Approaches of the Travelling Salesman Problem

An Analysis of Four Algorithms

Julia Redston
College of Computing
Georgia Institute of Technology
Atlanta GA USA
jredston3@gatech.edu

Joseph Cantrell
College of Computing
Georgia Institute of Technology
Atlanta GA USA
jcantrell32@gatech.edu

Daham Eom
College of Computing
Georgia Institute of Technology
Atlanta GA USA
deom7@gatech.edu

ABSTRACT

The traveling salesman problem is such that given a set of cities and the distances between each city pair, the shortest possible path that visits each city and returns to the origin city must be found. The problem is applicable in far more cases than the one described in the problem, often appearing in microchips, planning, and scheduling problems among others. The problem is NP-Complete, and currently, the most well-known algorithms for calculating a solution take exponential time to complete in relation to the input size. This makes the problem computationally complex, and difficult to solve with large datasets. For this reason, many algorithms have been developed that cannot guarantee the shortest path but can provide an approximate solution in much less time.

In this paper, we will look at several different algorithms that either solve or approximate the traveling salesman problem and analyze their runtimes as well as the accuracy of their solution. We will be running our algorithms using latitude and longitude data from real cities. The algorithms tested are branch and bound, a construction heuristic, and two local search algorithms. The construction heuristic used will be an approximate algorithm using a minimum spanning tree, one of these local search algorithms is random hill climbing choosing the best successor, and the second local search algorithm is a simulated annealing.

CCS CONCEPTS

• Theory of computation~Shortest paths

KEYWORDS

Traveling salesman problem, Graph algorithms, Branch and bound, Construction heuristic, Local search, Random hill climbing, Simulated annealing

ACM Reference format:

Joseph Cantrell, Julia Redston and Daham Eom. 2018. Approaches of the Travelling Salesman Problem: An Analysis of Four Algorithms. In *Proceedings of Georgia Institute of Technology Conference. ACM, Atlanta, GA, USA, 4 pages.*

1 Problem Definition

We will analyze the performance of four different algorithms to solve the traveling salesman problem. There are fourteen different files containing the information necessary to construct graphs of different points in a city. Each point is given as an x-y coordinate, and each graph has N points, where N varies from 10 to 230. The traveling salesman problem seeks to find the shortest cycle that visits all N points. The distance between two points u and v, c(u,v), is the Euclidean or Geographic distance between them, depending on the graph. Only one graph, ulysses16, uses Geographic distances. This traveling salesman problem is metric, and thus all edge costs are symmetric, where distance(u,v) is equal to distance(v,u). The edge costs also satisfy the triangle inequality, wherein the sum of two edges of a triangle is greater than the length of the remaining edge.

2 Related Work

The traveling salesman problem is important in many fields as it is applicable to a wide variety of problems, and many algorithms for solving or approximating solutions have been made. In the 1950's the first use of branch and bound was used to solve the problem, and in the 1960's the MST, approximation became popular [1]. In 1976 the Christofides algorithm was discovered which offered a 1.5 approximation [2].

There have been more recent discoveries and studies on the traveling salesman problem as well. In 2006 the longest known optimal tour was beaten when an optimal tour for an 85,900-city instance was calculated to solve the number of connections needed to create a custom microchip [3].

The traveling salesman problem has been used in many applications besides the obvious movement between cities, including creating microchips, using X-rays to analyze crystalline structures, and certain scheduling problems [4].

3 Algorithms

3.1 Branch and Bound

The brute force algorithm for TSP simply finds all possible Hamiltonian cycles of an input graph and returns a cycle with the shortest distance. This brute force algorithm guarantees an optimal solution, but the time complexity is O(n!) which is quite inefficient. Using a branch and bound algorithm improves the performance of solving TSP by limiting the total number of cycle

computations. For each "branch" there is a lower bound, initially set to the sum of distances from each city to its two closest cities. Then, the lower bound is updated by subtracting two shortest distances of an encountered city. Whenever a cycle's lower bound exceeds the current best solution, the branch is pruned and no further computations are processed for that cycle.

The branch and bound algorithm for TSP guarantees the optimal solution. However, it is still expensive to solve a TSP problem, especially with a large number of cities. In the worst case, if none of the branches are pruned, the branch and bound algorithm takes the same amount of time as the naive brute force algorithm. Therefore, both time complexity and space complexity of the branch and bound algorithm for TSP are O(n!).

