Graph Applications, Class Notes, CS 3137

1 Traveling Salesperson Problem

Web References:

- `http://www.tsp.gatech.edu/index.html`
- `http://www-e.uni-magdeburg.de/mertens/TSP/TSP.html` TSP applets

A *Hamiltonian* cycle is a special kind of cycle in a graph in which each vertex in the graph is visited exactly once (except the start and end vertex which are identical). Not every graph will admit to containing a Hamiltonian cycle. However, there is special case of the this problem in which a Hamiltonian path of shortest distance is sought in a **complete graph** on N vertices. This is known as the Traveling salesperson problem, as it mirrors the task of a salesperson who must visit N cities in the most efficient possible way.

How difficult is it to solve this problem? To get a quick idea of the complexity of the problem, we can say that any TSP path will be a listing of the N cities in some order. How many possible orderings are there? Given a starting city (we can choose any city since it will be somewhere on the path) there are $N-1$ choices for the second city, in the path, $N-2$ choices for the third city in the path, and so on, leaving $(N-1)!$ factorial possible paths: $(N-1)! = (N-1)\cdot(N-2)\cdot(N-3)\cdot\ldots\cdot 1$. Actually, this will generate paths which are circularly symmetric, so they constitute the same path. For example, listing a 4 city Hamiltonian cycle as 1-2-3-4-1 is identical to the cycle 1-4-3-2-1. The actual number of distinct paths is $\frac{(N-1)!}{2}$ because of this, but we can ignore symmetry to do the analysis below.

To determine which of these paths is shortest, we are forced to try them all. This is an extremely expensive computation. For 11 cities, 10! is the number of possible paths, or 3,628,800 paths we have to examine to see which is shortest. If we increase it to only 20 cites, we get $1.2\cdot 10^{17}$ paths!

How long does it take to compute $1.2\cdot 10^{17}$ paths? Lets assume that your computer can do about 1 billion instructions per second (1 GHz), and that each 20 city path takes about 20 operations to compute, so that you can calculate 50 million paths per second. So in 1 day, your computer can calculate this many paths:

$$\frac{1 Billion\ Instructions}{1 sec.}\cdot\frac{1 path}{20 instructions}\cdot\frac{60 sec.}{1 min.}\cdot\frac{60\ min.}{1 hour}\cdot\frac{24 hours}{1\ day}=\frac{4,320,000,000,000 paths}{1 day} \tag{1}$$

To find out how many days this computation would take, we divide:

$$\frac{1.2\cdot 10^{17}}{4.32\cdot 10^{12}} = .28\cdot 10^{5} days\ =\ 28000\, days\ =\ 76\, years \tag{2}$$

Such problems are practically intractable; in fact they belong to a special class of problems known at $NP\ \ Complete$. There are no known algorithms that can compute these problems in a reasonable amount of time. So what are we to do? On large search problems like this, we can use approximate methods that can come close to finding the minimum distance path, yet can be computed in a much shorter time than what we saw above. The basic method is 1) find a feasible solution, and then 2) improve that solution, moving it towards the optimal solution.

Is there a simple algorithm that will choose a TSP path that we can say will be bounded by never being greater than the optimal path by a constant factor? There is, and it is a variant of the Minimal Spanning Tree

(MST) we computed. Given an MST, If we perform a preorder traversal of this tree and number its nodes in the order we visit them, we can come up with a TSP path that is guaranteed to be no more than twice the MST cost. To see this, we define:

$$C(H_{opt}) = cost\ of\ optimal\ TSP\ path \tag{3}$$

$$C(T) = cost\ of\ MST \tag{4}$$

Since we can generate a spanning tree by removing a single edge from a TSP path, we know that:

$$C(T) \leq C(H_{opt}) \tag{5}$$

Now, define a *walk* of the graph as a path that mimics the MST, but when it reaches a "dead end", it simply backtracks to the vertex it came from and continues along the MST. This walk will traverse every edge of the MST twice, and if its cost is $C(W)$:

$$C(W) = 2C(T),\ and\ substituting \tag{6}$$

$$C(W) \leq 2C(H_{opt}) \tag{7}$$

which means the cost of a walk is within a factor of 2 of the optimal path. A walk isn't quite a TSP path, since we repeat vertices. However, we can turn the walk into a TSP path if we do a depth first search with a preorder numbering the nodes as we go along (see figure 1). This ordering then becomes our path. This cycle's cost can be no greater than the cost of the walk, since we omit some vertices along the walk's path. So if we denote this TSP path's cost as $C(H)$:

