Motion Control of a Drawing Machine

László Hárs

July 14, 1989

Chapter 1. Introduction

In this work we present an algorithm which solves a Travelling Salesman type problem. Our problem is different mainly in that the edge costs of the underlying graph are dependent from the neighboring edges of the constructed tour. If this dependence requires to examine only a short arc containing an edge which we want to modify (replace, delete or add to the tour) our algorithm can be applied with little modification, but if we have to examine long arcs, possibly the hole tour, the algorithm cannot handle problems of larger then several hundreds of points in reasonable time. If the problem has thousands of points, our algorithm finds good approximate solutions in acceptable time, if the points have nearly uniform distribution in a small dimension space, and their distance is a monotone function of the Euclidean distance in all directions.

Our task was to optimize or at least improve the time needed for the movement of an instrument which by means of a flash lamp (laser) burns lines and special shape holes onto a large sheet of photosensitive glass. It is used then to produce masks to print the wiring of complex electronic circuits.

The locations of the components are already fixed, that is the data given to us consist of the endpoints of the lines to be drawn, the "pen" type which is used for the drawing, i.e. the shape, size and rotation angle of the diaphragm, what controls the shape and thickness of the laser beam that draws the line; and the place, diaphragm type and its rotation angle to burn in the separate holes.

In the actual machine the flash lamp is fixed and there is a moving table. It can move in the x axis direction (horizontally) and in the same time in the y axis direction (vertically). It carries the sheet of glass on which to draw. For our purpose it does not make any difference if we think the glass is fixed and the drawing head moves.

Since the table has large mass, it takes time to accelerate it to top speed, and if it has to stop, to slow down. This results in a quite complicated "distance function" of the pairs of points, which gives the time needed to move the drawing head from one point to the other.

Both directions have their own motor which can work simultaneously with the other one, therefore the "distance" of two points is given by the maximum of the time intervals the table travels the distance of the difference of the x coordinates of the points, and independently the distance of the difference of the y coordinates. The two motors of the table can be different, therefore the time needed to travel a distance in direction x may be different from the time of direction y.

The wider the diaphragm the longer time it takes for the laser to burn enough light in the sheet, therefore we have upper bounds on the speed of the line drawing. Naturally, during movements without drawing, the maximum speed can be used.

The machine has a controller, which is not very smart. Although it does not stop the head if the consecutive visited points (where it shoots, or the endpoints of the drawn lines) lie on a horizontal or vertical line, but if the next point is not on the continuation of the current vertical or horizontal move, it stops, then starts a new straight move to the next point. It would be much better, not to stop, although the drawing head then moved on a curve, not on a straight line, and the job of the controller became more complex.

Also, if we want to change the speed, first the drawing head has to stop, then a new move starts with the new speed.

It has additional costs to change the type, angle or size of the diaphragm, and these may be large. There are only a few possibilities for the diaphragm, therefore it is not bad to handle "subtours", with constant values of these.

1.1. The concrete problem

The size of the drawing surface is approximately $8,400,000 \times 7,200,000$ machine units, i.e., $x \in [169\,607, 8\,424\,509]$ and $y \in [169\,607, 7\,160\,668]$. It means that representing the coordinates of the points at least 24 bit precision is necessary.

There are 46,681 points (line endpoints or single shot points). We can divide the problem into 14 subproblems, where the diaphragm type and its rotation angle are the same. Now the change of the size of the diaphragm has 0 cost, so we do not care about this. The number of points in the subproblems are: 27 422, 7 828, 4 868, 4 845, 1 228, 247, 151, 32, 32, 16, 4, 4, 3, 1. The last 4 of them are trivial, so we actually have 10 subproblems.

The cost function or the length of the edge (the "distance") of two points in both axes' direction has logarithmical growing rate of their Euclidean distance, i.e., sufficient accuracy is achieved by linear approximation between 2^i and 2^{i+1}. It means, that in both axes' direction 5 comparisons give the interval where a multiplication and an addition yield the cost. The maximum of the values of the two directions give the edge length, therefore some 20 simple instructions are enough to calculate the approximate cost. This is important, since in the large subproblems our computer could not store the distances of all the pairs of point. (Even if we have 4,000 points almost 8 millions of distances should be stored, which consumes 32 MBytes if we use the 32-bit integer representation of the compiler. Even if we do not distinguish between the large enough distances, 16-bit precision is necessary, which yields to 16 MBytes of storage requirement. This is still too large. On the other hand, the bucketing technique what we applied and will discuss later makes unnecessary to calculate the majority of these distances, therefore even if we could store all the distances, it were not very good.)

This cost function is used when the drawing head starts from 0 speed and completely stops at the other point. If more points are visited during the same vertical or horizontal move, the head does not slow down. The time, that is the cost of getting from the leftmost to the rightmost point or backwards (or between the topmost and the bottommost one) is the distance of these two points.

The originally used tours were got more or less by "zig-zag" scan. It means, first collect the lines of horizontal directions, order them by their y coordinate and put the ones of the same y coordinate in descending order (sorting the endpoints by x coordinates). Then start from the top right corner, go to left until the last line's endpoint, then go down to the next existing y value, to the leftmost line endpoint. Now go to the right, then down, to the left, and so on. Exhausting the horizontal lines do similarly with the vertical ones. The separate holes if they lie on the continuation of a line are shot "on fly". The left over single shots are done by a last zig-zag scan.

