Rough draft of tsp for group

Report: There is no graded design experience for this project, but your group should do the design experience together before starting to code.

1. [10] Correct implementation of the greedy TSP algorithm. Brief discussion and complexity of the algorithm.

Greedy time complexity - O(nlogn)

The greedy algorithm works by always making a choice that has the most immediate benefits. For our greedy TSP algorithm the route always adds on the neighboring city that has the lowest cost. If this route however does not lead to a complete route to every city the route is cleared and a new starting city is chosen. This method will continue until it finds a complete path. Our greedy TSP algorithm has an average time complexity of O(nlogn).

2. [35] Correct implementation of your own TSP algorithm including a discussion of the algorithm, why you chose it, and its pros and cons. Give proper attribution (references) for ideas you find externally. Clearly point out which ideas came from other sources, and which ideas are original contributions that you made. Include a discussion of the theoretical big-O complexity of your algorithm. Discuss how the empirical complexity matched your theoretical complexity. Include screenshot examples of typical results for your algorithm.

2-opt complexity = O(n^2)

3-opt complexity = O(n^3)

For our first fancy algorithm we chose a 2-opt algorithm. This 2-opt algorithms theoretical complexity is O(n^2), with our implementation being the same. Our algorithm swaps two city pathways at a time to check whether the swap improved total cost. If total cost was improved the route is changed to the new improved route. This continues in a double for loop until all path swaps have been tested. Once improvements no longer happen during swaps the while loop ends and the final route is returned.

3. [30] Include a table with columns (see below) for each of the TSP algorithms including the Branch and Bound TSP algorithm you implemented in Project 5 (use one of your individual project 5 implementations as a representative version of B&B; however, do not set your time limit at 60 seconds as you previously did). For the random algorithm, use the default algorithm provided with the original framework (this acts as a baseline for the greedy algorithm). You can play with all three levels of problem difficulty (Easy, Normal, Hard) during testing, but just use the Hard level for all of your reporting. For the greedy algorithm report the improvement over random as a fraction [calculate this as greedy\_cost/random\_cost]. This will help calibrate and make sure your greedy algorithm is implemented correctly. For B&B and your own algorithm, report the improvement over the greedy algorithm [calculate this as your\_cost/greedy\_cost]. You should create a table just like the one below with the city sizes shown below (15, 30, 60, 100, 200). Round average tour lengths to the nearest integer. Round time and % improvement to two significant digits beyond the decimal. The results for each cell (all 4 algorithms) should be the average of 5 runs with different random seeds for that number of cities. Do not try to find a particular set of seeds on which your implementation does well. They should be randomly chosen runs. You will fill in average time (seconds) and average tour length for the different numbers of cities. If an algorithm takes more than “reasonable time” (more than about 10 minutes or so) to solve for that number of cities for the majority of your trials, then just fill in that cell for that algorithm with “TB” (Too Big). For B&B and your algorithm, if a minority of the trials do not finish, take the average of the majority that does. The numbers in the example table below are made up. (Note: We give random and greedy algorithms an unfair advantage in our comparison as we allow them an arbitrary number of restarts when they fail in the hard version, which we do not grant to B&B and your algorithm. This points out another way in which they are inferior to your algorithm and B&B.

Random Greedy Branch & Bound Our Algorithm # Cities Time (sec) Path Length Time (sec) Path Length % of Random Time (sec) Path Length % of Greedy Time (sec) Path Length % of Greedy 15 0.001 7743 .004 4100 .52 3.26 3527 .86 1.47 3649 .89 30 60 100 200 .11 100761 .3 15798 .16 TB TB TB 165.1 14244 .90 …

TABLE

4. [15] To complement the required five rows shown in the table above, you should also add to your table some additional rows (different numbers of cities). In particular go as large as you can before getting “Too Big.” The last row of your table should be the largest number of cities which can be done in a reasonable time limit. Discuss and analyze the results in the table. In particular, make sure you have a few rows in the "sweet spot" for your algorithm, which is around the number of cities where it shows the best balance of lots of cities with good costs. Discuss and analyze the overall results from your table.

5. [10] Your write-up should look sharp and follow the form of a brief conference paper with abstract, introduction, algorithm explanation, complexity, analysis of results, and future work.

The Traveling Salesman Problem:

Adaptation of 2-Opt Local Optimization in Comparison to Branch & Bound Techniques

Adara Christensen

Ammon Christensen

Matt Christensen

April 14, 2021

Professor Tony Martinez

BYU, Computer Science

Abstract:

The 2-opt and 3-opt algorithms are heuristic methods that involve local search algorithms to solve the traveling salesperson problem. Our algorithms are a combination of both 2-opt and 3-opt local search problems. This paper will discuss and compare the benefits of 2-opt and 3-opt algorithms in comparison to performance on branch and bound and greedy tsp algorithms as well as an analysis of empirical data from both algorithms.

