	1555060	2003747	28.85%
Philadelphia	0.045	1395981	1626820	16.54%
Roanoke	42.779	655454	808235	23.31%
SanFrancisco	1.492	810196	1060717	30.92%
Toronto	3.147	1176151	1618300	37.59%
UKansasState	0.002	62962	65561	4.13%
UMissouri	1.846	132709	165116	24.42%

Table 7: Personal Computer 2

CPU	One Intel i7-7700k 4-core CPUs at 4.6 GHz (Kaby Lake)
RAM	32 GB Memory (4x8 GB DDR4 2133 Mhz dual channel DIMMs)
Disk	One 1 TB SSD

Each experiment utilized a simulated annealing approach with restarts upon cooling schedule conclusion, as well as temperature increase restarts upon convergence of solutions.

For the first experiment, each given instance was run ten times, with a maximum cut-off time of 60s, with random seeds. Average final result times, average relative error, and percentage optimal found are shown in Table 8. Box-and-whisker plots over each of the 10 iterations per instance are shown in Figure 2.

Table 8: Simulated Annealing vs. Optimal

Instance	Time (s)	Optimal	Simulated Annealing	Rel. Error
Atlanta	2.989	2003763	2003763	0.00%
Berlin	19.473	7542	7775.9	3.10%
Boston	26.686	893536	896475.6	0.33%
Champaign	27.265	52643	53154	0.97%
Cincinnati	0.418	277952	277952	0.00%
Denver	30.615	100431	105112	4.66%
NYC	21.125	1555060	1584206	1.87%
Philadelphia	8.731	1395981	1396388	0.03%
Roanoke	21.193	655454	961891.6	46.75%
SanFrancisco	24.977	810196	871425	7.56%
Toronto	27.402	1176151	1282788	9.07%
UKansasState	0.373	62962	62962	0.00%
UMissouri	23.75777778	132709	143311.4	7.99%

For the second experiment, we chose to generate qualified run-time distributions for our 10 Atlanta instances and 10 Champaign instances. We held a threshold solution quality constant, and ran 50 iterations for each instance and measured the probability of our algorithm generating a solution as good as the threshold solution quality, varied over run-time. The results for Atlanta are shown in Figure 4 and the results for Champaign are shown in Figure 5.

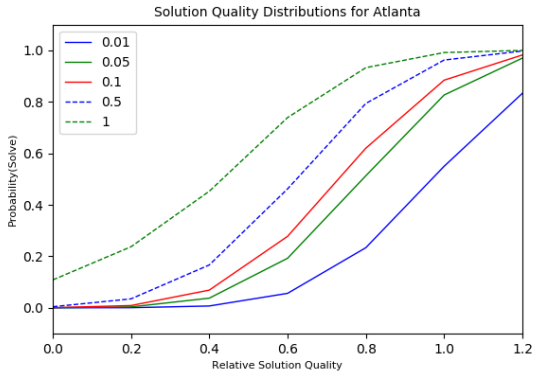

Figure 1: Simulated Annealing: Solution quality distributions for Atlanta.

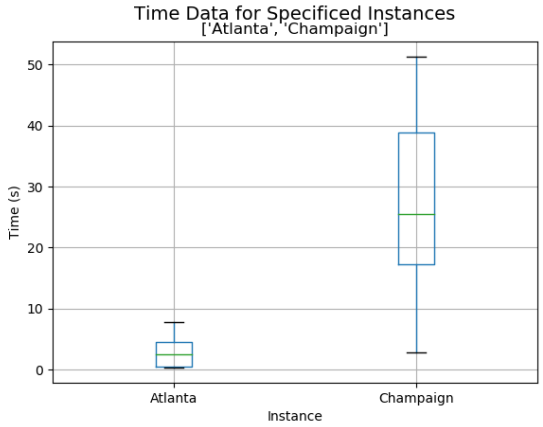

Figure 2: Boxplot: Average run-times for Atlanta and Champaign.

