1 INTRODUCTION

The Traveling Salesman Problem (TSP) is an NP-Complete computational problem which has been studied extensively for several decades [4]. In this paper, techniques for obtaining/approximating the solution to the symmetric Traveling Salesman Problem are developed and compared. A branch-and-bound algorithm using a minimum spanning tree to inform a lower bound on sub-problems is presented; which, given enough time, produces an exact solution to the problem. The use of approximation algorithms proves more prudent for larger problems while sacrificing solution quality; a fact which motivated the development of a greedy local search algorithm which explores 1-exchange neighbors. In addition, two local search algorithms a 2-Opt Hill Climb and Simulated Annealing Algorithm are explored. These local search algorithms are ideal for finding low error solutions in limited time, although no performance bounds are guaranteed beyond what is expected from the Greedy Algorithms that initially seed them. The results derived showed that the local search algorithms were extremely adept at providing relatively good solutions even in the presence of large data sets, however they struggled to consistently return optimal solutions even for the smaller data-sets. Of the two local search algorithms explored, the simulated annealing algorithm provided clear benefits especially for larger data sets which contain a higher number of local optima which can frequently trap other algorithms in a non optimal neighborhood.

2 PROBLEM DEFINITION

The symmetric traveling salesman optimization problem is formalized as follows:
Given a connected undirected graph G consisting of n nodes,$\{v_0, \ldots, v_{n-1}\}$, with edge weights d_{ij} between nodes i and j. Find a Hamiltonian Cycle, a path P* where each node has degree 2, with minimum weight.
In this paper, nodes in a 2 dimensional space were considered: $v_i = \begin{bmatrix} v_{ix} \\ v_{iy} \end{bmatrix} \in \mathbb{R}^2 \quad \forall i \in [0, n-1]$
With edge weights calculated using the Euclidean distance: $e_{ij} = ||v_i, v_j||_2 = \sqrt{(v_{ix} - v_{jx})^2 + ((v_{iy} - v_{jy}^2)} \quad \forall i \neq j, i \in [0, n-1], j \in [0, n-1]$
The solution is an element in the set of vertex sequences which are Hamiltonian Cycles given by:
$\mathcal{H} = \{(v_i)_{i=0}^{n-1} : v_0 = v_{n-1}, v_i \neq v_j \quad \forall i, j \in [0, n-1]\}$
The symmetric TSP is given by following optimization problem:
$P^* = (v_i^*)_{i=0}^{n-1} = \overset{argmin}{(v_i)_{i=0}^{n-1} \in \mathcal{H}} \quad \sum_{i=0}^{n-2} ||v_i, v_{i+1}|| + ||v_0, v_{n-1}||$

3 RELATED WORK

The Traveling Salesman Problem has been a widely studied in the fields of graph theory and computational algorithms for almost 200 years. Long considered impossible, formal methods for restricting the problem space using approximation algorithms, similar to the 2MST approximation used here. With formal methods to approximate routes to given accuracies solving the problem for large data-sets became feasible in practice. With a good starting point for lower bounds, methods such as the Branch-and-Bound method used in this paper could now solve large data-sets by limiting the search space using the previously calculated lower bounds given by the approximation guarantee. Recently in 2011 researchers from Stanford and McGill Universities were able to improve upon the approximation algorithm developed by Nicos Christofides in 1976 which had defined performance bounds of at most 50% greater than the true optimal solution when they developed an approximation technique which improved upon Christofides' algorithm by four hundredths of a trillionth of a trillionth of a trillionth of a trillionth of a percent, a staggeringly small value in most other scenarios. Although this improvement was relatively minimal, it broke open a wall that existed in the world of computer science for over 35 years, proving that in practice a better solution was possible. With the rise of Quantum Computing improved results for the Traveling Salesman Problem will not only be made possible by improved algorithms, but also by the exponential increase in processing power that the Quantum Technology brings. With this technology on the rise we are poised to see major improvements in the runtime of NP-Hard algorithms such as the TSP problem which will be extremely important given the increase in the use of Big Data throughout virtually every industry. [3]

4 ALGORITHMS

4.1 Branch and Bound (BnB)

Branch and bound algorithms are a class of algorithms which are guaranteed to find the optimal solution of a problem given enough time. The exactness of the algorithm comes with a trade-off for time complexity which remains exponential. Branch and bound algorithms maintain an upper and lower bound on the optimal solution to a problem and iteratively. The original problem can be considered a set of decisions about whether an edge is in the solution; furthermore, each subproblem can be defined as finding the optimal Hamiltonian Cycle given a set of edges which must be in the cycle, denote these as **decided edges**. Each problem can be answered by making a decision about a remaining undecided edge and then formulating a subproblem for the remaining edges; in such a manner the original problem can be decomposed into a series of subproblems. The data structure used to represent each subproblem is denoted as a 5-tuple $subproblem(L, \mathcal{D}, C, \mathcal{U}, k)$ consisting of:

- Lower bound, L : The lowest possible value for the cost of Hamiltonian cycle that can be constructed with Decided Edges and any of the remaining undecided edges.

