CSE-6140 Project: Travelling Salesman Problem

Chris Fleisher
GTID: 903421975
cfleisher3@gatech.edu

Sahith Dambekodi
GTID: 903542538
sdambekodi3@gatech.edu

Sanmeshkumar Udhayakumar
GTID: 902969263
sudhayakumar3@gatech.edu

Monica Gupta
GTID: 903514001
mgupta334@gatech.edu

INTRODUCTION

The travelling salesman problem (TSP) is a well known combinatorial problem. The idea of TSP is to find the shortest tour of a group of cities without visiting any town twice. Practically, it implies the construction of a Hamiltonian cycle within a weighted undirected graph. Therefore, this is a problem of combinatorial graph search [4]. The applications of the TSP problem are numerous – in computer wiring, job scheduling, minimizing fuel consumption in aircraft, vehicle routing problem, robot learning, etc. We have solved this problem using 4 different approaches: Branch and Bound, Approximation, .

PROBLEM DEFINITION

We define the TSP problem as follows: Given a complete undirected graph G that has a nonnegative integer cost associated with each edge, we must find a hamiltonian cycle of G with minimum cost.

RELATED WORK

The TSP is perhaps one of the most explored problems in computer science with numerous implementations and heuristics til date. Multiple exact and approximate approaches have been formulated in an attempt to find time efficient solutions. Some exact algorithms are branch-and-cut and the cutting plane method. The Lin-Kernighan heuristic is generally considered to be the best. From a theoretical standpoint, Christofides algorithm provides a 3/2-approximation for problem instances where edge costs satisfy the triangle inequality. The best to date approximation bound for graphic TSP is 7/5 (7/5-approximation for graphic TSP, 3/2 for the path version, and 4/3 for two-edge-connected subgraphs).

1. Branch and Bound

Branch and bound finds an exact solution to the TSP problem and the algorithm is of worst-case complexity [13]. It constructs a tree rooted at the start vertex and grows the tree by computing a lower bound on the best possible solution obtained by exploring the subtree at that node [6]. If the bound is worse than the current best bound found so far, we do not explore the subtree at that node. In our project, we have the coordinates of V cities. Modeling the cities as a graph $G(V,E)$, we have a fully connected graph where the edge weight is the Euclidean distance between the two vertices(cities). [11] [9]

Without loss of generality we begin the path with vertex 0 and set it as the root node in the tree. The bounding function uses the 2 shortest edges for each vertex, i.e. the lower bound of the root node is the sum of the 2 shortest edges for each vertex V and this sum is divided by 2 to remove repeated edge costs. While exploring a new node in the tree, say vertex 1, we add the edge cost 0-1 and subtract half the sum of shortest edges of 0 and 1 to find the lower bound for vertex 1. While exploring directly below vertex 1, say vertex 2, we add the edge cost 1-2 and subtract half the sum of shortest edge of 2 and second shortest edge of 1 to find the lower bound for vertex 2. We use an $n \times n$ adjacency matrix to represent the graph G where the matrix element $adj[i][j]$ represents the euclidean distance between vertex i and j.

Detailed description of the branch and bound algorithm:

1. Initialise the current bound to half the sum of the two shortest edges of all vertices. Maintain a boolean visited array to keep track of visited vertices, initialise to all false values. Set $visited[0] = true$. Maintain an integer path array to keep track of the current path. Set $path[0] = 0$ since we begin the path with vertex 0.

2. We use a recursive function TSPRec to recursively create the tree from level 1 onwards. The function takes the adjacency matrix, current bound, best weight, current weight, level and current path as parameters. The current weight is the sum of the edge costs of the path created so far. The current bound is the lower bound found at the previous level. The best weight can only be updated at level n.

3. While the level is not equal to n, we iterate through all vertices starting from 0 to $n-1$. If the vertex has not been visited, we add the edge cost, $adj[curr\_path[level-1]][i]$ to the current weight and then calculate the new lower bound for vertex i using the current bound.

4. If the new lower bound is lesser than the best weight so far, we mark the vertex as visited, add it to the current path and call TSPRec for the next level, i.e. go to step 3. If the new lower bound is not lesser that the best weight so far, we revert to the old current weight, current path and visited array and iterate to the next vertex.

5. If we reach level n, we add the weight from the last vertex back to the first to the current weight and check if it is lesser than the best weight. If lesser, we copy the current path to the final path and update the best weight. Otherwise, we return.

Pseudocode:

```
procedure TSP(adj)
    Initialise path as all -1                          ▷ current path
    Initialise visited as all false
    for $i = 0$ to $n-1$ do                 ▷ find initial lower bound
        c_bound += (firstMin(adj,i)+secMin(adj,i))
    c_bound /= 2.
    visited[0] = true
    path[0] = 0
    b_wt = ∞                                           ▷ best weight
    c_wt = 0                                         ▷ current weight
    TSPRec(adj,c_bound,b_wt,c_wt,1,path,visited)
```

```
procedure TSPREC(adj,c_bound,b_wt,c_wt,lvl,path,visited)
    if lvl == n then
        curr_res = c_wt+adj[path[lvl-1]][path[0]]
        if curr_res < b_wt then
            copyToFinal(path,final_path)
        return
    for $i = 0$ to $n-1$ do
        if visited[i]==false then
            tmp = c_bound
            c_wt += adj[path[lvl-1]][i]
            c_bound = calcBound(adj,lvl,c_bound,i)
            if c_bound+c_wt<b_wt then
                c_path[lvl] = i
                visited[i] = true
                BNBRec(adj,c_bound,b_wt,c_wt,lvl+1,path,
    visited)
            else
                revert(c_wt,c_bound,visited)
```

Time complexity:

Worst case, branch and bound will never prune a node and its complexity remains the same as brute force complexity i.e. $O(2^n * n^2)$. Worst case, there will be a total of 2^n nodes and we perform n^2 in each.

Space complexity:

Since the graph is fully-connected, the adjacency matrix takes up $O(n^2)$ space. The other arrays like visited, current path etc are $O(n)$ and thus the space complexity of the whole algorithm is $O(n^2)$.

Strengths:

Branch and bound gives the exact solution i.e. for the given cities, it returns the tour of all cities without visiting any city twice with the minimum possible total distance.

Weaknesses:

The complexity of branch and bound is exponential and it slows down considerably as the number of cities increases, making it impractical to use.

2. Approximation: MST + DFS

Our approximation algorithm theoretically is supposed to guarantee an approximation ratio <= 2. At a very high level, it achieves this approximation through, creating a minimum spanning tree (MST) and doing a depth first search (DFS) in which vertices/cities that are already visited are not marked repeatedly. The reason this algorithm works is because this is a metric TSP problem, where d(i,k) <= d(i,j) + d(j,k). [2] [5] [7]

1. When we find MST, MST <= optimal tour because MST goes through least weight edges, and doesn't form a cycle, so the optimal tour solution can't be less than the distance on a MST.

2. Doing depth first search on MST in which vertexes are allowed to be visited multiple times, causing each edge in MST to be visited twice, makes a pseudo-tour, PT = 2*MST <= 2* OPT tour, since MST <= optimal tour.