Introduction:

The traveling salesperson problem (TSP) was initially defined in the early 1800s by two men W.R. Hamiliton and Thomas Kirkman, solving puzzles based on the Hamiltonian cycle. Then the concept was transformed in the 1920s by Karl Menger. He called it the “messenger problem” (Home). Discovering the method of merely traveling to the closest city from one's current location did not result in the lowest travel distance. This traveling salesperson problem has since then received much attention from mathematicians and computer scientists alike.

The TSP algorithm is defined as NP-Hard. This means that the complexity will always increase as the number of destinations is increased and there is no “quick” solution (Ma, S.).

Due to the difficulty in solving a complete solution when considering both time efficiency and lowest/shortest cost there are many algorithms that have been created.

Some of the algorithms that have been discovered and that will be discussed and analyzed within this paper are the Greedy Algorithm, Branch and Bound Algorithm, 2-Opt Algorithm, and 3-Opt Algorithm. The greedy algorithm is one of the most simple in objective. This algorithm finds the shortest path always and travels there. However as discovered by Karl Menger this solution is not always the most optimal. The branch and Bound algorithm works to find the most optimal solution through traveling down branch networks and testing all the possible solutions. This algorithm may find the most optimal path but the time complexity to receive this solution is slow. Then there are the 2-Opt and 3-Opt algorithms which begin with a path and then work to improve it by swapping two to three paths at a time and replacing if the cost decreases. This method is faster than the Branch and Bound algorithm and also is able to improve upon the greedy algorithms lowest path.

Greedy Algorithm:

Complexity and implementation

The concept of a greedy algorithm is for the choice with the most immediate benefits to be used first. For our greedy TSP algorithm the route always adds on the neighboring city that has the lowest cost. Our algorithm begins with a random starting city and follows the pattern of continually choosing the lowest path. If the final route does not connect to every city the route is cleared and a new starting city is chosen. This method will continue until it finds a complete path.

Our greedy TSP algorithm has an average time complexity of O(n^2). This algorithm's main big-O effectors are a for loop that goes through all cities and finds the lowest cost neighboring city with a complexity time of O(n) . This city is then added to an array in O(1). If the length of the array matches the number of cities to be traveled to, the function exits the while loop and returns a solution. Otherwise it returns to the beginning of the while. This while loop since it is often traveled to at least n times runs in O(n) time. Making the total running time for the greedy algorithm O(n^2). The space complexity is simply O(n), because we never store more than 3 copies of the route or given solution. Thus O(n) + O(n) + O(n) = O(3n) = O(n).

*Source code for this greedy algorithm implementation will be provided in the file attached to this paper.*

Branch and Bound Algorithm:

Complexity and implementation

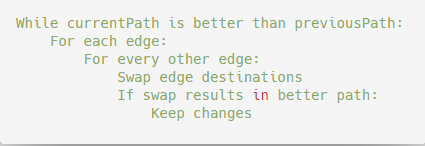
The Branch and Bound Algorithm implementation allows for a better solution than that of the greedy algorithm. This method finds the optimal solution by traveling through the many possible paths that can be taken. Path options that exceed the value of the initial BSSF are filtered out and pruned off the list of possible solutions. Due to the checking of many potential solutions the time complexity of Branch and Bound is rather high at O(b^n (n+1)!). This big-O time complexity is a worst case scenario and will not reach this speed very often. However due to the slower speed, cut-off times are often provided to limit the time taken to receive a solution. Space complexity is a similar story, worst case scenario compounds to O(!n), while the average case tends to be closer to O(p \* n^2). While the cutoff prevents total optimal solutions it provides for a better solution than that of the greedy algorithm.

.

2-opt and 3-opt Algorithms:

