The Algorithm

Local Search Popularity with 2-Opt

Presented as an efficient alternative to finding acceptable solutions to

The Travelling Salesman Problem

Submitted to

Dr. Tony Martinez

for

CS 312: Algorithm Analysis

Brigham Young University

Provo, Utah

December 11, 2014

By

Tommy Jensen, Sean Lane, and Epper Marshall

# Abstract

The Local Search Popularity with 2-Opt algorithm (LSP2Opt) is a hybrid of algorithm methodologies attempting to combine efficiency and accuracy in solving TSP. In this paper, we will analyze the paradigm behind Popularity, how a Local Search modification makes it reliably productive, and how the 2-Opt heuristic (not created by the authors) helps to make answers more optimal.

# I Introduction

This paper introduces an algorithm that was originally created by the authors while incorporating work done by others. An explanation of the Nearest Neighbor algorithm (known in class as “Greedy”) will set the stage for the Popularity algorithm’s paradigm as a greedy algorithm. Then, Popularity’s polynomial efficiency and its unique solution-finding efficiency when used with the Local Search algorithm design will be discussed. Additionally, the authors’ caching implementation of 2-opt will be shown to have great running time improvements over naively-implemented 2-opt without sacrificing any of the gains made by the brute force approach. Finally, the empirical analysis of LSP2Opt’s performance against Greedy and Branch and Bound will conclude this paper.

# II Greedy (Nearest Neighbor)

The Greedy algorithm (often known as “Nearest Neighbor”) is a commonly used approximate solver for TSP because it is easy to implement and gets an answer very quickly. However, it has the drawback that it can get stuck in local minima and never find an even close to optimal solution. In fact, city layouts exist that Greedy will find uniquely worst possible solutions for! Nevertheless, the authors found it useful as a source of inspiration.

### Implementation

The authors implemented Greedy according to its well-known pseudocode:

1. Pick a random starting city: O(1)
2. Find the shortest edge that goes to an unvisited city, and travel it: O(n)
3. Repeat Step 2 until the original starting city is reached, unless no routes are available, then continue: O(n)
4. If a dead-end was found, start over at Step 1 with a different start city (rare to occur even once, let alone in terms of the size of the input; not factored into big-O bound as it is a small constant)

Using this pseudocode, Greedy becomes an excellent base-line for comparison for other TSP solving algorithms. Later empirical analysis will show it has a predicable path-length improvement over the random solution (picking edges at random until a circuit is made) and it can run in less than a second for more than 1,000 cities at a time. With Step 3 running n times for each repeat of Step 2, this implementation of Greedy is O(n2) in run-time bound.

### Greedy Takeaway

The key to the Greedy algorithm is that it always takes the shortest available path to an unvisited city. This is what allows it to run so quickly, as it only requires O(n2) time to find a solution (n for each city, and n for checking each edge per city to travel down). However, this head-first approach blindly misses possible solutions that require taking a longer edge first.

# III Popularity

The Popularity algorithm is an original creation of the authors’ based on the Greedy paradigm. Like Greedy, it takes the best edge available to it from a city. However, it scores “best” by popularity, which is defined as the combination of an edge’s length and the average length of “short” edges leaving the destination city. This allows it to choose not just the shortest available edge, but a short edge that also leads to more known short edges.

### How Edge Popularity is Scored

An edge’s popularity score is comprised of two pieces of information: the edge’s length, and the average length of the short edges leaving the edge’s destination city. While the edge’s length is easy enough to find, the second piece requires more explanation.

First, for each city, the length of all outbound edges is averaged. Then, each outbound edge shorter than that average is averaged again. This average length of the shorter-than-average outbound edges serves as the second piece of information in calculating the popularity score of an edge leading to that city.

After all cities have had their short edge’s average length calculated, all edges get a popularity score by combining these two pieces of information as follows:

Above, *e* represents the edge receiving a score, *cityAverage* represents the calculated average length of the shorter-than-average outbound edges from *e*’s destination city, and *f* is a floating-point number used to weight the result.

