Traveling Salesman Problem

One version of the ***Traveling Salesman Problem*** (***TSP***) is to minimize the total airfare for a traveling salesman who wants to make a tour of n cities, visiting each city exactly once before returning home. Figure 6.4.3 shows a weighted graph model for the problem; the vertices represent the cities and the edge-weights give the airfare between each pair of cities. A solution requires finding a minimum-weight Hamiltonian cycle. The graph can be assumed to be complete by assigning arbitrarily large weight to edges that do not actually exist.

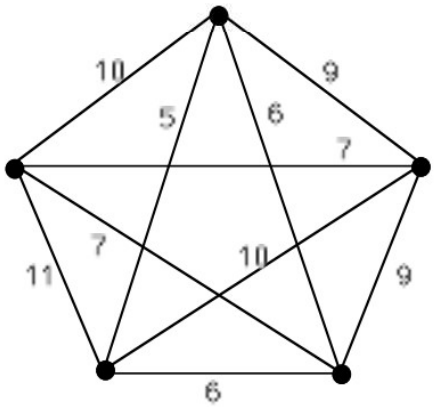

Figure 6.4.3 Weighted graph for a TSP.

The TSP has a long history that has stimulated considerable research in *combinatorial optimization*. The earliest work related to the TSP dates back to 1759, when Euler published a solution to the Knight's Tour Problem (see Exercises). Other early efforts were made by A.T. Vandermonde (1771), T.P. Kirkman (1856), and of course W.R. Hamilton (1856).

The problem of finding a minimum-weight Hamiltonian cycle appears to have been first posed as a TSP by H. Whitney in 1934. M. Flood of Rand Corporation recognized its importance in the context of the then young field of *operations research*, and in 1954, three of his colleagues at Rand, G. B. Dantzig, D.R. Fulkerson, and S.M. Johnson ([DaFuJo54]), achieved the first major breakthrough by finding a provably optimal tour of 49 cities (Washington, D.C. and the capitals of the 48 contiguous states). Their landmark paper used a combination of linear programming and graph theory, and it was probably the earliest application of what are now two of the standard tools in integer programming, *branch-and-bound* and *cutting planes*.

The next dramatic success occurred in 1980 with the publication by Crowder and Padberg of a provably optimal solution to a 318-city problem ([CrPa80]). To enumerate the problem's approximately 10^{655} tours at the rate of 1 billion tours per second, it would take a computer 10^{639} years. The Crowder-Padberg solution took about 6 minutes by computer, using a combination of branch-and-bound and *facet-defining inequalities*.

Heuristics and Approximate Algorithms for the TSP

DEFINITION: A ***heuristic*** is a guideline that helps in choosing from among several possible alternatives for a decision step.

Heuristics are what human experts apply when it is difficult or impossible to evaluate every possibility. In chess, the stronger the player, the more effective that player's heuristics in eliminating all but a few of the possible moves, without evaluating each legal move. Of course, this has the risk of sometimes missing what may be the best move.

DEFINITION: A ***heuristic algorithm*** is an algorithm whose steps are guided by heuristics. In effect, the heuristic algorithm is forfeiting the guarantee of finding the best solution, so that it can terminate quickly.

Since the TSP is *NP*-hard, there is a trade-off between heuristic algorithms that run quickly and those that guarantee finding an optimal solution. Time constraints in many applications usually force practitioners to opt for the former. An excellent survey and detailed analysis of heuristics are found in [LaLeKaSh85]. A few of the more commonly used ones are given here.

The simplest TSP heuristic is ***nearest neighbor***. Its philosophy is one of shortsighted greed: from wherever you are, pick the cheapest way to go somewhere else. Thus, the nearest-neighbor algorithm (appearing on the next page) is simply a depth-first traversal where ties are broken by choosing the edge of smallest weight.

Algorithm 6.4.1: Nearest Neighbor
Input: a weighted complete graph.
Output: a sequence of labeled vertices that forms a Hamiltonian cycle.

Start at any vertex v.
Initialize $l(v) = 0$.
Initialize $i = 0$.
While there are unlabeled vertices
    $i := i + 1$
    Traverse the cheapest edge that joins v to an unlabeled vertex, say w.
    Set $l(w) = i$.
    $v := w$

As is typical of greedy algorithms, the nearest-neighbor heuristic is very fast, and it is easy to implement. The algorithm sometimes performs quite well; for instance, for the weighted graph in Figure 6.4.3, it produces the optimal solution if it starts at the top

Vertex.