AN ABSTRACT OF THE THESIS OF

Robert W. Starr for the degree of Master of Science

in Computer Science presented on July 11, 1980.

Title: A Multi-Tour Heuristic for the Traveling Salesman Problem

Redacted for Privacy

Abstract approved: ______________

(Dr. Paul Cull)

This paper demonstrates the effectiveness of a heuristic for the Traveling Salesman Problem based purely on the efficient storage of multiple partial sub-tours. The heuristic is among the best available for the solution of large scale geometric Traveling Salesman Problems. Additionally, a version of the heuristic can be used to prove optimality of modest size problems (about 30 cities).

The heuristic gives an alternative to the standard technique of backtracking and starting over, by using main memory for alternate paths. It gives insight into the magnitudes of sub-tour sizes in a tour generation process, and gives a potential "assist" heuristic, to be used along with a more standard method in reaching very large scale Traveling Salesman Problems.

A Multi-Tour Heuristic for the Traveling Salesman Problem

by

Robert W. Starr

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed July 11, 1980

Commencement June 1981

APPROVED:

Redacted for Privacy

Professor of Computer Science

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of the Graduate School

Date thesis is presented July 11, 1980

Typed by Cathy Criswell Harper for Robert W. Starr

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

A MULTI-TOUR HEURISTIC FOR THE TSP

I. INTRODUCTION

The traveling salesman problem (TSP) has proved to be a hard nut to crack. Literally hundreds of different approaches have been leveled against this problem, both algorithmic and heuristic; some have been more successful than others, but none has been truly satisfying. There lies the lurking suspicion in the back of our minds that this problem just ought not to be this hard; after all, it is so easily defined, and, at least in the geometric case, people can do very well for $n < 100$. Could not modern computers, with processing speed and memory capacity increasing each year, do much better?

It is true that the TSP, even the geometric TSP, is NP-complete[1] (Garey, Graham, and Johnson : 1976). Thus, if NP not = P, which most computer scientists believe is the case, then the running time of any general algorithm for the TSP will be at least an exponential function

[1] I am using NP-complete in the broad sense to mean a problem which can be solved in deterministic polynomial time if and only if all problems in the class NP can be so solved. Thus I am using Cook's original definition of reducibility, but I am enlarging the class of problems considered to include problems other than decision problems. See Garey and Johnson (1979) pp 115-120 for a more detailed exposition of these matters.

of n. This does not necessarily make any such method impractical for moderately large n; for example, if T = 1.01 ** n (T the running time in seconds; n the number of cities) then a 500 city problem would require only 145 seconds of computer time. But, unfortunately, most exponential algorithms, including those for the TSP, do not behave like 1.01 ** n, but more like 2 ** n. Thus, in practice, all exact algorithms have run into the "exponential" bind, so that problems of n > 100 often prove impossible (see below).

Because of the difficulties with exact algorithmic approaches, several researchers have tried heuristic methods and have done better. Most notable in this regard is the local search technique of Kernighan and Lin (1973), which often gets an exact (or virtually exact) solution for n < 100, and runs in time proportional to n ** 2.2. However, even this method has theoretical limitations (see Papadimitriou and Steiglitz : 1977, and Rosenkrantz, et al. : 1977), which limit its effectiveness for a certain class of problems.

In this paper I will describe yet another heuristic which was leveled against the TSP. The general approach, as far as I can determine from the literature, has never before been attempted seriously. My goal was to handle moderately large geometric TSP problems (100 - 200 cities, or possibly higher) with a method that would in practice usually produce an exact tour, or something extremely close. I wanted the program to run reasonably fast (within a few minutes of

CPU time on a large scale computer) but not necessarily be greatly constrained by main memory limitations.[2] I felt this was reasonable since even megabyte micro-computers are expected in the not too distant future. I wanted to take a fresh approach to the problem; one that did not rely on tight computer processing logic, but rather on the quick availability of a large body of information.

The basic idea behind the heuristic is to build many tours simultaneously. The presupposition is that among the n! possible tours, only a relatively limited number are reasonable (see Appendix 1). More specifically, as the heuristic goes through the tour-building process it only needs to retain a (relatively) small number of "good-looking" sub-tours, at each step, in order to be relatively confident that <u>the</u> optimal solution will, in fact, not be thrown out in the pruning process. It does no backtracking (as do Kernighan and Lin (1973)), nor does it modify sub-tours as it goes (as do Karg and Thompson (1964)); it merely holds on to a large number of sub-tours, at each step, keeping those that look the most promising and pruning the rest. The power of the heuristic is dependent on a means

[2] In fact, I felt that a large memory capacity could be of use in this problem. Most combinatorial programs developed today do not use much memory other than, say, a distance matrix; and thus, essentially perform only in proportion to the power of the processor. They process their data serially; if alternate paths are considered at all they resort to backtracking logic. However, the use of main memory to handle alternate paths, instead of using backtracking logic, has much to commend it, especially as the cost of memory decreases.

of economically storing large numbers of sub-tours[3] (in fact, billions of them), and on discovering effective pruning criteria. These matters will be dealt with in more detail in section III. I will first, however, trace some of the past literature from which the multi-tour heuristic was derived (although, in fact, many of the ideas were arrived at independent of this literature).

[3]Though, in fact, the "large numbers of sub-tours" we keep is relatively small in comparison to the total number of sub-tours possible at each step.

II. HISTORICAL ROOTS

The Traveling Salesman Problem was first formulated in 1932 (Grotschel and Padberg : 1977). It can be defined as follows (from Bellmore and Nemhauser : 1968):

"...given a nonnegative integer n and an n-dimensional square matrix $C = \{c_{ij}\}$. Any sequence of $p+1$ integers taken from $(1,2,\ldots,n)$, in which each of the n integers appears at least once and the first and last integers are identical is called a tour. A tour may be written as

$$T = (i_1, i_2, i_3, \ldots, i_{p-1}, i_p, i_1).$$

By a feasible solution to the traveling salesman problem, we mean a tour. An optimal solution is a tour such that

$$Z(t) = \Sigma_{(i,j)\varepsilon t'} c_{ij} \text{ is minimized}$$

where $t' = \{(i_1,i_2), (i_2,i_3), \ldots, (i_{p-1},i_p), (i_p,i_1)\}$

is the ordered pair representation of t."

Alternative, but similar definitions have also come up in the literature. For example, Rosenkrantz, et al. (1977) define the TSP as do Bellmore and Nemhauser, but in addition require that the distance matrix C also satisfy the triangle inequality ($c_{ik} \leq c_{ij} + c_{jk}$ for $1 \leq i \leq n$, $1 \leq j \leq n$, $1 \leq k \leq n$). A well-known sub-class of the TSP is the geometric TSP. In this problem, the distance matrix $C = \{c_{ij}\}$ is derived from n points in the two-dimensional plane.

For the purposes of this paper, it is not essential which definition is used. However, it is doubtful that the heuristic (nor any other point insertion method) would be effective for problems not satisfying the triangle inequality. For illustrative purposes, all my examples were taken from the class of geometric TSPs.

The point of departure for this heuristic is the insertion technique of Karg and Thompson (1964). This technique can be briefly summarized as follows:

```
1. Begin with a random order of the points {p(i), i = 1, n}.
2. Let T(3) = {p(1), p(2), p(3)}.
3. For k = 4 to n
   T(k) = the shortest sub-tour which can be formed by
          inserting p(k) between two points of T(k-1)
   Eventually we derive T(n).
4. Take another random order of the points and repeat
   steps 2 and 3, constructing another T(n). Do this
   several times until the alloted computer time runs out.
5. Choose the shortest of these T(n) as the solution.
```

This is essentially the heuristic used by Karg and Thompson. They supplement it by exchanging pairs of sub-tours as they are being built, but this is not germane to the following discussion.[4]

The multi-tour building heuristic is similar to that of Karg and Thompson in that it is an insertion method. It builds a sub-tour of length k by inserting a city between two cities of a previous sub-tour of length k - 1. Every sub-tour it creates is created in that manner.