3. When removing vertexes from PT, such that each vertex is visited only the first time, we get Tapprox. Tapprox distance <= PT because d(i,k) <= d(i,j) + d(j,k). PT has distances d(i,j) + d(j,k) where i is vertex a, j is vertex a again when it is repeated, and k is the vertex after the visiting vertex a repeatedly. This PT with distances d(i,j) + d(j,k) has to be less than Tapprox with distances d(i,k).

4. So now combining all the relationships between variables, Tapprox <= PT <= 2*OPT.

Prim's Algorithm for MST

For the MST, the data structure used is an adjacency matrix that is a numCities by numCities 2d matrix that stores the euclidean distance between cities. We then use Prim's algorithm to find the MST on this adjacency matrix.

A brief description of the steps of Prim's algorithm is below:

1. Create boolean array "mstSet" which indicates whether a vertex at index i has been included in mstSet. Create array "parent", that at index v indicating vertex V, will be filled with the parent index 'u', of the child vertices added as edges and vertices are included in the MST. Create a variable, cost, that keeps track of cost of each vertex. initialize all costs as infinity initially, and indicate a starting vertex by setting this vertex as 0 cost [10].

2. while mstSet doesn't include all vertices:

- Pick a vertex u which is not in mstSet and has minimum cost, and add it to mstSET
- update cost of adjacent vertices to u, by assigning it to the weight of edge (u,V) is that weight is less than the current costs. If the cost, is updated of vertex v, update the parent, u, for that vertex that gives that minimum edge weight.

Prim's Algortihm for MST Pseudo-code

```
1: procedure PRIMS
2:     NumberOfCities = number of cities or
3:     number of vertices in graph
4:     create array "cost" size NumberOfCities, or the vertexes in an instant; init to $\infty$
5:     let $MST$ be array size $NumberOfCities$
6:     $MST[i] \leftarrow False \quad \forall i \in \{0, \cdots, |MST|\}$
7:     let $P$ be parent array size $NumberOfCities$
8:     $P[i] \leftarrow -1 \quad \forall i \in \{0, \cdots, |P|\}$
9:     $M \leftarrow$ adjacency Matrix
10:    Get random starting index, init cost to 0
11:    for $i = 0$ to $NumberOfCities - 1$ do
12:        $u =$ index at which there is min cost
13:        for $v = 0$ to $|V|$ do
14:            if $M[u][v] \wedge !MST[v] \wedge M[u][v] < cost[v]$ then
15:                $cost[v] \leftarrow M[u][v]$
16:                $P[v] \leftarrow u$
```

Time complexity of Prim's Algorithm for MST
Time complexity is $O(V^2)$ since we're using adjacency matrix, where V is the number of cities, or vertices of the graph. If used adjacency list, time complexity = O(Elog(V)) where E is paths between all the cities, and V is the number of cities. Since this is a complete undirected graph, meaning there are distances between all cities, E = V*(V-1)/2. So O(Elog(V)) = O(V*(V-1)/2*log(V)) = about O(V^2*log(V)) > O(V^2) for adjacency matrix for large V. Thus we picked adjacency matrix.

Space complexity of Prim's Algorithm for MST
Space complexity of Prim's MST: O(V^2) since adjacency matrix is V by V array, which is the biggest array.

DFS algorithm
Once we had the MST with each vertex assigned its parent vertex to represent the edges picked in the MST, we turned this into an adjacency list. Then we did depth first search (DFS) of this MST and kept track of all vertexes visited in order in the depth first search, not writing all vertices already visited.

A brief description of the steps in the DFS algorithm:

1. Have bool array "visited" of size numberofCitiesOrVertices to keep track of whether each vertex has been visited. Initialize to false at the beginning. Start visiting at random vertex or city. Call DFS
2. while mstSet doesn't include all vertices:
3. call recursive function DFS(starting vertex, "visited" array)
4. In DFS, when looping through adjacency list of starting vertex, if the vertex hasn't been visited yet, then call DFS function recursively with the input as this vertex, while also passing "visited" bool array to make sure vertexes aren't repeatedly visited. Else, just return nothing.

DFS algorithm pseudo-code
Let MST adjacency List be called 'adjMST'

```
1: procedure MAIN
2:     visited = bool array of size V numCities and initialize all false
3:     vertex[0] = True
4:     call function DFS(0,visited)
```

```
1: procedure DFS(vertex, visited)
2:     visited[v] = true
3:     for adjacent vertices w of v do
4:         if w isn't visited then
5:             DFS(w, visited)
```

Time complexity of DFS
Time complexity = O(V+E), where V is the number of cities, or vertices of the graph, and here E is paths between all the cities. If we used adjacency matrix, time complexity = $O(V^2)$. In MST, E = V-1. So O(E+V) = O(V-1+V) = about O(V) < O(V^2). Thus we picked adjacency list.

Space complexity of DFS
Space complexity of DFS is O(V+E) for the adjacency List, where V is the number of cities, or vertices of the graph, and here E is paths between all the cities. Since E = V-1, O(V+E) for space complexity = O(2V-1) = O(V). If we used an adjacency matrix, the space complexity would be O(V^2).

Overall Time Complexity of Approximation Algorithm
Overall time complexity = O(V^2) for MST + O(V) for DFS = O(V^2), where V is the number of cities, or vertices of the graph.

Overall Space complexity of Approximation Algorithm
Overall space complexity = O(V^2) for MST + O(V) for DFS = O(V^2), where V is the number of cities, or vertices of the graph.

Space complexity of DFS is O(V+E) for the adjacency List, where V is the number of cities, or vertices of the graph, and here E is paths between all the cities. Since E = V-1, O(V+E) for space complexity = O(2V-1) = O(V). If we used an adjacency matrix, the space complexity would be O(V^2).

Pros and Cons of DFS
1. Pros
   - Approximation Algorithm is fast O(V^2)
   - Approximation Algorithm guarantees an approximation ratio of 2, so the solution distance of the tour you get will be less than twice the optimal distance.

2. Cons
   - If getting a tour solution that could be 2*optimal solution is unacceptable, especially when distance on that tour is large, then aprroximation algorithm won't provide feasible solution.