The original time for making the whole mask was about 10.5 hours (the total cost of the drawing). This was reduced by our algorithms to about 4.5 hours, a 57% improvement. To achieve this improvement we used about 2.3 hours CPU time on an IBM 3080KX mainframe computer, run under VMS operating system. The programing was done in VSPASCAL, IBM's Pascal language version. The running time could be reduced with optimization of the code. However, after only 4 minutes of CPU time the algorithm already gives approximately 5.5 hours cost of tour, which is already a 48% improvement. This may be enough for practical purposes, since it is got really fast. (This 4 minutes CPU time could also be reduced by code optimization.)

Chapter 2. Problem formulation

The input consists of two files. The first file contains the records describing how and where to draw lines and shoot points, the second one contains the data for the cost function.

2.1. Drawing information

The file has a record for each point which can be a line endpoint or a single shot point. Each record consists of the following fields:

- diaphragm-type
- diaphragm-x-size
- diaphragm-y-size
- diaphragm-rotation-angle
- x-coordinate
- y-coordinate
- speed (1..3, with 1 denoting the maximum speed)
- 0: at single shot point / the other endpoint's index: at linend.

If two lines share a common endpoint or an endpoint is to be shot over again with a larger diaphragm, this point has to be listed twice, because they need not follow each other on the optimal tour of the drawing head. This means, that there can be (and actually are) points of distance 0 from each other.

2.2. Cost function

First we determine the speed used between the given two points. If they are endpoints of a line to be drawn, we use the given speed value, otherwise the maximum speed.

Let d_x and d_y denote the absolute values of the difference of the x and y coordinates of the two points, respectively. Let f_i denote the cost of moving 2^i machine units in direction x, and g_j denote the cost of moving 2^j machine units in direction y (with the given maximum speed).

If $2^i \leq d_x < 2^{i+1}$ the cost c_x of the x direction move is determined by linear interpolation, i.e., from the equation

$$\frac{c_x - f_i}{d_x - 2^i} = \frac{f_{i+1} - f_i}{2^{i+1} - 2^i}.$$

Similarly, if $2^j \leq d_y < 2^{j+1}$ the cost c_y of the y direction move is determined from

$$\frac{c_y - g_j}{d_y - 2^j} = \frac{g_{j+1} - g_j}{2^{j+1} - 2^j}.$$

The cost of moving between the two points now is calculated by adding $\max(c_x, c_y)$ to the cost of diaphragm change.

2.3. Objective

The objective is to find a (cyclic) permutation of the points, i.e., a round *tour*, such that the line endpoints are next to each other, the “total joined-move-cost” of this permutation be minimum, or at least “small enough”.

Here we mean by the total joined-move-cost the sum of the costs of the “moves”. A *move* means the maximal (cyclic) subsequence of points which have the same diaphragm type, angle and all have either the same x or the same y coordinate, and the other coordinates form a monotone sequence. (During a move the drawing head does not stop.)

Since this problem includes as a special case the Travelling Salesman Problem, we can not hope to find the exact optimum in all but the smallest subproblems ([1]). Therefore some approximate, heuristic solution is the best we can do in reasonable computation time.

Chapter 3. Optimization heuristics

As it has already been mentioned, we solve separately 10 subproblems, in each of them the diaphragm is unchanged. This simplifies the cost function, too, because no check is necessary for the diaphragm change.

There are two possibilities for the move constructions. Either use some heuristic algorithm to join points in moves (e.g., if they have common x or y coordinates, the same diaphragm, and are closer to each other than a certain threshold value, which may vary depending the neighborhood of the points), or during the optimization process always determine (dynamically) the moves.

The former approach is easier and has the advantage to reduce the problem to a standard Travelling Salesman Problem. (The nodes of the graph are the endpoints of the moves and the single shot points. We assign very large negative costs to the edges corresponding to the moves, to force them into the tour.) In this case all the standard TSP algorithms can be applied. This work was done by W. Schwärzler in [2].

We have chosen the dynamic-move approach, and this will be discussed here. It turned out, that better solutions can be found with it in short running time.

A two stage algorithm has been developed. First we construct an initial tour using a modified nearest neighbor heuristic algorithm, then we improve it by a similarly modified 2-opt algorithm. (See [3].)

3.1. Other possibilities

In [3] one can find a lot of heuristic algorithms for the Travelling Salesman Problem. We name here the algorithms as in [3]. Many of them can be modified to handle the dynamic move construction. But, since five of the subproblems are really large, even an $O(n^2)$ running time algorithm can be unacceptable.

For the initial tour construction stage a promising possibility would be the Arbitrary Insertion algorithm. It can be modified to handle the dynamic moves and to use a bucketing technique, to make it subquadratic in running time. In [3] there are experiments cited which show, that this usually gives shorter tour than the Nearest Neighbor procedure. On the other hand, it is slower and harder to implement. (In [8] the worst case performance of the Nearest Neighbor procedure is shown to be inferior to the worst case performance of the insertion algorithms.)

The Farthest Insertion, Shortest Insertion, Cheapest Insertion algorithms are even slower. Although they give good initial tours, too, they are in general not shorter than the one constructed by the Arbitrary Insertion to justify the extra running time.

