Nathaniel Breneman, breneman@oregonstate.edu
Josh Huff, huffj@oregonstate.edu
William Mack, mackw@oregonstate.edu
Professor Schutfort
CS 325 – Winter 2018
March 16, 2018

Final Project: Solving the Travelling Salesman Problem with Approximation Algorithms

Twice-around-the-tree Algorithm

Description

Twice-around-the-tree is an approximation algorithm to the traveling salesman problem based on the relationship between Hamiltonian circuits and spanning trees. Since removing an edge from a Hamiltonian circuit creates a spanning tree, using the minimum spanning tree provides a good approximation to the shortest tour. This is done by first constructing the minimum spanning tree (MST) of the graph. Then starting at an arbitrary vertex, we perform a walk around the MST, using a depth first search of the MST. Finally we scan the vertex list visited during the walk, and remove all repeated vertices except the first and last vertex will be the same. The vertices left in the list is a tour of the least distance visiting all vertices only once. Using Prim's algorithm to construct the MST has a time complexity of $O(n^2)$. Performing a depth first search of the MST is O(V + E). We scan each number in the vertex list only once, giving us linear time or O(n). Thus, this algorithm has an overall time complexity of $O(n^2)$ + O(V + E) + O(n), which reduces to simply $O(n^2)$.

Pseudocode

```
1. Create a minimum spanning tree T of G.
2. Starting at an arbitrary vertex, perform a walk W around T.
3. Scan vertex list W and remove all repeated occurrences of the same vertex.
4. Add the first vertex to the end of W to create a hamiltonian circuit. This circuit is the
   optimal tour of all vertices.
```

Held-Karp Algorithm

Description

The Held-Karp algorithm (hereafter "HKa") is a bottom-up dynamic programming approach to the TSP. Though our given assignment involves a symmetric TSP, HKa can be used to solve asymmetric TSPs as well, because it uses an adjacency matrix that can account for each edge regardless of different weights between vertices. HKa begins with a set of vertices, singling out one given vertex as both the start and endpoint. It recursively finds the least costing tours of the smallest subsets of the graph, and stores those values to be referred to when larger subsets are considered until finally the whole tour is assembled and the minimum comprehensive distance for it can be assessed.

The base cases for this recursion are as follows. The optimal tour equals 0 if there is only one element in the set of vertices, because the distance from a vertex to itself is zero. The base case for a set with two elements is the sum of the edges adjacent to the two vertices in the set. More accurately, it is a trivial example of the recursive case, because there are no alternatives; the paths themselves are the minimum tour by default. The recursive case is finding the minimum of the costs of traveling to the current node being considered from each of the nodes in the subset of those adjacent to it. Because these cumulative sums are essentially optimal subproblems, unsatisfactory options are "pruned off" as the algorithm progresses; this results in an exponential instead of factorial run-time.

An implementation of HKa will store the result of the vertex which is resolved to be the optimal choice for the preceding subset. That is, the algorithm works from "key" vertex while treating it as the endpoint, so it is effectively operating with the end as a reference and adding vertices as their optimal choices become apparent. Consequently, the reverse of this list is the optimal tour.

The time complexity is such that the distances held in memory are subproblems to be considered in linear time, or space times *n*. More formally, the time complexity for HKa is $(n – 1)(n – 2) 2^{n-3} + (n – 1)$ or $O(n^2 2^n)$ and the space complexity is $(n – 1) 2^{n-2}$ or $O(n2^n)$ (Held and Karp 199).

Pseudocode

1. Create an adjacency matrix of each vertex's Euclidean distance from all others.
2. Treat all vertices other than the origin as a set, S.
3. Loop through each vertex in S, performing steps 4 and 5 for each.
4. Beginning with the subset of endpoint's neighbors, perform step 5 on all subsets of S.
5. For each subset, apply the optimal subproblem solution, choosing the minimum value.
6. The minimal value of all these tours is the optimal. Return the reversed list of vertices.

Christofides Algorithm

Description

This algorithm is an approximation algorithm specifically designed for finding solutions to the Traveling Salesman problem. It works on instances of the problem that have each edge of the graph with a defined length (or distance) and satisfies the triangle inequality (the sum of two edges/distances in a triangle is greater than the distance of the third edge/distance.)

It works by first finding a minimum spanning tree of the graph: this is because the solution to the traveling salesman problem is a Hamiltonian cycle of minimum weight, so we want to begin with a structure that connects all vertices in the graph in a minimal way. Next, a subgraph of odd vertices is formed. The edges of this subgraph and the MST are then combined to form a connected multigraph. An Euler circuit from this multigraph is then taken, and lastly turned into a Hamiltonian circuit by skipping any vertices visited more than once. Because of the triangle inequality, we know that skipping these vertices is either going to have no effect on total distance or will reduce it; it cannot increase it.

The time complexity is the cost of finding the MST, plus the cost of finding the minimum-weight subgraph, plus the cost of finding the Euler circuit. The total cost then is O(n^3) (Christofides 4).

