Empirical Analysis of a Divide and Conquer Algorithm for the Traveling Salesman Problem

Presented by:

Justin Bryan

Payton Hall

John Whetten

Zachary Wallace

Abstract:

We present the implementation of a divide and conquer algorithm for the traveling salesman problem. We examine its efficiency and accuracy with respect to a naïve random tour algorithm, a greedy algorithm, and a branch and bound algorithm. Empirical evidence suggests a possible use-case for this divide and conquer within a specific range of problem sizes.

Introduction:

The traveling salesman problem has won a place in the popular consciousness as the archetypical example of a NP-complete problem ever since it was formally identified as such[1]. The general problem can be formulated in a number of ways, but we may summarize it as follows: *Given a list of points and the distances between those points, find the shortest path that passes through each of those points exactly once.* The traveling salesman problem has applications in a wide variety of technical and industrial fields, which has resulted in considerable research on the subject. Despite that, the problem remains intractable for data sets of any large size.

Heuristic algorithms are a common method of resolving this issue. While the most optimal solution is can only be found with an algorithm that runs in exponential time, many algorithms are capable of finding a sufficiently close to optimal solution in polynomial time. We present one such. For the remainder of this paper, we use the word "city" to refer to nodes on the graph, and "tour" to refer to a Hamiltonian path through those nodes.

Random Tour Algorithm:

This algorithm randomly generates an ordering of the cities on the graph, and attempts to traverse them in that order. If it succeeds, it returns that ordering of cities and the total cost of the traversal.

The random tour algorithm has a time complexity of approximately O(n), where we take *n* to be the number of cities. It randomly selects an ordering of nodes, which can be done in O(n) time, and it attempts to travel from each node to the next, which is a constant-time operation that is executed n times. Theoretically, this is only a lower bound, because it may have to generate several lists before it finds a valid tour, but unless the graph is quite sparse, or the number of nodes is large, the number of iterations necessary is negligible.

Unlike all of the other algorithms discussed, the random tour algorithm makes no attempt to optimize its tour. It only verifies that the tour it is providing is valid and visits all of the cities. While this means that it is of minimal use in most cases, we include it precisely because it provides an entirely un-optimized path. The path the random tour algorithm provides is useful as a method of empirically evaluating how effective other algorithms are at optimizing the length of their tour.

Greedy Algorithm:

The greedy algorithm starts from the first city on the graph. It then checks for the shortest available route to a city it hasn't already visited, moves to that city, and repeats the process until it has visited all cities.

The greedy algorithm must iterate through each node of the graph. At each node, it must examine all edges leading from that node to determine which of them is the shortest. The algorithm will traverse each node, so it will check all edges of the graph twice: once from each end. As such, the algorithm has a running time of O(e), where *e* is the number of edges. In a highly dense graph, *e* is approximately n^2, so we may approximate the running time of the greedy algorithm to O(n^2) in such cases.

The greedy algorithm is a fairly naïve implementation of a solution to the traveling salesman problem, as it rapidly provides a solution that is not very optimized. Similarly to the random tour algorithm, it provides a useful benchmark for comparison to more sophisticated algorithms.

Branch and Bound:

The branch and bound algorithm uses a tree of states, each of which represents a partial path through the cities. Each state contains a reduced-cost matrix, which is updated and reduced to create the child nodes of that state. The algorithm generates and traverses this tree using a priority queue of states, each of which has a key equal to the cost of reaching that state. If the algorithm finds a solution, it prunes all states in the tree (or queue) whose cost is higher than the cost of that solution.

The branch and bound algorithm's worst case time and space complexity arise from the nature of its queue. If the algorithm is forced to enumerate and evaluate each possible state on the tree, and does not prune at all, it will create O(n!) search states, each of which will require O(n^2) time to create. The worst-case scenario for time complexity is also O($n!n^2$), but the actual-case execution is generally better.