- Decided edges, $\mathcal{D}$: A set of edges which must be in the solution.
- Decided Edge Cost, C : Cost of the edges in the Decided Edges set.
- Connectivity Union Set, $\mathcal{U}$: A list of association between each node and the connected subgraph it belongs to.
- Decision index, k : An index into a list of edges for which all previous edges have been decided.

A characteristic of branch and bound algorithms is that they omit investigation of problems that are known to have suboptimal solutions; this is accomplished by calculating a lower bound on each subproblem and keeping track of the best solution encountered throughout the run of the algorithm and omitting subproblems which are guaranteed to have suboptimal solutions. If a subproblem's lower bound exceeds the quality of the best solution, it is apparent that no solution of the subproblem can be an optimal solution of the original problem. In this paper the lower bound for each subproblem is the minimum spanning tree which contains the decided edges and a subset of the undecided edges. Another key design decision of a branch and bound algorithm is the branching strategy, that is: how subproblems are chosen to be investigated. Based on empirical observations and previous works [2] a depth first search strategy for selecting subproblems proved more effective at finding solutions. It was discovered that the effectiveness of this algorithm was strongly influenced by seeding the algorithm with an initial good solution [2]; a hill climbing algorithm which optimizes each potential greedy solution was used for this task.

Algorithm 1 BnB($V = \{v_0, \dots, v_{n-1}\}$): Find minimum cost Hamiltonian Cycle for euclidean distances

```
Data: $\{v_0, \dots, v_{n-1}\}$ set of 2-D points
for all Unordered Pairs $\{i, j\}$ do
  Construct edge $e = (v_i, v_j, d_{ij})$
  Add e to list E of edges in increasing weight order: $E = E \cup \{e\}$
end for
Initialize best solution : best ← GreedyHillClimb( $\{v_0, \dots, v_{n-1}\}$ )
Initialize frontier : $frontier \leftarrow frontier \cup (0, \emptyset, 0, \{v_0, \dots, v_{n01}\}, 0)$
while frontier is not empty do
  Select last added subproblem in frontier and remove it from frontier: $subproblem \leftarrow frontier.pop()$

  Create a new subproblem without the next edge and add it to frontier:
  Add cost of undecided edge set to lower bound : $child_0.LB \leftarrow child_0.LB+$ subproblem's decided edge set cost
  Add cost MST of nodes which are not endpoints of edges in subproblem's decded edge set :
  $child_0.LB \leftarrow child_0.LB + Cost(MST(V - \{v : e = \{v, u\} \in subproblem.\mathcal{D}\}, E - \mathcal{D}))$
  Add cost of edges in cutset between nodes with endpoints in decided edge set and nodes without an endpoint in decided edge set to lower bound.
  copy decided edge set since nothing was added: $child.\mathcal{D} \leftarrow subproblem.\mathcal{D}$
  copy decided edge set cost since nothing was added: $child.C \leftarrow subproblem.C$
  copy union set since no edge was added to change connectivity of nodes: $child.\mathcal{U} \leftarrow subproblem.\mathcal{U}$
  increment decision index since a decision not to include the current index edge has been made: $child.k \leftarrow subproblem.k + 1$
  Add the child to frontier : $frontier \leftarrow frontier \cup child_0$

  if adding $e_{subproblem.k+1}$ does not result in any node having 3 edges in $subproblem.\mathcal{D}$ then
    if adding $e_{subproblem.k+1}$ does not create a cycle for edges in decided edge set then
      if adding $e_{subproblem.k+1}$ results in $subproblem.\mathcal{U}$ to be full connected then
        Traverse edges to yield Hamiltonian Cycle P
        if $cost(P) < cost(best)$ then
          Update best solution : $best \leftarrow P$
        end if
      else
        Create a new subproblem with the next edge and add it to frontier:

        Add $e_{subproblem.k+1}$ to decided edge set: $child.\mathcal{D} \leftarrow subproblem.\mathcal{D} \cup \{e_{subproblem.k+1}\}$
        Add cost of undecided edge set to lower bound : $child_1.LB \leftarrow child_1.LB+$ subproblem's decided edge set cost
        Add cost MST of nodes which are not endpoints of edges in subproblem's decded edge set :
        $child_1.LB \leftarrow child_0.LB + Cost(MST(V - \{v : e = \{v, u\} \in subproblem.\mathcal{D}\}, E - \mathcal{D}))$
        Add cost of edges in cutset between nodes with endpoints in decided edge set and nodes without an endpoint in decided edge set to lower bound.
        Add cost of $e_{subproblem.k+1}$ to decided edge set cost: $child_1.C \leftarrow subproblem.C + cost(e_{subproblem.k+1})$
        Modify union set by taking union of both of $e_{subproblem.k+1} = \{u, w\}$'s endpoints: $child.\mathcal{U} \leftarrow subproblem.\mathcal{U}.union(u, w)$
        increment decision index since a decision to include the current index edge has been made: $child.k \leftarrow subproblem.k + 1$
        Add the child to frontier : $frontier \leftarrow frontier \cup child_1$
      end if
    end if
  end if
end while
return best
```