For the third experiment, we chose to generate solution quality distributions for our 10 Atlanta instances and 10 Champaign instances. We held the maximum run-time constant and ran 50 iterations per instance to measure the probability of our algorithm generating a solution as good as a range of threshold solution qualities. The results for Atlanta are shown in Figure 1 and the results for Champaign are shown in Figure 3.

5.4 Genetic Algorithms

We conducted three experiments to evaluate the performance of the simulated annealing algorithm. For these experiments, we used a personal computer, whose specification is detailed in Table 9. The code was implemented in Python, and the interpreter version used was Python 3.7.

For the first experiment, each given instance was run ten times, with a maximum cut-off time of 60s, with random seeds. Average final result times, average relative error, and percentage optimal

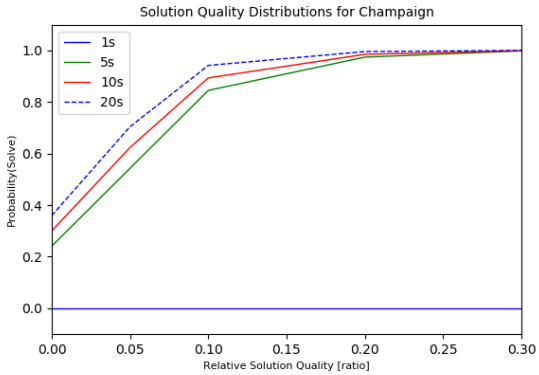

Figure 3: Simulated Annealing: Solution quality distributions for Champaign.

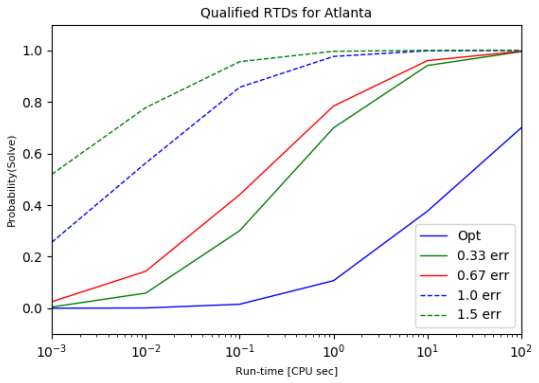

Figure 4: Simulated Annealing: Qualified run-time distributions for Atlanta, zoomed in.

Table 9: Personal Computer 3

CPU	Intel(R) i7-4510U 2-core CPUs at 2.00 GHz
RAM	8 GB Memory
Disk	One 1 TB HDD

found are shown in Table 10. Box-and-whisker plots over each of the 10 iterations per instance are shown in Figure 7.

For the second experiment, we chose to generate qualified run-time distributions for our 10 Atlanta instances and 10 Champaign instances. We held a threshold solution quality constant, and ran 50 iterations for each instance and measured the probability of our algorithm generating a solution as good as the threshold solution quality, varied over run-time. The results for Atlanta are shown in Figure 9 and the results for Champaign are shown in Figure 10.

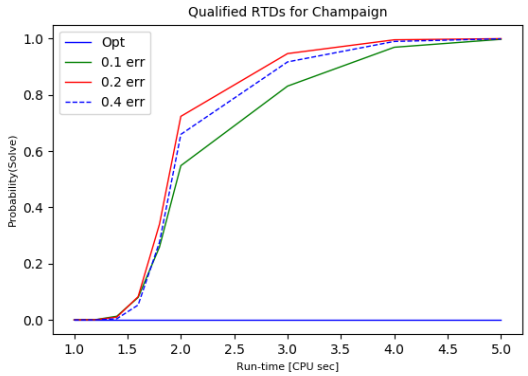

Figure 5: Simulated Annealing: Qualified run-time distributions for Champaign

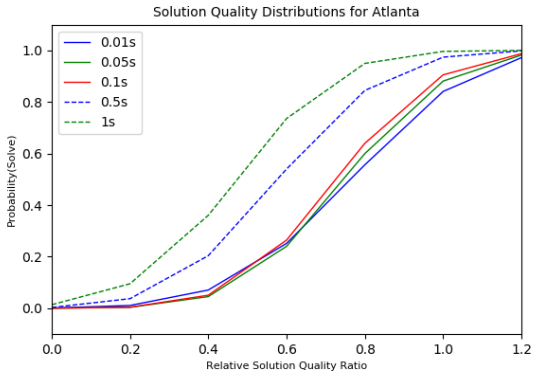

Figure 6: Genetic Algorithm: Solution quality distributions for Atlanta.