The heuristic does not, however, start with a random order of points. It is not given any second chances, so it is necessary for it to use the most useful order of points possible (note that "order of points" in this context means the order in which the points are introduced into the tour-building process, not the order in which the paths will be traversed). The order chosen is like the "farthest insert" method described by Rosenkrantz et al. (1977) (see section III.A. below). As he says, this method gives a series of T(k) which begin very quickly to approximate T(n), both in length and shape. It is especially useful to the multi-tour heuristic that the lengths of these sub-tours increase quickly; it is also useful to avoid bringing two close neighbors both into the sub-tour until as late as possible

---

[4]Note also the work of Norback and Love (1977), who start with the Karg-Thompson approach. They, though, are concentrating on speed in their heuristic, rather than great accuracy.

(see section III for elaboration). The "farthest insert" approach does well in both of these aspects.

I also find a parallel (or pre-cursor) to the multi-tour heuristic, though in a more remote sense, in the dynamic programming algorithm of Held and Karp (1962). In their construction of the optimal solution they build links of nodes, of increasing length. They build these linkds in parallel, systematically adding to the links point by point. However, when they add a node to a link, they add it at the <u>end</u>. They do not retain the entire link as they go, but only the length of the link and its end points; the link is later constructed in a separate pass. This keeps their algorithm simple, and effective for an exact solution for 10 - 15 city problems; but it does not seem to be adaptable to the pruning techniques I used in my tour-building.

There is another interesting similarity between my heuristic and that of Held and Karp's dynamic programming. They use up memory at a rate of 2 ** n: an n-city problem requires 2 ** n storage locations. In the multi-tour heuristic assuming <u>nothing</u> is ever pruned, 2 ** (n-2) nodes or storage locations are retained (although some of the storage locations are larger than others). This is so because both methods use <u>unordered</u> subsets of cities as the tour is being built.

III. DESCRIPTION OF THE ALGORITHM

I will describe the algorithm in three stages. This corresponds to the stages in which the idea formulated in my mind.

A. I began with the simple insertion heuristic propounded by Karg and Thompson (1964), refined by the "farthest insert" variation described by Rosenkrantz et al. (1977). In this variation, the heuristic does not start with a random sequence of points (each to be added to the preceding sub-tour); instead, it starts with <u>one</u> fixed, specially designed sequence (say {p(1), p(2), ... p(n)}). These points are defined as follows:

1. p(1) and p(2) are two points such that D(p(1), p(2)) is a maximum. (D is the distance function).
2. For k = 3 to n

   p(k) is chosen so that min(D(p(k), p(i)), i = 1, k - 1) is maximized.

   In other words, p(k) is chosen to be as far away as possible from the points {p(1), p(2), ..., p(k-1)}.

Next the tour-building process is initiated. T(3), obviously, would be {p(1), p(2), p(3)}. Then for T(4), with the insertion method, there are three possibilities: {p(1), p(2), p(3), p(4)}, {p(1), p(2), p(4), p(3)}, or {p(1), p(4), p(2), p(3)}. But, in fact, the heuristic doesn't choose any <u>one</u> of these three sub-tours at this time; it keeps all three of them (calling them, say, T(4,1), T(4,2), and T(4,3)).

In the same way, multiple T(5)'s are created. Each one of the T(4)'s will generate four T(5)'s. Thus, there will be 12 T(5)'s in all, labeled T(5,1), T(5,2), ..., T(5,12). These are stored in memory, in simple arrays.

Moving right along, the heuristic creates the T(6)'s; there are 60 of them. Then the T(7)'s (360 of them), and so on. However, as is apparent, if this course is continued for very long, the computer will run out of memory (and have enormous processing time). So the number of sub-tours kept must be limited at each step (i.e., for each tour size). This limit I will call L. Thus (assuming L <= 360), there are L T(7)'s: T(7,1), T(7,2), ..., T(7,L).

In setting this limit, L, I had to establish a criteria for determining which of the T's to keep, and which to drop (prune). I based this criteria purely on the basis of total length of sub-tour. Thus, the heuristic keeps the L shortest of the possible sub-tours for T(k) at each k.

This technique did moderately well for a simple heuristic (see appendix 2 for a sample problem run with several values for L). However, I felt I could do better. One of the problems with this heuristic is what I call the "multiplicative effect." Assume that we are up to p(8) in our insertion logic, and these 8 points are as in Figure 1.

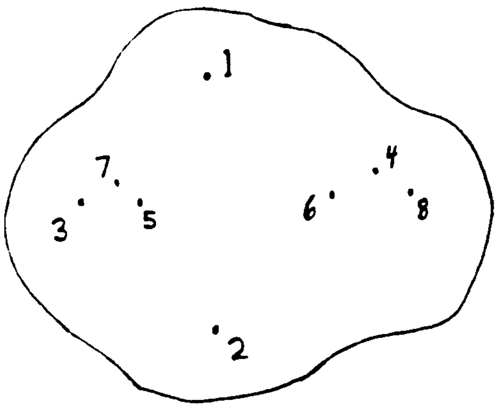


Figure 1. A set of points showing the "multiplicative effect."

Then consider the half sub-tour from 1 to 2 via 3, 5, and 7. This can be traversed (1, 3, 7, 5, 2) or (1, 5, 7, 3, 2) or (1, 7, 3, 5, 2). Each of these three are "good" paths (we are not being precise here) in contrast to, say, (1, 3, 5, 7, 2). Similarly, the other half sub-tour can be traversed (2, 6, 8, 4, 1) or (2, 8, 4, 6, 1) or (2, 6, 4, 8, 1). Thus the entire sub-tour can be traversed:

(1, 3, 7, 5, 2, 6, 8, 4, 1)
or (1, 3, 7, 5, 2, 8, 4, 6, 1)
or (1, 3, 7, 5, 2, 6, 4, 8, 1)
or (1, 5, 7, 3, 2, 6, 8, 4, 1)
or (1, 5, 7, 3, 2, 8, 4, 6, 1)
or (1, 5, 7, 3, 2, 6, 4, 8, 1)

or (1, 7, 3, 5, 2, 6, 8, 4, 1)

or (1, 7, 3, 5, 2, 8, 4, 6, 1)

or (1, 7, 3, 5, 2, 6, 4, 8, 1).

There are nine, or 3 X 3, "good" sub-tours, as a result of the fact that there are three good paths on either side. For large sub-tours, this difficulty becomes especially acute, as close "good" variants pop up in several corners of the sub-tour. Thus the size of L required to keep "good" entire sub-tours (the concatenation of "good" partial sub-tours) becomes prohibitively large.

B. It seemed necessary to overcome this "multiplicative effect." And, in fact, this did not appear to be all that difficult. All that would have to be done was to isolate the various sections of the graph, and have each section keep track of its own "good paths." Thus, if, at one point, section A kept 5 sub-tours, section B kept 3 sub-tours, and section C kept 6 sub-tours, then only 14 sub-tours in all would have to be kept, instead of the 90 that would be necessary if the graph were not sectioned off. This was certainly a big savings.

The rub is, of course, that it is impossible to know how to section the graph. If, as in Figure 2, the graph is segmented into three parts as shown, it is presupposed that the optimal tour first traverses all the points in section A, then those in section B, and then those in section C. If the optimal tour, in fact, at any place, oscillates between these sections, then it will never be found. This sub-division itself, by virtue of being set up in this way, will have eliminated the optimal tour. So, then, it is necessary to sub-divide in the right way; and yet it is impossible to know what the right way of sub-dividing is.

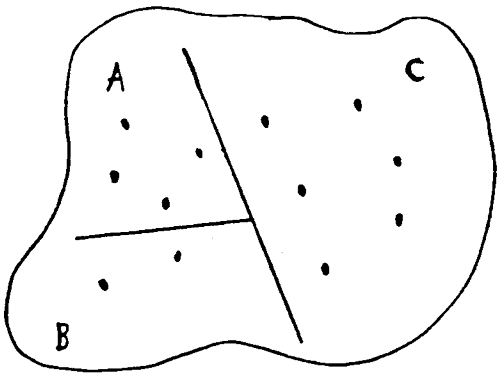


Figure 2. One subdivision (SUBD1) of the first 12 cities entered into a problem.