4.2 Approximation (Approx)

Approximation algorithms are of great utility when trying to quickly find a suboptimal solution that has guarantees on its quality with respect to the optimal solution. The algorithm of this class chosen for this study was the 2-MST approximation algorithm; in this algorithm the minimum spanning tree is found for the input nodes and subsequently traversed in depth first search order from an arbitrary node to yield a sequence of nodes. The edge sequence corresponding to the MST's depth first search vertex sequence is then taken as a solution to the problem. This algorithm has a guarantee of generating a solution which is at most twice as costly as the optimal solution [1]. The algorithm first begins by generating all the edges and sorting them; an operation with a runtime complexity of $O(mlogm)$ where m is the number of edges. A union find data structure is constructed by assigning each node a parent and a set rank; an operation which has a runtime complexity of $O(n)$. The edges in ascending cost order are iterated through, and the cheapest edge which can be added which does not create a cycle is added to the MST; this is known as Kruskal's algorithm and has a runtime complexity of $O(log(m))$. To yield the solution, an arbitrary vertex is selected as the root and a depth first search is performed; this operation has a runtime complexity of $O(n)$. Since the number of edges exceeds the number of vertices the overall runtime complexity is $O(mlogm)$.

Algorithm 2 2-MST($\{v_0, \ldots, v_{n-1}\}$): Approximate the minimum cost Hamiltonian Cycle for euclidean distances using a 2-MST algorithm

```
Data: $\{v_0, \ldots, v_{n-1}\}$ set of 2-D points
for all Unordered Pairs $\{i, j\}$ do
    Construct edge $e = (v_i, v_j, d_{ij})$
    Add e to list E of edges in increasing weight order: $E = E \cup \{e\}$
end for
Initialize set of edges for MST : $MST \longleftarrow \emptyset$
while Edges included != (n - 1) do
    Apply Union Find to Determine if least cost edge remaining makes a Cycle
    if least expensive edge e does not create a cycle based on union find then
        Add Edge e to Minimum Spanning Tree: $MST \longleftarrow MST \cup e$
    end if
end while
Select an arbitrary node v as the root
Perform depth first search of MST rooted at v to yield a vertex sequence $\vec{w}$
return $\vec{w}$
```

4.3 2-OPT Hill Climb (LS1)

The 2-OPT Hill Climbing Local Search Algorithm is a type of Local Search Algorithm which implements a 2-OPT Exchange to construct neighborhoods of solutions from an initial solution value. The 2-OPT Exchange looks to swap edges throughout the path with the hopes of swapping edges that overlap creating a non-optimal route of travel. Using a Hill Climbing Local Search approach, the algorithm moves through all exchange permutations from an initial solution, updating the current best route if a given exchange is successful in lowering the total route cost. For this implementation of the 2-OPT Exchange initial solutions were constructed using a simple Greedy Algorithm which considers a starting node, and then constructs a route by adding the lowest cost next step to the route given the desired node has not yet been visited. Furthermore, this algorithm considers each possible Greedy Route as an initial solution to further increase the search space. In practice, it proved more reliable to initialize the algorithm with a Greedy Solution rather than using a random path as the initial solution.

Algorithm 3 2-Opt_HC($\{v_0, \ldots, v_{n-1}\}$): Approximate the minimum cost Hamiltonian Cycle for euclidean distances using a Hill Climbing local search algorithm with 2-Opt exchange Neighborhood Creation

```
Data: $\{v_0, \ldots, v_{n-1}\}$ set of 2-D points
for all Unordered Pairs $\{i, j\}$ do
    Construct edge $e = (v_i, v_j, d_{ij})$
    Add e to list E of edges in increasing weight order: $E = E \cup \{e\}$
end for
while Unassigned nodes in $v$ do
    Assign Nodes to Route based on Greedy Nearest Neighbor implementation
end while
for $i = 1$ to length(Route Matrix) do
    for $j = i + 1$ to length(Route Matrix) do
        reverse array (route[i] to route[j]) and add it to newroute[i] to newroute[j]
        if cost(newroute) < cost(route) then
            route ⟵ newroute
        end if
    end for
end for
```