Notably, the branch and bound algorithm is guaranteed to find the optimal solution eventually. However, due to its exponential run time, it is often better to halt the algorithm early and take whatever intermediate solution it finds. The branch and bound algorithm is useful because it can be used as a Monte Carlo or Las Vegas algorithm, depending on the needs of the user.

Divide and Conquer Algorithm:

The divide-and-conquer algorithm we implement takes after Ishikawa et al. 2010[4]. In simplified form, it recursively performs the following actions:

1. The recursive function accepts a list of cities. If the list is of size $n = 1$, the algorithm halts and returns that list as a path.
2. If the list is larger than $n = 1$, the algorithm splits that list of cities in two, then recursively calls itself on each sub-list it generates.
3. The algorithm receives from the recursive call a path through each sub-list.
4. The algorithm then uses 2-OPT to find the most optimal merge of the two paths that it has been passed.

2-OPT checks each checks possible combinations of nodes (a_i, b_j), where a_i is a node in the first list, and b_j is a node in the second list. It checks which of all paths:

$$a_1, \dots a_i, b_{j+1} \dots b_n, b_1, \dots b_j, a_{i+1}, \dots a_n$$

has the lowest weight. Since each list of size n/2, and all combinations must be checked, $n^2/4$ calculations are performed, which comes out to an $O(n^2)$ complexity in total.

Since the algorithm is a divide and conquer algorithm, the master theorem provides its overall time complexity:

- the problem is divided into two subproblems, so $a = 2$
- each subproblem is of size $n / 2$, so $b = 2$
- merging each subproblem takes place in $O(n^2)$ time, so $d = 2$

Since $d > \log_b a$, the time complexity of the algorithm is $T(n) = O(n^d)$, which is $O(n^2)$.

The divide-and-conquer algorithm is interesting because it recursively builds local path optimization into global optimization. By doing so, it guarantees a result in polynomial time while still optimizing at multiple scales.

Advantages:

The divide and conquer algorithm's time complexity is a major advantage. The traveling salesman is NP-complete, and so no polynomial-time algorithm exists for it. In fact, since it is widely believed that $P \neq NP$[3], it seems probable at present that no polynomial-time algorithm *can* be developed for the traveling salesman problem. Since the divide and conquer problem runs in quadratic time, it is much quicker to find approximations than others.

In addition, the divide and conquer algorithm has an important advantage over the branch and bound algorithm that merits some discussion. The divide and conquer algorithm always returns a result at the end of its $O(n^2)$ running time. On the other hand, the branch and bound

algorithm is not guaranteed to find any solution, even a partially optimal one, until very nearly the end of its run time. In the worst-case scenario, the branch and bound algorithm visits every node on the tree except those on the lowermost level, preventing it from pruning any nodes and forcing it to visit each state on the tree.

Disadvantages:

The most obvious disadvantage of the divide and conquer algorithm is that it is a heuristic algorithm. While it runs in quadratic time, which is highly useful given that the problem it is approximating has so far only proven solvable in exponential time, the necessary drawback is that it can only provide a solution that has been optimized, not one that is globally optimal.

Additionally, while the growth of a quadratic-time algorithm is greatly superior to the growth of an exponential-time algorithm, it is still possible for such a problem to quickly move from practical at smaller sample sizes to impractical at larger sample sizes.

Finally, the divide and conquer algorithm produces no intermediate solutions. Unlike the branch-and-bound algorithm, which can provide a partially optimized path if halted before completion, the divide and conquer algorithm must execute fully before it will provide any path through all cities.

Methods:

The algorithm generates a "scenario", which is a data structure that contains a list of nodes, each with a set distance to other nodes. This distance is not necessarily Euclidean. Each node connects to most other nodes, but it is not guaranteed to connect to all of them.

In our results below, we include the average running time and total solution weight of each algorithm for certain numbers of cities. For the greedy algorithm, we include the improvement of the greedy algorithm over the random tour algorithm. This number represents the ratio between the greedy algorithm and the random tour algorithm. A lower percentage represents a greater improvement by the greedy algorithm over the random tour.