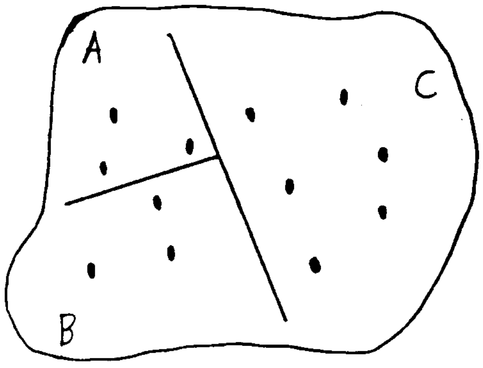


Figure 3. Another subdivision (SUBD2) of the 12 cities in Figure 2.

The answer, of course, is to keep track of multiple ways of sub-dividing. So, as shown in Figure 3, there is another way of sub-dividing these 12 cities (call this SUBD2). This sub-division will be kept, as well as the one shown in Figure 2 (SUBD1). In fact, several sub-divisions will be kept. Each of these sub-divisions will maintain its several sections of cities; each section of cities will keep track of multiple tours through its cities.

These sub-divisions do not have to be processed entirely independent from each other. So, for example, section C of SUBD1 and section C of SUBD2 consist of the same set of cities. It would be possible then for SUBD1 and SUBD2 to both set a pointer to this particular set of cities; and so the sub-tours for these cities, along with their pruning criteria, etc., need only be held in one place.

I did not implement this variation, although it seemed to have promise. Another, much better, and more general method became apparent to me at that time.

C. This method, too, breaks the "multiplicative effect." It, too, sections off the graph of cities, and keeps track of multiple sub-divisions. But it also sub-divides the sub-divisions; in fact, its main work, in a sense, is that of sub-dividing.

The sub-divisions described in section B were fairly arbitrary. At that time (I was thinking of the geometric case), I was considering using the convex hull to determine the sub-divisions. So Figure 4 shows two sub-divisions of a set of points (note that here the points of the convex hull are actually <u>between</u> two sub-divisions -- the dividing point -- rather than <u>in</u> one of them). This all looks fine. But suppose the set of points is as in Figure 5. Here the convex hull seems to be less useful. So what do we do? I never found a satisfactory answer.

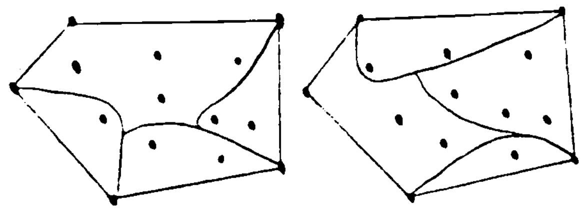


Figure 4. Two subdivisions of a set of points based on their convex hull.

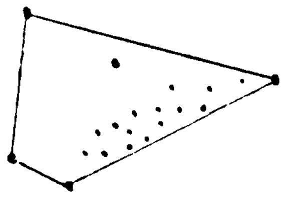


Figure 5. An example where subdivision based on the convex hull is ineffective.

The method of this section does not have such a problem. It lets its "order of entering points" precisely determine the subdivisions. It causes new sub-divisions any time a point is added.

What happens is this. The heuristic starts off with three points, using the "farthest insert" method, {p(1), p(2), p(3)}. p(1) is the beginning and end of the tour (the home city); and p(2) is the "mid" point of the tour. (By "mid" point we merely mean a point along the path from the beginning to the end point. The mid point is the one that is first inserted between these two points.) Additionally, p(3) is the mid point of the p(1) - p(2) path (not the p(2) - p(1) path). This choice just determines orientation; only one sub-tour is possible with three cities.

Then p(4) is inserted into the problem. As noted in section A, there are three possible sub-tours now. However, the heuristic does not split into three (or point to three) sub-tours directly. Instead, it makes two two-way choices. First, it determines that p(4) can be either on the "left" side of the p(1) - p(2) - p(1) path (the side that contains p(3)), or it can be on the "right" side. Now the placing of p(4) on the right side is fairly straightforward: it leads merely to the path p(2) - p(4) - p(1). However, the placing of p(4) on the left side is a bit more complicated; it would have to be inserted somewhere in the path p(1) - p(3) - p(2). And clearly, there are two different places it may be placed: between p(1) and p(3), or between p(3) and p(2). So here is the second two-way split. The organization of this data is shown in Figure 6.

Each of these nodes represents a set of paths. Each path of this path set will contain the points listed with the node, both the three points on top and those in parentheses.

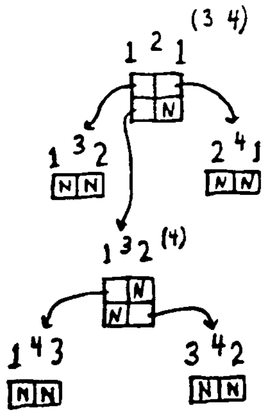


Figure 6. The splitting of four points into path sets.

The three numbers on the top of each of these nodes represent the beginning, mid, and end points of the path (or each path of the set of paths). The numbers in parentheses represent additional points which belong in each of the paths of the path set. Those nodes which have no numbers in parentheses represent a "bottom" (bottom of the tree): a path set consisting only of one path, that of the three points. These nodes contain no pointers.

The nodes which contain one or more numbers in parentheses represent those path sets which contain more than three points. These need to be broken down into nodes representing "half paths" (or half path sets) of this path set. However, the split into half path sets is not unique; in fact, 2 ** m different ways of splitting are possible (where m is the amount of points in the parentheses).[5] Each of these ways of splitting is represented by a layer in the node.

Each way of splitting represents one possible division of the m points into two places: left of the mid point, or right of the mid point. So (see Figure 7) the points {p(9), p(12), p(13)} have to be split among the two paths p(4) - p(8) and p(8) - p(9). Eight ways (or 2 ** 3) are possible, as shown.

---

[5]The top node, however, can only have 2 ** (m - 1) ways of splitting. This is because point p(3) is forced on the left side by convention, and thus it does not contribute to the two-way choices.

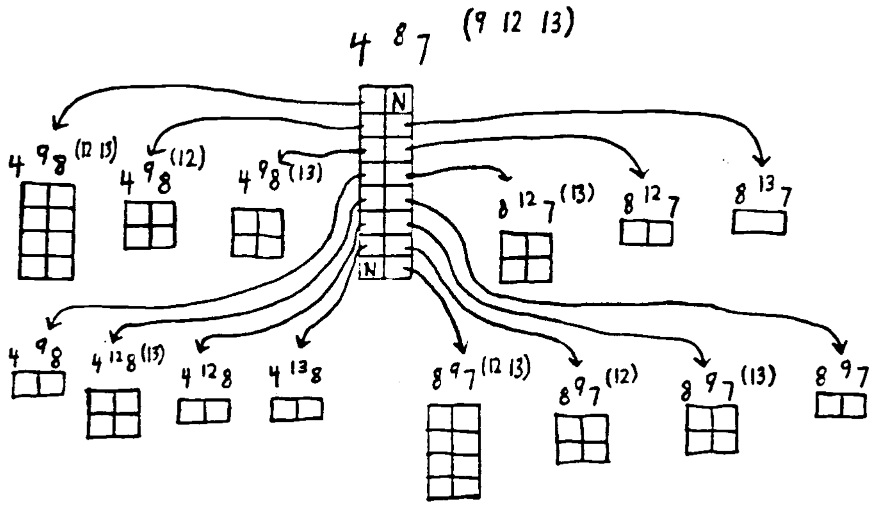


Figure 7. A complete path set of six points.

Note that some nodes can be pointed into from more than one place (see Figure 8 for an example).

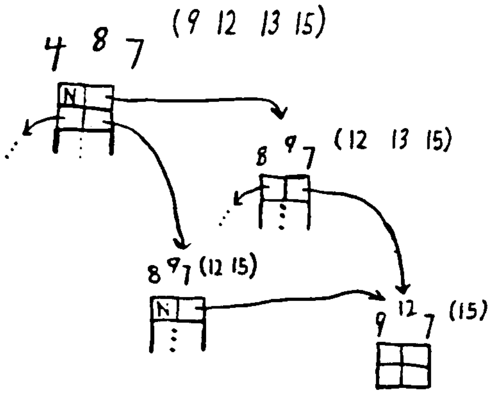


Figure 8. A node that is pointed to from more than one place.