The pseudo-code is as follows:

```
costs = n x n matrix, costs[i][j] represents a distance from i to j
bound = sum of two shortest edges of all cities / 2

function(bound, sum_length, path):
   if all cities are checked:
      best_so_far = current_path
      best_length = current_length
      halt
   endif
   for i to num:
      prev = previous node of path
      if prev != i and i is not visited:
         sum_length += costs[prev][i]
         temp = bound
         bound -= sum of two shortest edges of current node / 2
         if bound + sum_length < best_length:
            mark i as visited
            path.add(i)
            function(bound, sum_length, path)
         endif
         if prev != i:
            sum_length -= costs[prev][i]
         endif
         bound = temp
         remove marks of nodes less than current level.
      endif
   endfor
end
```

3.2 Construction Heuristic

A heuristic can be thought of as a mental shortcut, a rule of thumb, or an educated guess. A heuristic cannot guarantee correctness but is either generally true or "close enough" to the optimal solution. For this problem, we used a minimum spanning tree (MST) as an approximate algorithm for the traveling salesman problem. The MST-Approx algorithm guarantees a solution that is at most two times the length of the optimal solution, so it is 2-approx. This algorithm is useful because it is relatively fast, but at worst can be twice the optimal solution, making it useful when finding a solution in a timely manner is more important than accuracy.

First, we use Prim's Algorithm, which is $O(n^2)$, to determine the minimum spanning tree. Then we double every edge in the MST. We traverse the resulting graph to find a path, never visiting an edge more than once, and when a node is repeated in the path we remove it.

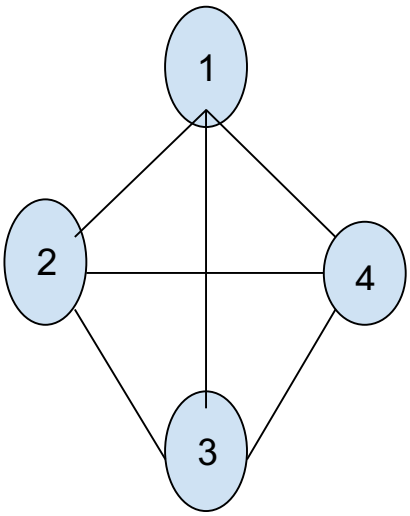


fig. 1

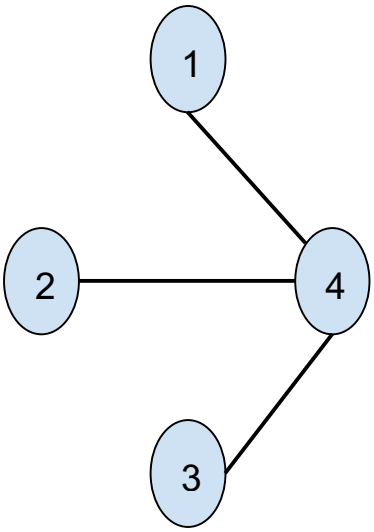


fig. 2

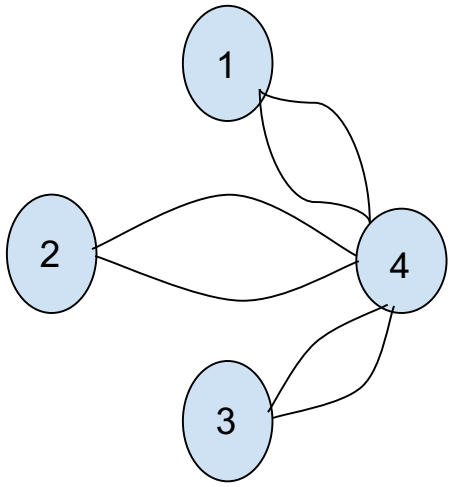


fig. 3

For example, if we take the graph in fig. 1, we can find the MST in fig. 2. Doubling every edge results in fig 3. Traversing the graph results in the path 1-4-2-4-3-4-1, and removing every duplicate node that becomes 1-4-2-3-1. This final path is an approximate solution for the traveling salesman problem.

This algorithm requires we run Prim's Algorithm once, which takes $O(n^2)$ time. Then we traverse the MST once, calculating distances as we go, which takes O(n) time. Thus the approximate algorithm has a runtime complexity of $O(n^2)$. The space complexity is O(n), as all that needs to store is the array of n nodes, the array size n of the shortest distance to each of those n nodes, and the array of the solution path of size n.

The pseudocode is below. The function getTourLength(nodes) will calculate and sum the distance between each node in nodes. This distance includes the distance between the last node and the first node. The function getDist(a,b) will calculate the distance between nodes a and b.

```
function getTourLength(Vertex[ ] nodes):
  float length = 0
  for i=1 to i=length(nodes) - 1:
    length += getDist(nodes[i-1], nodes[i]
  length += getDist(nodes[length(nodes)-1], nodes[0])
  return length

function mst_approx(Graph G):
MST = primsAlg(G)
MST =   doubleAllEdges(MST)
totalLength =  0
tour = []
StartNode = MST.nodes[0]
prevNode = startNode
node = startNode
while len(tour) < len(G.nodes):
  if node not in tour:
      tour.add(node)
      totalLength += G.getDist(prevNode, node)
       prevNode =   node
  node = node.next()
totalLength += G.getDist(firstNode, lastNode)
tour.add(endNode)
```

3.3 Local Search 1

The random hill climbing algorithm is a local search algorithm that finds a solution to TSP, but is not guaranteed to be optimal. The algorithm begins with a solution path containing the problem's vertices in random order. For each vertex i, the algorithm will evaluate all the other vertices (the neighborhood) to find the best vertex with which to swap with i. The best vertex is chosen based on its improvement to the solution, ie how much it decreases the path length. If the solution distance would not decrease, no vertices are swapped. The final solution will be given at the end of these two nested loops.