For the tour improvement stage the k-opt (with $k \geq 3$) or the Kernighan-Lin algorithms ([5]) were much too slow. The only promising procedure would be the modification of the Or-opt algorithm to cut and insert parts of the recently constructed moves. But we can expect only some 5% more reduction of the tour cost in expense of a 4 to 5 fold increase in the running time. For practical purposes the necessary 15 hours CPU time is already unacceptable, therefore we did not implement this Or-opt version.

Chapter 4. Nearest neighbor algorithm

The idea is very simple.

```
Chose and mark an arbitrary point. It forms a path of length 0.
While unmarked points are left
    Scan all the unmarked points
        Find the one nearest to the most recently marked point
    Mark and add this point to the path
Connect the last point to the first one closing the tour.
```

To find the nearest point from the end of the path in our case is not so simple. If the path ends with a line endpoint, which line we have not yet insert into the path, the other endpoint of the line must always be chosen. Otherwise we perform the scan. During this scan we actually save 5 closest points: one to the North from the path endpoint (with the same x coordinate and smaller y coordinate), one to the East, South and West and one which has different x and y coordinates.

Because the distance is an increasing function of the Euclidean distance in any direction we can save time not calculating the distance to the points which have the same x or y coordinate, but only comparing the difference of the inequal coordinates.

We have to treat separately the points with equal x and y coordinates to the coordinates of the last point of the path. It is sure, that this point will be the closest, but not always good to continue the path there. This point can be a line endpoint, which line perpendicularly leads us far from the path endpoint which may lie in the middle of a possible move. Therefore the other endpoint of this perpendicular line should be compared against the other points, and if it wins, not this but the first endpoint is attached to the path. We implemented this version of the algorithm, but it gave the same initial tours as without this modification, showing that in the actual problem this possibility does not occur. However, we kept the code handling this situation in the program, for other practical problems may need it.

This procedure in this form is unfortunately of $O(n^2)$ time complexity, what makes it unpractical for subproblems with more than 1,000 nodes. Fortunately, the points of equal distance from a given point lie on the boundary of a rectangle, which is almost a square, (its center is the given point). As this distance gets larger so do the rectangles.

Also, the points in the larger subproblems are quite uniformly distributed on the drawing surface. These make it possible to use a bucketing technique, similar to the one discussed in [4], to accelerate the algorithm. In practice it worked very fast and gave good enough solution. It may be sufficient to use only this part of the algorithm.

4.1. Bucketing

Let us cut the drawing surface into congruent rectangles, called *buckets*. To find the nearest point to a fixed point P, first we look the points falling into the same bucket, then the points in the 8 neighboring buckets (which are incident to the first one). Next we continue the search in one "level" further around, until an unused point is found in a bucket incident to one of the the buckets of the previous level. We have to completely finish the last level even if a free point is found, and one other level more. We have to keep the nearest to P of the points found. Because P may lie close to a corner of its bucket, the extra level of search is necessary to find the really nearest point. (It still may not be the exact nearest point, because the buckets we used are only close to the ideal shape, but the error is so small and rare, it does not degrade the solution.)

With this technique we do not have to calculate the majority of the distances of the pairs of points, since usually in early levels we find the closest point.

4.1.1. Bucket construction

Since the Nearest Neighbor heuristic is not sensitive to that if we sometimes provide not the exact nearest neighbor, but a point wich is in almost the same distance, in construction of the buckets the accuracy is not very important. Therefore we implemented a fast and simple approximate algorithm instead of the slower and more complex exact one. The quality of the solution using the approximate algorithm is practically the same as using the exact one.

The implemented bucketing is the following: The point $P(x, y)$ belongs to bucket $B(i, j)$ if $i = \lfloor x/2^{14} \rfloor$ and if $j = \lfloor y/2^{14} \rfloor$. (These divisons are done by simple right shift of the numbers, which are faster then the divisions.)

We have 226,008 buckets: $i \in [0, 515]$; $j \in [0, 438]$. Each bucket is actually a pointer, pointing to NIL if the bucket is empty, or to one of the points belonging to this bucket. Each point has an associated pointer to the next point in the same bucket (or to itself if there is no other point in the bucket), closing finally to a circular list.

During the path construction we have to delete the used points from the lists. To be able to do it effectively double linked lists are necessary, what makes the construction process slower. The deletion itself takes several instructions, therefore it is better just mark the points already on the path under construction. It is a single step, and now one way linked lists are enough to keep the points in the boxes together.

It seems that the number of the buckets are too large. Almost all of them are empty. But our experiments show that this is the best value in our example drawing problem. (The cost of the tour we get, of course, does not change much, but the running time was the smallest with 2^{14}.) This may vary from problem to problem. If the points are really uniformly distributed fewer buckets will be enough. There is some work which takes time proportional to the square of the number of the points in any one of the buckets: The distances of all of the pairs are determined. Therefore no bucket should contain really many points. This explains, why we have to use so many buckets if the points (as in our example) are not absolute uniformly distributed.