These nodes, of course, eventually have to be pruned to keep the processing and memory requirements within reason. I now turn to a discussion of the pruning criteria.

At first I thought it would be a good idea to prune by limiting the size of the nodes. So the top node would have a maximum of, say, 100 layers (or pairs of pointers). The second level nodes (those pointed at by the top node) would be allowed 10 layers each, the third level nodes 5 layers each, and all other nodes 3 layers each. Of course, many sets of parameters were tried here. But although this method did manage eventually to solve a 60 city problem, it had some inherent disadvantages, and was eventually dropped. One difficulty (A) was that not all second level nodes are alike; some have more points than others. Additionally, (B) some of these second level nodes contributed to the best of the total sub-tours, whereas others had just barely escaped the pruning cut-off point. It would have seemed good to give extra consideration (more layers) to those second level nodes which contributed to a better total sub-tour.

The answer to difficulty A would be to consider the number of points in the node as a factor in making the layer-allowance. The answer to difficulty B would be to base the cut-off on <u>length</u> of sub-tour rather than on layer-allowance.

I did not pursue the first of these tacks, although it certainly had potential to improve the heuristic, and could be incorporated at a later time. I felt at the time that the second tack would prove more fruitful.

Each of the nodes, then, was to retain the length of its best path. (This length would be stored in the header of the node; see appendix 3 for a detailed description of the node structure.) Refer to Figure 9. The p(12) - p(17) - p(13) node here has 5 layers. Each of these layers points to a left half path set and a right half path set; and each of these path set nodes has <u>its</u> best path length in its header. The length, then, that appears in the header of the p(12) - p(17) - p(13) node is the <u>minimum</u> of the sums of the lengths of the left and right sub nodes; in this case, the 4th layer (248.82 + 282.91 = 531.73) gives the minimum. Of course, for a bottom node, the length is merely the sum of the two lines connecting the three points.

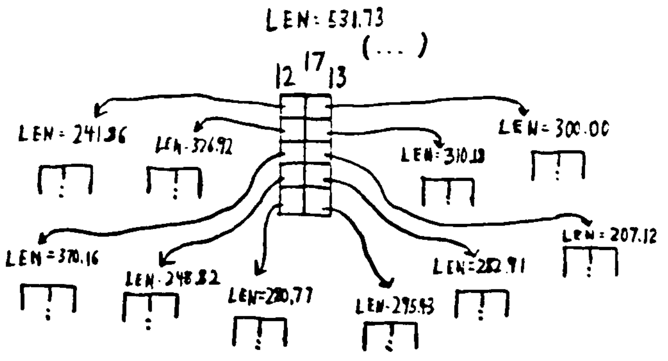


Figure 9. The derivation of lengths on the nodes.

The idea, then, was to prune those layers of a node whose length (the left side plus the right side[6]) was too long. Of course, a percentage of the best length could be used as the cut-off; for example, every layer whose length was greater than 1.3 times the best length would be dropped. And, in fact, this technique was used (in part). However, it did not answer difficulty B above.

To answer difficulty B it was needed to keep the length of the complement of a node, that is, the length of the piece of the sub-tour which includes all points currently entered into the problem except those in the node. This length will be called the length from above (LFA). For example, consider the node of Figure 7, and assume that 14 points have been entered into the problem so far. Call this node N(1). N(1) represents the path including points (all these points and only these points) {p(4), p(9), p(8), p(12), p(13), p(7)} with p(4) first, p(7) last and the others in one or more arrangements. But the complete sub-tour includes also points {p(1), p(2), p(3), p(5), p(6), p(10), p(11), p(14)}; and N(1) has to fit in as a piece of the whole.

N(1) is then part of a super-ordinate node, which in turn is part of another node, etc., until the top node is reached (see Figure 10).

[6]Note that the length of an empty pointer (represented by an N within the layers) is merely the distance between the mid point and the beginning or end points (beginning point if N is on the left, end point if N is on the right).

We see here that N(1) is pointed into "from above" from two other nodes N(2) and N(3). Now N(2) points to N(1) from its right side, and its corresponding left side has (best) length 220.41. Moreover, the length from above (LFA) of N(2) is 591.00. Thus, N(1)'s LFA, as derived via N(2), is 811.41. As for N(3), it points to N(1) from its left side, and its corresponding right side has length 84.61; and its LFA is 630.45. So N(1)'s LFA as derived via N(3) has

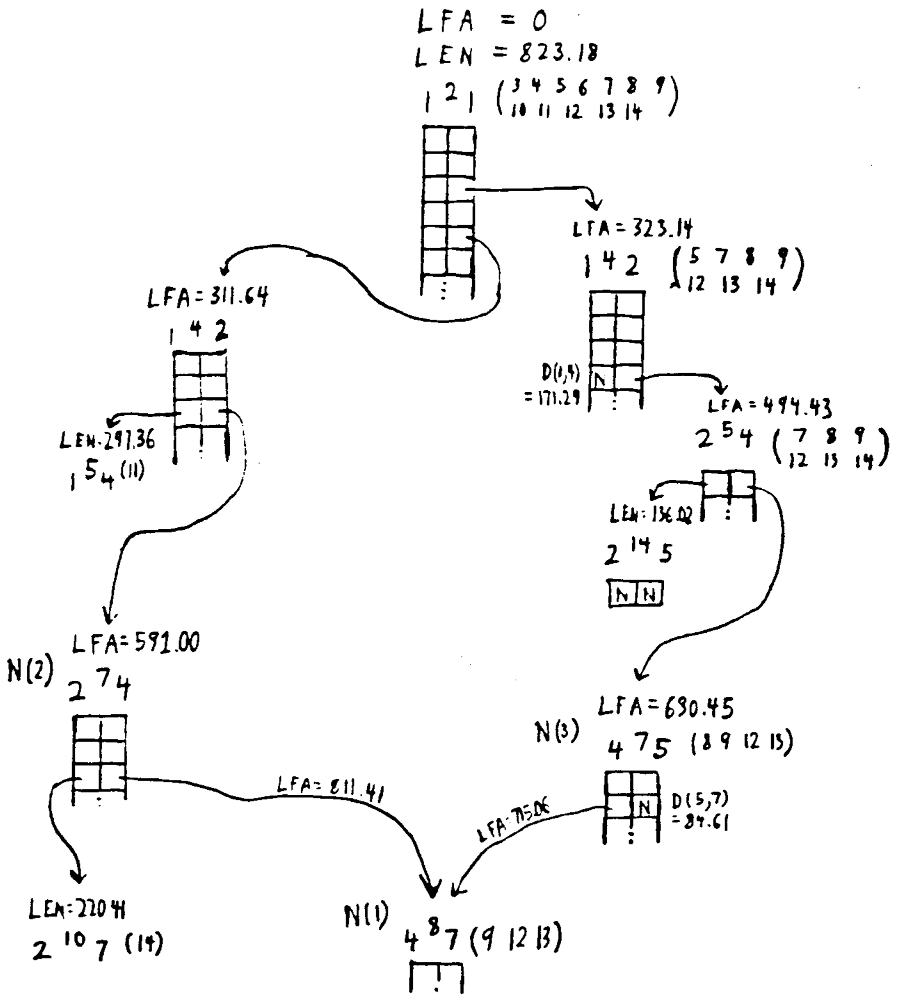


Figure 10. The derivation of the LFA's (Length from Above) on the nodes.

length 715.06. The LFA assigned to N(1) is then the lower of these two values. (Note that the LFA's are derived recursively from the top. Obviously, the LFA of the top node is 0).

The best <u>total</u> length, then of any sub-tour that a node could be part of is the sum of its length and its LFA. This is what was used to compare against the cut-off value. The cut-off value was an absolute value which increased as each point was added to the problem. Various criteria were used to create these values.[7]

One further modification was made (refer to Figure 11). Suppose that at this time the cut-off value is 700. Since the LFA of N(1) is 400, the maximum length of any of its paths is 300. N(1) has several paths, the best of which totals 208.36 and the worst 293.03.