Table 10: Genetic Algorithms vs. Optimal

Instance	Time (s)	Optimal	Genetic Algorithm	Rel. Error
Atlanta	4.529	2003763	2003763	0.00%
Berlin	39.233	7542	7784.9	3.22%
Boston	42.802	893536	948353.3	6.13%
Champaign	43.57	52643	53434.3	1.50%
Cincinnati	1.3	277952	277952	0.00%
Denver	56.349	100431	111204.7	10.73%
NYC	52.971	1555060	1623225	4.38%
Philadelphia	18.314	1395981	1395981	0.00%
Roanoke	52.446	655454	764587.5	16.65%
SanFrancisco	53.414	810196	842513	3.99%
Toronto	57.159	1176151	1220635	3.78%
UKansasState	1.332	62962	62962	0.00%
UMissouri	52.794	132709	147623.5	11.24%

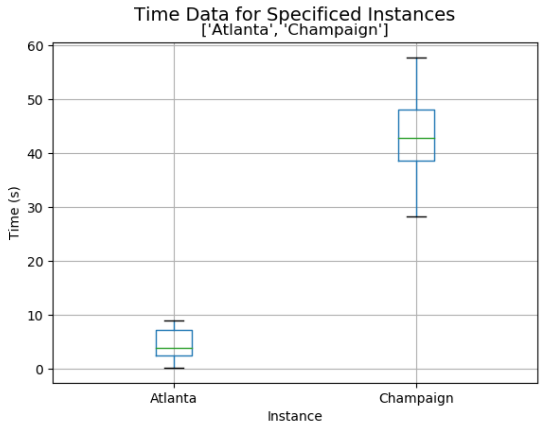

Figure 7: Boxplot: Average Genetic Algorithm run-times for Atlanta and Champaign.

For the third experiment, we chose to generate solution quality distributions for our 10 Atlanta instances and 10 Champaign instances. We held the maximum run-time constant and ran 50 iterations per instance to measure the probability of our algorithm generating a solution as good as a range of threshold solution qualities. The results for Atlanta are shown in Figure 6 and the results for Champaign are shown in Figure 8.

6 DISCUSSION

The best version of the branch-and-bound algorithm (i.e. the one using the hybrid approach, which takes both the lower bound and the path length into account for selecting the partial solution to expand next) was set to run for 10 minutes per instance, and we empirically verified that it has only practical application for small instances, with at most 20 cities. Specifically, we got a relative error of around 1% for the instance *Atlanta*, which has 20 vertices. For some instances with more than 40 and less than 60 vertices (e.g. *Boston* and *Champaign*), we still got relative errors of less than 10%. This was expected, considering that the branch-and-bound algorithm is exact (i.e. returns the optimal solution if given enough time), at the cost of having exponential execution time in the worst case.

The MST 2-approximation algorithm was found to generate reasonable results in a very short amount of time. In the iterative MST experiment, most instances were able to find the best MST solution in less than one second. Additionally, all instances were able to find the single-root MST solution in less than 0.2 second and all solutions were still within the approximation guarantee. This was expected, given that the MST algorithm is a heuristic developed to sacrifice quality while generating solutions quickly and within a given quality guarantee. The iterative MST results were also only provided marginal improvements when compared to the single-root results, while requiring a much longer amount of time. On average, the iterative algorithm improved tour quality by 4.76% while taking 6509% more time. A remarkable example of this is the Roanoke instance where the tour cost was reduced from 838282 to 808235 while taking 42.8 seconds as opposed to 0.2

