# Project 4 - The Traveling Salesman Problem

Matt Schreiber,
CS325 - Analysis of Algorithms,
Winter Quarter 2014

March 11, 2014

## Introduction

For this project, I implemented four separate algorithms in Perl: *a)* A greedy, or nearest-neighbor, algorithm; *b)* A minimum spanning tree algorithm; *c)* Christofides' algorithm; and *d)* a min-max ant colony optimization algorithm . Although the algorithms differ in a variety of ways, there are a few respects in which they are similar or identical. First, when traversing a path in the graph, each algorithm marks off vertices which have already been visited. This restricts the algorithms to choosing from among unvisted vertices for their next leg, ensuring that the path does not include any cycles. Furthermore, two algorithms (nearest-neighbor and ant colony) strongly prefer (in the case of nearest-neighbor, insist upon) choosing as next node the node connected to the current position by the shortest edge, while the other two algorithms choose an arbitrary next node, having already restricted path choices to optimized subgraphs of the original graph.[1]

## Nearest-Neighbor Algorithm

This is the simplest of the four algorithms. It begins by choosing a random vertex from within the graph, then selects next vertices solely on the basis of what is locally optimal. In other words, the algorithm always selects as next node that node which is connected to the current node by the shortest edge.

As the algorithm proceeds through the graph, it maintains a list of vertices that it has already visited, and excludes these from consideration when choosing what vertex to move to next. Despite its extreme simplicity, the nearest-neighbor algorithm produces results at a relatively high speed whose accuracy is comparable to the more sophisticated algorithms.

## Minimum Spanning Tree Algorithm

*Because this algorithm was adapted directly from another person's work,* [2] *it was not chosen as the default algorithm for my program.*

The algorithm begins by constructing a minimum spanning tree of the graph, [3] i.e. a tree that satisfies the conditions that *1)* all nodes are connected (there are no disjoint subgraphs); and *2)* the total weight of all edges is at least as low as that of any other spanning tree of the graph.

[1] In a sense (or rather, two senses) of 'optimized' that will be explained shortly.

[2] *See* Mankowski, Walt, "Approximation Algorithms in Perl", `http://www.mawode.com/~waltman/talks/approx_ppw.pdf`

[3] Using the `minimum_spanning_tree` routine from the Perl CPAN Module `Graph`, documentation at `http://search.cpan.org/~jhi/Graph-0.96/lib/Graph.pod`

Once the tree has been constructed, the algorithm chooses an arbitrary node in the tree, then begins simply walking through the tree, selecting a neighbor arbitrarily (in whatever order `perl` decides to use when iterating through the list of the current node's neighbors) so long as it hasn't already been visited. The neat little trick is that, because the vertices and edges of a Euclidean Traveling Salesman Problem necessarily obey the triangle inequality, it is possible to 'shortcut' past a path in the minimum spanning tree, proceeding to a neighbor directly along whatever path was in the original graph, without adding extra distance.

Let me give an example: suppose we've already gone through points $a$ and $b$ in the tree, and we are now at point $c$. Point $d$ is not connected to $c$ in the minimum spanning tree, although it is connected to point $a$. However, we need not travel back through $a$ and $b$ to get to $d$: since, in the original graph the distance $a - c$ (the hypothenuse of a triangle) cannot be shorter than the distance $a - b - c$ (the sum of the triangle's two legs), the distances $c - d$ will never be longer than the distance $c - b - a - d$.

The algorithm simply 'skips' to neighbor nodes, even if they are not connected in the minimum search tree, stopping when there are no unvisted nodes.

## Christofides' Algorithm

This algorithm is something of a jazzed-up version of the minimum spanning tree algorithm, and it uses the same traversal method as that algorithm. After creating the minimum spanning three, the algorithm excises all nodes of even degree (those connected to an even number of edges). From the set of remaining nodes, the algorithm constructs a minimum perfect matching, i.e. a set of [4] edges in which no edge shares a vertex with any other edge, where all vertices in the graph are connected to an edge – the 'perfect' part – and where the sum of the weights of the edges is the same or less than for any other matching – the 'minimum'' part).

The next step is to combine the edges of the matching with the edges from the minimum spanning tree, which creates a graph in which it is possible to to create an Eulerian circuit, i.e. a path through the graph that passes only once through each edge [5]. The algorithm then converts this into a Hamiltonian path by using the same 'shortcut' method as the the minimum spanning tree algorithm, in other words, substituting direct connections between nodes, from the original connected graph, for any path that passes through an already-visted node. Again, this is possible without any gain in total path weight because the edges in the original graph satisfy the triangle inequality.

## Min-max Ant Colony Algorithm

This was, by a fair margin, the most complicated algorithm to implement. The basic idea is to emulate the emergent pattern-finding capabilities of any ant colony, in which ants leave pheromone trails that increase the likelihood that further ants will choose to follow that ant's pathway, rather than cut another one. This is done by assigning each edge a 'pheromone' value in addition to its weight; when a 'worker ant' process is considering which edge to take from a given node, the choice is based on a probabilistic function that rates the 'attractiveness' of a path based on a weighted comparison of each edge's weight, preferring shorter edges to longer ones, and its pheromone strength, preferring paths which more ants have taken.

The min-max variation of the ant colony heuristic sets minimum and maximum values for pheromone strength, initializing all paths in the graph to the maximum at the outset and 'evaporating' the pheromone strength at a specified rate after the termination of each exploratory phase. Each ant starts at a randomly-chosen node in the graph, walking

[4] Using Joris van Rantwijk's `Graph::Matching` module, available at `https://www.google.com/url?q=http://jorisvr.nl/maximummatching.html`. Because this module produces *maximum* rather than minimum matches, getting the expected result involded a little trickery with negating edge weights, as described in my `tsp.pl` source.

[5] As the Wikipedia page for the Christofides Algorithm (`http://en.wikipedia.org/wiki/Christofides_algorithm`) points out, this is possible because all nodes in the graph are now of even degree, so for every unique entry into a node there is at least one unvisited edge by which to exit

the edges until no unvisited nodes remain. When the worker processes terminate, the main process compares the lengths of the paths, updating the value of the shortest path found yet if the shortest path of the iteration is the shortest overall. The main process then chooses, based on a simple semi-randomized equation, whether to increase by a specified amount the pheromones on the global best path, or the best path for the given iteration.

As more and more iterations execute, the paths of the ants are more and more likely to converge upon the global best path found so far, since this path will contain edges that are both *a priori* preferable, due to their short length relative to other edges from a given node, and *a posteriori* desirable due to their high pheromone strength.

## Miscellaneous

Although this strategy does not form a part of any of the four algorithms *per se*, I also implemented multiprocessing in order to generate and test multiple solutions simultaneously. This can be disabled by passing the arguments `--processes 1` to the program.