Pseudocode

As described by Goodrich and Tamassia on page 513:

1. Construct a minimum spanning tree, M, for G.
2. Let W be the set of vertices of G that have odd degree in M and let H be the subgraph of G induced by the vertices in W. That is, H is the graph that has W as its vertices and all the edges from G that join such vertices . . . Compute a minimum-cost perfect matching, P, in H.
3. Combine the graphs M and P to create a graph, G′, but don't combine parallel edges into single edges. That is, if an edge *e* is in both M and P, then we create two copies of *e* in the combined graph, G′.
4. Create an Eulerian circuit, C, in G′, which visits each edge exactly once (unlike in the 2-approximation algorithm, here the edges of G′ are undirected).
5. Convert C into a tour, T, by skipping over previously visited vertices.

2-Opt Algorithm

Description

The 2-opt heuristic was introduced in the late 1950s by G. A. Croes. Croes described the process of approximating the optimal tour as following three general steps: finding a valid trial path; applying edge inversions that transform said trial into a smaller, still working tour; and consulting the adjacency matrix for edge weights that might contribute to shortening the tour as a result of their inclusion (793).

Essentially, 2-opt can be described as a local search algorithm where two edges are temporarily removed from a tour, creating two subgraphs that are to be reconnected by two new edges (Mersmann 3). If exchanging the pairs of edges results in a shorter path, the result becomes the new tour. This process is repeated at every vertex in the tour, such that every local “neighborhood” of the tour is optimized, leading to the tour becoming optimized globally. Of course, because the trial tour chosen at the beginning of execution does not necessarily resemble the actual optimal tour, it is not necessarily the case that 2-opt will eventually produce the optimum result. Instead, it provides a result within an approximation ratio of 2.

Pseudocode

Adapted from pseudocode described by Slootbeek on page 6:

1. Assemble a “trial” tour that satisfies the TSP requirements without regard for tour length.
2. Set a best distance variable to the current route’s total distance.
3. Repeat steps 4 through 12 until the loops terminate without finding an improved route.
4. Loop through each vertex, I, in current route, performing step 4 for each
5. Loop through each vertex, K, where K starts at I + 1, performing step 5 for each.
6. Set a new route to the result of calling 2-opt method (steps 7 through 10) with I and K:
7. Take route[0] to route[i – 1] and add them in order to new route
8. Take route[i] to route[k] and add them in reverse order to new route
9. Take route[k+1] to the route’s end and add them in order to new route.
10. Return the new route.
11. Set a new distance to the total distance of new route.
12. If the new distance is less than the best distance, the new route becomes the current route.

Summary of Additional Considered Algorithms

Branch-and-cut / branch-and-bound algorithms

The primary deterrent to our pursuit of this option was logistics. Operating LINDO via Citrix was facile for linear programming exercises, but none of us could accurately estimate how much time and effort it would take to develop confidence in our understanding of the algorithm, find a open-source linear programming suite for our chosen language, and efficiently leverage its methods in our own program to successfully find optimal tours. Even if we were to overcome these difficulties, there was the very real possibility of flip not supporting some component.

Brute Force

Intuitively, we understood this would certainly result in the correct solution, but the run-time of a brute force approach to this problem is factorial, so this option is completely unusable to all but the smallest problem sets. Because our sample problem sets included node counts in the hundreds and tens of thousands, this was never a serious consideration.

Lin-Kernighan heuristic

According to Helsgaun, the original implementation of the Lin-Kernighan heuristic boasts an average running time of $O(n^{2.2})$ (7). What's more, it is relatively efficient in finding an optimal solution (Helsgaun 2). However, implementation seems complex enough that we were concerned about the feasibility of finishing a working program within our given deadline. Additionally, the algorithm's designers warn that its performance depends heavily on implementation choices; our initial choice of language would likely be a huge bottleneck in that respect. For these reasons, we decided to treat Lin-Kernighan heuristic as a last resort option.

Selected Algorithms: Nearest Neighbor and 2-opt

Our choice of Python 3 was made under the assumption that we eliminated all of the poorest choices (that is, those that were heavily dependent on implementation choices to reach their advertised performance, such as the Lin-Kernighan heuristic), and any reasonably efficient implementation of the remaining algorithms would satisfy the project requirements. Naturally, if (when) the results turned out poorer than anticipated, we could defer changing our entire strategy in favor of porting our current codebase to a similar implementation in C/C++.

The Christofides algorithm was an obvious first choice because of the 3/2-approximation bound it boasts. The pseudocode gave us pause, however, as it described concepts outside the scope of this course and our independent research revealed that the Christofides algorithm is, itself, composed of several other algorithms, some of which were very simply described but prohibitively time-consuming to actually implement. In particular, the need to compute minimum weight perfect matching called for the use of a "blossom" algorithm, of which there are several descriptions and examples online—and none of them were intuitively comprehensible. If this were a graduate-level course and we had had the entire term to research and implement Christofides, it would certainly remain a first choice, but the prospect of completing that same amount of work in the two weeks before final exams was daunting.