The pseudocode for local search is as follows. The randomize(seed) function will randomize the random number generator according to the integer seed. The length(A) function will return the number of elements in the array A. The function randomInt(a,b) will generate a random number from a to b, inclusive. The array function append(a) will append an element a to an array, while the function remove(a) will remove an element from the array.

```
function hillClimb(Vertex[ ] nodes, int seed):
  randomize(seed)
  Vertex[ ] choices = nodes
  Vertex[ ] init = [ ]
  //generate initial random solution
  for i=0 to i=length(nodes) - 1:
    choice = randomInt(0, length(choices) - 1)
    init.append(choices[choice])
    init.remove(choices[choice])
  float bestLength = getTourLength(init)
  //check if swapping parts of the solution decrease tour length
  for i=0 to i=length(init) - 2:
    bestSuccessor = null
    bestSuccessorNum = null
    bestSuccessorLength = ∞
    //find the best successor to swap with position i
    for j=1 to j=length(init) - 1:
      Vertex[ ] successor = init
      Vertex temp = successor[i]
      successor[i] = successor[j]
      successor[j] = temp
      successorLength = getTourLength(successor)
```

```
    if successorLength < bestSuccessorLength:
      bestSuccessor = successor
      bestSuccessorNum = j
      bestSuccessorLength = successorLength
  Vertex[ ] climb = init
  Vertex temp = climb[i]
  climb[i] = climb[bestSuccessorNum]
  climb[bestSuccessorNum] = temp
  float climbLength = getTourLength(climb)
  //check if the best successor swap decreases tour length
  if climbLength < bestLength:
    init = climb
    bestLength = climbLength
 return init
```

The time complexity of the random hill climbing algorithm is $O(n^2)$ because there are two nested loops which will both iterate through each vertex. Parsing the graph takes O(n) time, so the final time complexity is $O(n^2)$. The space complexity of this algorithm is O(n) because the only data structures used are arrays holding the vertices.

The strength of the random hill climbing local search lies in the speed with which it can gather a solution by iteratively attempting to improve a random initial solution. However, this algorithm will fail to find the optimal solution if there are local optima in the search space. The algorithm will generate better solutions until reaching a local optimum wherein no further improvement can be made. This can result in a very suboptimal solution if the search space of candidate solutions has many scattered local optima.

Random hill climbing was chosen because it produces a solution very quickly. Choosing the best successor in the neighborhood was done to find the smallest tour length possible by decreasing the tour length as much as possible at each iteration. However, the problem of local optima still exists with this strategy of evaluating successors. There was no automatic tuning or configuration done with this algorithm.

3.4 Local Search 2

The traditional hill climbing search algorithm has advantages of relatively simple implementation and inexpensive cost of searching. However, the hill climbing algorithm is not considered as an ideal algorithm because the process can be stuck at a local optimum. To resolve the incorrect convergence, simulated annealing algorithm is designed. The algorithm uses temperature and cooling factor to check other possible sub-optimal solutions with descending possibility of searching. More specifically, from a one of the solutions, the simulated annealing algorithm searches its neighbors by randomly swapping two nodes in the path. At the same time, the simulated annealing uses a possibility which is an exponential of negative of difference between new solution and old solution over temperature. Therefore, when the temperature value is high, the algorithm actively searches other possible solutions with high possibility, but as the temperature is decreased by the cooling factor, it reduces the frequency of searching other solutions. Then, when the temperature reaches to a threshold, the algorithm returns the best possible optimal solution. Because of the searching possibility, the algorithm can explorer searching space without stucking at a point.

The space complexity of this algorithm is O(n) since the only required data structures are lists of size n which stores paths of a possible solution and the best solution. Unlike the other algorithm, the time complexity is depending on the temperature, and cooling factor with a threshold and the rough time complexity is $O(n^k)$ where k is number of iterations for temperature decay until passing the threshold. To obtain the best possible solution for this TSP, the temperature is set to the maximum 64 bit integer value supported by Python, cooling factor is 0.999, and the threshold is 0.0001. Any automated tuning or configuration are not used for this algorithm.