### How Popularity Weight Factor is Calculated

The popularity weight factor is the key to allowing Popularity to achieve better than Greedy path solutions to TSP. Note that factors with great absolute values will result in each edge being effectively scored only by its length, and then the results would be no different than Greedy. Factors with very low absolute values, on the other hand, result in edges being effectively scored only by the length of edges leaving the city it goes to. Empirical testing showed that this yields less optimal solutions than Greedy itself, which makes sense because the algorithm can choose to travel to cities with short outbound edges, but then never take those edges!

Therefore, a balance between the two numbers is required to acquire ideal solutions. Through empirical analysis of several sets of data, the authors determined that the solutions produced by using various factors is not evenly distributed, but tends to produce local minima. A recurring location of a local minimum relative to the path length produced by Greedy was discovered as follows:

Above, *f* is the popularity scoring weight factor, and *LG* is the length of the Greedy algorithm’s solution to the same set of cities. While the authors cannot prove at this time why local minima occur around this factor value, they theorize that it is related to the limited size of the provided framework’s city-spawning arena (in other words, that this factor has more to do with the cities’ density than their cardinality).

Regardless of proof, empirical analysis of data in the authors’ “work environment” showed that this does regularly produce better results than Greedy, though often the ideal factor for this local minimum would be just slightly lower or higher than the calculated one, causing less than desirable results. For this reason, a better way to find good factors for Popularity to use is required for reliable results. This will be discussed in Section III.

### Popularity’s Run-Time Bounds

Popularity, like Greedy, is a polynomial time algorithm which allows it to return answers quickly for large sets of cities. A breakdown of its non-trivial steps follows:

1. **Calculate the length of Greedy solution: O(n2)** Greedy was shown to be O(n2) in the previous section.
2. **Calculate the average length of “short” edges for each city: O(n2)** This requires a loop through each city (n) and for each city, looping through each of its edges to find the average length (n) and again through to find the average length of just the shorter-than-average edges (n). O(2n2) reduces to O(n2).
3. **Calculate the popularity score for each edge: O(n2)** Iterates through each edge (worst case n2) and calculates its score by its length and the destination city’s score from Step 1.
4. **Build path for TSP: O(n2)** This starts at a chosen city, then goes down the most popular edge that leads to an unvisited city. Like Greedy itself, this is O(n2) because it iterates once for each city, and for each city iterates through its edges looking for unvisited destination cities. Restarts in the event of a dead-end.

Therefore, it can be seen that Popularity has a runtime bound of O(n2) like Greedy does. This makes it a very powerful option as a starting route to optimize because it can be done quickly. It also tends to give better routes than Greedy does (if it is given a good weight factor to use). The next section will address finding optimal factors to give Popularity so it can in turn give good solutions to TSP.

# IV Local Search in Popularity

Local Search algorithms are useful in solving complex problems such as TSP because they can efficiently find local best solutions. While there are some well-known applications for local search algorithms to TSP that will be discussed later, the authors applied it to Popularity to help it choose its popularity weight factor.

When Local Search Popularity (LSP) runs for the first time, it uses the function mentioned in Section III to calculate an estimated factor that lies at a local minimum for Popularity results. When Popularity finishes running, LSP compares it to the length of the path that Greedy found. Then, it adjusts the factor accordingly so that minimum-path producing factors are zoomed in on (to find the true local minimum they are next to) and that less-valuable results are avoided.