$$C(H) \leq C(W) \leq 2C(H_{opt}) \tag{8}$$

1.1 Approximate algorithms for TSP paths

A simple way to find a TSP path is to use a local, greedy algorithm, known as the **Nearest-Neighbor (NN)** path. In this method, we always travel to the nearest city that hasn't been yet visited from the most recent city we have visited. At the end, we simply link the first and last cities to finish the path. For the city distance matrix below the optimal TSP path is 1790 miles, and the NN method will give a value of 2080 miles starting from Washington, which is 16% more than optimal. Interestingly, the NN path will vary depending on the starting city. Therefore, if we compute NN from each vertex, we can take the minimum of these paths as our best TSP path (the shortest path is 1995 miles beginning at Bufflo).

The NN method can perform poorly sometimes because we build it very locally. Another way to look at the problem is known as the method of **Coalesced Simple Paths** which is a variant of Kruskal's MST method: add edges of minimum cost as long as the problem's constraints are satisfied. In MST, we always added the edge of minimum cost under the constraint that no cycles were created. In TSP, we do the same: add the edge of minimum cost as long as no cycles are created **AND** no vertex has degree greater than 2. This will generate a unique TSP cycle if there are distinct edge costs. For the city distance matrix its minimum cost path is 2056 miles, which is 14% greater than optimal.

Insertion methods are also used, in which we insert a vertex to increase the size of a path. Starting with a simple cycle of k vertices, we keep adding vertices that minimize the change in the path's new cost. For example, if we have a an edge (u,v) in the path, we add the vertex x such that:

$$dist(u, x) + dist(x, v) - dist(u, v)\ is\ a\ minimum \tag{9}$$

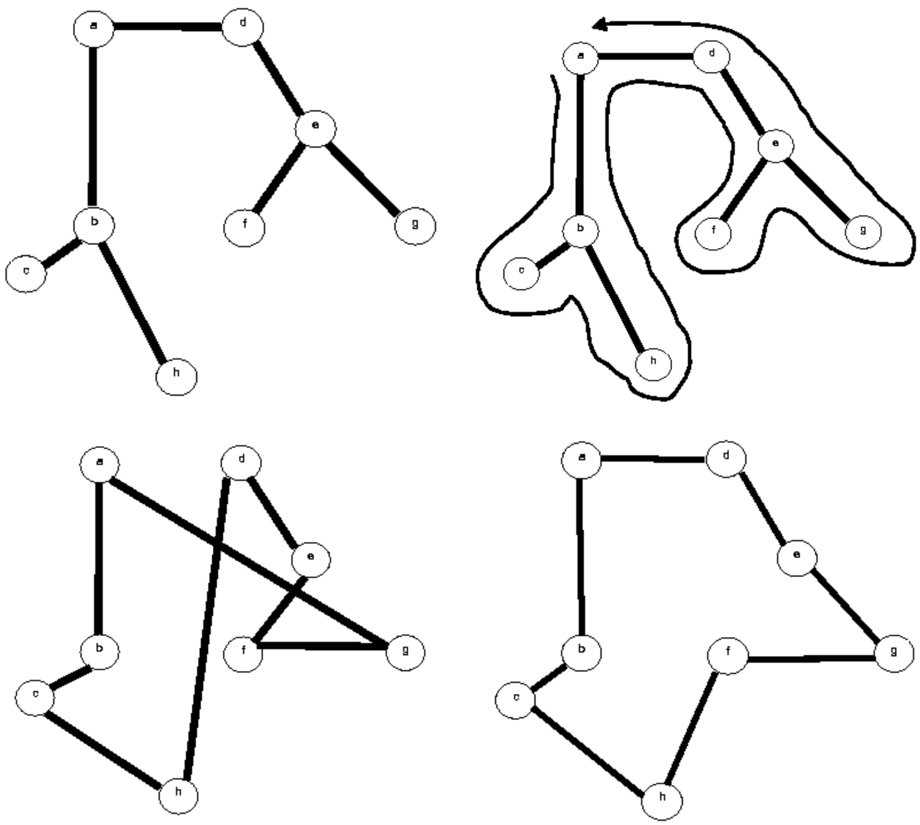

Figure 1: Given a spanning tree of a graph, there is a simple way to construct a Hamiltonian circuit of the graph using Depth First Search. Top left: A spanning tree of a graph. Top Right: A depth first search of the spanning tree. Bottom Left: A Hamiltonian circuit of the graph formed by linking the vertices according to a preorder numbering of the nodes from the Depth First Search. Bottom Right: An improved circuit formed by removing any crossing edges in the graph. Step 1: Crossing Edges a-g and e-f are exchanged for a-f and e-g. Step 2: This still leaves crossing edges a-f and h-d, which are then exchanged for edges a-d and h-f, yielding the circuit without crossing edges.