The pseudo-code is as follows:

```
function simulated_annealing():
  temperature = max_int
  cooling_factor = 0.999
  threshold = 0.0001
  best_length = max_int
  current_path = one of the possible solutions
  while temperature > threshold:
    if iterations == 1000 and candidates are all same:
      current_path = one of the possible solutions
    endif
    new_path = current_path after swapping two nodes
    diff = new_path_length - current_path_lengh
    if diff < 0 or e^(-diff / temperature) > random number:
      current_path = new_path
    endif
    if current_length < best_length:
      best_path = current_path
    endif
    temperature = temperature * cooling_factor
  endwhile
  return best_path, best_length
```

The simulated annealing is selected because it has advantages on both time complexity and the solution quality. As the property of a local search algorithm, the simulated annealing algorithm does not guarantee an optimal solution. However, it generates a suboptimal solution that is very close to the global optimal

solution in noticeably short amount of time comparing to the branch and bound algorithm. Increasing temperature and cooling rate (but less than 1), and decreasing threshold can improve the solution quality, but this will also increase the time complexity which is still acceptable.

4 Empirical Evaluations

The four algorithms were run by a machine with an Intel i7-7700HQ processor with 2.8 GHz clock speed and 16 GB DDR4 RAM. The operating system used is 64-bit Windows 10 version 1803. The implemented code is written, compiled, and executed using Python 3. The four algorithms were all executed with a maximum time of 600 seconds, after which they return the best solution generated so far. For each of the fourteen graphs, the local search algorithms were run ten times with ten different random seeds. Results for these ten runs were averaged to provide a better overview of the local search algorithms.

For each algorithm we report the time taken to find the final solution, the total distance in the solution's tour, and the error relative to the optimal solution. A relative error of 0 indicates the optimal solution was found, whereas a relative error of .5 indicates the solution found was 50% higher than the optimal solution. Branch and bound was used to calculate the optimal solution for the smaller graphs. Because of the exponential nature of branch and bound, some graphs were too large to afford an optimal solution in a reasonable timeframe, and these graphs do not have a reported relative error.

4.1 Branch and Bound

DataSet	Time (s)	Solution Quality	Relative Error
Atlanta	600	2497224	0.2463
Berlin	600	19149	1.539
Boston	600	2203741	1.466
Champaign	600	212857	3.043
Cincinnati	2.85	277952	0
Denver	600	540412	
NYC	600	7163135	
Philadelphia	600	3624919	
Roanoke	600	6847051	
SanFrancisco	600	5581318	
Toronto	600	8917606	
UKansasState	0.57	62962	0
ulysses16	600	7046	0.02726
UMissouri	600	658567	

The only converged test cases for branch and bound are Cincinnati and UKansasState with 10 nodes and the solutions obtained from the branch and bound are optimal. However, any test cases with more than 10 nodes failed to converge within a 10 minute limit. It is shown in the test case ulysses16 that the input size is critical to the overall performance of the branch and bound. With 10 nodes, the branch and bound solves Cincinnati and UKansasState in 2.85 seconds and 0.56 seconds, respectively. However, adding 6 more nodes (ulysses16 with 16 nodes) results in reaching the time limit. Additionally, using test cases with a large number of nodes and same time limit ends up with higher errors of 1.539, 1.466, and 3.043.

4.2 Approx (MST)

DataSet	Time (s)	Solution Quality	Relative Error
Atlanta	0.00166	2488307	0.2418
Berlin	0.0072	10114	0.341
Boston	0.0042	1107063	0.239
Champaign	0.0101	64760	0.2302
Cincinnati	0.00145	318227	0.1449
Denver	0.0305	129206	
NYC	0.0159	1927253	
Philadelphia	0.00488	1697409	
Roanoke	0.249	796030	
SanFrancisco	0.0413	1085013	
Toronto	0.0503	1652074	
UKansasState	0.00196	70318	0.1168
ulysses16	0.00369	7803	0.1376
UMissouri	0.0603	170427	

Every instance of the MST approximation algorithm easily ran within the 600 second time limit, with the slowest run being for Roanoke which still completed in less than half a second.. Of the

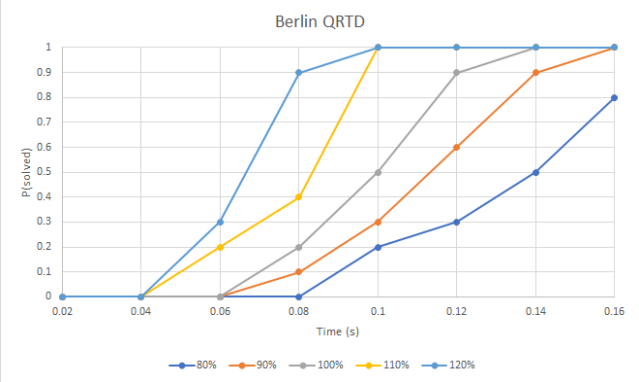
graphs we have the optimal solutions for, the average relative error compared to the optimal solution is .2073, well under the guaranteed upper bound of 1, given that this is a 2-approx algorithm. The largest error is for Berlin at .341, about 1.6x the average, it is the largest graph at 52 nodes. The smallest cities, Cincinnati, UKansasState, and ulysses16, all have a relative error below the average.