4.4 Simulated Annealing (LS2)

The Simulated Annealing Algorithm is designed to tackle one of the most prominent problems with the Hill Climb Local Search setup, becoming "stuck" at local optima solutions. In a strict hill climb algorithm, if a particular solution represents a local optima, the algorithm will be unable to break free thus providing a lower quality solution. The Simulated Annealing algorithm uses a cooling concept to allow the algorithm to move to solutions with higher costs than the current best route early on in the process when the annealing temperature is still high. By calculating an acceptance probability that is determined using the current temperature, the algorithm allows the solution space to move out of local optima with higher probabilities when the cooling temperature is high. As the algorithm runs a cooling constant α is used to decrease the system temperature, thus decreasing the probability of selecting a solution that gives a higher cost route. To traverse a neighborhood a 2-OPT exchange with random nodes is performed. The highly random nature of this algorithm causes the solution quality over time to vary greatly run to run, but does however offer the benefit of traversing different solutions spaces each run increasing the likelihood for a low error as both the length and volume of runs is increased.

Algorithm 4 Sim_Anneal($\{v_0, \dots, v_{n-1}\}$): Approximate the minimum cost Hamiltonian Cycle for euclidean distances using a Hill Climbing local search algorithm with 2-Opt exchange Neighborhood Creation

```
Data: $\{v_0, \dots, v_{n-1}\}$ set of 2-D points
Current_Route: $\{c_0, \dots, c_{n-1}\}$ Set of Location Nodes denoting the current route for annealing
Best_Route: $\{b_0, \dots, b_{n-1}\}$ Set of Location Nodes denoting the best route calculated so far for annealing
Temperature: $T$ Current Annealing Temperature used
Cooling Ratio: $\alpha$ Ratio used to cool the temperature as Simulated Annealing is run
for all Unordered Pairs $\{i, j\}$ do
    Construct edge $e = (v_i, v_j, d_{ij})$
    Add e to list E of edges in increasing weight order: $E = E \cup \{e\}$
end for
while Unassigned nodes in $v$ do
    Assign next node in route as the remaining node with the shortest distance between itself and the current node
end while
while Temperature $\geq$ Ending Temperature do
    Generate Random 2 Exchange Permutation
    if Current Solution Cost > Neighbor Route Cost then
        Update Current Route
    else
        Calculate Probability Using Current Temperature
        if Probability > Randomly Generated Probability then
            Update Current Route
        end if
    end if
    if Current Cost $\leq$ Best Cost then
        Update Best Route
    end if
end while
```

5 EMPIRICAL EVALUATION

5.1 Comprehensive Tables

Performance of the algorithms is summarized in tables 1 through 4. The empirical data makes it clear that the approximation algorithm has the worst performance among the studied algorithms, but the smallest runtimes. These findings are in line with expectations since the approximation algorithm is deterministic and will not continue randomly searching after it has found a solution.

Among the local search algorithms the 2-OPT hill climb algorithm generally performs as well as or worse than the simulated annealing. Both algorithms were run with the same timeout, which indicates that randomly seeded hill climb algorithm is much less effective at improving solution quality relatively quickly.

Finally, branch and bound is clearly the most expensive algorithm in terms of runtime and generally does not perform better than the local search algorithms. During development it became clear that the branch and bound algorithm was very sensitive to the solution it used to initialize the upper bound.

Table 1: Approx Algorithm Performance Table

Instance	Time[s]	Solution Quality	Relative Error
Atlanta	0.00	2488307	0.2418
Berlin	0.02	10114	0.3410
Boston	0.00	1107063	0.2390
Champaign	0.02	64760	0.2302
Cincinnati	0.00	318227	0.1449
Denver	0.03	129206	0.2865
NYC	0.02	1927253	0.2393
Philadelphia	0.00	1697409	0.2159
Roanoke	0.27	796030	0.2145
SanFrancisco	0.03	1085013	0.3392
Toronto	0.06	1652074	0.4046
UKansasState	0.00	70318	0.1168
UMissouri	0.05	170427	0.2842

Table 2: BnB Algorithm Performance Table

Instance	Time[s]	Solution Quality	Relative Error
Atlanta	0.02	2003763	0.0000
Berlin	0.55	8087	0.0723
Boston	402.47	958728	0.0730
Champaign	42.53	60181	0.1432
Cincinnati	52.53	277952	0.0000
Denver	570.83	112944	0.1246
NYC	1.55	1783236	0.1467
Philadelphia	177.14	1482811	0.0622
Roanoke	541.27	757720	0.1560
SanFrancisco	6.59	857727	0.0587
Toronto	9.62	1218693	0.0362
UKansasState	22.58	62962	0.0000
UMissouri	119.11	150638	0.1351

Table 3: LS1 Algorithm Performance Table

Instance	Time[s]	Solution Quality	Relative Error
Atlanta	0.01	2003763	0.0000
Berlin	0.41	8087	0.0723
Boston	0.04	999953	0.1191
Champaign	0.54	61010	0.1589
Cincinnati	5.15	214510	0.2282
Denver	2.81	116743	0.1624
NYC	0.80	1783236	0.1467
Philadelphia	0.05	1498008	0.0731
Roanoke	18.00	783018	0.1946
SanFrancisco	2.55	857727	0.0587
Toronto	6.31	1218693	0.0362
UKansasState	2.09	42791	0.3204
UMissouri	1.81	153554	0.1571