Chapter 5. 2-opt-type improving algorithm

Let the number of points in the current subproblem be N. Fix an orientation of the initial tour (then we have directed edges). The 2-opt algorithm works as follows (see [3]):

```
Chose an arbitrary (directed) edge AB.
Repeat Until N consecutive steps failed to improve the tour.
    Scan each other edge CD
        If replacing edges AB and CD with edges AC and BD
          (and therefore reversing the arc BC) reduces the cost
          of the tour
        Then rearrange the tour in this way;
          Substitute AB with DE, (if E is the point next to D);
          Continue the outer loop.
    Substitute AB with BE, (if E is the point next to B).
```

The obvious necessary modification of this algoritm is not to delete edges which correspond to lines to be drawn. With this change the algorithm works on our problem, but unacceptably slowly. Also, we have to tell how to check if a rearrangement really decreases the cost of the tour.

In general a 2-opt improving algorithm can take much more than $O(n^2)$ steps, what we have said is unacceptable. But starting from an already quite good initial tour and restricting the number of tries make it a practical algorithm.

5.1. Closest-points-only modification

A well known variation of the 2-opt improving algorithm for dense graphs (see [5]) taking for C only the k closest points to A, with some fixed number k. In our problem, however, it is not known in advance if two points with the same x or y coordinate will belong to a move. If they do, their distance will be shorter then what we can calculate now. Therefore we collect also the closest m points in all the directions North, East, South and West, if they are not much farther (say 4 times) than the kth closest diagonal point.

At first glance it seems that $m > 1$ is unnecessary, but this is not the case. A point e.g., close to another one on a horizontal line may belong to a vertical line to be drawn, and so it can lead us far from our horizontal line, breaking a possible long move. $m = 2$ or $m = 3$ are reasonable choices, and really give better solutions.

Then we can check if D is among the listed closest points of B, and deal with only edge CD if it is. Other possibilities are here, to accept D if it is not farther from B then l times the distance of the farthest point from B. $l = \infty$ gives the largest number of points D to work with, but we can save the a distance calculation and a comparison at each step, therefore it is quite a good choice. Another variation (if $l < \infty$) to always accept a point D if it is on a horizontal or vertical line with point C. This is good, because it helps finding long moves, which are drawn fast. We implemented this version.

This modification requires some preprocessing (for $k = 10$ and $m = 2$ about one third of the running time of the Nearest Neighbor initial tour construction), but then the 2-opt algorithm runs much faster.

5.1.1. Bucketing

For finding the k closest points to all of our points in the large subproblems we again use bucketing. The same buckets can be used as in the Nearest Neighbor heuristic, but we have to search more levels, until we find the k closest points (plus one more level, for the same reason mentioned earlier).

Some extra buckets are searched for points in the East, West, North and South directions until m points are found with the same x or y coordinates as the point under consideration, or the buckets are so far, that they contain points of more then four times the distance of the farthest found diagonal point.

5.2. Improvement check

Since a move (where the drawing head does not stop) may contain the edge AB we are dealing with, it is not easy to calculate the change of cost an edge exchange causes.

Let $A = (x_a, y_a)$, $B = (x_b, y_b)$, and UV be the arc corresponding to the move containing edge AB. If $x_a \neq x_b$ & $y_a \neq y_b$ then $U = A$, $V = B$.

If $x_a = x_b$ & $y_a \neq y_b$ then we follow the tour backwards from point A and forwards from point B while the points we meet have x coordinates equal to x_a and while their y coordinates form a monotone sequence. The last point conforming with these in the backward direction is U, in the forward direction is V.

If $x_a \neq x_b$ & $y_a = y_b$ then we do the same as in the previous case, except the role of the x and y coordinates are exchanged.

If $x_a = x_b$ & $y_a = y_b$ then the situation is a bit more complex. First we try to extend the arc in both directions with points having the same x coordinates. If we fail to in backward or forward direction, then we try to extend the arc in that direction with points having the same y coordinates. With this procedure we can actually end up with two moves, if one direction uses the points with the same x coordinates and the other uses the points with the same y coordinates. We keep these double-move arcs, because neither one move could be preferred.

One more complication can arise: While searching for U we may reach point D (or for V we reach C). If this happens, the later generated arcs and this one may mix themselves up when we try to calculate the cost change of the rearranged tour. Fortunately, now the hole DA path is short, and rearranging the tour according the tried edge exchange is fast, and if it increases the cost, we can easily restore the original tour. (The rearrangement is actually reversing a section of the tour.)

We determine this move-arcs for edge CD, too. If one side is too short, the other arc endpoints give the limits, between which we have to calculate with rearrangement the cost change of the tour. Since these points are never too far from each other on the tour, we do not have to spend much time in improvement check.

Next we determine the move-arcs for the "cross edges" AC and BD. (Now the tour points which are the *next* and *previous* vary respectively.)

If we ever find any two arcs reaching each other, we reverse the short side of the tour, calculate the cost change, and restore the tour if no improvement is found.

Finally, we chose the arc starting point got from AB or from AC, which are farther from A, and similarly the farther arc endpoints from B, C and D. These mark the short sections of the tour, and only here the arc costs should be calculated to be able to check the cost change of the tour. (In an arc got for a cross edge AC or BD again we have to build up the moves from the edges, with different *next* and *previous* links as in the cases of the normal edges AB and CD.)