To address the local search algorithm problem of getting stuck on the first local minimum encountered and never finding other minimums, LSP starts drastically increases the rate at which the Popularity weight factor increases and decreases, always alternating the direction the values goes. This allows the factor to “jump” out of local minimum “holes” when they are deemed explored. Empirical analysis shows that this is usually effective, as LSP typically finds two to four significantly different factor values as producers of best solutions so far.

|  |  |  |  |
| --- | --- | --- | --- |
| # Cities | Greedy (Length) | LSP (Length) | Percent Improvement |
| 15 | 4,007 | 3,632 | 9.36% |
| 200 | 15,926 | 14,560 | 8.58% |
| 500 | 27,129 | 24,326 | 10.33% |

LSP simply runs Popularity until it thinks that it has found the best minimum it can. This is achieved in the authors’ implementation by using a “forgiveness” counter. Forgiveness is an integer initialized to a value (the authors used 50), then decremented each time Popularity doesn’t find a better solution. The forgiveness factor is reset to its starting value whenever a best path is found. This allows LSP to explore fairly thoroughly for local minima without taking exponential amounts of time doing so.

# V 2-Opt in LSP2Opt

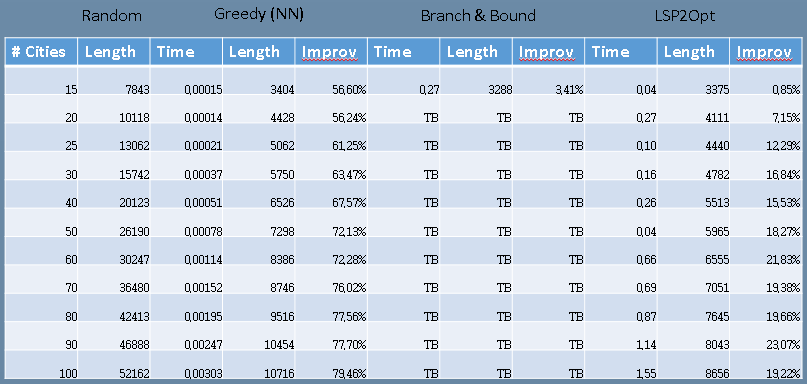
// stuff!!!!!