We may iterate over all choices of u,v to select this minimum.

Another simple refinement is to take an existing path, and analyze if 2 edge pairs can be rearranged to produce a shorter path. If the path has an edge from A to D and an edge from C to B, we can see if connecting A to B and C to D will reduce the cost:

Baltimore	0	345	514	355	522	189	97	230	39
Buffalo	345	0	430	186	252	445	365	217	384
Cincinnati	514	430	0	244	265	670	589	284	492
Cleveland	355	186	244	0	167	507	430	125	356
Detroit	522	252	265	167	0	674	597	292	523
New York	189	445	670	507	674	0	92	386	228
Philadelphia	97	365	589	430	597	92	0	305	136
Pittsburgh	230	217	284	125	292	386	305	0	231
Washington	39	384	492	356	523	228	136	231	0

(b) The distance matrix

```
- - - - - A***B - - - -       - - - - - A   B - - - - -
                                         * *
                                          *
                                         * *
- - - - - C***D - - - -     - - - - - - C   D - - - - -
```

If $dist(A, B) \ + dist(C, D) \ < \ dist(A, D) + dist(C, B)$ we perform the switch. We can prove that if 2 edges cross each other, they can never be part of the optimal path. So checking for these edges and then switching them as above can reduce the path length. To prove this add an intermediate node Z where the path;s cross:

```
- - - - -  A   B - - - - -   - - - - - A   B - - - - -
            * *                         * *
             *                           Z
            * *                         * *
 - - - - - C   D - - - - -   - - - - - C   D - - - - -
```

$$dist(A, D) + dist(C, B) = dist(A, Z) + dist(Z, B) + dist(C, Z) + dist(Z, D) \tag{10}$$

Because $dist(A, B) < dist(A, Z) + dist(Z, B)$ and $dist(C, D) < dist(C, Z) + dist(Z, D)$ we can always improve the tour by making this switch.

This known as 2-opting, and a path in which all edge pairs are analyzed this way is 2-optimal. We can also look at sets of 3 edges edges to perform 3-opting, and on up to an arbitrary k-opting. However, the cost increases quickly. There are 8 possible ways to reconnect 3 pairs of edges, and $2^{k-1}(k-1)!$ alternatives in general in k-opting.

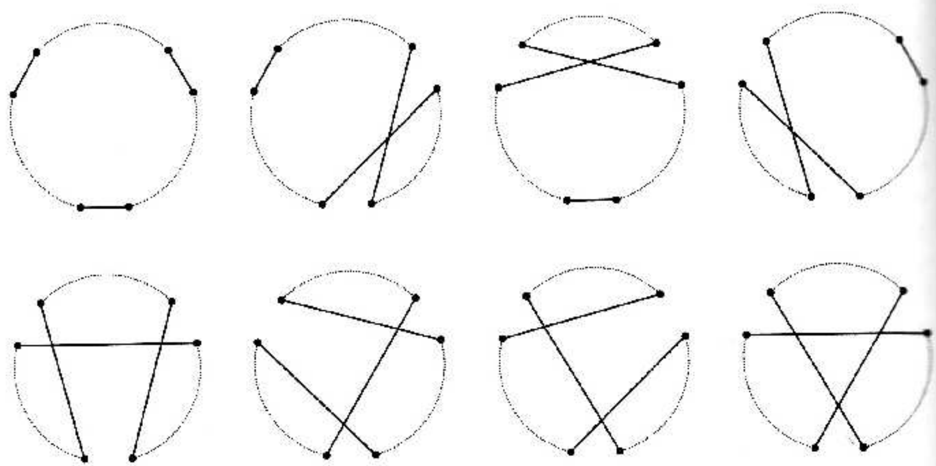

Figure 6.3: The Eight Ways in Which Three Paths Can Be Reconnected

Figure 2: 3-opting has many choices of replacement paths to try to reduce the tour cost.