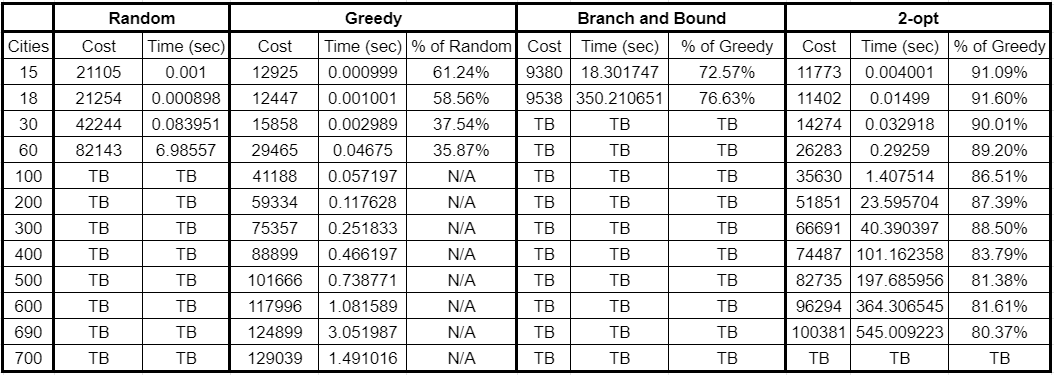
Complexity and implementation

The 2-opt Algorithm is implemented by taking a given valid route and looking to improve it by comparing edge costs when two edges are interchanged. Simply put, we disconnect the path at a given point and see if by switching the order of two nodes and reconnecting the path to see if we can find better improvement.

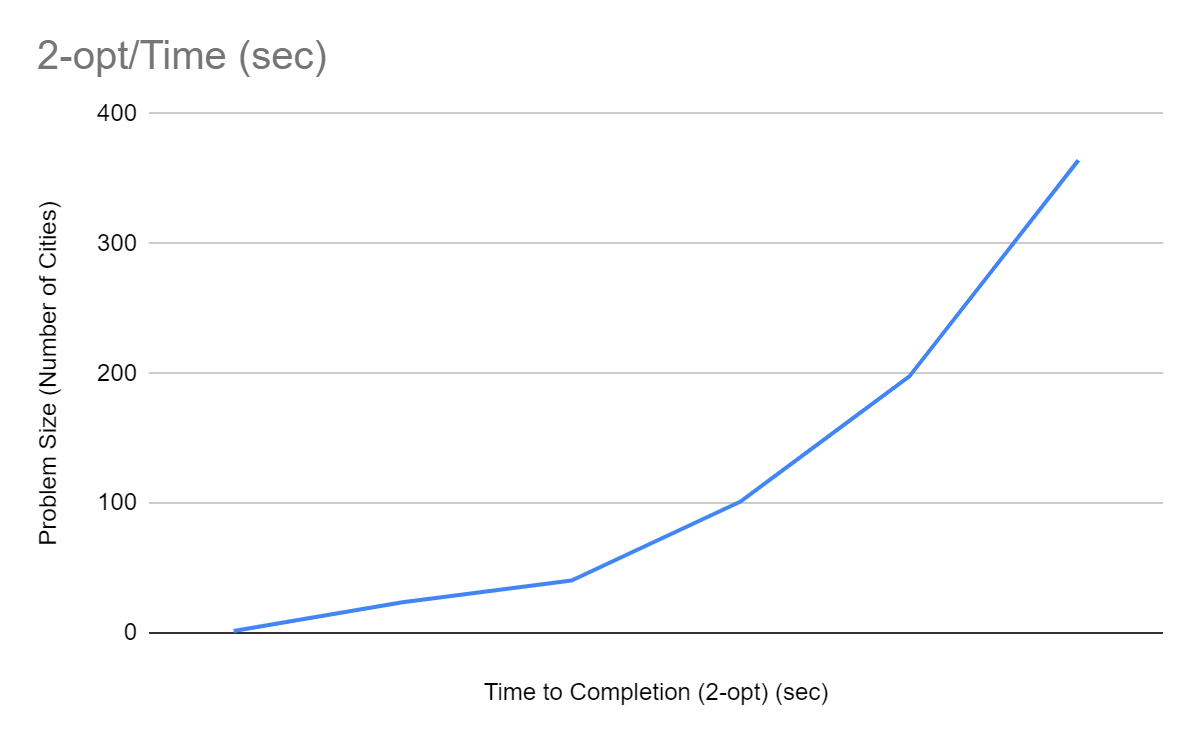


This provides a relatively simple way to improve a given solution in polynomial time. The time complexity of 2-opt is O(n^2). This is because the algorithm must compare every edge in the path to every other edge in the path, and may do this multiple times if improvements continue to be found. The 3-opt Algorithm does the same thing, but switches all combinations of 3 edges and runs in O(n^3) time complexity. Additionally, both algorithms have a space complexity of O(n) because only 3 solutions/routes are ever stored - O(n) + O(n) + O(n) = O(3n) = O(n).