Please cite any sources of information that you used to inform your algorithm design
-http://www.cs.tufts.edu/ cowen/advanced/2002/adv-lect3.pdf //

3. Simulated Annealing
We elected to build a version of Simulated Annealing as one of our local search algorithms given it's relative speed of execution [?], recognizing that we would need to complement our implementation with domain specific heuristics to improve overall solution quality. Simulated Annealing generates monotonically improving solutions by randomly sampling the neighborhood search space based on a decaying schedule. The algorithm progresses thru the search space by keeping track of the best result found as it advances. The decay schedule, referred to as the temperature schedule (T), biases early execution towards exploration and latter execution towards exploitation, making the algorithm increasingly likely to get stuck at a local optima over the execution lifetime. Terminating conditions vary, but are often triggered after a given number of iterations, a given runtime, or if an incremental solution improvement has not occurred for a pre-defined number of iterations. The following SA pseudocode outlines the key aspects of our implementation, with various components subsequently described.

```
1: procedure SA($c, \alpha, D$)    ▷ cutoff, cooling rate, distances
2:     $P \leftarrow$ LazyNeighbors($D$)    ▷ path array
3:     $score \leftarrow$ GetScore($P, D$)    ▷ current score
4:     $T \leftarrow score$    ▷ T schedule
5:     $dims \leftarrow |P|$
6:     while $runtime < c$ do    ▷ search until cutoff
7:         $steps \leftarrow dims * (dims - 1)$    ▷ steps in T round
8:         while $steps > 0$ do
9:             $N, idxs \leftarrow$ Exchange3($P$)
10:            $nextscore \leftarrow$ GetScore($N, D$)
11:            $\Delta E \leftarrow nextscore - score$
12:            if $\Delta E < 0$ then    ▷ accept better
13:                $score \leftarrow nextscore$
14:                $P \leftarrow N$
15:            else    ▷ Metropolis condition
16:                $p \leftarrow e^{\frac{\Delta E}{T}}$
17:                $r \leftarrow$ random $\in [0, 1]$
18:                if $r > p$ then    ▷ accept worse
19:                    $score \leftarrow nextscore$
20:                    $P \leftarrow N$
21:                else    ▷ revert swap
22:                    $P \leftarrow$ Revert3($P, idxs$)
23:            $steps \leftarrow steps - 1$
24:        $T \leftarrow \alpha T$    ▷ geometric cooling
25:    return $P$    ▷ best path found before cutoff
```

For each iteration, Simulated Annealing randomly samples a candidate based on the current solution's neighboring state space. The algorithm accepts the candidate based on the Metropolis condition, as follows:

$$p_{accept}(T, P, P') = \begin{cases} 1 & \text{if } f(P') \leq f(P) \\ e^{\frac{f(P) - f(P')}{T}} & \text{otherwise} \end{cases}$$

The Metropolis condition always accepts the candidate path, P', if it has a lower distance than the current path, P, as determined by the scoring function $f(\cdot) =$ GetScore defined as follows:

```
1: procedure GetScore($P, D$)    ▷ path $P$, distances $D$
2:     $score \leftarrow 0$
3:     for $i = 0$ to $|P| - 1$ do    ▷ increment for each stop
4:         $score \leftarrow score + D[i][i+1]$
5:     $start \leftarrow P[0]$    ▷ starting location
6:     $end \leftarrow P[|P| - 1]$    ▷ ending location
7:     $score \leftarrow score + D[start][end]$    ▷ round trip!
8:     return $score$    ▷ score for $P$
```

Alternatively, if the candidate path has a higher distance than the current path, the Metropolis condition accepts the inferior route with probability p_a. The acceptance probability for a longer path is therefore proportional to the magnitude of the score differential and the current temperature T. The temperature is geometrically reduced by α after each iteration, which lowers the following iteration's probability of accepting a longer path [1].

We attempted to reduce the likelihood our implementation would get stuck at a locally optimal but globally poor solution by expanding the neighborhood search space from a 2-exchange to a 3-exchange (Exchange3 and corresponding Revert4). Expanding the neighborhood improves the chances that improved regions will be considered at each iteration. The downside of this bird's eye view is that the algorithm will not only have a wider area to search, but may potentially jump to a region affording an immediate improvement but poorer final solution [3]. Jumping to a potentially unpromising region of the search space is of even greater concern early in the execution given the higher probability of acceptance dictated by the Metropolis condition. Despite these concerns, we found that coupling a wider neighborhood search space with a quickly cooling temperature schedule improved our solution quality without significantly increasing the runtime.

```
1: procedure Exchange3($P$)    ▷ path $P$
2:     $i, j, k \leftarrow$ random $\in (0, 1, \cdots, |P| - 1)$    ▷ select 3
   indices randomly
3:     swap $P[i]$ and $P[j]$
4:     swap $P[i]$ and $P[k]$
5:     return $P, (i, j, k)$    ▷ exchanged path $P$ and indices
```

```
1: procedure REVERT3($P, (i, j, k)$)                ▷ path $P$ and indices
2:     swap $P[i]$ and $P[k]$
3:     swap $P[i]$ and $P[j]$
4:     return $P$                                   ▷ pre-exchange path $P$
```

The algorithm is particularly sensitive to the temperature schedule as it is a key determinant for exploring the candidate neighborhood [3]. We found that setting the initial temperature relative to the initial path distance generated a more appropriate cooling schedule than manual configuration. Setting the starting temperature based on the initial path distance ensured that the acceptance probability was proportional to the range of possible score differentials, often denoted as ΔE, or change in energy. We also spent considerable time tuning the temperature cooling rate, α, by running multiple trials on each of the underlying datasets and comparing the mean solution quality. Unsurprisingly, small changes to α often caused significant performance divergence across the provided datasets given the varying magnitude of each example's distance matrix. One potential future improvement might be to normalize the distance matrix prior to running the Simulated Annealing algorithm so that the temperature was set proportionally across datasets.

Our initial implementation naively selected a random path to start, which resulted in a very poor initial solution but significant improvement over the algorithm's runtime. We then found that initiating the path via a basic nearest neighbor heuristic materially improved the quality of our final solution, but quickly caused our algorithm to get stuck at local optima. Retuning our initial temperature and α parameters for the updated initialization enabled the algorithm to more frequently escape local optima, resulting in further improvements to solution quality at minimal cost to runtime across the provided datasets.

```
1: procedure LAZYNEIGHBORS($D$)                     ▷ distance $D$
2:     let $P$ be path array of length $|D|$
3:     let $C$ be candidate array of length $|D|$
4:     $P[i] \leftarrow i \forall i \in (0, \cdots, |D|)$          ▷ path array
5:     for $i = 0$ to $|P| - 1$ do                  ▷ set each path element
6:         $start \leftarrow P[i]$
7:         $best \leftarrow start$
8:         $cost \leftarrow \infty$
9:         for $c \in C$ do                         ▷ nearest remaining
10:            if $D[start][c] < cost$ then
11:                $best \leftarrow c$
12:        $P[i+1] \leftarrow best$
13:        remove $best$ from $C$
14:    return $P$    ▷ return nearest neighbor heuristic path $P$
```

Time Complexity
Our nearest neighbor initialization, `LazyNeighbors`, iterates over each location and assigns the nearest remaining candidate as the next stop in the path. The total initialization time complexity is $O(n^2)$, consisting of $O(n)$ iterations each making $O(n)$ candidate checks and updates which require constant time.

Although our implementation terminated based on the cutoff, we will analyze the time complexity based on the cooling schedule as that is the effective runtime bound. By the master theorem, geometric temperature cooling requires $O(logn)$ time. Each temperature round requires $O(n^2)$ for $dim * (dim - 1)$ steps. Our existing implementation then requires $O(n)$ for each step, bounded by our naive `GetScore` procedure while path swaps all occur in $O(1)$ based on our utilization of the C++ vector data structure. The total time complexity of our initial Simulated Annealing implementation is therefore $O(n^3 logn)$. Persisting the distances of each recent swap would instead of iterating thru each location would improve the `GetScore` runtime to $O(1)$ and the initial implementation's time complexity to $O(n^2 logn)$. The temperature schedule dominates the algorithm's runtime resulting in an overall time complexity of $O(n^3 logn)$ for our current implementation.