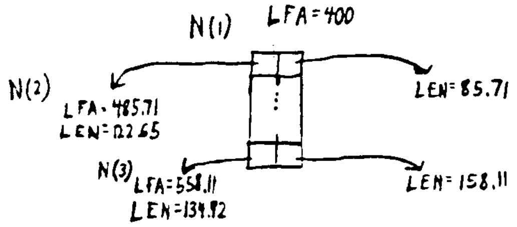


Figure 11. An example of further pruning on the "good" children

[7]For a small enough problem, this cut-off number can remain constant, thus giving a proof that the answer obtained is indeed the optimal. This, in fact, was done for a 30 city problem (see appendix 1).

Now the LFA of N(2) is 485.71, so that the maximum length path kept in this node will be 214.29. This is quite a bit more than the best path length, 122.65. So N(2) therefore is likely to contain many paths, even some of which are terribly longer than its best. N(3), on the other hand, since its LFA is 558.11 can only have a path whose length is <= 141.89. Since its best path length is 134.92, there is very little leeway. So N(3) will most likely have few paths, maybe only one.

Of course, this situation is not unwarranted. N(2) should have better consideration since it contributes to the best path set in N(1). But, in fact, the advantage given here to N(2) seemed to be too much. A path of length 214.29 in N(2) did not seem a good one, in balance, to keep around. To see this more clearly, note that the paths in N(2) would be attached to those in N(4). And N(4)'s LFA is 522.65, so that N(4) could also have a large path, one of length 177.35. Since a path attached to one of length 214.29 would produce a 391.64 size path, which is way beyond the 300 ceiling for this node. So allowing the full leeway on the best path of the node seemed excessive; instead, only a percentage was allowed. (Many were tried; in the end I settled on about 50%).[8]

[8]More precisely, I let this percentage depend on the LEVEL of the node (see appendix 3 for definition of LEVEL).

This, then, is the logic finally used for the heuristic. Now I will turn to a discussion of some "problem areas": conditions in a graph which would cause difficulties for the heuristic.

IV. ADVERSE CONDITIONS

I will describe here two conditions. These can be easily visualized in a geometric problem, although the same principles will generally apply to any TSP satisfying the triangle inequality. It is striking how simple these "adverse conditions" seem to be conceptualized visually and in the human mind, and yet are so seemingly difficult to capture effectively in computer logic (see section VI for further comments along this line).

Let me begin by drawing a graph that will cause the heuristic obvious difficulties (refer to Figure 12). We see here that, after 5 points have been entered into the problem, the path p(2) - p(5) - p(4) - p(3) does not look good. In fact, this path will even cross itself. But, looking at all the points, it seems that certainly p(2) - p(5) - p(4) - p(3) will be part of the optimal tour. So, at this step of k = 5, we see that the heuristic will be sorely tempted to prune its (eventually) optimal sub-tour.

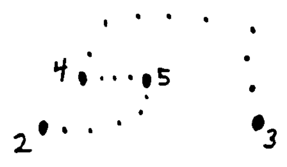


Figure 12. An example of a "snaking" optimal path.

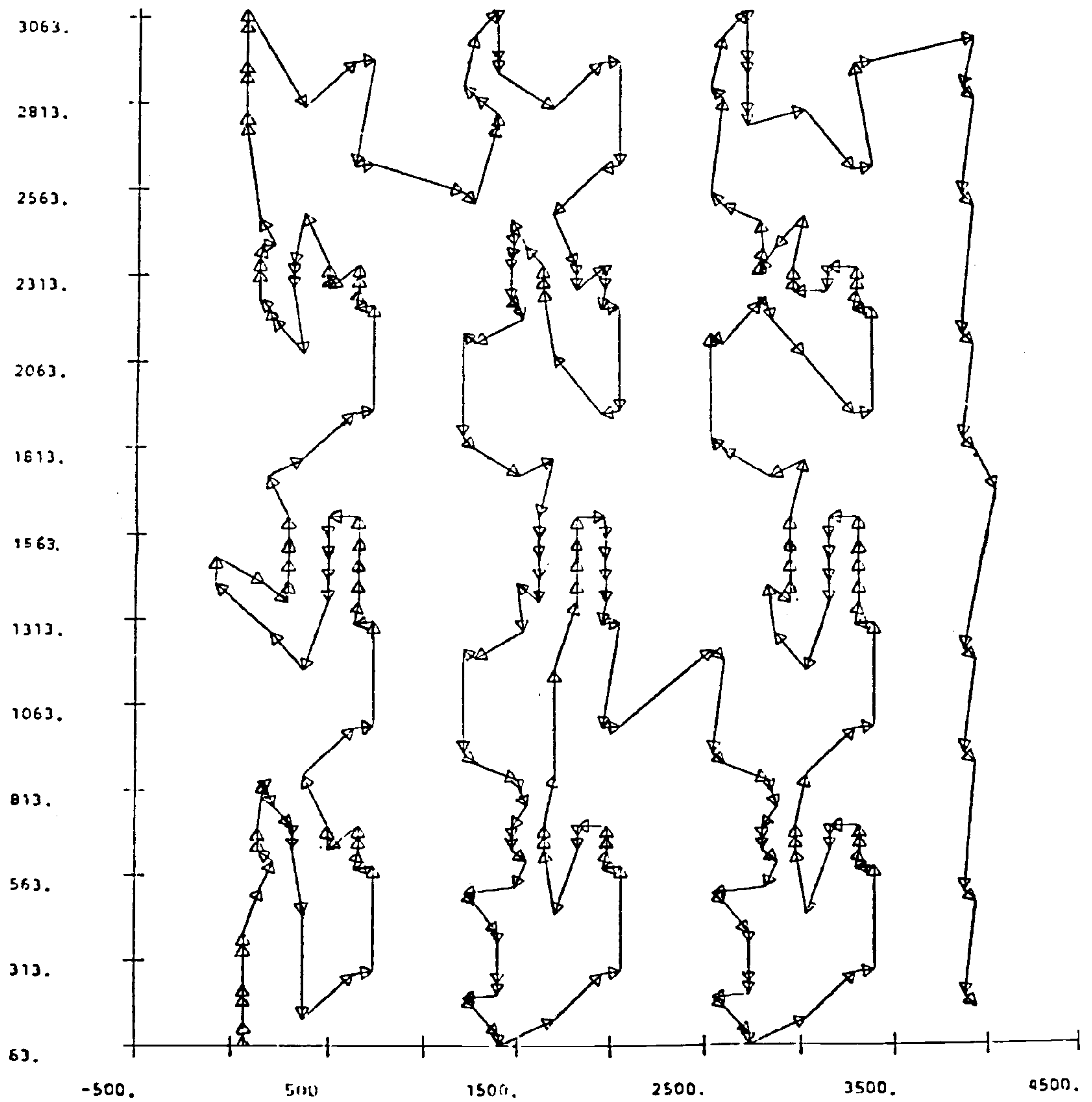


Figure 13. 318-City Problem

In a large problem, these snaking type paths can occur on numerous occasions (see Figure 13, a reproduction of Padberg's solution to the Kernighan-Lin 318 city (open tour) problem (Padberg and Hong : 1977), for an example[9]). This type of snaking is also similar to the "diamonds" discussed by Papadimitriou and Steiglitz (1978). I will now go on to briefly describe the two conditions, both of which are caused by snaking optimal paths.

A. Suppose, at any step, the new point to be inserted into the problem forms a cusp on the (eventually) optimal sub-tour (refer to Figure 14). After the first 6 points are added there exist, among others, the "good" paths P1 = {p(1) - p(6) - p(3) - p(2) - p(4) - p(5)- p(1)}, P2 = {p(1) - p(5) - p(6) - p(3) - p(2) - p(4) - p(1)}, and P3 = {p(1) - p(6) - p(3) - p(2) - p(5) - p(4) - p(1)}. Now it can be "seen" (I am assuming here that personal intuition and visual perspective cannot entirely be discounted) that P2, in fact, is on the optimal tour. However, when point p(7), possibly, will not be accepted as part of P2;[10] its insertion would cause a large increase

[9]But note that these snakes are in local areas, so that our overcoming of the "multiplicative effect" might give us some power here.

