The Traveling Salesman Problem

Rebecca Sagalyn

August 28, 2013

1 Introduction

Given a set of cities and known distances between each pair of cities, the TSP aims to find a tour in which each city is visited exactly once and the distance travelled is minimized. I chose to use the Christofides' algorithm to construct a tour and the 2-opt algorithm to improve it.

2 Christofides' Approximation Algorithm

Christofides' is a tour construction heuristic that can be applied to graphs with the following property: A complete graph $G = (V, E, w)$ with edge weights that satisfy the triangle inequality $w(x, y) + w(y, z) \leq w(x, z) \qquad \forall x, y, z \in \mathcal{V}$

1. Find a minimum spanning tree T of G
2. Let O be the set of vertices with an *odd* degree in T
3. Find a minimum cost perfect matching M for these vertices
4. Add M to T to obtain multigraph H
5. Find a Eulerian tour of H
6. Convert the Eulerian tour into Hamiltonian path by skipping visited nodes (using shortcuts)

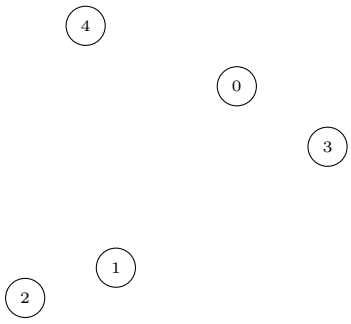

Figure 1: Nodes

2.1 Minimum Spanning Tree

I used Prim's Algorithm to find the minimum spanning tree in G:

function MST($G = (V, E)$)
  for v in V **do**
    $key[v] \leftarrow \infty$
    $parent[v] \leftarrow NULL$
    insert v into Q
  end for
  $key[0] \leftarrow 0$
  while !$Q.empty()$ **do**
    $v \leftarrow Q.removeMin()$
    for u adjacent to v **do**
      if $u \in \mathcal{Q}$ and $weight(u, v) < key[u]$ **then**
        $parent[u] \leftarrow v$
      end if
      $key[v] \leftarrow weight(u, v)$
    end for
  end while
end function

The algorithm maintains an array Q with the vertices that are not yet in the tree (initially, Q is empty). It iterates through each vertex v not yet in the tree (using vertex 0 as the intial vertex) and chooses the minimum weight edge (u, v) where u is already in the tree. Thus, this is the lightest edge crossing the cut (since it connects an edge in the MST with an edge not yet in the MST). Then vertex v is added to the tree. The algorithm continues until Q is empty and thus all vertices have been added to the MST.

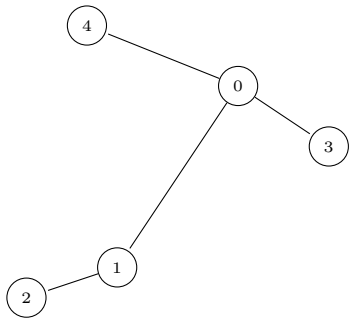

Figure 2: Find the MST

2.2 Vertices With An Odd Degree in MST

The next step is to find vertices with an odd degree in the MST. Since I store the MST in an adjacency list using C++ vectors, this procedure only involves checking the vector size at each index (node) of the list and is $O(n)$.

2.3 Weighted Perfect Matching for Odd Vertices

A connected graph has an even number of vertices of an odd degree. We now find a perfect matching among these vertices so that all vertices have an even degree. Ideally, we would find a minimum matching, but instead I used a greedy algorithm to find an approximate minimal matching.

function PERFECTMATCHING
  Input: *odds* (list of odd vertices), G (adjacency list)
  while (!*odds.empty*) **do**
    $v \leftarrow odds.popFront()$
    $length \leftarrow \infty$
    for $u \in \wp dds$ **do**
      if $weight(u, v) < length$ **then**
        $length \leftarrow weight(u, v)$
        $closest \leftarrow u$
      end if
    end for
    $G.addEdge(closest, v)$
    $odds.remove(closest)$
  end while
end function

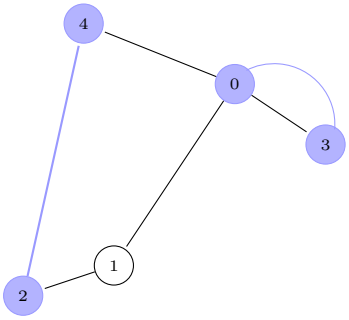

Figure 3: Find a perfect minimal-*ish* matching on odd vertices

The set of matched vertices is now added to the MST, forming a new multigraph.

2.4 Eulerian Tour

Next we find a euler circuit starting at any arbitrary node in our multigraph. If our node has neighbors, we push our node on a stack, choose a neighbor, remove the edge between them from the graph, and make that neighbor the current vertex. If our vertex has no neighbors left, we add it to our circuit and pop the top vertex from the stack to use as our current vertex. We continue tracing a tour in this manner until the stack is empty and the last vertex has no more neighbors left.

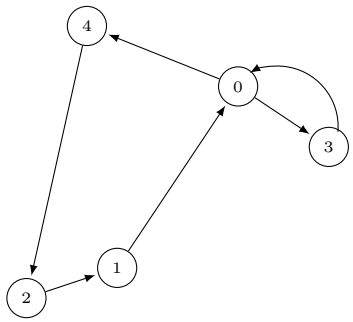

Figure 4: Find a Euler circuit

2.5 Hamiltonian Path

Finally, we turn our Euler circuit into a Hamiltonian path by walking along the Euler tour, checking at every stop whether that node has already been visited. If it has, we skip that node and move on to the next one. Since our graph satisfies the triangle inequality, shortcutting vertices in this manner will not increase the length of our path.

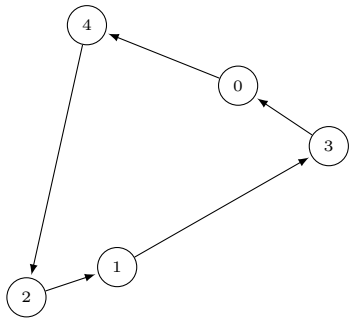

Figure 5: Convert the Euler circuit into a Hamiltonian path by skipping visited nodes

3 Two-Opt

After a tour was constructed using the Christofides heuristic, I applied the 2-opt improvement algorithm to optimize the path. The 2-opt algorithm examines each edge in the tour. For each edge, it looks all non-adjacent edges, and determines whether removing the two edges and reconnecting them would shorten the tour. If it does, the edges are swapped. The search continues until it no longer improves the path.

References

[1] Hyung-Chan An, *Improving Christofides' Algorithm for the s-t Path TSP*. Cornell University. www.graph-magics.com/articles/euler.php

[2] *Jarnik's (Prim's) Algorithm.* www.personal.kent.edu/ rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/primAlgor.htm

[3] wwww.seas.gwu.edu/ simhaweb/champalg/tsp/tsp.html

[4] *Eulerian Path and Circuit.* www.graph-magics.com/articles/euler.php