5.3. Implementation

There are many ways to represent the tour. The simplest and fastest is to store the consecutive points in an array, but this is not suitable with the closest-points-only modification of the 2-opt algorithm. Some additional information is needed to find a given point, and this information has to be updated when the tour is rearranged. Therefore it is better to use a linked list. Each point has a pointer to the next one on the tour. But we have to follow a path backwards, too, so an other pointer is needed to the previous tour point. What we have implemented is this double linked circular list. (See e.g., [6].)

A serious drawback of this representation is that the reversing during the 2-opt part of the algorithm the long paths may consume significant time. Unfortunately we can not use the reversible list representation of [6], because it does not give the information at a random point which of its two neighbors is the next or the previous point on the tour, according to a fixed tour orientation. If we chose the wrong neighbor, we may cut the tour into two separate cycles when we try to improve with a path reversing.

We could use a tree representation to speed up the reversings (see [7]), but if we start from an already quite good tour, as we do after the Nearest Neighbor part, there are much less reversings than tries, so the loss of time during the improvement checks is larger then the gain in the reversings. This is why the double linked circular list representation was the best, and is used in the final version of the program.

The following simple functions compare one coordinate of points indexed by i and j, supposed that the other are equal. If the first coordinate is equal, we follow either the `Next` or the `Prev` links until inequal coordinates are found. (Functions `X(i)` and `Y(i)` provide the coordinates of point i.)

```
(******************* Greater than Next/Prev X/Y *********************)

Function GNX( i, j : Integer): Boolean;
Begin  While X(i) = X(j) Do j := Next[j]; GNX := X(i) > X(j) end;

Function GNY( i, j : Integer): Boolean;
Begin  While Y(i) = Y(j) Do j := Next[j]; GNY := Y(i) > Y(j) end;

Function GPX( i, j : Integer): Boolean;
Begin  While X(i) = X(j) Do j := Prev[j]; GPX := X(i) > X(j) end;

Function GPY( i, j : Integer): Boolean;
Begin  While Y(i) = Y(j) Do j := Prev[j]; GPY := Y(i) > Y(j) end;
```

It is simple to reverse an arc of the tour. (The procedure `Swap` exchanges its two arguments.)

```
(*****************************************************************)
(* Procedure Reverse                                             *)
(* Reverses the arc (i1,i2)                                      *)
(*                                                   17.05.1989.*)
(*****************************************************************)
```

```
Procedure Reverse( i1, i2: Integer);
Var  u: Integer;
Begin
  Next[Prev[i1]] := i2;  Prev[Next[i2]] := i1;
  u := Next[i1]; Next[i1] := Next[i2]; Next[i2] := Prev[i2];
  Prev[i2] := Prev[i1]; Prev[i1] := u;
  While u <> i2 do Begin
    Swap(Next[u],Prev[u]);
    u := Prev[u]  end;
end;
```

To calculate te cost of an arc the following function `NCost` is used. The total cost of the tour is determined by the same way. Here the function `MinSp(u,v,spd)` gives the minimum of the speed of the edge (u, v) and *spd*. This is necessary, since in a move during drawn segments we can not increase the speed, only if the drawing head first stops.

```
(*********************************************************************)
(* Function NCost                                                    *)
(* gives the cost of arc (u,v) with Next-links, determines tablemoves*)
(* u: start, v: end of some tablemove. Speed=lowest (interline gaps).*)
(*                                                       19.05.1989.*)
(*********************************************************************)
Function NCost( u,v: Integer): Integer;
Var  c, t, w, spd, Xu, Yu: Integer;
      GT: Boolean;
Begin   c := 0;
  Repeat
    w := Next[u]; Xu := X(u); Yu := Y(u);
    spd := MinSp(u,w,1);
    If Xu = X(w) Then Begin
      t := Next[w]; GT := GNY(u,w);
      While (Xu = X(t)) & (w <> v) &
            ((GT & (Y(t)<=Y(w))) or (^GT & (Y(w)<=Y(t)))) Do Begin
        spd := MinSp(w,t,spd); w := t; t := Next[w] end;
    End Else
    If Yu = Y(w) Then Begin
      t := Next[w]; GT := GNX(u,w);
      While (Yu = Y(t)) & (w <> v) &
            ((GT & (X(t)<=X(w))) or (^GT & (X(w)<=X(t)))) Do Begin
        spd := MinSp(w,t,spd); w := t; t := Next[w] end;
    End;
    c := c + ECost( u, w, spd);
    u := w;
  Until u = v;
  NCost := c;
end;
```