In theory, the 3-opt version should result in a better path, since more total path combinations are considered. Our empirical results showed that when the starting path is derived from a greedy algorithm, there is rarely a difference between the 2-opt and the 3-opt algorithm. However, when the starting route is chosen at random, there is a significant improvement when using 3-opt, at the cost of greater time complexity. In all cases, the empirical time complexity was as expected, with variability in the improvement of the path travelled. When compared to the branch and bound algorithm, this provides a less optimal solution, but finishes in a much faster time. Our empirical analysis also found that as the city count increased, the improvements upon Greedy with the 2-opt algorithm also increased. (See table below)



The efficiency of the 2-opt algorithm, compared to the cost of the greedy results (2-opt cost/greedy cost), increases because of the likelihood that an improvement in an early iteration will make future changes and iterations possible. This means that our rate of improvement only increased with the size of the problem, as the gains of future iterations compound with each improvement. This can be seen in the table of empirical analysis figures as seen above. We start to see steady improvement as the problem size grows past 100 cities. However, because early changes create more possibilities for future iterations, which also create improvements that enable more iterations and thus start a positive feedback loop. This causes the time to completion to increase also.



Graphing the time to completion for 2-opt, we can see that because of the positive feedback loop, the time scales on an exponential level. Thus, the sweet spot for this algorithm is for relatively large scale problems (about 100-1000 cities for consumer grade hardware) where the 2-opt algorithm has a larger search base to find improvements to the cost of each tour significantly while also maintaining a manageable time to completion.

It should also be noted that the results for each cell (all 4 algorithms) are the average of 5 district runs, each with different random seeds for that number of cities.

In conclusion, the 2-opt algorithm offers a comparable solution to a greedy approach to the Traveling Salesperson problem for small scale problems, both in time/space complexity and time to completion, with around a ~10% improvement for problems below a size of 10. However with larger problems a bigger improvement can be seen while also dramatically increasing the time to completion. Thus, the cons of the 2-opt algorithm are little improvement over a greedy approach for small problem sizes (less than 100 cities) in addition to not being able to handle incredibly large problem sizes of 1000+ cities due to time complexity. However, the major pro to this approach is being able to compute a significant and important range of problems (100-1000 cities) in a very reasonable time (less than 10 minutes) with significant improvements over a greedy approach (up to 20% improvement of cost).

Future Work:

In terms of future work, we also investigated the possibility of using a 3-opt algorithm approach in addition to 2-opt. With the 3-opt algorithm, we investigate a larger search space, at the cost of having a larger time complexity of O(n^3). This is because when we want to reconnect three different edges, there is more than one way to reconnect them. As we disconnect 3 edges from the route there are 7 possible new ways to reconnect the route (and we try all of them to find the most optimal path). As the number of edges increases, the number of possibilities increases as well. Thus, a k-opt algorithm will always have a time complexity of O(n^k). We anticipate that this may lead to lower cost paths, but this is not guaranteed.

Improvements to be found may be very small or nonexistent, and as the complexity increases, we should expect to approach the ideal path. This is especially prominent when using a greedy solution for the initial route. In our initial empirical testing, we did not find any significant difference between 2-opt and 3-opt when the size of the problem was below 50 cities. This leads us to believe that in order to find significant differences, we would need to run tests on increasingly larger problem sizes. This might result in testing taking more than a reasonable amount of time on consumer grade hardware. While this may produce a better solution, the time and computation required for such marginal improvements may not be reasonable. However, while using a random solution for the initial route we saw noteworthy improvements between 2-opt and 3-opt. This is because of how optimized the cost of routes are for greedy solutions, which lead to little room for improvement between 2- and 3-opt approaches. However, this is not scalable, because on an asymmetric problem with some infinite distances the random algorithm is unable to find a solution on problem sizes over 100 cities.

Conclusion:

The traveling salesperson problem has many potential solutions depending on the need for optimization and time efficiency. Due to our focus on an optimal time solution with improved cost path for large scale problems (100-1000), we found the 2-opt algorithm was the most beneficial. This algorithm provided significant improvements from greedy, with only modest increases in time complexity, both theoretically and empirically. Furthermore, the space complexity remains the same, and the 2-opt algorithm is handling problem sizes well into the hundreds. Our algorithm can also be used well beyond that, provided it is given enough time to find a solution. If a more optimal solution is required, more sophisticated methods may be used, but our solution is relatively low-cost both in time and space complexity, and provides a solution that is very near optimum.

References:

Home. (n.d.). Retrieved from <https://www.theorsociety.com/about-or/or-methods/heuristics/a-brief-history-of-the-travelling-salesman-problem/>

Ma, S. (n.d.). Understanding The Travelling Salesman Problem (TSP). Retrieved from https://blog.routific.com/travelling-salesman-problem#:~:text=The Travelling Salesman Problem

GitHub repository link:

<https://github.com/mrchristensen/TravelingSalesperson>

Screenshots:

