Computing for the Physical and Social Sciences (COS 323) – Fall 2001

Assignment 2: The Traveling Salesman Problem (W. Cook)
Due Date: October 25, in class

Given the cost of travel between each pair of a finite number of cities, the *traveling salesman problem* (TSP) is to find the cheapest *tour* passing through all of the cities and returning to the point of departure. The optimal tour through the 13 cities of Mercer County is given in Figure 1.

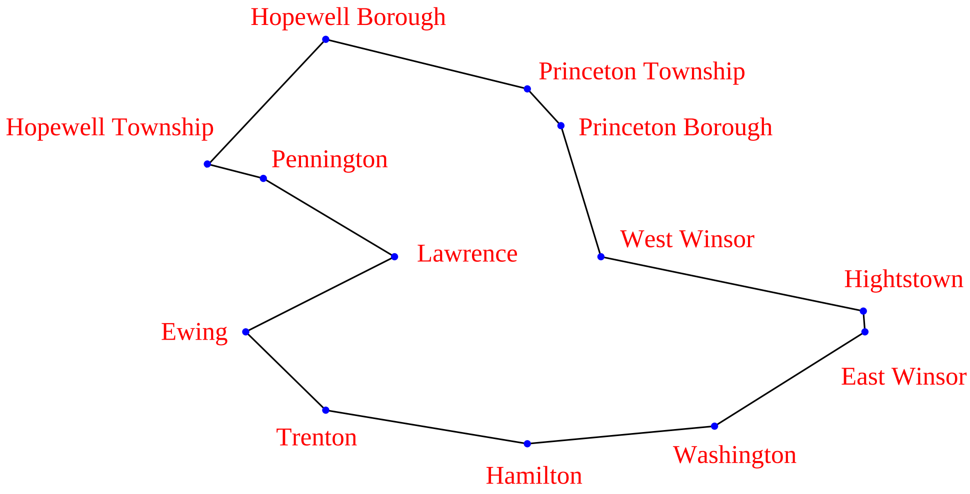

Figure 1: Optimal Tour

The simplicity of the TSP model, coupled with its apparent intractability, makes it an ideal platform for exploring new algorithmic ideas, and it has long been a primary subject for the study of both exact and heuristic methods in discrete optimization. Surveys of work on the TSP can be found in Lawler, Lenstra, Rinnooy Kan, and Shmoys (1985), Reinelt (1994), Jünger, Reinelt, and Rinaldi (1995), and Johnson and McGeoch (1997). A guide to on-line literature and a description of current results can be found on the web page:

`http://www.math.princeton.edu/tsp/`

We will use the TSP to introduce combinatorial computing and to discuss heuristic algorithms.

In this assignment we consider TSP instances specified by sets of points in the plane, using the TSPLIB (see Reinelt (1991)) Euclidean distance function to determine the cost of travel between pairs of cities: the cost of the trip between city (x_1, y_1) and city (x_2, y_2) is the Euclidean distance between the points, rounded to the nearest integer. A C-function to compute the distance is:

```
static int edgelen (double x1, double y1, double x2, double y2)
{
        double t1 = x1 - x2, t2 = y1 - y2;
        int temp;

        temp = (int) (sqrt(t1 * t1 + t2 * t2) + 0.5);
        return temp;
}
```

Note that this is a symmetric distance function, in the sense that it costs just as much to travel from (x_1, y_1) to (x_2, y_2), as it does to travel from (x_2, y_2) to (x_1, y_1).

Part 1: Enumeration Algorithms

In a sense, the TSP is very easy to solve: we just need to compute the length of each tour and record the shortest one. A difficulty, of course, is that the number of tours that we need to check grows quite quickly with the number, n, of cities. (How fast does it grow? What if the distance function is not symmetric?) Still, as Bentley (1997) demonstrated in his nice introduction to the TSP, small instances can be solved by a series of tweaks to a straightforward enumeration algorithm. We will explore this in the first part of the assignment, to introduce common methods for improving combinatorial codes, and also to see the limits of brute-force approaches to combinatorial problems.

To begin, we need to choose a data structure to specify a TSP tour. A natural approach is to consider a permutation of length n which gives the order the cities appear as we go through the tour (we label the cities $0, 1, \ldots, n-1$). An easy way to check all tours is to run through all permutations and compute their length with a function `tourlen(int *perm)`, where `perm` is an array of ints that contains a permutation of 0 through $n-1$. A C-code for `tourlen` is:

```
static int tourlen(int *perm)
{
        int i, val;

        val = len(0,n-1);
        for (i=1; i < n; i++) val += len(i-1,i);

        return val;
}
```

where n is a global variable giving the number of cities, and len(i,j) is a function that returns the distance from city i to city j (the length of the *edge* (i, j)).

Bentley (1997) gives the following code to run through all $n!$ permutations, using a recursive call to find $(n-1)!$ permutations after we fix the first element at each of its n possible values.

```
static void easysolve(int *perm)
{
        int i, besttour = MAXINT;

        for (i=0; i < n; i++) perm[i] = i;
        searchit(n, perm, &besttour);
}

static void searchit(int m, int *perm, int *besttour)
{
        int i, val;

        if (m==1) {
              val = tourlen(perm);
              if (val < *besttour) *besttour = val;
        } else {
              for (i=0; i < m; i++) {
                    swap(i, m-1, perm);
                    searchit(m-1, perm, besttour;
                    swap(i, m-1, perm);
              }
        }
}
```

The function swap(i,j,perm) exchanges the cities perm[i] and perm[j].

Problem 1. Build the above code fragments into a working code for the TSP. Speed up the code by a factor of n by fixing the starting city once and for all (the TSP does not depend on the starting city) and storing it in perm[n-1], so that we enumerate only $(n-1)!$ permutations instead of $n!$ permutations. Using random geometric instances (for an n city instance, choose n random integral points in the n by n square), estimate the growth in the running time of your code as a function of n. What is the largest instance you could expect to solve in a day on your workstation? ∎

To test the correctness of your code, you can use the 8 cities given by the coordinates: $(4, 4), (2, 1), (3, 1), (5, 7), (5, 3), (4, 7), (2, 0), (3, 2)$. This instance has optimal value 16.

Problem 2. The above code computes the length of the tour from scratch for each permutation. This goes against one of the basic principles of computational work: *we should* *always try hard to avoid repeating the same calculations.* To help in this case, modify the code to keep track of the length of the partial tour as we go along, that is, keep track of the length of the path given by $\texttt{perm}[\texttt{m}], \texttt{perm}[\texttt{m}+1], \ldots, \texttt{perm}[\texttt{n}-1]$; this involves the use of an extra variable `psum` to pass into `searchit()`. How large of a problem can you now expect to solve in a day? ∎

Problem 3. The square root in the `edgelen()` function is time-consuming to compute (and our code calls this function very many times). Try to speed up the code by precomputing a distance matrix containing the distance between every pair of cities (so `edgelen()` becomes a simple look-up). Time your code to estimate your speed up. ∎

Problem 4. Note that the algorithm runs through all permutations, even if `psum` is larger than `besttour`. If this is the case, we need not search further with that partial tour and we can simply return in the function `searchit()`. This is an example of *pruning* a search tree. With this change (it should only be a line or two) how large of a instance can your code now solve? ∎

For this improved code, you can also use the 13-city Mercer County instance as a (larger) test case; it has optimal value 852 and the data for the instance is given on web page:

`http://www.math.princeton.edu/tsp/world/work/princeton.html`

Part 2: Heuristic Algorithms

In this part of the assignment we will work with a 588-city TSP consisting of locations in the New Jersey (see Figure 2). The (x, y) coordinates for this `nj588` instance can be found on the web page given above.

Instances with as many cities as nj588 are far too large to be treated with the enumeration algorithms we discussed in Part 1.

Problem 5. Estimate the time your best code will need in order to solve an instance having 588 cities? ∎

Nonetheless, many applications of the TSP are of this size, or even larger. A common approach for the TSP and for other difficult problems in discrete optimization is to consider *heuristics*, that is, methods that are not guaranteed to produce optimal solutions, but which are designed to produce fairly good solutions at least some of the time. There is a very large literature on TSP heuristics; a good survey is given in Johnson and McGeoch (1997).