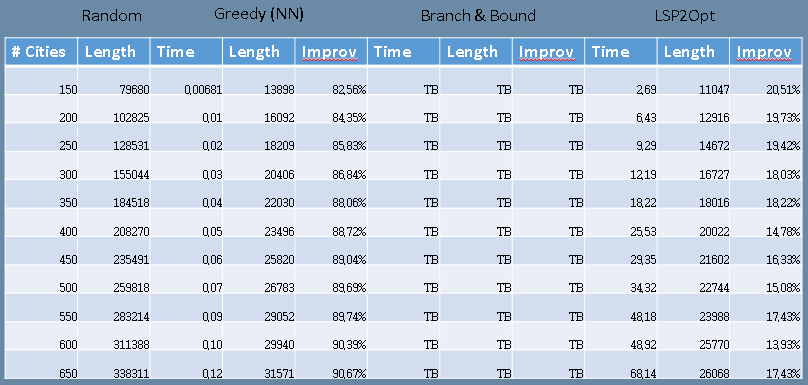
# VI Final Empirical Analysis

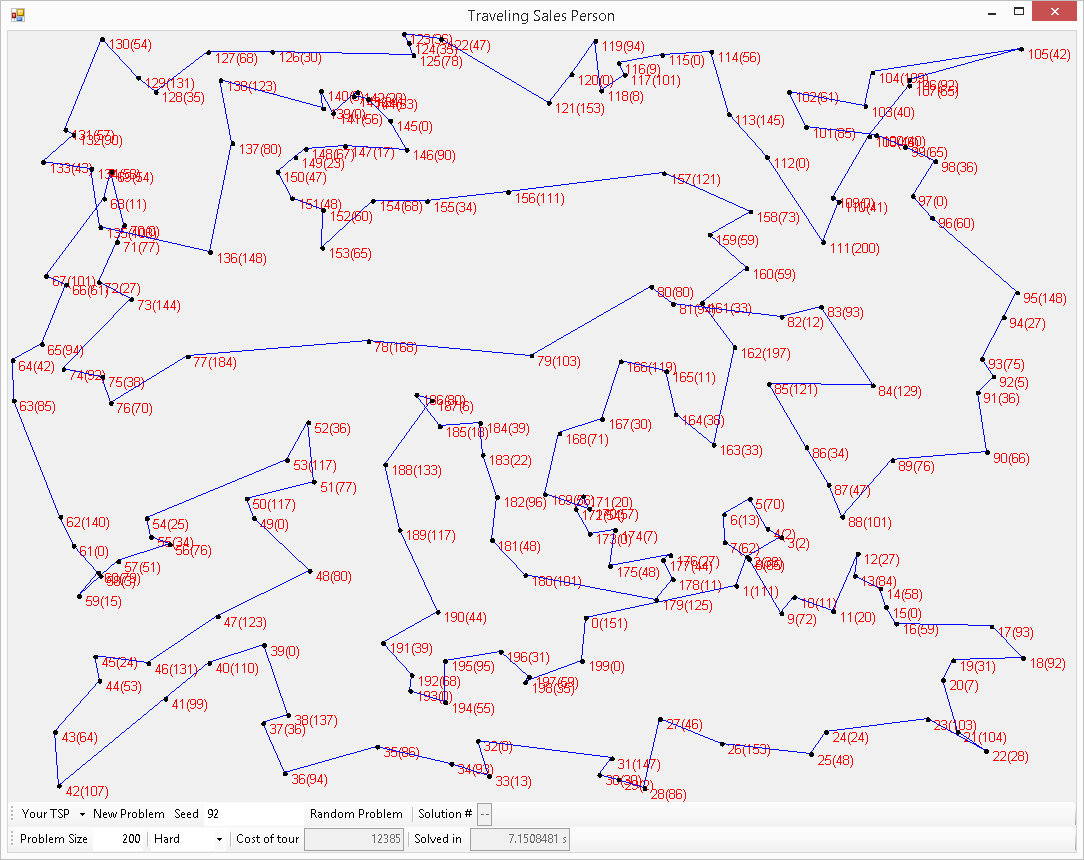
Now that the theory behind LSP2Opt has been fully discussed, a conclusion with the empirical results of its performance against Greedy and Branch and Bound follows. In the tables, the Greedy’s improvement was based off of Random’s performance, and B&B’s and LSP2Opt’s were based off of Greedy’s. Note that TB means that the algorithm took too long it give a response (used for B&B especially when no initial solution was ever found before time ran out).