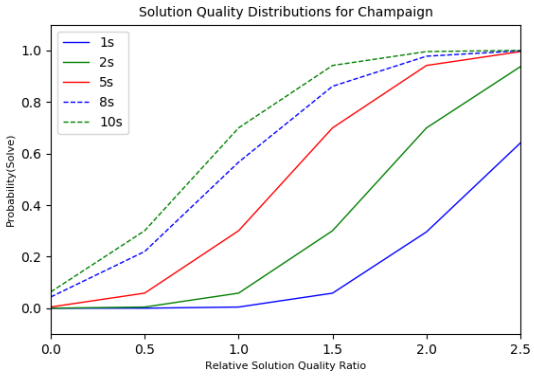

Figure 8: Genetic Algorithm: Solution quality distributions for Champaign.

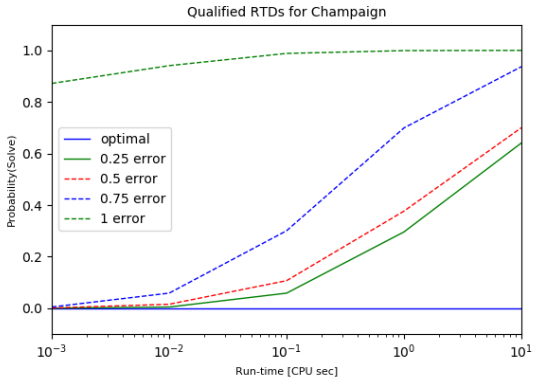

Figure 10: Genetic Algorithm: Qualified run-time distributions for Champaign

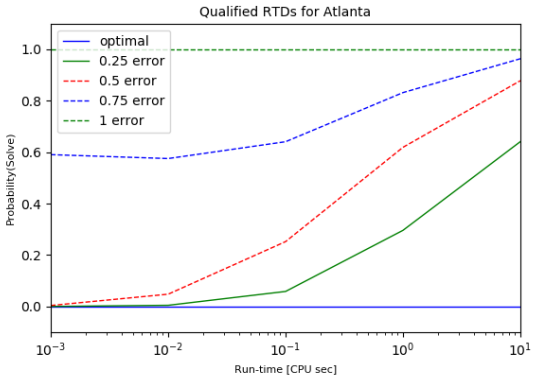

Figure 9: Genetic Algorithm: Qualified run-time distributions for Atlanta, zoomed in.

seconds. This suggests that the MST approximation should only be used if a reasonably accurate solution needs to be generated quickly, and other iterative methods are preferable if more time is available.

Simulated annealing as a probabilistic meta-heuristic for the traveling salesman problem provided good results for graph sizes of the order 20-50 nodes. However, since the amount of local minima that need to be probabilistically overcome to reach the global minimum increases significantly with respect to the number of nodes, simulated annealing would need much slower cooling schedules for high node numbers. We can see this in the QRTDs for both Atlanta ($n_{atl} = 20$) and for Champaign ($n_{champ} = 55$), as compute time for within 0.2-0.3 of OPT for the larger dataset is roughly as fast for the smaller dataset, but optimal values take much longer to find for larger datasets. This supports the analysis that early annealing finds better solutions quickly, but slows down as the number of crossovers producing better results decreases.

Genetic Algorithm is a local search algorithm for the traveling salesman problem provided good results for graph sizes of the order 20-50 nodes. However, since the algorithm relies a lot on randomly exploring new states(mutation), if the number of nodes is higher, we need a lot of iterations and big population size. We can see this in the QRTDs for both Atlanta ($n_{atl} = 20$) and for Champaign ($n_{champ} = 55$), as compute time for within 0.5 of OPT for the larger dataset is roughly as fast for the smaller dataset, but optimal values take much longer to find for larger datasets. This supports the analysis that early populations finds better solutions quickly, but slows down as the number of crossovers producing better results decreases.