Data Structures and Space Complexity
The space complexity for our overall implementation, as well as both the `LazyNeighbors` and temperature components, was dominated by the 2-dimensional adjacency matrix, D, which we represented with an array. We also chose to use the C++ vector data structure to represent all paths given it's previously mentioned constant time swap performance. We were able to iteratively update all other requisite data enabling us to represent step variables with primitive data structures.

Strengths and Weaknesses
As mentioned previously and highlighted by our empirical analysis, Simulated Annealing is able to quickly find results, but risks getting stuck at local optima [12]. The standalone implementation performs quite poorly, necessitating modifications to improve the ability to escape local optima prior to exhaustively searching the local neighborhood. Another weakness we found is that the algorithm is very sensitive to the temperature schedule. Given appropriate domain knowledge, this could potentially be viewed as a strength, but alas our lack of Champaign knowledge made this a particularly frustrating parameter to try and tune.

4. Genetic Algorithm
We use a Genetic Algorithm as the second Local Search Algorithm. Genetic Algorithm is based of evolutionary algorithms based on optimization according to survival of the fittest concept [15]. The algorithm will not find the optimal solution, but will give a good approximation based on the time amount of time the algorithm is made to run. [14]

Definitions:

- **Gene**: Represents a city in (x, y) coordinates.
- **Individual/Chromosome**: A single route satisfying the TSP constraints.
- **Population**: A collection of possible routes for TSP.
- **Parents**: Two routes that will be combined to create a new route.

- **Mating pool**: Collection of parents that are used to create the next generation of population.
- **Fitness**: The cost function of each route. Lower is better and helps evaluate the quality of a route.
- **Crossover**: Method by which a pair of parents create a offspring for the new generation of paths.
- **Mutation**: A way to introduce variation in the population by randomly swapping two cities in a route.

Method:

Create Initial Population

A population of P chromosomes are randomly generated to be the initial population where each chromosome is a path that visits all the cities and returns to the starting node. Each chromosome is created by choosing a random city from unvisited cities, adding it to the path, and continuing until all cities have been visited exactly once. Each of these solutions must be valid and satisfy the TSP constraints.

```
1: procedure CROSSOVER($P_1, P_2$)    ▷ Parents $P_1$ and $P_2$
2:     let $C_1$ and $C_2$ be the children being generated
3:     let $S_1$ and $S_2$ be randomly selected splitpoints from 0 to $|V|-1$ where $S_1 < S$
4:     let $G_1$ and $G_2$ be the gene markers denoting which cities have already been selected. All are initialized to 0
5:     for $i = 0$ to $S_1$ do
6:         $C_1[i] \leftarrow P_1[i]$
7:         $C_2[i] \leftarrow P_2[i]$
8:         $G_1[P_1[i]] \leftarrow 1$
9:         $G_2[P_2[i]] \leftarrow 1$
10:    for $i = S_2$ to $|V|-1$ do
11:        $G_1[P_1[i]] \leftarrow 1$
12:        $G_2[P_2[i]] \leftarrow 1$
13:    for $i = S_1$ to $S_2$ do
14:        if $G_1[P_2[i] == 1$ then
15:            $C_1[i] \leftarrow$ random city $c$ where $G_1(c)$ is 0
16:            $G_1[c] \leftarrow 1$
17:        else
18:            $C_1[i] \leftarrow P_2[i]$
19:        if $G_2[P_1[i] == 1$ then
20:            $C_2[i] \leftarrow$ random city $c$ where $G_2(c)$ is 0
21:            $G_2[c] \leftarrow 1$
22:        else
23:            $C_2[i] \leftarrow P_1[i]$
24:    for $i = S_2$ to $|V|-1$ do
25:        $C_1[i] \leftarrow P_1[i]$
26:        $C_2[i] \leftarrow P_2[i]$
27:    if Path in $C_1$ doesn't exist in population $P$ then
28:        $InsertBinarySearch(C_1)$
29:    if Path in $C_2$ doesn't exist in population $P$ then
30:        $InsertBinarySearch(C_2)$
31:    Delete from end of population $P$ so that $|P|$ is equal to size of population $N$
```

Determine Fitness

To simulate survival of the fittest, the fitness of each individual in the population is evaluated by calculating the cost of the path being taken. Each chromosome will have a fitness value which is the cost of the path. This serves as a measure of the quality of the solution. Based on the fitness, the population is sorted, with the individuals having the best fitness/ lowest path cost being placed at the start. These individuals represent the elite members of the population and represent the ancestors which have the best potential of producing a solution that is closest to the optimal.

Select Parents

Each individual in the population is considered as part of the current mating pool. Two parents are selected randomly from the mating pool and will create two children for the next generation. The two children are added to the population and then two worst individuals in the new population are removed and are not considered for mating in the next generation.

Crossover

After two parents are selected, crossover is performed to create two offspring. Two chromosome split points are randomly selected between the values 1 and $|V|$ where $|V|$ is the total number of cities. Until the first split point, the first child will get the genes from the first parent and the second child will get the genes from the second parent. Mark which genes have already been received. From the first split point to the second split point, the first child will get the genes from the second parent and the second child will get the genes from the second split point. If a gene has already been selected during the first iteration until split point 1, then randomly select a gene from the ones that have not been selected. From the second split point till the end of the genes from the parents, the first child gets the genes from the first parent and second child gets the genes from the second parent. If a gene has already been selected, then randomly choose a gene from the remaining unused genes. The two offspring are then added to the population if an identical individual doesn't already exist. Calculate the fitness value for each of the offspring and then use binary search to determine the position to insert each child in the population list. This is done to remove the need to sort the list every time a new child is added.

```
1: procedure MUTATION($C_1, C_2$)    ▷ Children $C_1$ and $C_2$
2:     let $m$ be mutation rate
3:     let $mutate$ be generated a random value between 0 and 1
4:     let $gene$ be the gene which will be mutated.
5:     if $mutate < m$ then
6:         $gene \leftarrow$ random value between 0 and $|V|-1$
7:         Swap values in $C_1[gene]$ and $C_2[gene]$
8:     return $C_1$ and $C_2$
```

Mutation

In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. If a mutation is being made, randomly select a gene and swap the gene values between the two offspring. This adds some variability to the population so that there is a smaller probability of getting stuck at a local optimum.

Data Structure and Space Complexity

The space complexity for our overall implementation, was dominated by the 2-dimensional adjacency matrix, D, which we represented with an array. The space taken up by the population will be be $O(NV)$ where N is the size of the population and V are the number of cities. Total space complexity will be $O(V^2 + NV)$. We also chose to use the C++ vector data structure to represent all paths given it's previously mentioned constant time insertion performance. We were able to iteratively update all other requisite data enabling us to represent step variables with primitive data structures.