4.3 Local Search 1

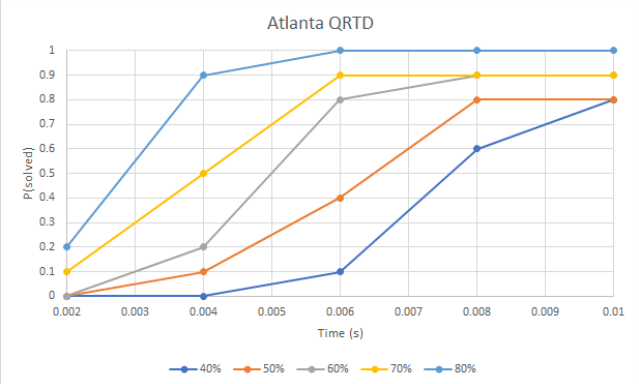
DataSet	Time (s)	Solution Quality	Relative Error
Atlanta	0.01	2773238.1	0.384
Berlin	0.15	13296.1	0.7629
Boston	0.07	1337172.3	0.4965
Champaign	0.17	91702.2	0.742
Cincinnati	0.001	287315.3	0.03369
Denver	0.6	214791.7	
NYC	0.34	2982677.1	
Philadelphia	0.03	2051776.1	
Roanoke	12.58	2037156.3	
SanFrancisco	1.02	2152361.2	
Toronto	1.35	3567721.2	
UKansasState	0.0012	74847.7	0.1888
ulysses16	0.014	8206.8	0.1965
UMissouri	1.23	281068.9	

The solutions for the random hill climbing algorithm with best successor strategy are acquired extremely quickly. The table above shows results averaged across ten runs with different random seeds. The smallest graph Cincinnati, with 10 nodes, is computed in about .001 seconds, while even the largest graph, Roanoke with 230 nodes, is computed in 12.58 seconds on average.

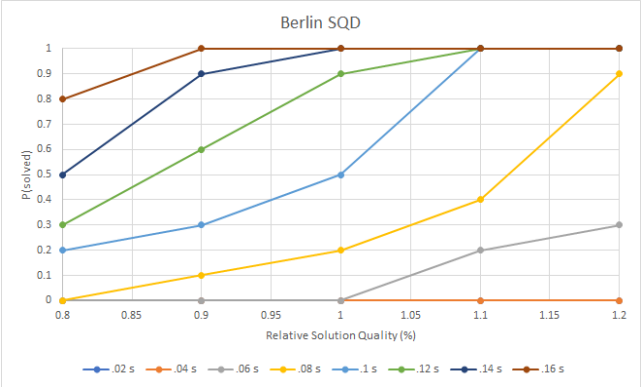
The relative solution error varies from about 3% for the smallest graph Cincinnati to 76% for Berlin. For a small graph like Cincinnati, there are less local optima at which the hill climbing algorithm can become stuck, resulting in errors as small as 3%. UKansasState is also 10 nodes but has about 19% error, while ulysses16 is 16 nodes and has about 20% error. This indicates UKansasState has many more local optima than Cincinnati. Champaign is 55 nodes and has a much larger error of about 74%, while Berlin with 52 nodes has an error of about 76%. In these larger graphs, there are many more opportunities for local optima at which the algorithm can become stuck.



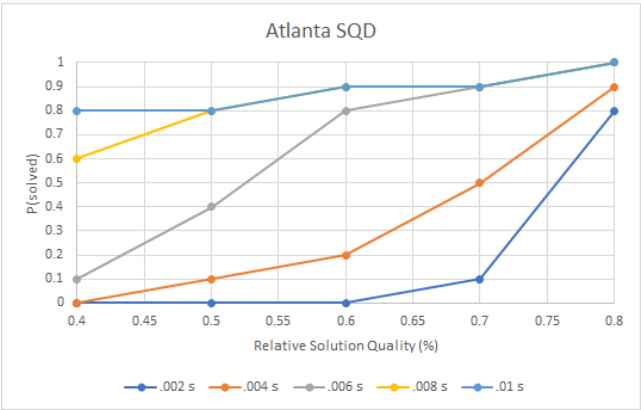
The Berlin Qualified Runtime Distribution (QRTD) graph above displays the fraction of the ten algorithm runs which, at a specific cutoff time, have generated a solution within a certain percentage of error (q*) from the optimal solution. This graph is a medium size of 52 nodes. No algorithm runs could produce a solution within 120% of the optimal within .04 seconds. All algorithm runs produced a solution within 90% of the optimal solution within .16 seconds. Within a similar .16 seconds, only 80% of the algorithm runs could produce a solution within 80% of the optimal solution.