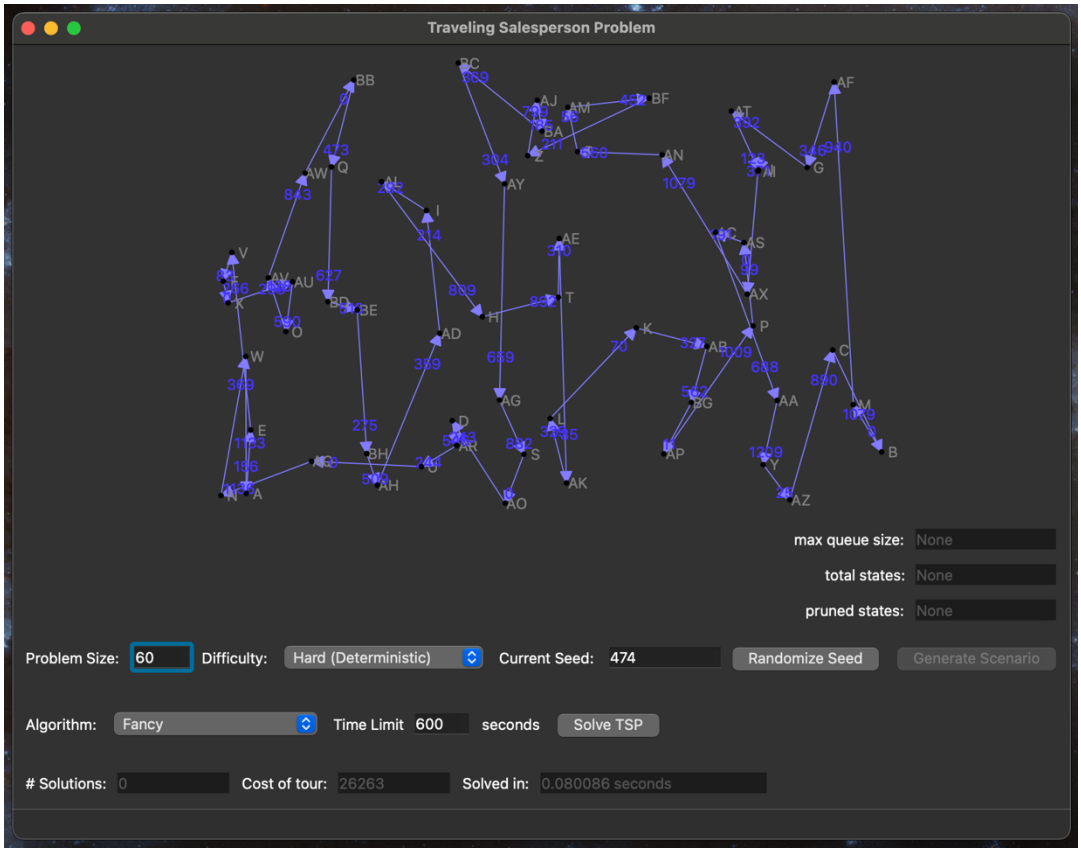
Likewise, the branch and bound algorithm and the divide and conquer algorithm include their improvement over the greedy algorithm, where applicable. This represents what percent of the greedy algorithm's cost each other algorithm's cost fills. Again, a lower percentage represents a greater improvement by those algorithms over the greedy algorithm. Percentages over 100% represent a point where the greedy algorithm's cost is on average lower than the other algorithm's cost.