Time Complexity

There will be G generations which is a hyperparameter and represents the stopping point if there is no time stoppage. Let the population be P. The insertion using binary search will be $O(log(N))$ for finding the point and then insertion at that point. The insertion time will be based on the data structure (in this case a vector which has $O(1)$) which is used. In the crossover method, we iterate through all the genes and select a particular one at each point. This will be equal to a time complexity of $O(V)$. Mutation procedure is $O(1)$. Overall time complexity will be $O(G(log(N) + V))$.

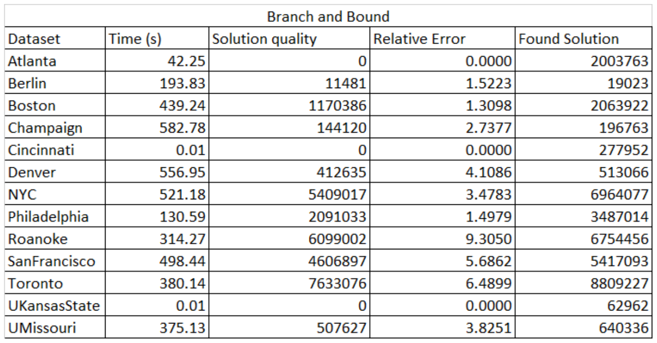
Advantages

- The algorithm can find good solutions in relatively less time.
- Random mutations gives the algorithm a chance to break out of a local optima.

Disadvantages

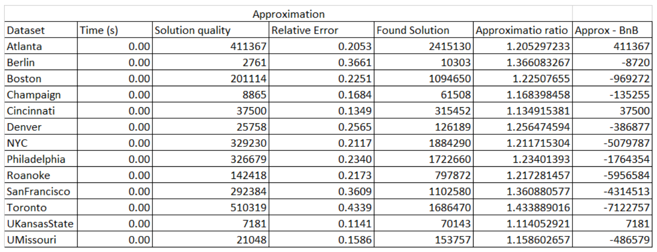
- Not guaranteed to find the optimal solution in all cases due to randomness in relation to creating new offspring and mutations.
- Choosing good hyper parameters related to size of population, mutation rate and number of generations requires multiple experiments.

EMPIRICAL EVALUATION



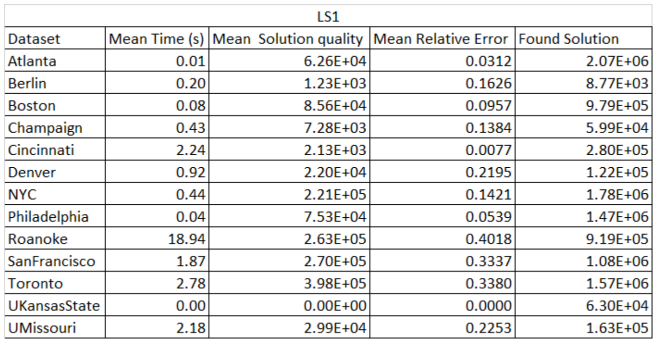
Branch and Bound				
Dataset	Time (s)	Solution quality	Relative Error	Found Solution
Atlanta	42.25	0	0.0000	2003763
Berlin	193.83	11481	1.5223	19023
Boston	439.24	1170386	1.3098	2063922
Champaign	582.78	144120	2.7377	196763
Cincinnati	0.01	0	0.0000	277952
Denver	556.95	412635	4.1086	513066
NYC	521.18	5409017	3.4783	6964077
Philadelphia	130.59	2091033	1.4979	3487014
Roanoke	314.27	6099002	9.3050	6754456
SanFrancisco	498.44	4606897	5.6862	5417093
Toronto	380.14	7633076	6.4899	8809227
UKansasState	0.01	0	0.0000	62962
UMissouri	375.13	507627	3.8251	640336

Figure 1. Branch And Bound Comprehensive Table



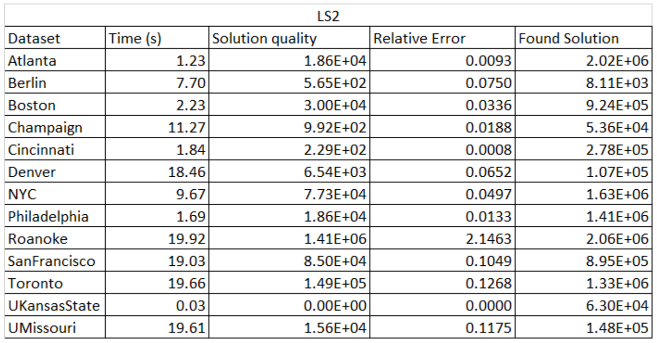
Approximation						
Dataset	Time (s)	Solution quality	Relative Error	Found Solution	Approximatio ratio	Approx - BnB
Atlanta	0.00	411367	0.2053	2415130	1.205297233	411367
Berlin	0.00	2761	0.3661	10303	1.366083267	-8720
Boston	0.00	201114	0.2251	1094650	1.22507655	-969272
Champaign	0.00	8865	0.1684	61508	1.168398458	-135255
Cincinnati	0.00	37500	0.1349	315452	1.134915381	37500
Denver	0.00	25758	0.2565	126189	1.256474594	-386877
NYC	0.00	329230	0.2117	1884290	1.211715304	-5079787
Philadelphia	0.00	326679	0.2340	1722660	1.23401393	-1764354
Roanoke	0.00	142418	0.2173	797872	1.217281457	-5956584
SanFrancisco	0.00	292384	0.3609	1102580	1.360880577	-4314513
Toronto	0.00	510319	0.4339	1686470	1.433889016	-7122757
UKansasState	0.00	7181	0.1141	70143	1.114052921	7181
UMissouri	0.00	21048	0.1586	153757	1.158602657	-486579

Figure 2. Approximation Comprehensive Table



LS1				
Dataset	Mean Time (s)	Mean Solution quality	Mean Relative Error	Found Solution
Atlanta	0.01	6.26E+04	0.0312	2.07E+06
Berlin	0.20	1.23E+03	0.1626	8.77E+03
Boston	0.08	8.56E+04	0.0957	9.79E+05
Champaign	0.43	7.28E+03	0.1384	5.99E+04
Cincinnati	2.24	2.13E+03	0.0077	2.80E+05
Denver	0.92	2.20E+04	0.2195	1.22E+05
NYC	0.44	2.21E+05	0.1421	1.78E+06
Philadelphia	0.04	7.53E+04	0.0539	1.47E+06
Roanoke	18.94	2.63E+05	0.4018	9.19E+05
SanFrancisco	1.87	2.70E+05	0.3337	1.08E+06
Toronto	2.78	3.98E+05	0.3380	1.57E+06
UKansasState	0.00	0.00E+00	0.0000	6.30E+04
UMissouri	2.18	2.99E+04	0.2253	1.63E+05