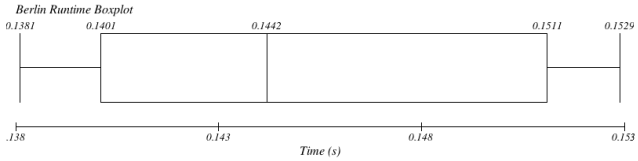
The QRTD for Atlanta, a small graph of 20 nodes, is shown above. Runs of the hill climbing algorithm fail to reach a consensus more than shown in the Berlin QRTD. Only the highest q* value of 80% had all algorithm runs find a solution within q* of the optimal solution. By contrast, only 70% of algorithm runs could find a solution within 40% and 50% of the optimal solution. In comparison to Berlin, where all but one value of q* reach a consensus, this shows a higher variability in the results of algorithm runs for Atlanta.



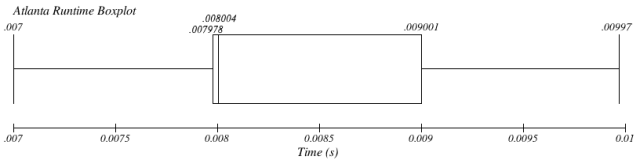
The Solution Quality Distribution (SQD) for Berlin is shown above. For each time cutoff, the graph plots the fraction of ten algorithm runs which have generated a solution within a particular percentage of the optimum solution. The time cutoffs .1 seconds, .12 seconds, .14 seconds, and .16 seconds have all ten algorithm runs reach a consensus at a relative solution quality of 1.1 (110% of the optimal solution). These time cutoffs all generate some fraction of solutions with a relative solution quality of .8, with .16 seconds having 80% consensus and .1 seconds having 20% consensus. The remainder of cutoff times, .02 seconds through .08 seconds, fail to generate any solutions with a relative solution quality of .8 and fail to reach a consensus at the highest relative solution quality of 1.2.



The Atlanta SQD is shown above. Here, .006 seconds, .008 seconds, and .01 seconds reach a consensus among all ten algorithm runs at a relative solution quality of .8 (within 80% of the optimal solution). Within .01 seconds, 80% of algorithm runs had a relative solution quality of .4. Within .004 seconds and .002 seconds, however, 0% of algorithm runs could generate a solution with relative solution quality of .4. The algorithm runs in Berlin converge to a consensus more quickly and frequently than the runs for Atlanta.



The above boxplot for Berlin shows the different quartiles of overall runtimes for the hill climbing algorithm runs. The range of runtimes is .1381 seconds to .1529 seconds, with an average close to the middle of these two values, .1142 seconds. The sparse placement of the quartiles shows a fairly even distribution of runtimes skewed slightly toward smaller values.



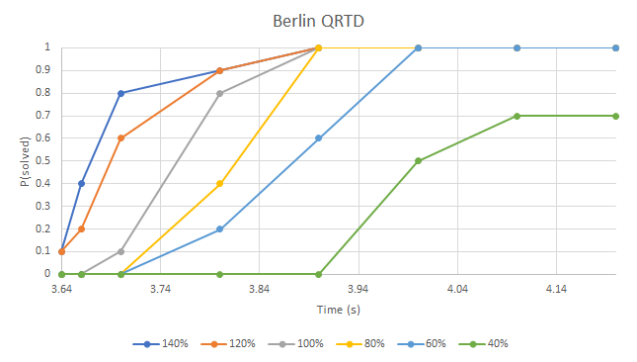
The above boxplot for Atlanta shows a minimum algorithm completion time of .007 seconds with a maximum of .00997 seconds. The median of .008004 seconds is skewed toward smaller values (the midpoint between the low and high values is .0085 seconds). The first quartile is strikingly close to the median at .007978 seconds, demonstrating most algorithm runs took about .008 seconds to complete. Like Berlin, the Atlanta algorithm runtimes are skewed toward smaller values.

4.4 Local Search 2

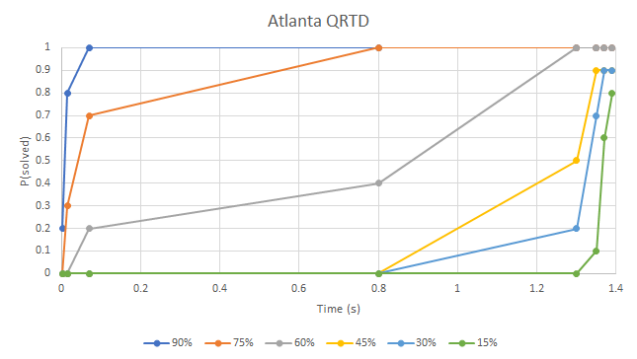
DataSet	Time (s)	Solution Quality	Relative Error
Atlanta	1.443	2103432.3	0.04974
Berlin	4.657	9733.7	0.2906
Boston	3.103	1046856.1	0.1716
Champaign	4.811	69859.2	0.3270
Cincinnati	1.215	297332.2	0.06972
Denver	8.779	152431.0	
NYC	6.310	2243137.4	
Philadelphia	2.228	1563803.1	
Roanoke	20.87	1936664.7	
SanFrancisco	9.163	1558695.7	
Toronto	10.24	2444077.1	