For "cross-edges" the cost of the containing arc is calculated by the following function. There is one more similar procedure `PNCost`, where the roles of the links `Next, Prev` are exchanged.

```
(*********************************************************************)
(* Function NPCost                                                   *)
(* gives the cost of arc (AA..A,B..BB) with Next-Prev links.        *)
(*                                                       09.06.1989.*)
(*********************************************************************)
```

```
Function NPCost( AA,A,B,BB: Integer): Integer;
Var  c, t, u, v, spd, XA, YA : Integer;  GT: Boolean;
Begin   v := B; u := A; XA := X(A); YA := Y(A);
  spd := MinSp(A,B,1);
  If XA = X(B) Then Begin

    t := Prev[A]; GT := GNY(B,A);
    While (XA = X(t)) & (u <> AA) &
          ((GT & (Y(t)<=Y(u))) or (^GT & (Y(u)<=Y(t)))) Do Begin
      spd := MinSp(u,t,spd); u := t; t := Prev[u] end;

    t := Prev[B]; GT := GNY(A,B);
    While (XA = X(t)) & (v <> BB) &
          ((GT & (Y(t)<=Y(v))) or (^GT & (Y(v)<=Y(t)))) Do Begin
      spd := MinSp(v,t,spd); v := t; t := Prev[v] end;
  End;

  If (YA = Y(B)) & (u = A) & (v = B) Then Begin

    t := Prev[A]; GT := GNX(B,A);
    While (YA = Y(t)) & (u <> AA) &
          ((GT & (X(t)<=X(u))) or (^GT & (X(u)<=X(t)))) Do Begin
      spd := MinSp(u,t,spd); u := t; t := Prev[u] end;

    t := Prev[B]; GT := GNX(A,B);
    While (YA = Y(t)) & (v <> BB) &
          ((GT & (X(t)<=X(v))) or (^GT & (X(v)<=X(t)))) Do Begin
      spd := MinSp(v,t,spd); v := t; t := Prev[v] end;
  End;

  c := ECost( u, v, spd);
  If AA <> u Then c := c + NCost(AA,u);
  If BB <> v Then c := c + NCost(BB,v);
  NPCost := c;
end;
```

To generate the arc, watching if it hits another arc, we use the following procedure. Again there are two more variants of this, `GetArcPP` and `GetArcNN`, where only the links `Prev` or the links `Next` are used.

```
(**********************************************************************)
(* Procedure GetArcNP                                                 *)
(* Finds the first and last node of a tablemove containing edge AB   *)
(* using the Next/Prev links                                          *)
(*                                                        18.05.1989.*)
(**********************************************************************)

Procedure GetArcNP( A,B :        Integer;    {Starting edge}
              Var AB0,AB1,                   {Arc tail, head}
                  nab0,nab1:     Integer;    {Number of nodes to AB*}
                  F0,F1:         Integer;    {Forbidden nodes}
              Var Short0,Short1: Boolean);   {Hits the forbidden node?}

Var  GT: Boolean;  X0,Y0, X1,Y1, Xt, Yt, t : Integer;
Begin
  nab0:= 0;        nab1:= 0;       X0 := X(A);      Y0 := Y(A);
  AB0 := A;        AB1 := B;       X1 := X(B);      Y1 := Y(B);

  If X0=X1 Then Begin
```

```
    GT := GPY(B,A);  t := Prev[A];  Xt := X(t); Yt := Y(t);
    While (X0=Xt) & ^Short0 & ((GT & (Yt<=Y0)) or (^GT & (Y0<=Yt)))
    Do Begin
      AB0 := t; t := Prev[AB0]; nab0 := nab0 + 1;
      Y0 := Yt; Xt := X(t); Yt := Y(t);
      If AB0 = F0 Then Short0 := TRUE;
    end;
    GT := GNY(A,B);  t := Next[B];  Xt := X(t); Yt := Y(t);
    While (X0=Xt) & ^Short1 & ((GT & (Yt<=Y1)) or (^GT & (Y1<=Yt)))
    Do Begin
      AB1 := t; t := Next[AB1]; nab1 := nab1 + 1;
      Y1 := Yt; Xt := X(t); Yt := Y(t);
      If AB1 = F1 Then Short1 := TRUE;
    end;
  end;
  If Y0=Y1 Then Begin
   If nab0 = 0 Then Begin
    GT := GPX(B,A);  t := Prev[A];  Xt := X(t); Yt := Y(t);
    While (Y0=Yt) & ^Short0 & ((GT & (Xt<=X0)) or (^GT & (X0<=Xt)))
    Do Begin
      AB0 := t; t := Prev[AB0]; nab0 := nab0 + 1;
      X0 := Xt; Xt := X(t); Yt := Y(t);
      If AB0 = F0 Then Short0 := TRUE;
    end;
   end;
   If nab1 = 0 Then Begin
    GT := GNX(A,B);  t := Next[B];  Xt := X(t); Yt := Y(t);
    While (Y0=Yt) & ^Short1 & ((GT & (Xt<=X1)) or (^GT & (X1<=Xt)))
    Do Begin
      AB1 := t; t := Next[AB1]; nab1 := nab1 + 1;
      X1 := Xt; Xt := X(t); Yt := Y(t);
      If AB1 = F1 Then Short1 := TRUE;
    end;
   end;
  end;
end;
```