Figure 3. Simulated Annealing Comprehensive Table average over 10 trails



LS2				
Dataset	Time (s)	Solution quality	Relative Error	Found Solution
Atlanta	1.23	1.86E+04	0.0093	2.02E+06
Berlin	7.70	5.65E+02	0.0750	8.11E+03
Boston	2.23	3.00E+04	0.0336	9.24E+05
Champaign	11.27	9.92E+02	0.0188	5.36E+04
Cincinnati	1.84	2.29E+02	0.0008	2.78E+05
Denver	18.46	6.54E+03	0.0652	1.07E+05
NYC	9.67	7.73E+04	0.0497	1.63E+06
Philadelphia	1.69	1.86E+04	0.0133	1.41E+06
Roanoke	19.92	1.41E+06	2.1463	2.06E+06
SanFrancisco	19.03	8.50E+04	0.1049	8.95E+05
Toronto	19.66	1.49E+05	0.1268	1.33E+06
UKansasState	0.03	0.00E+00	0.0000	6.30E+04
UMissouri	19.61	1.56E+04	0.1175	1.48E+05

Figure 4. Genetic Algorithm Comprehensive Table average over 10 trails

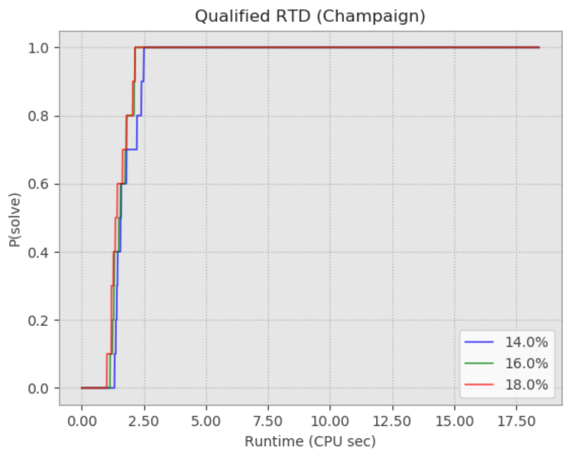


Figure 7. Genetic Algorithm Champaign QRTD

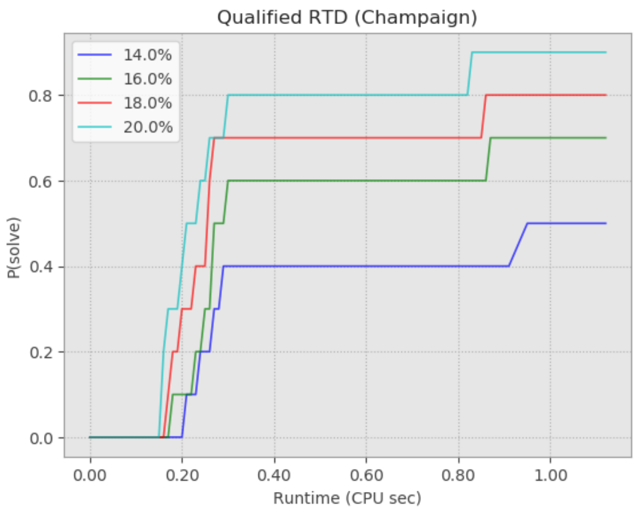


Figure 5. Simulated Annealing Champaign QRTD

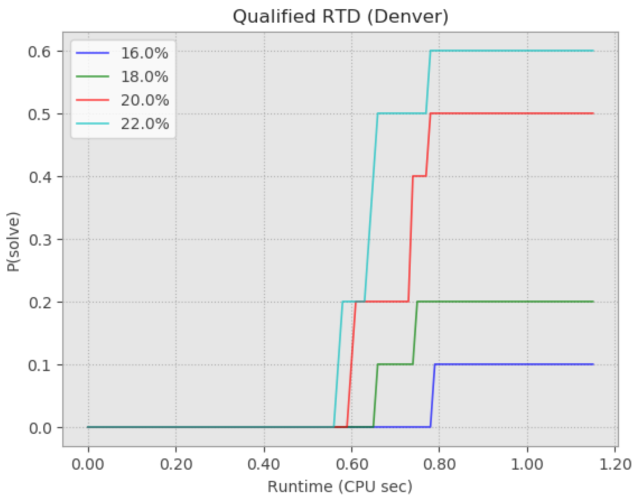


Figure 6. Simulated Annealing Denver QRTD

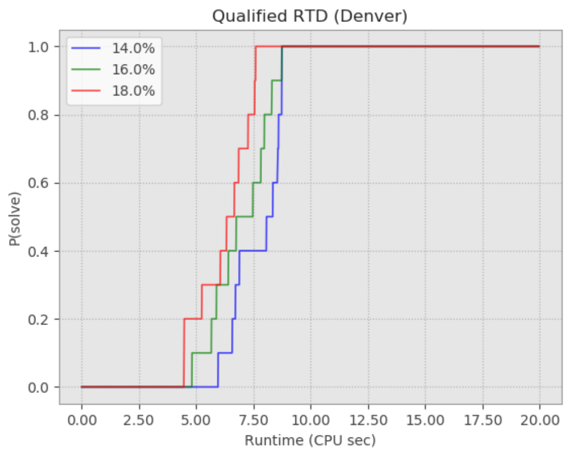


Figure 8. Genetic Algorithm Denver QRTD

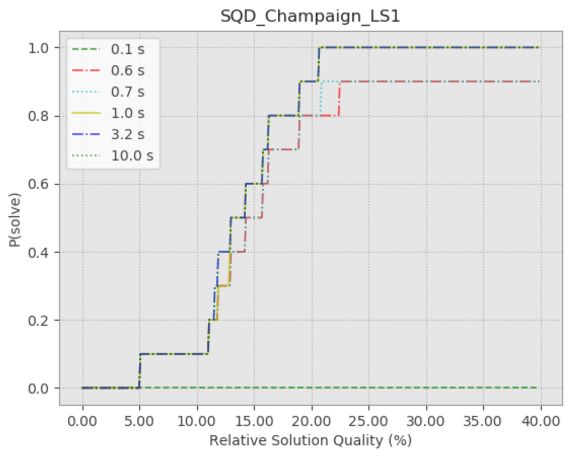


Figure 9. Simulated Annealing Champaign SQD

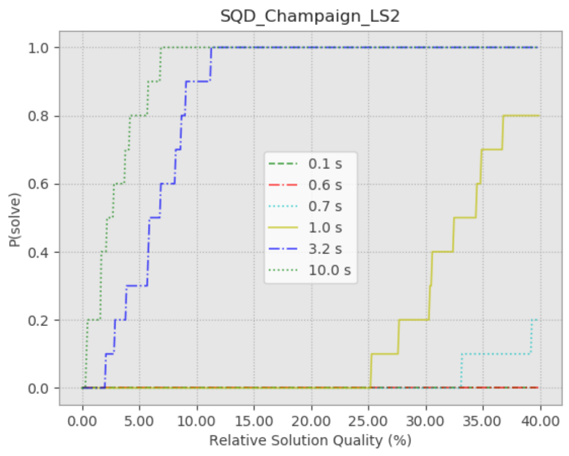


Figure 11. Genetic Algorithm Champaign SQD

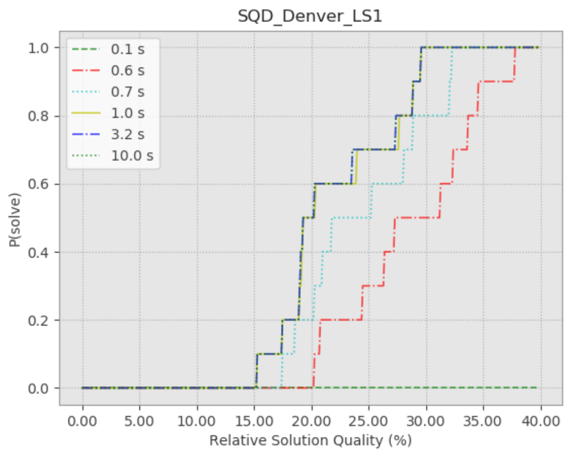


Figure 10. Simulated Annealing Denver SQD

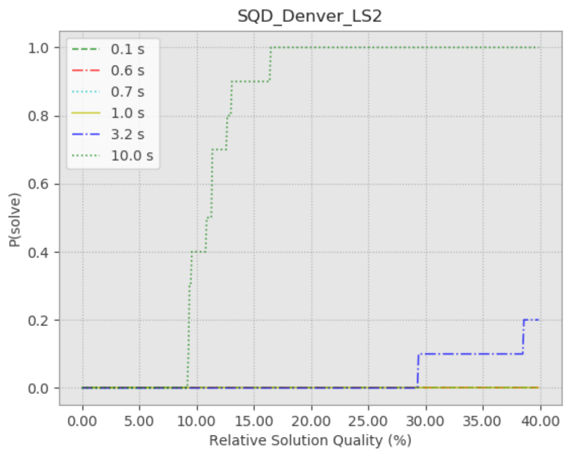


Figure 12. Genetic Algorithm Denver SQD

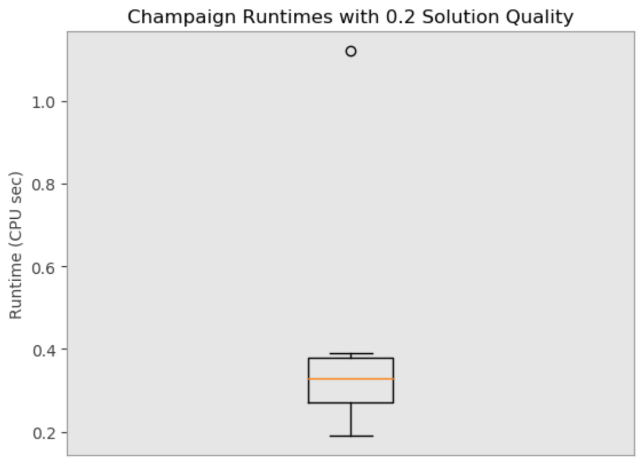


Figure 13. Simulated Annealing Champaign Boxplot

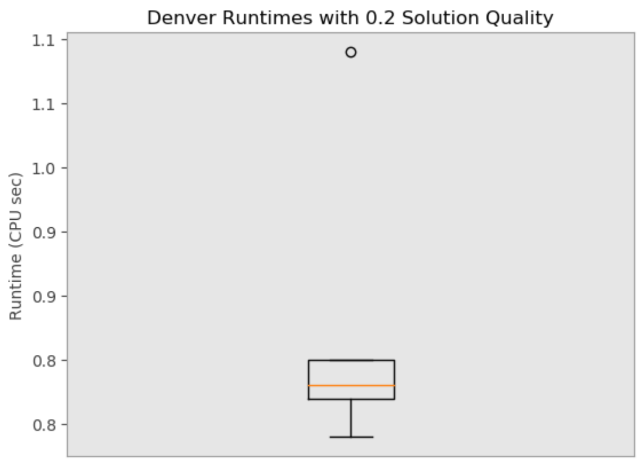


Figure 14. Simulated Annealing Denver Boxplot

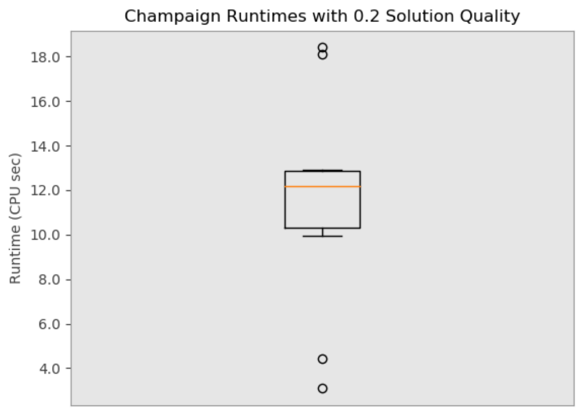


Figure 15. Genetic Algorithm Champaign Boxplot

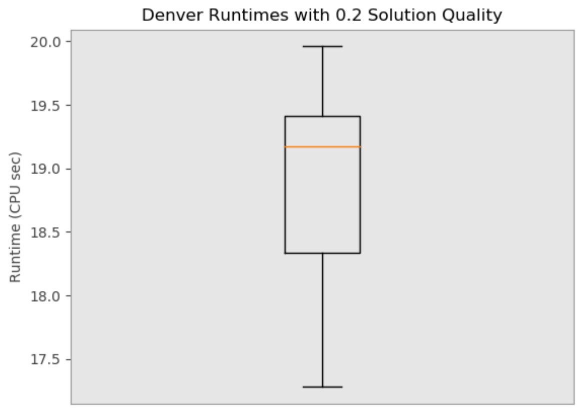