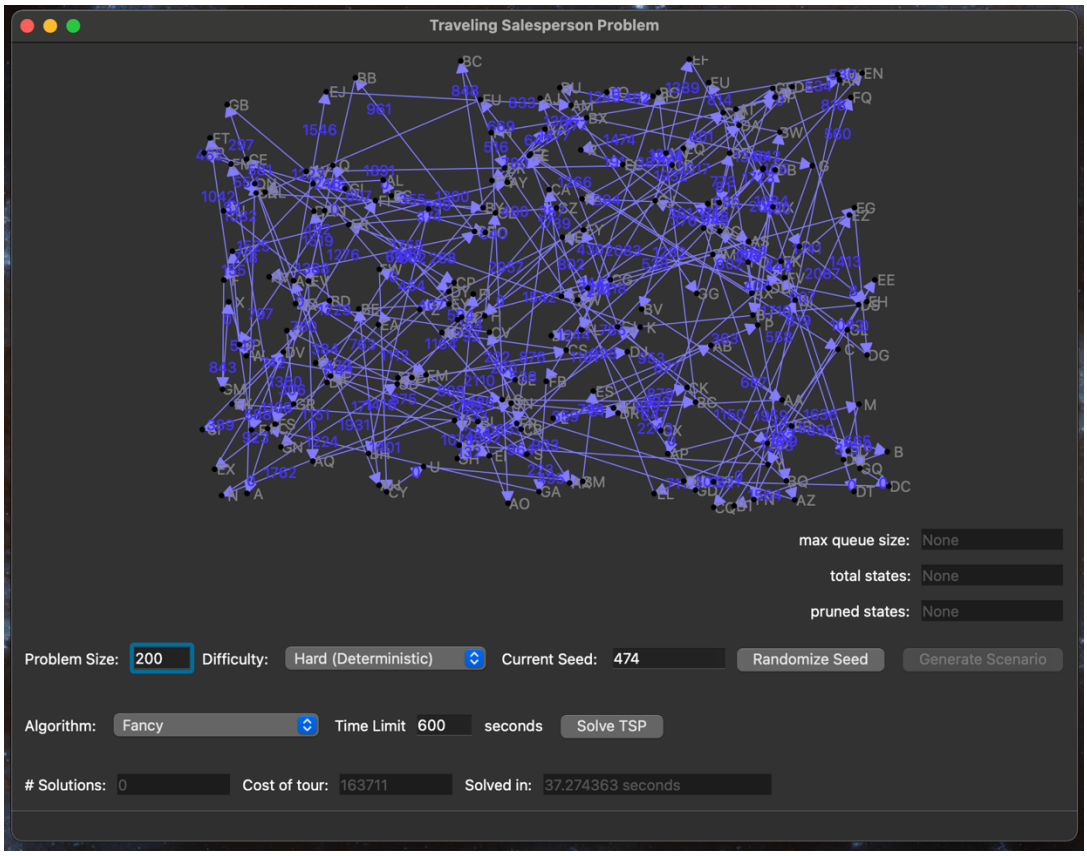
Results:

	Cities:	15	25	30	40
Random	Time:	0.0013	.025	0.042	0.34
	Weight:	19627	32667	39173	56316
Greedy	Time:	0.0017	.0022	0.0037	0.0055
	Weight:	11900	16690	18780	22154
	Improvement:	60.63 %	51.09 %	47.84 %	39.34%
Branch and Bound	Time:	2.68	TB	TB	TB
	Weight:	9744	TB	TB	TB
	Improvement:	81.88 %	TB	TB	TB
Divide and Conquer	Time:	0.0035	.012	0.020	0.038
	Weight:	11746	15178	16723	25301
	Improvement:	98.71 %	90.94 %	89.05 %	112.38 %

	Cities:	60	100	200	400
Random	Time:	55.94	TB	TB	TB
	Weight:	76031	TB	TB	TB
Greedy	Time:	0.0081	.020	.054	.19
	Weight:	27620	38815	59007	92335
	Improvement:	39.32 %	TB	TB	TB
Branch and Bound	Time:	TB	TB	TB	TB
	Weight:	TB	TB	TB	TB
	Improvement:	TB	TB	TB	TB
Divide and Conquer	Time:	0.15	1.41	38.61	TB
	Weight:	38427	73577	166876	TB
	Improvement:	139.13 %	189.56 %	282.81 %	TB