7 CONCLUSION

We implemented four algorithms to solve the TSP. The branch-and-bound algorithm is exact (i.e. returns the optimal solution if given enough time), at the cost of having prohibitively large execution times for the reasonably large instances found in practical applications. The MST approximation algorithm provides a polynomial time solution, at the cost of not providing a high quality solution but with an approximation guarantee of 2. The simulated annealing provides a meta-heuristic for local search with probabilistic accepting of strictly worse candidates to escape local minima. The genetic algorithm has no guarantees of optimality, or even an approximation bound, but tends to run reasonably fast while giving very good results most of the time. Quality of results and runtime also depends on hyper parameters like population size, number of mutations, crossovers and evaluation function.

We carried out an empirical evaluation of these algorithms, solving thousands of instances overall. Our dataset comprises 13 distinct instances of the most common case of the TSP, where cities correspond to points in a 2D plane and their distances are Euclidean (L2 norm). The smallest instances have 10 cities only, while the biggest instance has 230 cities.

We conducted two experiments to evaluate the performance of the branch-and-bound algorithm: the first one using a best-first

approach, selecting the partial solution with the lowest lower bound to expand at each iteration, and the second one using a hybrid approach that also takes the depth of a partial solution into account. The second approach (hybrid) was set to run for 10 minutes and clearly outperformed the first one, which was set to run for 6 hours. From our empirical evaluation, we concluded the branch-and-bound algorithm has only practical application for small instances, with at most 20 cities.

We conducted two experiments to evaluate the performance of the MST 2-approximation algorithm: the first one using a single root approach, using the first city in the instance tsp file as the root of the MST, and the second one using an iterative approach that considers all possible roots and outputs the best result. The second approach (iterative) generated better quality results than the first one but ran for a much longer time, which defeats the purpose of an approximation providing a quick result. From our empirical evaluation, we concluded that the MST algorithm is only advantageous if a reasonably accurate solution is needed in a short amount of time, and other iterative methods are preferable if more time is available.

In addition, we conducted three sets of experiments to evaluate the performance of simulated annealing: 1) 10 iterations of the algorithm with $max_time = 60s$ for every instance, 2) qualified run-time distributions for the Atlanta and Champaign instances, and 3) solution quality distributions for the same instances. We can conclude that simulated annealing is out-performed by branch-and-bound for exceedingly small instances, but out-performs branch-and-bound and the heuristic approximation for larger instances, up to a point. Empirically, it seems that point is within the 50-100 range.

Similarly, we conducted 3 sets of experiments to evaluate the performance of genetic algorithm: 1) 10 iterations of the algorithm with $max_time = 60s$ for every instance, 2) qualified run-time distributions for the Atlanta and Champaign instances, and 3) solution quality distributions for the same instances. We can conclude that genetic algorithm is out-performed by branch-and-bound for exceedingly small instances, but out-performs branch-and-bound and the heuristic approximation and even Simulated Annealing for larger instances, up to a point. Of all the algorithms that we implemented, Genetic Algorithm seems to do best for graph with large nodes (see Roanoke).

Finally, our results show for most n, local search outperforms branch-and-bound, except for very small n. While the approximation algorithm exhibits better time-performance, the nature of the approximation algorithm yields significantly worse fitness results. Furthermore, the genetic algorithm solver outperforms the simulated annealing solver for larger n in terms of fitness performance. Considering the differences in hardware, it is likely that genetic algorithms are more time-performance than simulated annealing as well.

The Travelling Salesman Problem (TSP) has been extensively studied for decades, with many approaches being proposed to solve it optimally or not. Nowadays, it still sustains academic and practical interest. Therefore, it was a great fit for our empirical study of different approaches to solve (or approximate the solution of) NP-hard optimization problems. In this way, we got hands-on experience in solving an intractable problem that is of practical importance.

REFERENCES

[1] [n. d.]. Branch-and-Bound alternative to best-first approach. https://gatech.instructure.com/courses/60478/external_tools/81. Accessed: 2019-11-28.