Figure 16. Genetic Algorithm Denver Boxplot

System Specifications

- CPU: 2.9 GHz Dual-Core Intel Core i5
- RAM: 8 GB 2133 MHz LPDDR3
- Language: C++
- Compiler: Apple clang version 11.0.0 (clang-1100.0.33.12)

Lower bound on approximation results

You can see from the Comprehensive Table for approximation algorithm that the lower bound on the optimal solution quality that was obtained empirically from approximation algorithm is 510319 for Toronto. Note we defined solution quality as (found approximation solution - optimal solution). Thus this is the difference of approximation solution from optimum. However, the max approximation ratio was 1.43 for San Francisco.

Solution quality difference of approximation solution - branch and bound solution lower bound is 411367 at Atlanta.

Simulated Annealing

The Simulated Annealing algorithm was too fast and too furious, consistently generating lower quality solutions than our Genetic Algorithm implementation. The best solution was typically found within the first second, at which point our prodigal algorithm haplessly continued searching until the cutoff. Interestingly, improvements upon the nearest neighbor initialization clustered after an initial exploration period, which for the Champaign trials occurred at 0.3 seconds and for the Denver trials occurred at 0.8 seconds, as evidenced by their respective QRTD plots.

After the initial batch of improvements the algorithm likely accepting a worse solution located in an unfavorable region of the search space, which prevented consistently better solutions from being found. The Champaign and Denver boxplot outliers are particularly interesting as they represent times when the algorithm hit the proverbial jackpot while traversing the 3-exchange neighborhood.