Table 4: LS2 Algorithm Performance Table

Instance	Time[s]	Solution Quality	Relative Error
Atlanta	0.03	2003763	0.0000
Berlin	9.42	7712	0.0225
Boston	10.40	908113	0.0163
Champaign	10.75	53896	0.0238
Cincinnati	0.01	280282	0.0084
Denver	7.68	109965	0.0949
NYC	9.42	1666127	0.0714
Philadelphia	9.51	1396495	0.0004
Roanoke	7.18	757971	0.1564
SanFrancisco	9.30	845412	0.0435
Toronto	8.66	1212031	0.0305
UKansasState	0.00	63664	0.0111
UMissouri	14.63	147186	0.1091

5.2 Local Search Algorithm Statistics

5.2.1 QRTDs. The Qualified Runtime Distribution (QRTD) graphs for UMissouri show the major difference between the two local search algorithms, being that the Simulated Annealing algorithm not only performs better but is able to continue to increase its accuracy with a much higher probability as the algorithm continues to run. Figure 4 additionally shows that the Simulated Annealing algorithm is able to reach a good solution quickly, and then iterate on it initially relatively quickly. However, we see that as the solution moves closer to the optimal set the slope of the line begins to flatten out as the improvements take longer to make, something that intuitively makes sense as the set of possible solutions which are an improvement over the current solution decrease, while the search space remains stable.