With these we can write the 2-opt type improving procedure `ImproveTour`. The global variable `TXCost` contains the Total eXchange Cost. The global array `Line` contains 0 for points corresponding to single shot, and the other endpoint's index for points corresponding to a line end to be drawn.

```
(***********************************************************************)
(* Procedure ImproveTour                                               *)
(* Improves a tour with Next/Prev representation, starting at node n1*)
(* having m nodes.                                          17.05.1989.*)
(***********************************************************************)
Procedure ImproveTour( n1, m: Integer);
Var  i, j, k, Fails, XCost, PA, MaxD,
     nab0,nab1, ncd0,ncd1, nac0,nac1, nbd0,nbd1,
     A,B,C,D, AA,BB,CC,DD, AB0,AB1, CD0,CD1, AC0,AC1, BD0,BD1: Integer;
     ShortBC, ShortDA: Boolean;
Begin
  TXCost := 0; A := n1; Fails := 0;
Repeat
 B := Next[A];
 If Line[A] <> B Then Begin
  PA := Prev[A];  MaxD := ECost(B,Nbor[Frst[B]],1);
  For j := Last[A] Downto Frst[A] Do Begin         { Closer...Farther }
    C := Nbor[j];
    If (C = PA) or (C = A) or (C = B) Then CONTINUE;
    D := Next[C];
    If (Line[C] = D)  or ( (X(B) <> X(D)) & (Y(B) <> Y(D))
     & (ECost(D,B,1) > MaxD) )  Then CONTINUE;     {/No BD edge exists}
    ShortBC := FALSE; ShortDA := FALSE;

    GetArcPN( A,B, AB0,AB1, nab0,nab1, D,  C,   ShortDA,ShortBC);
    GetArcPN( C,D, CD0,CD1, ncd0,ncd1, AB1,AB0, ShortBC,ShortDA);
    GetArcPP( A,C, AC0,AC1, nac0,nac1, CD1,AB1, ShortDA,ShortBC);
    GetArcNN( B,D, BD0,BD1, nbd0,nbd1, CD0,AB0, ShortBC,ShortDA);

    If nac0 > nab0 Then AA := AC0 Else AA := AB0;
    If nbd0 > nab1 Then BB := BD0 Else BB := AB1;
    If nac1 > ncd0 Then CC := AC1 Else CC := CD0;
    If nbd1 > ncd1 Then DD := BD1 Else DD := CD1;

    If ShortBC Then Begin
      XCost := NCost(AA,DD); Reverse(B,C);
      XCost := XCost - NCost(AA,DD); Reverse(C,B) end Else
    If ShortDA Then Begin
      XCost := NCost(CC,BB); Reverse(D,A);
      XCost := XCost - NCost(CC,BB); Reverse(A,D) end Else
    XCost := NCost(AA,BB) + NCost(CC,DD) -
        NPCost(AA,A,C,CC) - PNCost(BB,B,D,DD);

    If XCost > 0 Then Begin
      TXCost:=TXCost+XCost;
      Reverse(B,C);
      B := D;
      Fails := 0;
      LEAVE;                        {For j: No more A-nbor is considered}
    end; {Improve}
  end; {For j}
 end; {If AB free}
 Fails := Fails + 1;
 A := B;
Until Fails >= m;
end;
```

Chapter 6. Computational experiments

There are several parameters we can tune to achieve the best solution or to reduce the running time. These are:

- The number of the buckets (or the divisor d of the coordinates)
- The number of the considered closest diagonal points (k)
- The number of the considered closest horizontal/vertical points (m)
- The limit to take into consideration the point D in the 2-opt algorithm (l).

As usual, there is no absolute best choice, except for the number of buckets. In our example the division by 2^{14} was the fastest, but other examples may require a different choice. (Here, too, for Subproblem 2. the best was 2^{13}, but it requires so many buckets, we needed a 16 MByte virtual machine. For Subproblem 8. the best choice was 2^{16}. The above mentioned optimum is true for the sum of the running time of the 5 largest subproblems. A little improvement can be achieved with different d values for the different subproblems, but it is not significant.)

For the other parameters we have a trade off between the running time and the quality of the solution found. A parameter which allows more cases to be considered yields better solution but in longer running time. If we allow about 3 hours CPU time, the best was to put the last three parameters to $(8, 2, \infty)$, or to $(6, 2, \infty)$ if we are satisfied with a little worse solution in 2 hours 17 minutes of CPU time.

The improvements and the running times are summarized in the following tables. Unfortunately, the timer routine which was available for the test is not very accurate. Usually we can assume about 5% accuracy, but sometimes the error can be more than 20%.

Subproblem 2. is the largest one, and its contribution the final solution is also the largest. Therefore, looking at only Table 13. can be somewhat misleading, since it mainly reflects how the program performs on Subproblem 2.

In general, the best value for l seems to be ∞. It does not require much more running time than $l = 2$, but the solution is better.

For m the best value is 2. With 1 the solution is worse with little CPU time saving, and with 3, the solution is not improved, only the running time gets longer. The reason of this is the following: If we consider points for the 2-opt exchange which lie on the same line as the fixed edge but in a long distance, in the earlier stage of the algorithm we may find improvements producing "zig-zag" tours. These can not be optimal, later other exchanges are needed to get rid of them.

Finally, as anyone would think, increasing k gives better solutions in longer running time. We have to chose its value according to the allowable computation time.

Subproblem	Nodes	$d = 2^{13}$	$d = 2^{14}$	$d = 2^{15}$	$d = 2^{16}$
2	27422	126	154	358	857
1	7828	66	37	59	69
3	4868	40	25	43	64
7	4845	42	24	44	63
8	1228	41	16	21	5
Total	46191	315	256	525	1058

CPU Time (sec) for finding the closest points. ($k = 6$ $m = 1$)

Table 1.