UKansasState	0.7762	62962.0	0
ulysses16	3.163	6908.4	0.007202
UMissouri	10.06	223329.6	

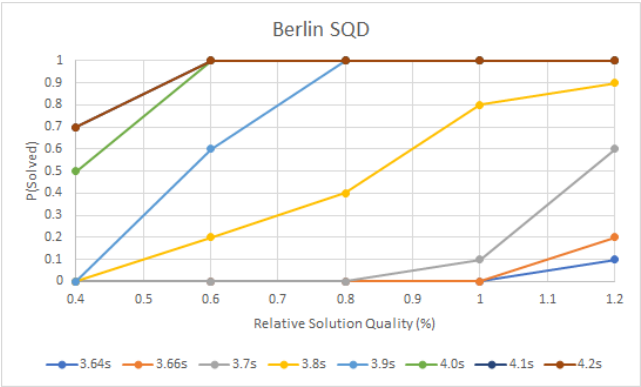
The simulated annealing runs 10 times for each test case and the results are averaged as shown in the above table. Since the parameters for the simulated annealing are set to focus on searching more possible solutions rather than quickly converging, the running time is generally long, but most relative errors are acceptable. It finds the optimal solution for the UKansasState case. The running time is proportional to the size of the TSP case and using larger cases tends to return higher relative error. To be specific, the algorithm yields the lowest relative error as 0 for UKansasState case and the highest error as 32.7% for Champaign test case. The longest running time taken is 20.87 seconds with Roanoke TSP with 230 locations and the shortest running time is 0.7762 seconds with UKansasState TSP of 10 locations.



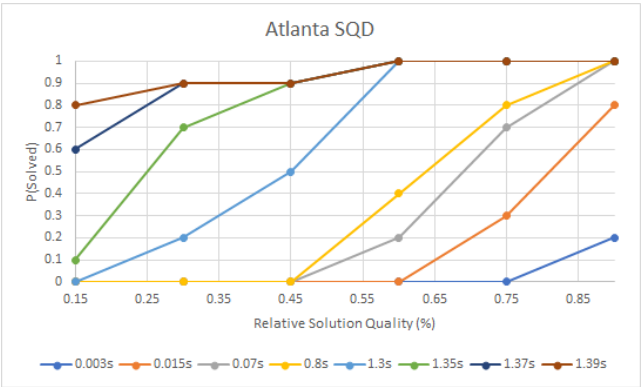
The Berlin QRTD graph is created based on the results of 10 runs of the simulated annealing algorithm for Berlin TSP. All the simulated annealing runs obtain a solution quality of 80% of the optimum or higher within 3.9 seconds. Also, all runs result in 60% of solution quality, but only 70% of runs are able to get 40% solution quality.



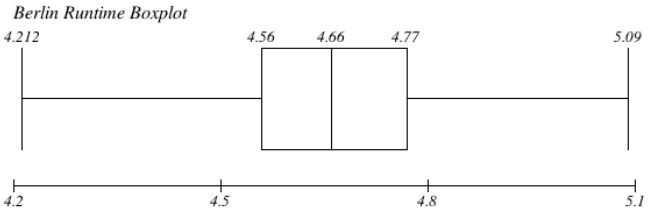
The Atlanta QRTD indicates that 60% to 90% solution quality is achieved by all algorithm runs within 1.3 seconds. However, not all of the runs reach the 15% to 45% solution quality. 9 out of 10 runs produce 45% and 30% of solution quality in 1.4 seconds and only 8 runs produce a solution quality of 15%. Also, none of algorithm runs can make any result of 15% to 45% solution quality until 0.8 seconds



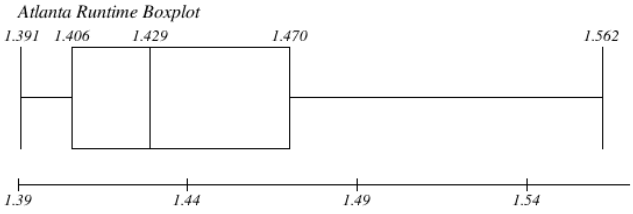
In the Berlin SQD, all algorithm runs in 3.9 seconds to 4.0 seconds reaches to the 80% of solution quality while all runs in 3.8 seconds to 3.64 seconds cannot fully reach to any solution quality. In 3.7 seconds to 3.64 seconds, none of runs generates solution quality less than 100% and 110%.