[2] M Sajid I Hussain A Shoukry S Gani A Hussain, Y Muhammad. 2017. Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator. *Computational Intelligence and Neuroscience* 2017, 7430125 (2017). https://doi.org/10.1155/2017/7430125

[3] Emile HL Aarts, Jan HM Korst, and Peter JM van Laarhoven. 1988. A quantitative analysis of the simulated annealing algorithm: A case study for the traveling salesman problem. *Journal of Statistical Physics* 50, 1-2 (1988), 187–206.

[4] Sanjeev Arora. 1998. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)* 45, 5 (1998), 753–782.

[5] Egon Balas and Paolo Toth. 1983. *Branch and bound methods for the traveling salesman problem.* Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA MANAGEMENT SCIENCES RESEARCH GROUP.

[6] Yair Bartal and Lee-Ad Gottlieb. 2013. A linear time approximation scheme for Euclidean TSP. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE, 698–706.

[7] Heinrich Braun. 1990. On solving travelling salesman problems by genetic algorithms. In *International Conference on Parallel Problem Solving from Nature*. Springer, 129–133.

[8] Joseph Chang. 2007. More on Markov chains, Examples and Applications). *Stochastic Processes* (02 2007), 43–86.

[9] Xiutang Geng, Zhihua Chen, Wei Yang, Deqian Shi, and Kai Zhao. 2011. Solving the traveling salesman problem based on an adaptive simulated annealing algorithm with greedy search. *Applied Soft Computing* 11, 4 (2011), 3680–3689.

[10] Vincent Granville, Mirko Krivanek, and J.-P Rasson. 1994. Simulated Annealing: A Proof of Convergence. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 16 (07 1994), 652 – 656. https://doi.org/10.1109/34.295910

[11] John Grefenstette, Rajeev Gopal, Brian Rosmaita, and Dirk Van Gucht. 1985. Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, Vol. 160. Lawrence Erlbaum, 160–168.

[12] Michael Held and Richard M Karp. 1970. The traveling-salesman problem and minimum spanning trees. *Operations Research* 18, 6 (1970), 1138–1162.

[13] Michael Held and Richard M Karp. 1971. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical programming* 1, 1 (1971), 6–25.

[14] Youssef Harrath Jihene Kaabi. 2019. Permutation rules and genetic algorithm to solve the traveling salesman problem. *Arab Journal of Basic and Applied Sciences* (2019), 283–291.

[15] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.

[16] Joseph SB Mitchell. 1999. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal on computing* 28, 4 (1999), 1298–1309.

[17] Heinz Mühlenbein, Martina Gorges-Schleuter, and Ottmar Krämer. 1988. Evolution algorithms in combinatorial optimization. *Parallel computing* 7, 1 (1988), 65–85.

[18] Jean-Yves Potvin. 1996. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research* 63, 3 (1996), 337–370.

[19] Satish B Rao and Warren D Smith. 1998. Improved approximation schemes for geometrical graphs via\spanners" and\banyans,". In *30th ACM Symposium on Theory of Computing (STOC'98)*. Citeseer, 540–550.

[20] Alexander Schrijver. 2001. On the History of Combinatorial Optimization (Till 1960). *Handbooks in Operations Research and Management Science* 12 (08 2001). https://doi.org/10.1016/S0927-0507(05)12001-5

[21] David B Shmoys and David P Williamson. 1990. Analyzing the Held-Karp TSP bound: A monotonicity property with application. *Inform. Process. Lett.* 35, 6 (1990), 281–285.

[22] Christopher C Skiścim and Bruce L Golden. 1983. Optimization by simulated annealing: A preliminary computational study for the tsp. In *Proceedings of the 15th conference on Winter Simulation-Volume 2*. IEEE Press, 523–535.

[23] S Tschoke, R Lubling, and Burkhard Monien. 1995. Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. In *Proceedings of 9th International Parallel Processing Symposium*. IEEE, 182–189.

[24] Ton Volgenant and Roy Jonker. 1982. A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation. *European Journal of Operational Research* 9, 1 (1982), 83–89.