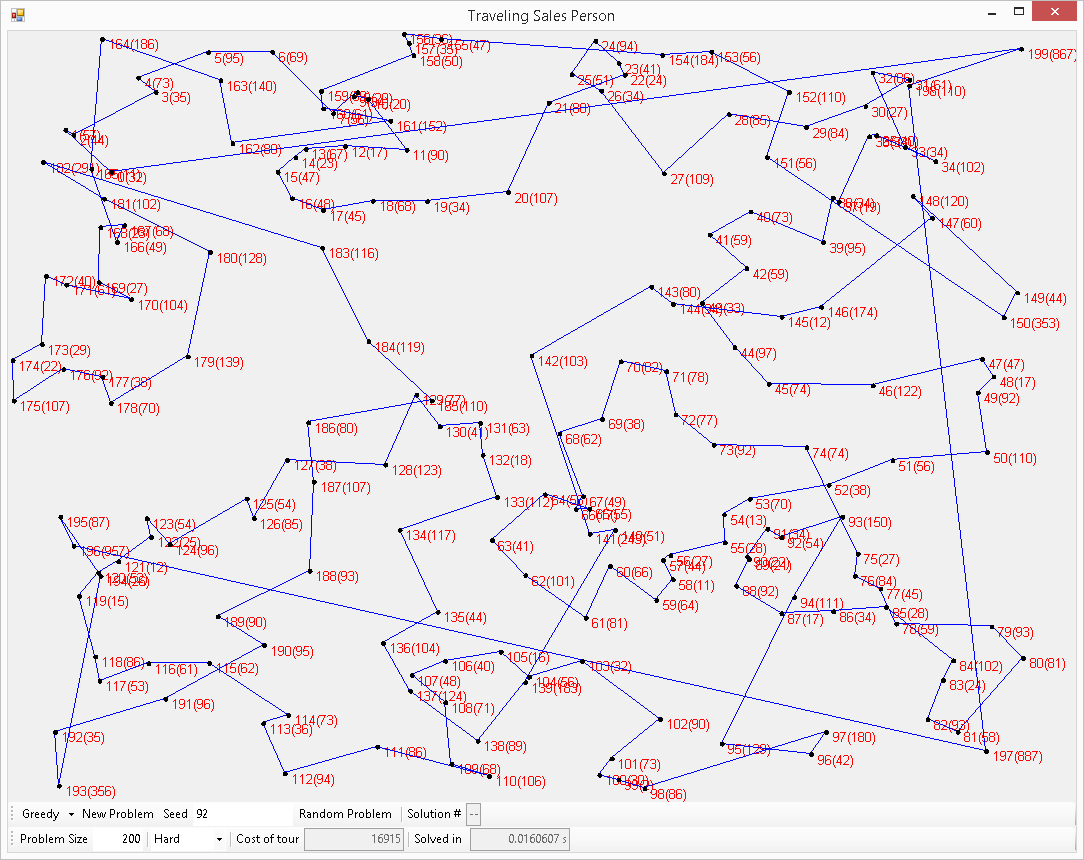
LSP2Opt uses a time-out feature centered around the 10 minute time limit imposed by the assignment. The LSP portion will only run for up to 200 seconds or until its first solution, whichever is longer, even if it still thinks it can find better minimum solutions to allow 2-Opt time to run. This was a design decision to maximize the number of cities the algorithm could handle in 10 minutes and still improve upon Greedy. LSP2Opt will return after 600 seconds with its BSSF whether or not LSP or 2-opt has finished, but they will both get a chance to run unless LSP takes the full 10 minutes to find its first solution. The authors were not able to find this ceiling, as LSP reached its 200 second limit before LSP2Opt in total reached its 600 second limit. This causes an apparent “increase” in LSP2Opt’s polynomial exponent around 1400 cities which is the result of the artificially imposed time limit on LSP. Without the artificial time limits, the authors estimate that LSP2Opt would reach an average run time of 600 seconds by about 1,750 cities.

Also interesting to note is that LSP2Opt performs far poorer on small sets (see 15, 20, and 25 city size test). This is because the very strengths of LSP prevent it from finding an acceptable solution in many small sets. In sets with few cities, the average of a city’s short outbound edges tends to be much larger, and has a negative effect on the edge selection, which leads to higher chances of finding a dead-end. Therefore, while the Pros of LSP2Opt include fast run times and better-than-Greedy answers, its Cons include being needlessly inefficient on small problem sets.









# VII Conclusion / Suggestions

The authors conclude that LSP2Opt is an ideal candidate for a TSP-solving algorithm for finding better than greedy solutions in small amounts of time for large city sets. It runs in the same time complexity class as Greedy, allowing it to find solutions to problems with thousands of cities, while also improving consistently on Greedy’s solutions.

While the authors are satisfied with their work, they pose open ideas concerning improvement upon LSP2Opt for the reader and the community at large. The behavior of the distribution of path lengths discoverable by Popularity weight factors could be better understood by applying stochastic algorithms to many data sets to find patterns in the location of local minima relative to known factors. In addition, can further optimizations be applied to 2-Opt without removing the thorough sub-range coverage? While there is always work to be done on TSP, the authors rest their case on LSP2Opt as a viable option for those with a need for high quality answers in fast run-time.

# VIII References

For this paper, a number of resources were used to gain greater light and knowledge on various algorithm topics. They are listed here:

* G. A. CROES (1958). *A method for solving traveling salesman problems*. Operations Res. 6 (1958) , pp., 791-812.
* S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani (2006). *Algorithms*, Section 9.3 Local Search, pp. 297-305