[10]I am not being precise here; P2 is actually a connection of paths pointed at by the tree strucutres defined in section III.

in length. On the other hand, p(7) easily fits within P1 or P3 and thus will not cause a great increase of length if it is placed within either of these paths. So, when pruning time comes, and the competition is fierce, P2 may very likely be dropped, destroying any chance of the optimal tour being found.

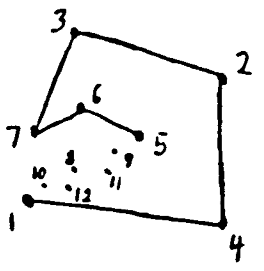


Figure 14. An example showing the problem of cusps

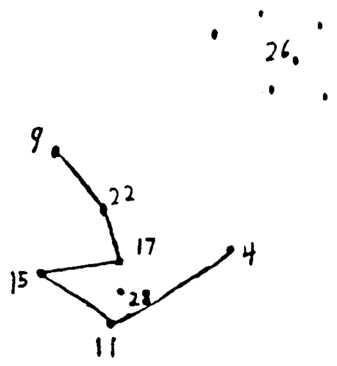


Figure 15. An example showing the importance of look-ahead.

B. Refer to Figure 15. Here we see that path P4 = {p(4) - p(11) - p(15) - p(17) - p(22) - p(9)} is chosen over P5 = {p(4) - p(17) - p(11) - p(15) - p(22) - p(9)}. (Note than P4 and P5 are both in the same path set, say PS1). However, if we observe the point p(28) we see that P5 may possibly be a better choice than P4. Of course, this presents no problem if P4 and P5 can both be kept. But suppose, say, p(26), in an entirely different part of the graph, is inserted, and does not fit in well with the complement of PS1;

suppose, in fact, that in the pruning PS1 just barely gets past the cut-off, but that its "lee-way" (see section III) is so small that it must drop P5. Now that is unfortunately, because if the heuristic could just look ahead to point p(28), it would determine to keep P5 instead of P4.

In this heuristic <u>no</u> use at all is made of the points not yet entered into the problem. So, as in Figure 14, the points {p(8), p(9), p(10), p(11), p(12)} could not be used at all in evaluating P2. If some of these "useful" points are entered too late, the heuristic can fail (i.e., not produce the optimal tour).

V. SUMMARY OF RESULTS

Below is a tabulation of the results and running times of the multi-tour heuristic on eight problems taken from Krolak et al. (1970). The same pruning factors were used in each of these runs; each run used 2.25 megabytes of virtual memory on an IBM 3033 processor. Note, however, that the 100 city problems could run in less memory (usually less than 1 megabyte, depending on the problem); additionally, the heuristic ran optimally for K24 and K28 with tighter pruning factors and thus less CPU time (about half). For this tabulation, though, I wanted to run the same program for each problem.

Problem	Number of Cities	Known Optimal Tour	Multi-Tour Result	Krolak's Result	CPU time (minutes: seconds)
K24	100	21285.5	21285.4	21282.0[11]	1:41
K25	100	22148.0	22194.9	22193.3	2:08
K26	100	20744.7	20744.7	20852.3	1:30
K27	100	21294.3	21419.1	21294.3	4:51
K28	100	22068.5	22068.5	22115.6	5:51
K30	150	-------	25524.8	26794.0	4:56
K32	200	-------	29546.3	26563.1	7:17
K33	200	-------	29468.3	29678.2	10:26

[11]There is a slight discrepancy here between my length and Krolak's. The tours, in fact, are the same. I suspect there is a rounding problem here; I use full floating point in all my length calculations; perhaps Krolak doesn't.

As can be seen, the heuristic works well on these problems, getting the optimal tour on three of the five 100 city problems. On the three larger problems this heuristic comes out with a tour shorter than that of Krolak. It is not my intention here to compare this method with others, counting machine CPU seconds. What is important is that the heuristic can "solve" problems of this size, using no sophistication other than the holding of multiple sub-tours. It is also interesting to note that the tour of a 30 city problem can actually be proven optimal (see appendix 1) via this technique.

The heuristic as it stands is not in general as effective practically as that of Kernighan-Lin (1973). It uses much more main memory (but see note 1) and is not as accurate per CPU second (however I believe it is fair to say the methods are roughly comparable in power, at least for the geometric case). It is likely that with further research in pruning techniques and in altering building sequence, the heuristic would be improved, perhaps greatly. But I am not sure of this.

My hope is that the ideas presented here will prove a spring-board for others. The multi-tour technique is not intrinsically incompatible with other approaches. I haven't yet pieced this together, but I would be greatly interested in seeing where this idea would lead one of those who has worked on the problem before, using a common technique (e.g., the assignment problem). However, in the next section I give some of my own ideas for the improvement of the basic method.

VI. POSSIBILITIES FOR IMPROVEMENT

There are several tacks which suggest themselves as possibilities for improvement. Some seem relatively easy to implement; unfortunately, these appear, at least to me, to offer only moderate improvement. Others offer more potential, but I have yet to come up with a way to get them into program logic. Let me begin with the easiest to implement and then I'll work up to those with most potential.

A. The pruning logic and cut-off factors could likely be improved with experimentation. For now I am using the value

$$\text{CUT-OFF LENGTH} = \text{BEST-LENGTH} * \left(\frac{1 - \left(\frac{k-10}{n}\right)^{1.8}}{\frac{8*n}{100} + 2} + \frac{370}{(k+5)^{3}} \right)$$

for the cut-off length while inserting the k'th point. Additionally, I am using a factor to further prune "half path sets" (see discussion in III.C.). These factors were derived experimentally, and interact with each other in subtle ways; further research with these factors, or the introduction of others, could markedly cut down running time.

B. The order in which the points are entered into the problem is based on the "farthest insert" method (see Rosenkrantz et al. : 1977). There are two advantages to this order: 1) the total length becomes approximated very quickly; and 2) there is less likelihood of a "great shuffle" being caused by one of the later points. However, other techniques suggest themselves which also share these advantages. One idea is to use a "farthest insert" variation, using results from the minimal spanning tree or assignment problem (see E below). Another idea would be to assign the entry order dynamically; that is, to not fix the order before the tour building process is begun, but rather, at point k, to let the best sub-tour(s) of the k - 1 preceding points determine the k'th point, instead of merely using the k - 1 points only. The idea here would be to choose that point which would lead to the greatest optimal sub-tour of k points.[12]

It may be thought here that a random order of point introduction would be appropriate here. Then various point orders could be tried, and the best one chosen (as, e.g., in Karg and Thompson : 1964). However, for that to work, tight pruning criteria must be used to get quick results. The philosophy behind the multi-tour method, on the other hand, is to keep the pruning criteria as liberal as possible, so that many sub-tours are generated in parallel. Rather than back-tracking and trying over, we would choose to increase the cut-off values, and thus retain more possible paths. Thus we will never have to reprocess the same information twice.

[12] I should note that a simple version of this technique was tried and did not seem beneficial. So I gave up for the time being.

C. This method may be used along with another, in a mutually reinforcing way, giving its strength where the other is weak, and vice versa. I would note here, for example, that one of my early tests on K26 produced a tour that was 1-opt (in Kernighan-Lin's terms) from the optimal tour; in fact, only one point needed to be moved. So running Kernighan-Lin on the result of the multi-tour result would almost instantly produce the optimal tour. In fact, I suspect that using several of my resulting tours as input to Kernighan-Lin will have the same effect as starting with random orders of points, but will, of course, generate the local minima more quickly.

D. In this section and the next I will speculate a bit. The suggestions posited in A and B will almost certainly lead to some improvement; possibly they will make "perfect" solutions for 200-city geometric problems almost certain. But I suspect that they will not reach much beyond that point (although they should give "close" approximations, comparable to, say, the Kernighan-Lin method). There seems to be a "hugeness" in the combinatorial complexity produced by these problems that cannot be grasped by such a heuristic. Yet, may it be possible to beat this hugeness; at least for a longer distance?

I think there may be a way. In tracing through the solution of P60 I printed a list of all the path sets held in memory as the 18'th point was being added. In examining these path sets, and looking at the graph, I noticed that at least half of them were "obviously" bad (see Figure 16 for an example). It is obvious here that the path {p(14) - p(18) - p(11)} cannot be on the (eventually) optimal subtour: p(12) would have to fit somewhere in the path (one can see visually that the path containing p(12) cannot loop around these points; the "looping" points just aren't there (as they are in Figure 17)). So the idea then would be to somehow mark the path {p(14) - p(18) - p(11)} as invalid, and in the pruning phase to somehow retrieve this information and prune this path automatically.