Consequently, Held-Karp was considered as an alternate option in case Christofides proved to be too difficult to implement in the given time frame. Further research on Held-Karp showed that, because of its massive memory requirements, it was ineffective for all but the smallest input sizes. There are great strides being made in supercomputing, but seeing as our program must be written for flip (which has only ~100 GB of RAM and is capable of providing a mere 24 threads), such is beyond the purview of this project. Parallelizing the TSP was not integral to our strategy from the beginning, primarily because we understood that our implementation language, Python 3, would vastly limit the effectiveness of most parallel programming efforts (particularly, multi-threading) because of the global interpreter lock feature. Furthermore, as budding software engineers, we are sensitive to the notion of premature optimization; our first concern was meeting the requirements and solving the problem correctly, with the extra credit for speed in the competition cases a distant second.

Because we were running out of viable options, we decided to try 2-opt and twice-around-the-tree separately, believing one of them would be sufficient. Even having been exposed to how favorable data conditions can have a dramatic effect on an algorithm's performance (e.g. passing a partially-sorted list of integers to insertion sort results in a far better outcome than its quadratic big-O), we expressed a shared consternation at the 1.25-approximation threshold. Some sources indicated that, in some cases, 2-opt can be marginally closer to the optimal than even Christofides, which since its conception in 1976 has reigned as the best general-purpose TSP approximation algorithm. This project was very instructive in its reminder that a guaranteed upper bound for proximity to the optimal does not necessarily promise any level of performance at run-time.

Since combining twice-around-the-tree and 2-opt didn’t produce the results we hoped for, we considered other algorithms that we previously deemed as unreliable, including the Nearest Neighbor (NN). NN is simple enough in principle: beginning with the home vertex, choose the least expensive edge to the next unvisited vertex until a valid tour is established. The NN is purportedly a workable solution for the special case we’re to address for this project—that is, a Euclidean TSP (Gutin, et al. 82). However, at the outset, we did not consider the possibility of our solution falling outside the 1.25 approximation threshold an acceptable risk. As Bang-Jensen, et al. demonstrate by proving in Theorem 4.3 of their work, NN will sometimes return the worst tour available even when presented with a significant number of better options (125). As tempting as it was to have a very quick, easy implementation, we agreed that we would be gambling to use the NN algorithm and consequently discarded it from consideration entirely.

Ultimately, we realized that we needed the best of both worlds of the NN and 2-opt algorithms. After some trial and error, we revisited the literature and discovered that our 2-opt was not achieving the results we expected of it because its trial tour was not carefully constructed. As we learned in our research, the algorithm is capable of locally optimizing edge choices, but the key limitation is that it can only optimize the tour it is given. Treating the NN algorithm as a preliminary method to provide 2-opt with a trial tour that stands closer to the theoretical optimal results in a serviceable approximation in an appropriate time. Our approach consists of running NN on twelve threads, each of which then runs 2-opt on its respective trial tour. The resulting tour is the least distance choice from among those candidate solutions.

Best Tours

Example instances

Tour Number	Distance	Time (seconds)
1	130861	0.034
2	2976	0.131
3	1930587	163.231

Competition instances

Tour Number	Distance	Time (seconds)
1	5844	0.023
2	8167	0.027
3	15080	0.165
4	19747	0.870
5	27822	4.063
6	39344	75.104
7	61821	159.514

Works Cited

Bang-Jensen, Jørgen, et al. "When the greedy algorithm fails." *Discrete Optimization*, vol. 1, no. 2, Nov. 2004, pp. 121–127., doi:10.1016/j.disopt.2004.03.007.

Christofides, Nicos. "Worst-case analysis of a new heuristic for the travelling salesman problem." Management Sciences Research Report No. 388. Pittsburgh, PA: Carnegie-Mellon University, 1976.

Croes, G A. "A Method for Solving Traveling-Salesman Problems." *Operations Research*, vol. 6, no. 6, 1958, pp. 791–812., doi:10.1287/opre.6.6.791.

Goodrich, Michael, and Roberto Tamassia. *Algorithm Design and Applications*, Wiley, 2015.

Gutin, Gregory, et al. "Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP." *Discrete Applied Mathematics*, vol. 117, no. 1-3, 15 Mar. 2002, pp. 81–86., doi:10.1016/S0166-218X(01)00195-0.

Held, Michael, and Richard M. Karp. "A Dynamic Programming Approach to Sequencing Problems." *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, Mar. 1962, pp. 196–210., doi:10.1137/0110015.

Helsgaun, Keld. "An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic." *European Journal of Operational Research*, vol. 126, no. 1, 1 Oct. 2000, pp. 106–130., doi:10.1016/S0377-2217(99)00284-2.

Mersmann Olaf, et al. "Local Search and the Traveling Salesman Problem: A Feature-Based Characterization of Problem Hardness." Learning and Intelligent Optimization : 6th International Conference, Paris, France, January 16-20, 2012, Revised Selected Papers. Springer, 2012.

Puntambekar, A.A. “Coping with the Limitations of Algorithm Power.” Analysis And Design Of Algorithms, pp. 12–53-12–54.

Slootbeek, Jaap J. A., “Average-Case Analysis of the 2-opt Heuristic for the TSP.” MA thesis, University of Twente, 2017.