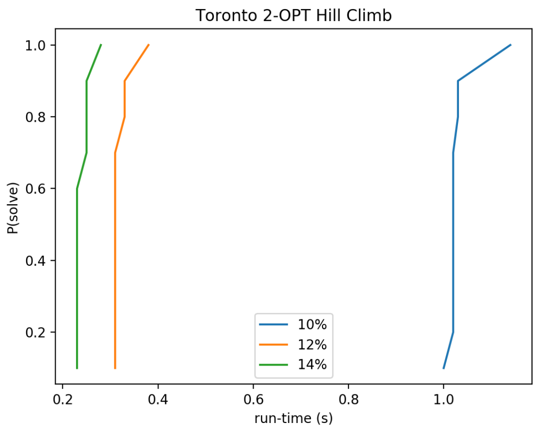

Figure 1: QRTD for the 2-OPT Hill Climb on the Toronto Dataset

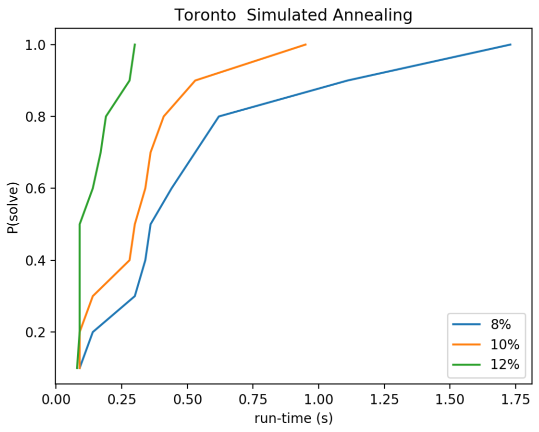

Figure 2: QRTD for the Simulated Annealing on the Toronto Dataset

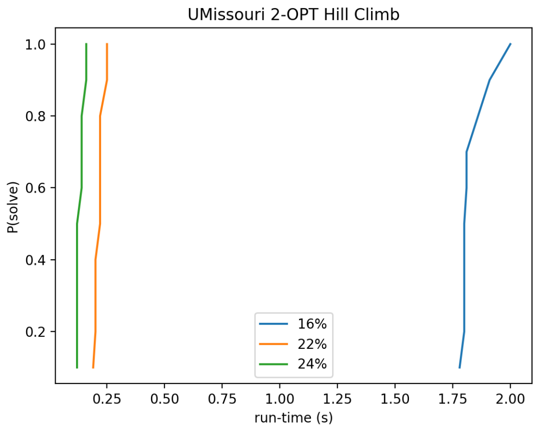

Figure 3: QRTD for the 2-OPT Hill Climb on the UMissouri Dataset

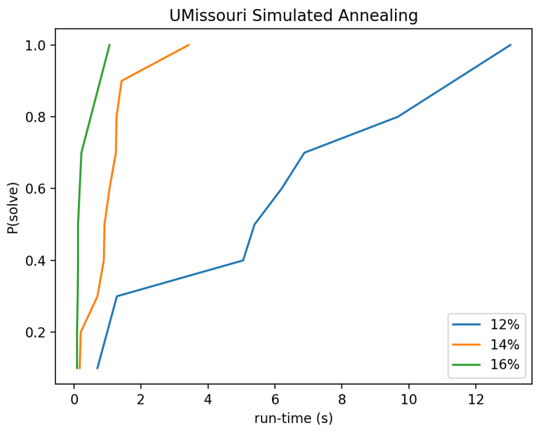

Figure 4: QRTD for Simulated Annealing on the UMissouri Dataset

5.2.2 SQDs. The Solution Quality Distribution (SQD) graphs show the difference in the algorithms due to the highly random nature of the Simulated Annealing implementation in comparison to the 2-OPT Hill Climb implementation. When looking at Figure 5 and Figure 7 we observe that the solution qualities relative to time over all of the runs are almost always the same when the values are truncated to two decimal places. This is in contrast to the Simulated Annealing algorithms shown in Figure 6 and Figure 8 where the time is takes to reach a specified solution quality varies much more. Considering the different use-cases for an algorithm such as this there are situations where both cases are desireable, depending on whether speed or dependability are more important.

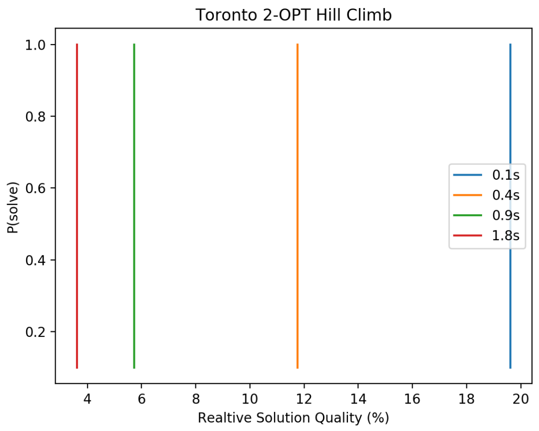

Figure 5: SQD for 2-OPT Hill Climb on the Toronto Dataset

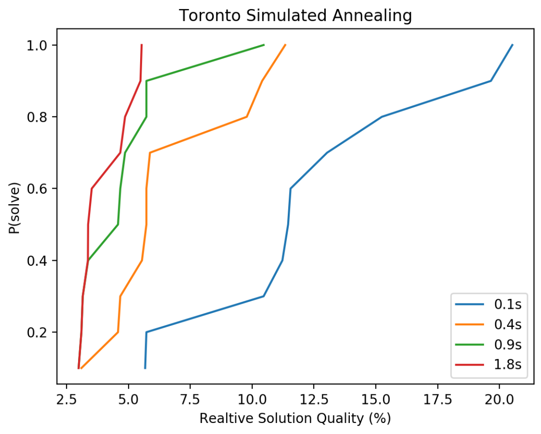

Figure 6: SQD for Simulated Annealing on the Toronto Dataset

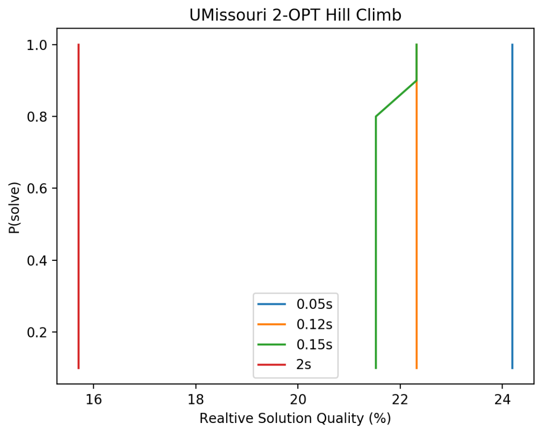

Figure 7: SQD for the 2-OPT Hill Climb on the UMissouri Dataset

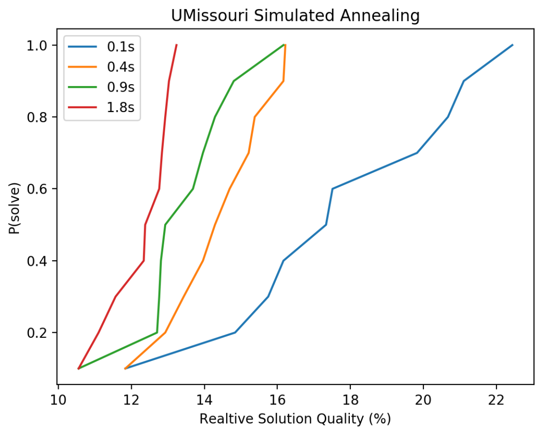

Figure 8: SQD for Simulated Annealing on the UMissouri Dataset

5.2.3 Box Plots. The box plot detailed in Figure 10 is especially interesting with regards to the way which the run-times change as the error of the algorithm decreases. The first box plot shows the run-times to reach a relative error of 10%, while the third plot shows the run-times for a solution error of 17%. The lower error calculations not only have greater magnitude, but also a much higher variance which shows not only the obvious increase in execution time but the increase in run-time variability.