In Atlanta SQD, all runs in 1.3 seconds to 1.39 seconds produce 60% solution quality and above. However, none of algorithm runs in 0.8 seconds to 0.003 seconds cannot make any solution quality until 45%. All runs in 0.8 seconds and 0.07 seconds generates 90% solution quality, but 7 runs in 0.015 seconds and only 2 runs in 0.003 seconds reach to the 90% solution quality.



The Berlin runtime boxplot is generated in a range of 4.212 seconds to 5.904 seconds with first quartile of 4.562 seconds, median of 4.658 seconds, and third quartile of 5.770 seconds. This indicates that most algorithm runs take 4.6582 seconds to complete while the longest running time is 5.904 seconds and the shortest running time is 4.212 seconds



Unlike the Berlin runtime boxplot, the above boxplot shows non-evenly distribute running times. The maximum running time is 1.562 seconds, minimum running time is 1.391 seconds. The first and third quartiles are 1.406, and 1.470, and the median is 1.429. The distribution shows that most algorithm runs generally take 1.429 seconds which is far from the maximum value and thus the maximum running time is not typical.

5 Discussion

Each of these algorithms performed quite differently. Branch and Bound was able to find an optimal solution for smaller graphs, but often exceeded the cutoff time of 600 seconds and did not find the optimal solution. Branch and bound was the most precise but slowest algorithm tested. The solutions reported after exceeding the cutoff time of 600 seconds (for larger graphs) had a much greater error than the other algorithms.

The approximation algorithm found a solution the fastest of the four algorithms, taking a fraction of a second to find solutions. However the resulting solutions tended to have a higher error than the branch and bound or the simulated annealing algorithm, although it generally had lower error than the random hill climbing algorithm. On average the relative error for the mst approximation was about 50% that of the hill climbing algorithm. The random hill climbing algorithm only had a lower relative error once, when run on Cincinnati. The random hill climbing algorithm error here (.03369) is significantly smaller than any relative error of the heuristic function, with the heuristic function algorithm resulting in about 400% the error of the random hill climbing algorithm.

The random hill climbing algorithm was the second fastest algorithm and suffered the most error of the algorithms. Random hill climbing took much longer than the construction heuristic algorithm for larger graphs; it took 51 times as long for the Roanoke graph of 230 nodes. However, random hill climbing was able to match the speed of the construction heuristic algorithm on smaller graphs such as Cincinnati and UKansasState, each having 10 nodes. The random hill climbing had errors 100% to 2600% higher than the simulated annealing algorithm for all but two instances. In one instance, Cincinnati, the random hill climbing algorithm outperformed the simulated annealing algorithm with only 50% of the amount of error found in simulated annealing. In another instance, UKansasState, the random hill climbing algorithm became stuck at a local optimum with 20% error, while the simulated annealing algorithm found the exact solution.

The simulated annealing algorithm took the second longest to produce a solution but was also the second most accurate. Simulated annealing took much longer than random hill climbing for smaller graphs, needing 1215 times more time for Cincinnati and 647 times more time for UKansasState, both graphs of 10 nodes. However, simulated annealing only needed twice as much time to find a solution for Roanoke, a graph of 230 nodes.

6 Conclusion

All of the algorithms we examined and tested were very different, with different time complexities big O's, and accuracy, but each algorithm has its use. In situations where getting the optimal solution is the primary task regardless of the time complexity, then branch and bound would be an ideal algorithm. If it is required that solutions can be found in under ten minutes, but otherwise accuracy is more important than speed, the simulated annealing algorithm is a better fit. If instead the main focus was to get a solution as fast as possible and an approximation solution was acceptable then the mst heuristic approximation algorithm or random hill climbing would be of better use. If the truth was somewhere between those two extremes then one of the local search algorithms may be ideal, depending on how much time and what margin of error the situation allows.

Each algorithm has its own strengths and weaknesses, with no algorithm being necessarily better or worse. When encountering the traveling salesman problem, discretion should be used in determining which algorithm is best suited for the situation at hand.

Ultimately, the traveling salesman problem is a versatile problem that applies to a wide variety of situations, so it is unsurprising that possible solutions differ greatly and are applicable in different situations.

REFERENCES

[1] Lawler, E., Lenstra, J., Rinnooy Kan, A. et al. J Oper Res Soc. 1986. 37: 655. DOI: https://doi.org/10.1057/jors.1986.117

[2] Nicos Cristofides. 1976. *Worst-case analysis of a new heuristic for the traveling salesman problem.* Graduate School of Industrial Administration. Carnegie Mellon University, PA.

[3] D. Applegate, R. E. Bixby, V. Chvatal, W. Cook. 2006. *The Traveling Salesman Problem.* Princeton University Press, Princeton

[4] Rajesh Matai, S. S., & Mittal, M. L. 2010. Traveling salesman problem: An overview of applications, formulations, and solution approaches, traveling salesman problem, theory and applications. In Traveling salesman problem, theory and applications. *InTech.* DOI: http://dx.doi.org/10.5772/547.