Perhaps the most simple and natural heuristic for the TSP is the *nearest-neighbor algorithm*: start at any city and visit the nearest city not yet visited, then return to the starting city when all other cities are visited.

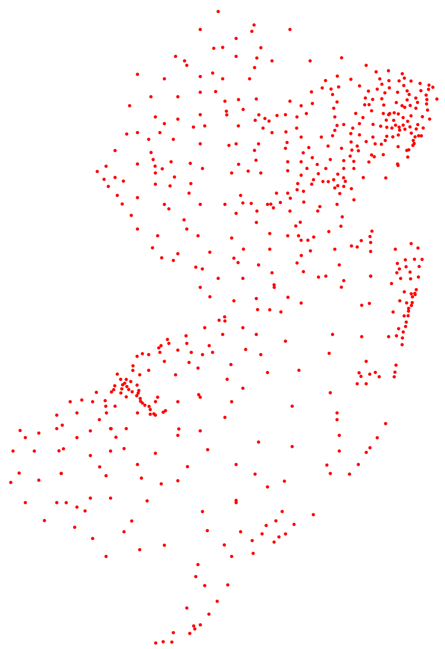

Figure 2: 588 Cities in New Jersey

Problem 6. Design a code for the nearest-neighbor algorithm and test the code on `nj588`. How does the running time and the solution quality vary with the choice of the initial city? Using random geometric instances, give an estimate of the growth in the running time of your code as a function of n, the number of cities. ∎

Nearest-neighbor is an example of a *tour construction* heuristic, in that it builds a tour from scratch. Other examples in this class include *addition* and *insertion* algorithms that start with a sub-tour on 2 or 3 cities, and add one city at a time until a full tour is obtained, for example *nearest-addition* adds the city that increases the sub-tour by the least amount amongst all cities not yet in the sub-tour.

Tour improvement methods form a second class of TSP heuristics; they take as input a specified tour and attempt to improve it by a sequence of operations. A classic example of this type of algorithm is called *2-opt* (Lin (1965)). The basic step of 2-opt is to delete two edges from the tour and reconnect the remaining fragments by adding two new edges, see Figure 3. (Note that once we choose the two edges to delete, we do not have a choice in which edges to add—there is only one way to add new edges to obtain a tour.)

The 2-opt algorithm repeatedly looks for 2-opt moves that decrease the cost of the tour tour, that is, the sum of the lengths of the two edges that are deleted is greater than the sum of the lengths of the two edges that are added. The algorithm terminates when there are no longer any improving 2-opt moves available. The final tour is said to be *2-optimal*.

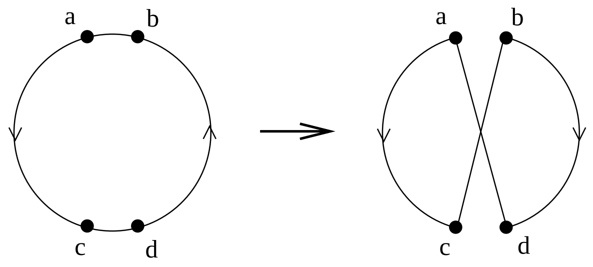

Figure 3: 2-opt Move

Note that a 2-opt move is the same as inverting a subsequence of cities in the tour, for example, in Figure 3 the initial tour is

$$a, \ldots, c, d, \ldots, b, a$$

while the final tour is

$$a, \ldots, c, b, \ldots, d, a$$

so the segment $(d, \ldots, b)$ is inverted. We call this operation `flip(d,b)` (we assume the tours are oriented, so the inversion is well-defined).

In dealing with combinatorial algorithms like 2-opt, it is often useful to separate the logic of the algorithm from the data structure that is used to represent the combinatorial object (in our case the tours). This allows us to easily experiment with different representations, which often determine the speed of the resulting code. In the case of 2-opt, it is enough to know that we have a `flip()` function, and functions `next(v)` (that returns the city that comes immediately after `v` in the tour) and `prev(v)` (that returns the city that comes immediately before `v` in the tour). Equipped with these functions, together with an initialization routine to feed in a tour, and a routine that returns the tour represented by the data structure, we can make an abstract implementation of the 2-opt algorithm. A study of such tour data structures is given in Fredman, Johnson, McGeoch, and Ostheimer (1995). On the web page given above, you can files flipper.c and flipper.h that implement `flip()`, `next()`, and `prev()` (these are taken from the *Concorde* computer code of Applegate, Bixby, Chvátal, and Cook (1999)).