Our SQD plots revealed that Simulated Annealing provided relatively consistent solution qualities irrespective of the runtime, which conceptually makes sense as the algorithm is prone to get stuck at local optima. One way we could improve results would be to randomly restart after a specified number of iterations without an improvement, similar to the idea of combining Genetic Algorithm with Simulated Annealing [8]. This would potentially free the algorithm to explore a different region of the state space without significantly impacting execution time.

As previously evidenced in the comprehensive table, the large Roanoke dataset proved particularly nettlesome for the Simulated Annealing algorithm. Reviewing the trace file and SQD plot showed that our algorithm generated consistent improvements over the 20 second runtime, leaving us pointing the finger at our nearest neighbor initialization. Future incarnations could improve upon our initialization by either selecting an alternative starting node, or outright discarding the the path in favor of a randomly generated path, if the generated path's quality was prohibitively low such as for Roanoke. Clearly nearest neighbor initialization risks generating a suboptimal candidate solution with significant cost imbalances, which may be exacerbated by larger datasets.

Genetic Algorithms

The Genetic Algorithm took consistently longer to settle at a value, but this was traded off for lower mean relative error, as can be seen from the comprehensive table.

Our RTD plot reveals that for past 10 CPU seconds, 100 percent of the tests get relative solution quality within 14 percent for both Denver and Champaign. Thus, if you want to get below 14 percent relative solution quality, run the genetic algorithm over 5 seconds.

Our SQD plots reveals that running the genetic algorithm below 1 second will make 0 percent of the tests to pass even up to 40 percent relative solution quality. This shows that genetic algorithm takes longer than simulated annealing algorithm.

The box plot reveals that the Champaign run time for the genetic algorithm had a tight bound of about 3 seconds difference in run time, while Denver had even tighter bound on average in run time of 1 second of the box to achieve 0.2 solution quality.

DISCUSSION

Looking at the results of Branch and Bound in Figure 2 we see that for smaller cities like Atlanta and Cincinnati where the number of nodes is less than 20, we get an error rate of 0. This is because the time limit of 600 seconds is enough time for the branch and bound algorithm to explore all possible path combinations in the solution space and is able to find the optimal solution. However, the larger the city, the higher the distance between the calculated solution and the optimal solution which results in a higher relative error. This is expected as the larger the number of vertices, the more time branch and bound takes to find the optimal solution and a time cutoff of 600 seconds will not be enough to iterate through all possible solutions. Given more time to run the algorithm it is reasonable to assume that we will keep improving our solution quality until we reach the optimum.

Looking at the results of the Approximation Algorithm we can observe that we get solutions that are within the theoretical bounds for approximation ratio of 2. The approximation algorithm generates a feasible solution in polynomial time and the quality of this solution is relatively close to the optimum. We can see from Figure 2 that the time taken for each of the cities is negligible which follows from the fact that time complexity is $O(V^2)$ and the largest V is 230 from city Roanoke. The worst approximation ratio we get is 1.43 for Toronto which is well within the theoretical approximation ratio of 2. The theoretical approximation ratio can be improve to 1.5 by using the Christofides algorithm which works by using MST and a combination of Eulerian Circuit and Hamiltonian Circuit.

Of our local search algorithms, Genetic Algorithms produced higher relative solution quality results across the datasets than Simulated Annealing while maintaining a reasonable execution timeframe. Getting stuck at local optima caused Simulated Annealing to effectively terminate quickly in contrast to the higher theoretical time complexity of $O(n^3 logn)$. The Genetic Algorithms empirical runtime was proportional to the size of the dataset, supporting our time complexity analysis. Interestingly, the Genetic Algorithm relative solution quality was inversely bifurcated by runtime, with longer trials producing a lower solution quality than those that terminated earlier. Simulated Annealing consistently generated a similar relative solution quality across runtimes.

CONCLUSION

The Travelling Salesman Problem is one of the most popular problems in current academia related to efficient algorithms. In this paper we have detailed 4 methods that attempt to solve this problem. These 4 methods differ in what parameter we are aiming for namely time and correctness of solution. Branch and Bound focuses on getting the optimal solution without regarding the time needed to run the algorithm. The Approximation Algorithm focuses on getting a solution as quick as possible in polynomial time but at the cost of a higher error rate with no scope for improving with more time. The Local Search algorithms are a middle ground between these two approaches wherein they focus on getting a solution quickly but then continuously iterate on this solution until a local or global optimum is reached. These algorithms are also time constrained but find feasible solutions and continuously make them better. There are many more different methods that can be explored in attempting to solve this problem which we leave to others in this field.

REFERENCES

[1] Harry Cohn and Mark Fielding. 1999. Simulated annealing: searching for an optimal temperature schedule. *SIAM Journal on Optimization* 9, 3 (1999), 779–802.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[3] Holger H. Hoos and Thomas Stützle. 2004. Stochastic local search: Foundations and applications. Elsevier, 2004. (2004).

[4] Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi. 1995. The traveling salesman problem. *Handbooks in operations research and management science* 7 (1995), 225–330.

[5] Gilbert Laporte. 1992. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research* 59, 2 (1992), 231–247.

[6] Eugene L Lawler and David E Wood. 1966. Branch-and-bound methods: A survey. *Operations research* 14, 4 (1966), 699–719.

[7] Waqar Malik, Sivakumar Rathinam, and Swaroop Darbha. 2007. An approximation algorithm for a symmetric generalized multiple depot, multiple travelling salesman problem. *Operations Research Letters* 35, 6 (2007), 747–753.

[8] Theodore W. Manikas and James T. Cain. 1996. "Genetic Algorithms vs. Simulated Annealing: A Comparison of Approaches for Solving the Circuit Partitioning Problem". Computer Science and Eng. (1996).

[9] Mirta Mataija, Mirjana Rakamarić Šegić, and Franciska Jozić. 2016. Solving the travelling salesman problem using the Branch and bound method. *Zbornik Veleučilišta u Rijeci* 4, 1 (2016), 259–270.

[10] Bernard ME Moret and Henry D Shapiro. 1992. An Empirical Assessment of Algorithms for Constructing a Minimum Spanning Tree. *Computational Support for Discrete Mathematics* 15 (1992), 99–117.

[11] Ferrante Neri, Carlos Cotta, and Pablo Moscato. 2011. *Handbook of memetic algorithms*. Vol. 379. Springer.

[12] B. Freisleben P. Merz. 1997. "Genetic local search for the TSP: New results". Proc. 1997 IEEE Int. Conf. Evolutionary Computation., pp. 159-164. (1997).

[13] G Terry Ross and Richard M Soland. 1975. A branch and bound algorithm for the generalized assignment problem. *Mathematical programming* 8, 1 (1975), 91–103.

[14] J. Scholz. 2019. "Genetic Algorithms and the Traveling Salesman Problem a Historical Review". (2019).

[15] SN Sivanandam and SN Deepa. 2008. Genetic algorithms. In *Introduction to genetic algorithms*. Springer, 15–37.