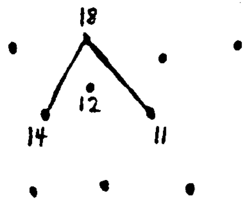


Figure 16. An example of a path that can be seen visually to be bad.

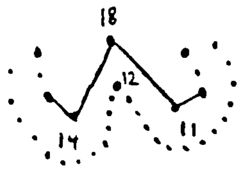


Figure 17. An "unusual" situation where p(12) would not necessarily have to be part of the p(11)-p(18)-p(14) path.

Here a person-machine approach could be used. A person could be shown a set of paths during the tour-building process, and cross some off via a light pen on a screen. I suspect this approach may be time-consuming; however, it may be that a few timely "person-purges" early on in the tour-build would go a long way in cleaning the tree structure.

Additionally, geometric criteria or minimal spanning tree logic may be applied to automatically make these purges. It would seem that the knowledge a person uses in pruning could, in part, be captured effectively by computer logic. I have not found a way to do this, but am not convinced that it is impossible.

E. One wholly neglected resource is the "not-entered points". As the heuristic constructs the sub-tours, and determines which to keep and which to prune, it makes use <u>only</u> of the "entered" points. Now, if a person were to follow along through a trace of the heuristic, he would see that this is a great loss. This was seen above, in Figures 15 and 17. It is as if the heuristic has a blindfold over one eye; more importantly, the heuristic does not have a true picture of the global structure of the problem it is dealing with.

It would seem beneficial to make use in some way of the entire set of points. And this would most likely involve the minimal spanning tree or the assignment problem (AP) solution. (Or, in the spirit of this paper, a set of near minimal spanning trees or AP solutions). The minimal spanning tree(s) could be used, for example, as follows. Suppose the path (p(17) - p(35) - p(20)) is in the minimal spanning tree. Then after p(17) and p(20) have been added we could somehow mark the p(17) - p(20) line as desirable (in anticipation of p(35)), and give it consideration in the pruning phase. I will not elaborate further on this theme but I think the point is clear. I also suspect that a vertex penalization, as in Dantzig et al. (1953), could be of help. The distance matrix could be dynamically modified as each point is entered. Additionally, "average" points, each representing a cluster of points, could be entered at first (see, for a parallel, the Rubber Band Tour Generator of Krolak (1970)); each of these points would be gradually replaced by two other "average" points, representing smaller clusters, until all points are entered. This later variation could minimize the problem of "cusps."

Many other variations could be mentioned. They all involve some dynamic modifications of the distance matrix, or a means of favoring, or discriminating against, certain paths. The concern here is that the implementation of one of these "helps" would cost more in CPU time or memory than it would actually save in improving the heuristic. One hopes that a simple, yet elegant, insight will be found.

VII. SUMMARY

In part, the heuristic was successful. It did not reach an optimal solution in every known case, nor run as quickly as I would have hoped; but it in general performed as well or better than Krolak's (1970) results. It reached optimality in three of the five 100 city problems, and was well within 1% of the optimal in the other two 100 city cases. It came out with a shorter tour than Krolak's in the three larger problems; I doubt that these tours, in fact, are optimal; but I do not know of any other solutions to these problems for comparison.

The Kerighan-Lin heuristic reached optimality in all five 100 city problems. I do not know exactly how long their program would run for these problems on the IBM 3033, but judging from their figures for the GE 635, I suspect it would run significantly faster than the multi-tour heuristic. So, practically, it is a better method. As far as I know, the Kernighan-Lin heuristic is the best available at the present time for these types of problems.

The multi-tour heuristic is an example of a relatively successful method for solving the TSP (at least the geometric case) that is based purely on the efficient storage of multiple partial sub-tours. It gives optimal or near optimal solutions for problems in the 100-200 city range, and can prove optimality for relatively small problems (about 30 cities).

The heuristic is in the genre of "tour-building" (Bellmore and Nemhauser : 1968), using a variation of the "insert" method of Karg and Thompson (1964). It does no backtracking, but instead keeps multiple alternate sub-tours. It does not use "local search" logic, so is not subject to the theoretical limitations described in Papadimitriou and Steiglitz (1977).

The multi-tour heuristic has potential for improvement as a stand-alone model; but possibly its best long-term contribution will be as an "assist" heuristic to one of the more standard methods based on the minimal spanning tree or assignment problem solutions.

BIBLIOGRAPHY

Bellmore, M. and Nemhauser, G. L. 1968. The Traveling Salesman Problem: A Survey. Opns. Res. 16, 538-558.

Christofides, N. 1975. Graph Theory: An Algorithmic Approach. Academic Press: London.

Dantzig, G., Fulkerson, R., and Johnson, S. 1954. Solution of a Large-Scale Traveling-Salesman Problem. Opns. Res. 2, 393-410.

Garey, M. R., Graham, R. L., and Johnson, D. S. 1976. Some NP-Complete Geometric Problems. Proc. Eighth ACM Symposium on Theory of Computing, 10-22.

Garey, M. R. and Johnson, D. S. 1979. Computers and Intractability. W. H. Freeman: San Francisco.

Grotschel, M. and Padberg, M. W. 1977. On the Symmetric Travelling Salesman Problem I: Inequalities. T. J. Watson Research Center, IBM Research, Yorktown Heights, New York.

Held, M. and Karp, R. M. 1962. A Dynamic Programming Approach to Sequencing Problems. SIAM J. 10, 196-210.

Karg, L. L. and Thompson, G. L. 1964. A Heuristic Approach to Solving Traveling Salesman Problems. Management Science 10, 225-248.

Krolak, P., Felts, W., and Marble, G. 1970. Efficient Heuristics for Solving Large Traveling Salesman Problems. Presented at 7th Mathematical Programming Symposium, Hague, Netherlands.

Lin, S. and Kernighan, B. W. 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. Opns. Res. 21, 498-516.

Norback, J. and Love, R. 1977. Geometric Approaches to Solving the Traveling Salesman Problem. Management Science 23, 1208-1224.

Padberg, M. W. and Hong, S. 1977. On the Symmetric Travelling Salesman Problem: A Computational Study. T. J. Watson Research Center, IBM Research, Yorktown Heights, New York.

Papadimitriou, C. H. and Steiglitz, K. 1977. The Complexity of Local Search for the Traveling Salesman Problem. SIAM J. Comput. 6, 76-83.

Papadimitriou, C. H. and Steiglitz, K. 1978. Some Examples of Difficult Traveling Salesman Problems. Opns. Res. 26, 434-443.

Rosenkrantz, D. J., Stearns, R. E., and Lewis, P. M. 1977. An Analysis of Several Heuristics for the Traveling Salesman Problem. SIAM J. Comput. 6, 563-581.

APPENDICES

APPENDIX 1

Proof of Optimality of a 30 City Tour

A. Figure 18 shows the optimal tour of P30, a 30-city problem that was generated randomly in the (0 - 100) unit square. The tour was obtained, and optimality was proved, by the following procedure.

1. Pick a length value (the "ceiling value") that is known to be greater than or equal to that of the optimal tour. The length of any known tour will do; but the idea is to pick this number to be as small as possible.