Problem 7. Design a 2-opt code (you can use the `flipper` routines to maintain your tour). How do the results vary with your starting tour (compare nearest-neighbor starting tours with random starting tours)? Estimate the running time of the code as a function of n. ▮

Extra Credit

Problem X1. In the enumeration algorithm developed in Problem 4, we prune the search based on the sum of the distances for the partial tour fragment that has been constructed.

In particular, we ignore the fact that the remaining cities cannot be joined to the partial tour at 0 cost. There are a number of ways to include these remaining cities in the pruning technique, for example the cost of the smallest edge between the remaining cities can safely be added to the variable `psum` in the pruning test. One way to make a big improvement is to compute a minimum spanning tree over the cities not yet in the tour fragment, and add its cost to `psum`, that is we can prune the search if $\texttt{psum} + \texttt{spanning}(\textit{remaining cities}) \geq *\texttt{besttour}$. Run tests to show how this improves the running time of the algorithm. How large of an example can you now solve? What is taking the largest portion of time in your runs? Consider possible improvements, for example, can we avoid recomputing the minimum spanning tree for the same set of cities over and over? ∎

Problem X2. The are a number of ways to improve the running time of 2-opt, at the cost of producing slightly worse tours (that is, the tours may not actually be 2-optimal). One method, proposed by Bentley (1992), is to associate a 0-1 "don't-look bit" with each city, to allow us to skip over cities that did not lead to 2-opt moves in previous searches. Initially all bits are set to 0, and after an unsuccessful 2-opt search from a given city (that is, we consider the city as the first point in a 2-opt move) we set the bit for the city to 1; we only try searches from cities that have a 0 don't-look bit. This method is perhaps too aggressive since after some 2-opt moves we might want to search again from a previously unsuccessful city. Bentley suggests the following compromise: when we find a 2-opt move involving the cities a, b, c, d, we set to 0 the don't-look bits associated with each of the four cities. See how this technique improves the running time of your 2-opt code, and measure the corresponding increase in tour length. ∎

Problem X3. Another way to speed up 2-opt is to only consider 2-opt moves that start with an edge that is in a pre-defined set of "neighbors", for example, when we search from city a, we delete a tour-edge (a, b), and we only consider adding edges (b, c) where c is in the list of b's neighbors. A simple neighbor set to consider is the nearest 5 or 10 cities for each city in the instance. Again, how does this impact the running time and tour quality? Are there better choices for neighbors? ∎

Problem X4. Going the other direction, how can we improve the quality of the tours obtained by 2-opt? Martin, Otto, and Felten (1992) proposed a general method, called *chaining* to improve local optimization algorithms. The idea is to take the locally optimal tour, T, produced by 2-opt, and "kick" it so that the resulting tour is no longer 2-optimal. We then re-apply 2-opt to produce a new locally optimal tour T'. If T' is shorter than T, then we replace T by T', and otherwise we continue with T. We continuing kicking as long as we see improvements from time to time (a reasonable choice is to try n kicks, for an n-city instance). The suggested kick is to make a certain 4-opt move, called a *double-bridge*, that is illustrated in Figure 4. One possibility is to choose a random double-bridge by selecting 4 cities at random (and avoiding the case where tour neighbors are selected). This is a

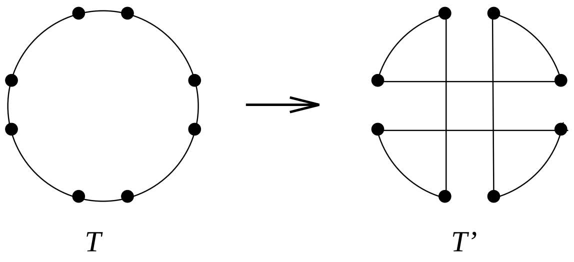

Figure 4: A double-bridge

powerful improvement on 2-opt and you should see the tour quality improve substantially. ∎