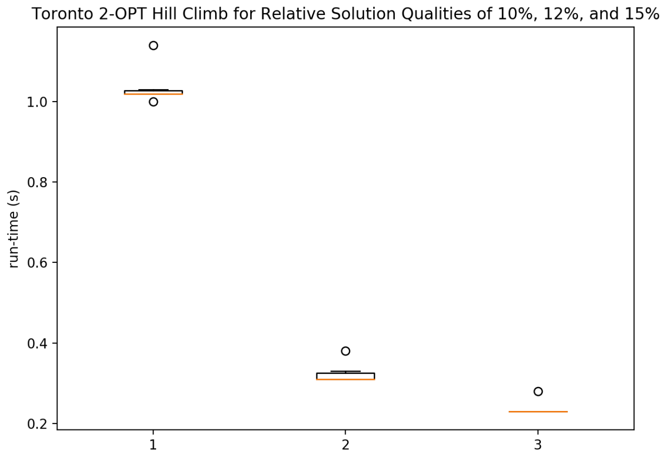

Figure 9: Box Plot for 2-OPT Hill Climb on the Toronto Dataset

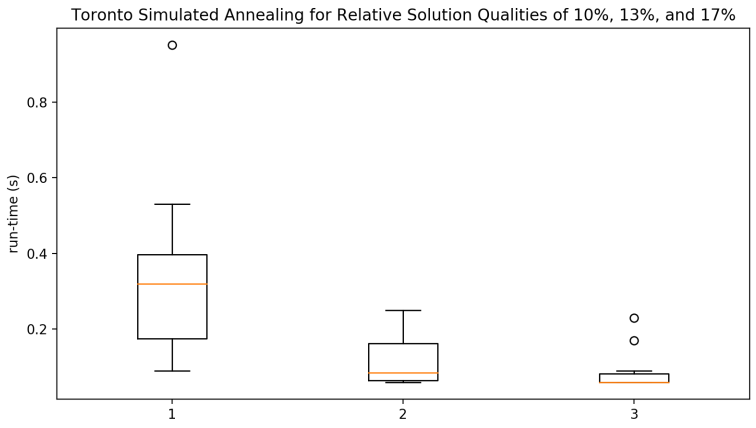

Figure 10: Box Plot for Simulated Annealing on the Toronto Dataset

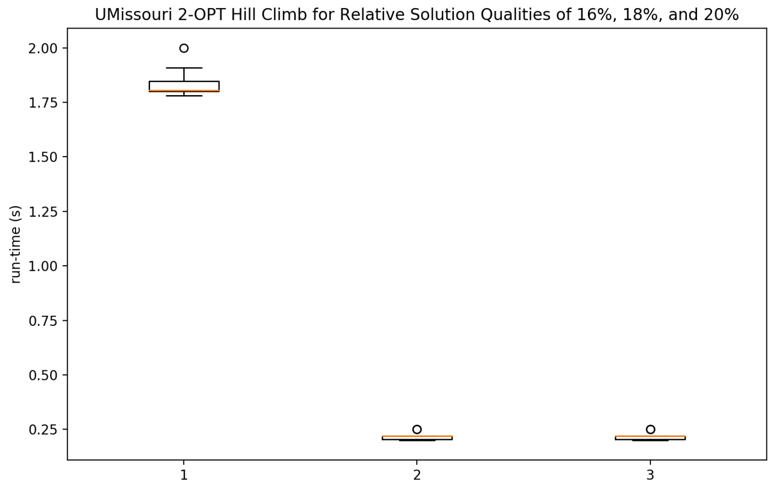

Figure 11: Box Plot for the 2-OPT Hill Climb on the UMissouri Dataset

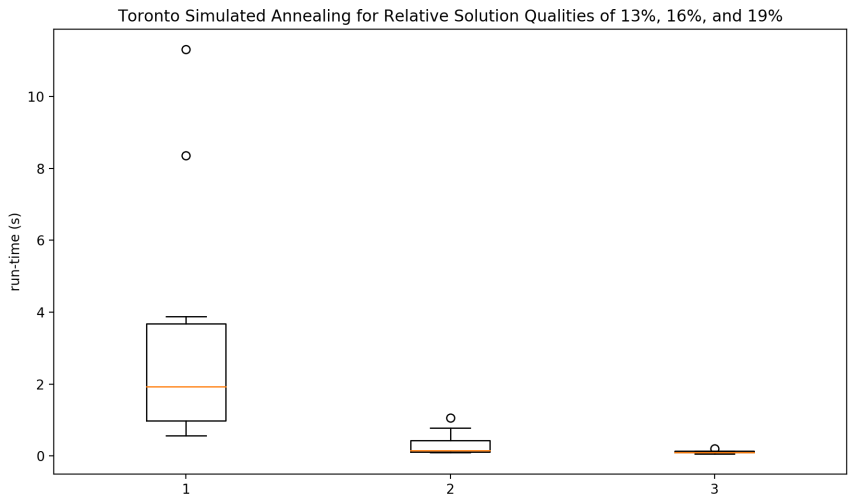

Figure 12: Box Plot for Simulated Annealing on the UMissouri Dataset

6 DISCUSSION

6.1 Branch-and-Bound

The branch and bound algorithm was highly sensitive to the initial solution it used to select an upper bound. If the upper bound solution was poor, then the algorithm struggled to find new solutions as it did not omit many subproblems; surprisingly, if the solution was too good the algorithm also performed poorly as it spent an excessive amount of time performing a depth first search in which it was simply discarding most subproblems. Another reason the algorithm may have performed poorly is due to non-optimal implementation of the Union-Find data structure and Kruskal's algorithm as well as high amount of memory copies when generating child subproblems. In general each iteration of this algorithm was very expensive compared to the local search alternatives which coupled with the large number of subproblems in the search space yielded an algorithm with relatively poor performance.

6.2 Approximation Algorithm

The approximation algorithm serves as a guaranteed method by which to quickly achieve a solution with some guaranteed quality and should be exclusively used as such. Its practical usage appears very limited with the exception of providing heuristics for problems that may inform a more sophisticated algorithm.

6.3 Local Search

The 2-Opt Exchange and Simulated Annealing algorithms each performed relatively well for the majority of data-sets based on algorithm run-time versus performance. For each algorithm instance, a short algorithm runtime was able to produce results which were in most cases within 5% of the optimal solution. Each algorithm setup used a simple Greedy approximation algorithm as the initial path for local search. When testing different initial paths, the Greedy path proved to provide a good mix between accuracy and execution time as a strong initial solution was presented quickly by the algorithm, allowing more local search iterations per time-frame. The Simulated Annealing provided a clear benefit over the 2-opt exchange hill climbing setup however, which was due mainly to the ability of the algorithm to consider a broader neighborhood than the strict neighborhood that the 2-Opt exchange argument considered. To more clearly identify this, the α value was varied with a clear decrease in performance observed when the α value was deviated far from the eventual selection of .98 in either direction. This α value proved to have a reasonable affect on the annealing time-frame which allowed the algorithm to consider the entire search space thoroughly enough while still reaching a reasonable solution in a