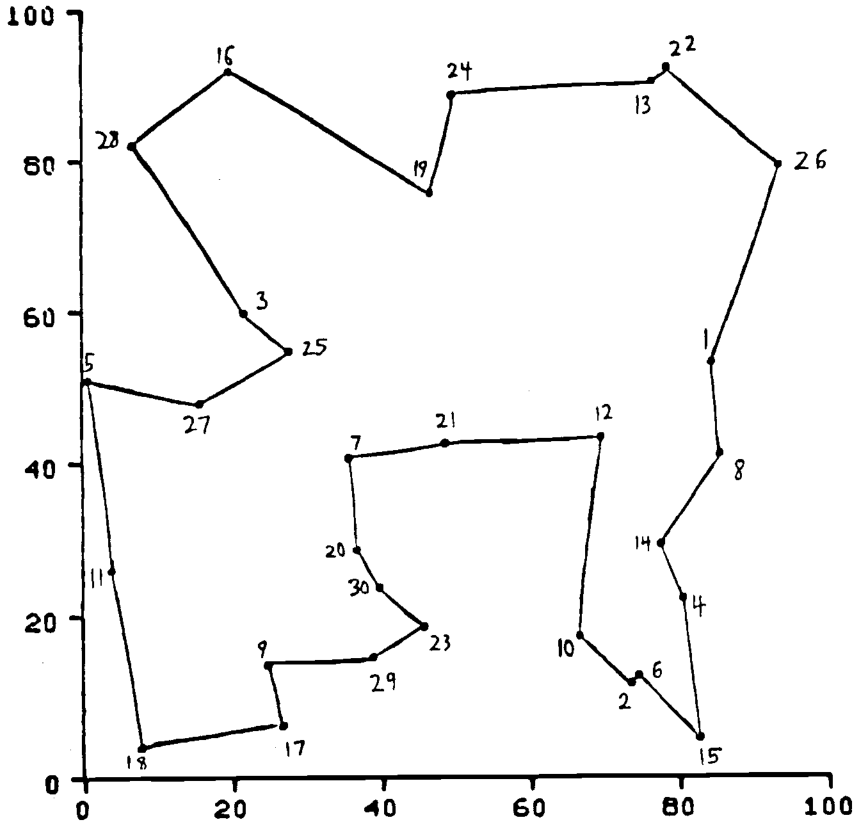
2. Go through the multi-tour building process, using this ceiling value as the cut-off point: at any step prune only those paths which do not contribute to a sub-tour of length less than or equal to the ceiling value.

3. The resulting shortest tour will necessarily be optimal. This is so, since the optimal tour (and any of its sub-tours) will have total length which will not exceed the ceiling value. Thus none of these pieces of the optimal tour will be pruned, and so the optimal tour will survive after the pruning is done. It obviously will be the shortest tour of those surviving tours, by the definition of optimality.

The optimal tour of P30 is 460.294. If 470 is used as the ceiling, the generation and proof will run in 8 seconds CPU time.

B. The number of nodes (each representing a path set) remaining after this "constant ceiling" pruning of P30 is as follows.

CEILING	NODES REMAINING
461	31
462	44
463	46
464	57
465	68
466	95
467	129
468	146
469	152
470	189



85	54
74	12
22	60
81	23
1	51
75	13
36	41
86	42
25	14
67	18
4	26
70	44
77	91
78	30
83	5
20	92
27	6
8	3
47	76
37	29
49	43
79	93
46	19
50	89
28	55
94	80
16	48
7	82
39	15
40	24

length = 460.294

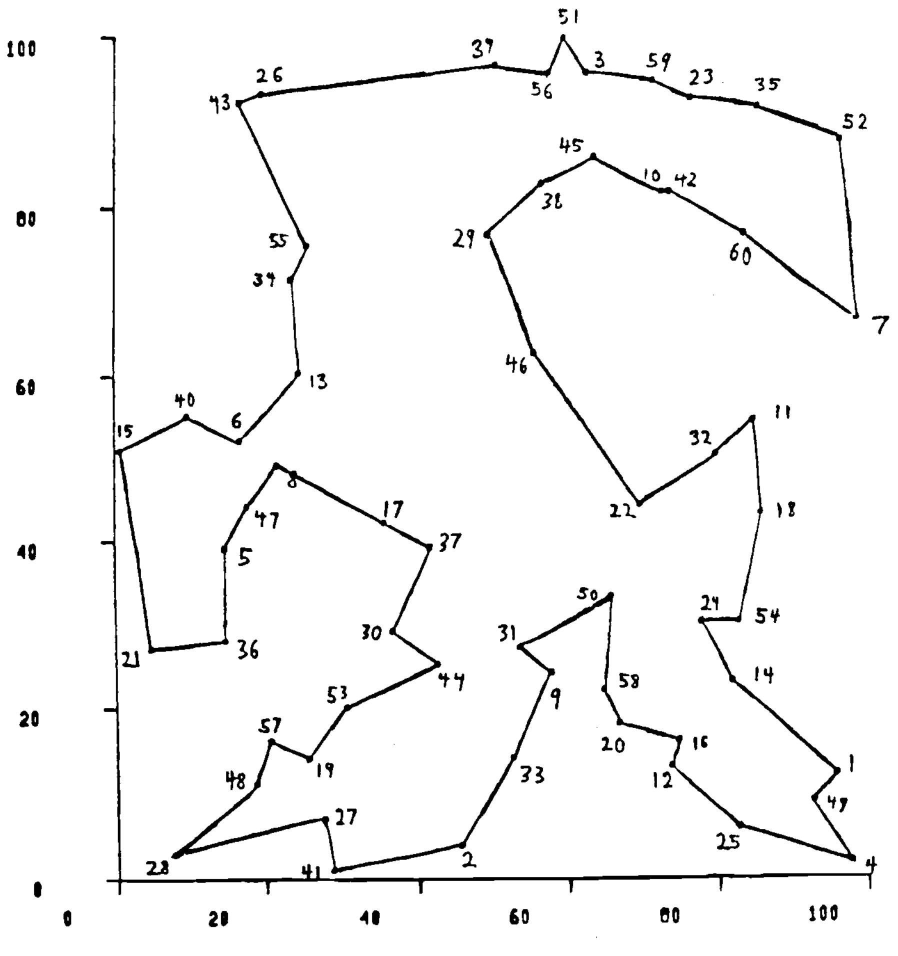
Figure 18

APPENDIX 2

Analysis of the basic multi-tour algorithm of section III.A.

A. Figure 19 shows the optimal tour of P60, a 60-city problem that was generated randomly in the (0 - 100) unit square. Its length is 634.714. Lengths obtained by the basic method for various L (see section III.A. for the definition of L) were:

L	LENGTH
6	678.622
7	678.622
8	678.622
32	671.315
40	659.667
64	659.667
104	659.667
224	653.490



96	12	46	4
63	95	98	2
15	39	17	52
99	65	22	49
58	24	73	81
85	54	74	13
25	60	82	23
1	51	75	16
36	42	86	43
26	14	67	18
5	27	70	44
77	92	78	30
83	6	20	93
28	7	8	3
50	76	37	29
54	27	80	50
53	14	24	71
86	91	15	28
42	39	57	82
51	96	10	55
29	1	74	81
17	92	43	25
64	85	56	62
18	44	19	11
93	9	65	33
60	99	97	87
31	20	83	30
26	75	58	95
21	16	65	22
72	94	84	76

co-ordinates numbered left to right in list above, i.e., point 1 is (96,12), point 2 (46,4), etc.

length = 634.714

Figure 19

APPENDIX 3

Description of the Header of the Node

Below is a description of the information that is contained in each node.

Beginning Point - The leftmost point of (each of the paths of) the path set.

Mid Point - The "middle" point of (each of the paths of) the path set.

Ending Point - The rightmost point of (each of the paths of) the path set.

Length - The smallest of the lengths of the paths of the path set.

New Length - The smallest of the lengths of this path set, if the current point (the one being entered into the problem) is included in the path set.

Length from Above - Described in detail in section III.C.

Level - The number of nodes (inclusively) in the chain from the top node to this node. If there is more than one chain to this node, the shortest chain is used.

Usage - The number of nodes which immediately point down to this node.

Spawned Segment Pointer - A pointer to the node which is spawned from this node by the addition of the current point.

Number of Children - The number of (pairs of) child nodes this node immediately dominates.

Left Child - The pointer to a left half of this node. A left half refers to a path which begins with _Beginning Point_ and ends with _Mid Point_. If the _Left Child_ contains a null pointer, the 2-point path (line) _Beginning Point_ - _Mid Point_ is the reference.

Right Child - The pointer to a Right half of this node. A right half refers to a path which begins with _Mid Point_ and ends with _Ending Point_. If the _Right Child_ contains a null pointer, the 2-point path (line) _Mid Point_ - _End Point_ is the reference.

There are as many _Left Child_ - _Right Child_ pairs as is specified in _Number of Children_.