Problem X5. The most successful TSP heuristics to date are based on an algorithm of Lin and Kernighan (1973). The core of the algorithm is a search method for tentatively performing a sequence of `flip`'s such that each initial subsequence appears to have a chance of leading to a shorter tour. If the search is successful in finding an improved tour, then the sequence of `flip`'s is made and a new search is begun. Otherwise, the tentative `flip`'s are discarded and we begin a new search. A detailed description of the algorithm is given in Applegate, Bixby, Chvátal, and Cook (1999a). Design a variant of Lin-Kernighan and test it on nj588. (Note that the current champion amongst TSP heuristics is a Lin-Kernighan variant proposed by Keld Helsgaun (2000) less than a year ago—so despite the great body of work that has gone into the TSP, it might still be possible to make substantial improvements in existing methodology.) ∎

Problem X6. Design your own tour-construction or local-improvement algorithm for the TSP. How does it perform on nj588? What are the running time versus tour-quality tradeoffs? ∎

Problem X7. There are a number of well-studied instances of the TSP for which the optimal value is not yet known. The instances are specified in TSPLIB format (a standard way of giving the (x, y) coordinates) and they are available at the web pages:

`http://www.math.princeton.edu/tsp/world/countries.html`

`http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`

see also the page

`http://www.math.princeton.edu/tsp/unsolved.html`

Study how your codes behave on these instances. You might improve the best known result for an unsolved problem! ∎

Problem X8. Design a branch-and-bound code to find the optimal value for instances larger than those that can be handled by the direct enumeration methods described in Part 1. Using a spanning-tree (or 1-tree) lower bound, you should be able to solve instances with 40 or more cities. ∎

References

D. Applegate, R. Bixby, V. Chvátal, and W. Cook (1999). *Concorde.* Available at `http://www.math.princeton.edu/tsp/concorde.html`

D. Applegate, R. Bixby, V. Chvátal, and W. Cook (1999a). "Finding tours in the TSP". Available at `http://www.math.princeton.edu/tsp/papers`

J. L. Bentley (1992). "Fast algorithms for geometric traveling salesman problems", *ORSA Journal on Computing* **4**, 387–411.

J. Bentley (1997). "Faster and faster and faster yet", *Unix Review* **15**, 59–67.

M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer (1995). "Data structures for traveling salesmen", *Journal of Algorithms* **18**, 432–479.

K. Helsgaun (2000). "An effective implementation of the Lin-Kernighan traveling salesman heuristic", *European Journal of Operational Research* **126**, 106–130

D. S. Johnson and L. A. McGeoch (1997). "The traveling salesman problem: a case study in local optimization", in: *Local Search in Combinatorial Optimization* (E. H. L. Aarts and J. K. Lenstra, eds.), Wiley, New York, pp. 215–310.

Jünger, M., G. Reinelt, G. Rinaldi. 1995. "The traveling salesman problem", in *Handbook on Operations Research and Management Sciences: Networks* (M. Ball, T. Magnanti, C. L. Monma, and G. Nemhauser, eds.), North-Holland, Amsterdam, pp. 225–330.

E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, eds. (1985). *The Traveling Salesman Problem*, Wiley, Chichester.

S. Lin (1965). "Computer solutions of the traveling salesman problem", *The Bell System Technical Journal* **44**, 2245–2269.

S. Lin and B. W. Kernighan (1973). "An effective heuristic algorithm for the traveling-salesman problem", *Operations Research* **21**, 498–516.

O. Martin, S. W. Otto, and E. W. Felten (1992). "Large-step Markov chains for the TSP incorporating local search heuristics", *Operations Research Letters* **11**, 219–224.

G. Reinelt (1994). *The Traveling Salesman: Computational Solutions for TSP Applications*, Springer-Verlag, Berlin.

G. Reinelt (1991). "TSPLIB – A traveling salesman problem library", *ORSA Journal on Computing* **3**, 376–384. An updated version is available at `http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/`.