relatively short period of time. The 2-Opt algorithm struggled with a smaller neighborhood as only 2-Opt exchanges starting from all Greedy solutions were considered, and only those neighborhoods who provided a clear one to one improvement over the current best were used. The power of the Simulated Annealing setup becomes clear when looking at both of these local search algorithms as a similar 2-Opt Exchange neighborhood is considered in both, however the ability of Simulated Annealing to allow worse routes temporarily with the hope that they eventually lead to a more desireable solution proved to be key.

7 CONCLUSION

As is true with any engineering design, the type of implementation that should be used when looking to solve the TSP is heavily based on the type of results that are desired. All of the algorithms that were tested proved optimal for various situations depending on whether or not the driving factors for the implementation were speed or accuracy of the final solution in relation to the optimal. For example, the Branch-and-Bound algorithm proved to be reasonable when the dataset was relatively small, time was not a factor, and the desired outcome was to obtain a solution close or equal to the optimal. When the solution space ballooned the branch-and-bound did not provide as good of results compared to the extra time that it took to traverse the solution space, often becoming stuck traversing branches which in the end proved worse or not much better than previous solutions. For these larger cases both the Simulated Annealing and 2-OPT Hill Climb algorithms performed in many cases better than the Branch-and-Bound algorithm in considerably less time. Had the Branch-and-Bound algorithm been given hours to run instead of the designated 10 minutes, the solution returned likely would have been much closer to optimal than the Local Search algorithms were able to provide. However, when compute power and time is a constraint the Local Search algorithms proved to be the optimal method for evaluating the TSP algorithm. The Approximation algorithm in all cases returned with the greatest error, something which is to be expected based on the algorithm implementation. However, the extremely short runtimes make the approximation algorithm reasonable when looking to computer an initial solution, or when solution error is not the most important factor to consider execution time is prioritized.

REFERENCES

[1] [n.d.]. approximation algorithm. http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture28-ApproximationAlgorithm.htm

[2] [n.d.]. branch and bound algorithms - principles and examples. https://janders.eecg.utoronto.ca/1387/readings/b_and_b.pdf

[3] ERICA KLARREICH. 2013. *Computer Scientists Take Road Less Traveled.* https://www.quantamagazine.org/computer-scientists-find-new-shortcuts-to-traveling-salesman-problem-20130129/

[4] Gilbert Laporte. 1992. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research* 59, 2 (1992), 231âĂŞ247. https://doi.org/10.1016/0377-2217(92)90138-y