Screenshots:


Traveling Salesperson Problem
max queue size: None
total states: None
pruned states: None
Problem Size: 60
Difficulty: Hard (Deterministic)
Current Seed: 474
Randomize Seed
Generate Scenario
Algorithm: Fancy
Time Limit 600 seconds
Solve TSP
Solutions: 0
Cost of tour: 26263
Solved in: 0.080086 seconds


Traveling Salesperson Problem
max queue size: None
total states: None
pruned states: None
Problem Size: 200
Difficulty: Hard (Deterministic)
Current Seed: 474
Randomize Seed
Generate Scenario
Algorithm: Fancy
Time Limit 600 seconds
Solve TSP
Solutions: 0
Cost of tour: 163711
Solved in: 37.274363 seconds

Discussion:

The random tour algorithm mostly performed as would be expected, except at higher numbers of cities. This is because, as mentioned in the algorithm analysis and methods sections, the algorithm starts by generating a random list of cities, not a path. Since not all cities are guaranteed to have a defined distance to each other, as the size of *n* increases, it becomes increasingly likely that there is a point *i* in the list a_1, ... a_i, a_{i+1}, ... a_n where there is no path from a_i to a_{i+1}. As such, it actually becomes exponentially more difficult for the algorithm to find a valid path.

However, if we initialize the scenario such each city is a point on a Euclidean plane, and therefore ensure that there is a defined distance between each city, we observe that the random tour algorithm does perform efficiently:

	Cities:	15	30	60	100	200
Random	Time:	.00011	.00017	.00024	.00036	.00040
	Weight:	15566	43181	73526	139710	276201

The greedy algorithm has a similar issue: it is possible that the algorithm reaches a "dead end" where all paths from the present node lead to nodes that are already visited. However, because the graph is dense, this only occurs when most of the graph has already been traversed, so this is a relatively uncommon issue. It is noteworthy, however, when comparing the greedy algorithm to others, that this does influence the algorithm's results– our data only includes those situations where the greedy algorithm succeeds, which artificially deflates its actual running time.

The branch and bound algorithm's performance was about as expected: due to its exponential time complexity its average time until completion soon became impractically long. It usually returned intermediate results, but since we are comparing each algorithm's final result to each other, those are not included. Of note is that at $n = 200$, branch and bound failed to return even a sub-optimal intermediate path.

Our algorithm outperforms the greedy algorithm on the range $n = 15$ to $n = 30$. Unlike branch and bound, our algorithm returns a path quickly on values $n > 20$. It is possible for branch and bound may also return an sub-optimal path in a similar timeframe, but it is not guaranteed to. This suggests a niche for our algorithm on data sets of $n = 20$ to $n = 30$. Further refinement of the algorithm's implementation could improve this range.

References:

1. Karp, R.M. (1972). Reducibility among Combinatorial Problems. In: Miller, R.E., Thatcher, J.W., Bohlinger, J.D. (eds) Complexity of Computer Computations. *The IBM Research Symposia Series.* Springer, Boston, MA. https://doi.org/10.1007/978-1-4684-2001-2_9
2. Dantzig, G., Fulkerson, R., & Johnson, S. (1954). Solution of a Large-Scale Traveling-Salesman Problem. *Journal of the Operations Research Society of America*, *2*(4), 393–410. http://www.jstor.org/stable/166695
3. Gasarch, W. I. (2019). Guest Column: The Third P =? NP Poll. *SIGACT News Complexity Theory, Column 100*. SIGACT News, University of Rochester, Rochester, NY.
4. Ishikawa, K., Ikuo, S., Masahito, Y., Masashi, F. (2010). Solving for large-scale travelling salesman problem with divide-and-conquer strategy. *SCIS & ISIS 2010*.