Subproblem	Nodes	$d = 2^{13}$	$d = 2^{14}$	$d = 2^{15}$	$d = 2^{16}$
2	27422	110	137	331	1562
1	7828	54	32	49	127
3	4868	26	18	30	110
7	4845	24	17	29	108
8	1228	40	13	13	5
Total	46191	254	217	452	1912

CPU Time (sec) for the Nearest Neighbor procedure

Table 2.

k m l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6 1 1	7.0	67.4	37	376
6 1 2	13.6	69.7	37	1267
6 1 ∞	13.5	69.7	38	849
6 2 1	7.3	67.5	38	1091
6 2 2	13.6	69.7	38	2518
6 2 ∞	14.1	69.9	38	2226
8 1 1	7.7	67.6	41	455
8 1 2	13.8	69.7	42	1169
8 1 ∞	14.3	69.9	41	1926
8 2 1	7.8	67.6	42	778
8 2 2	13.3	69.6	42	1788
8 2 ∞	14.3	69.9	42	1839
10 1 1	7.7	67.6	47	583
10 1 2	14.6	70.0	47	3074
10 1 ∞	15.0	70.2	47	1483
10 2 1	7.9	67.7	47	800
10 2 2	15.2	70.3	47	2832
10 2 ∞	15.1	70.2	47	2951
12 2 ∞	16.0	70.5	53	5475
15 3 ∞	14.7	70.1	63	4689

Subproblem 1. (Nodes = 7 828) Original Cost = 8 073 274
Nearest Nbor Cost = 2 829 409
Nearest Nbor Improvement = 64.9%

Table 3.

k	m	l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6	1	1	13.5	49.9	154	3801
6	1	2	23.5	55.7	155	3558
6	1	∞	24.0	56.0	155	3941
6	2	1	13.1	49.7	156	4114
6	2	2	23.7	55.8	157	6244
6	2	∞	23.8	55.9	157	4043
8	1	1	14.6	50.5	169	2629
8	1	2	24.9	56.5	170	5383
8	1	∞	25.0	56.5	170	5447
8	2	1	15.0	50.8	172	4299
8	2	2	24.7	56.4	172	5833
8	2	∞	25.1	56.6	172	7723
10	1	1	15.7	51.1	191	4123
10	1	2	25.6	56.9	191	7362
10	1	∞	25.3	56.7	191	5441
10	2	1	16.1	51.4	194	7023
10	2	2	25.6	56.9	194	7631
10	2	∞	26.1	57.2	194	9124
12	2	∞	26.4	57.4	216	8637
15	3	∞	23.0	55.4	257	14813

Subproblem 2. (Nodes = 27 422) Original Cost = 17 385 566

Nearest Nbor Cost = 10 064 911

Nearest Nbor Improvement = 42.1%

Table 4.

k	m	l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6	1	1	5.8	40.4	25	254
6	1	2	11.0	43.7	25	738
6	1	∞	11.3	43.9	25	604
6	2	1	6.5	40.8	25	425
6	2	2	10.9	43.7	25	738
6	2	∞	11.6	44.1	26	1121
8	1	1	6.7	41.0	27	521
8	1	2	11.5	44.0	28	604
8	1	∞	12.8	44.8	28	715
8	2	1	6.3	40.7	28	343
8	2	2	11.1	43.8	28	963
8	2	∞	11.7	44.2	28	911
10	1	1	7.3	41.4	31	429
10	1	2	12.0	44.4	31	961
10	1	∞	13.1	45.0	31	1227
10	2	1	7.6	41.6	32	549
10	2	2	12.6	44.7	32	1501
10	2	∞	12.6	44.7	32	1077
12	2	∞	13.0	44.9	35	1141
15	3	∞	13.2	45.1	40	2600

Subproblem 3. (Nodes = 4 868) Original Cost = 2 782 181
Nearest Nbor Cost = 1 759 281
Nearest Nbor Improvement = 36.7%

Table 5.

k	m	l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6	1	1	8.6	41.7	24	339
6	1	2	13.9	45.1	24	991
6	1	∞	13.5	44.9	24	675
6	2	1	9.0	42.0	25	810
6	2	2	13.7	45.0	25	761
6	2	∞	12.7	44.4	25	512
8	1	1	8.1	41.5	26	414
8	1	2	14.5	45.5	27	835
8	1	∞	15.1	45.9	27	797
8	2	1	9.3	42.2	27	930
8	2	2	15.7	46.3	27	1052
8	2	∞	14.6	45.6	27	1191
10	1	1	10.2	42.7	30	531
10	1	2	16.2	46.6	30	1640
10	1	∞	15.5	46.1	30	1241
10	2	1	10.5	43.0	30	744
10	2	2	14.5	45.5	30	1149
10	2	∞	14.4	45.4	30	1303
12	2	∞	15.6	46.2	33	1495
15	3	∞	15.8	46.3	39	2948

Subproblem 7. (Nodes = 4 845)

Original Cost = 2 807 871
Nearest Nbor Cost = 1 788 933
Nearest Nbor Improvement = 36.2%

Table 6.

k	m	l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6	1	1	5.5	57.0	16	12
6	1	2	8.5	58.4	16	48
6	1	∞	8.8	58.5	16	37
6	2	1	5.8	57.2	16	19
6	2	2	8.5	58.4	16	43
6	2	∞	9.1	58.7	16	46
8	1	1	6.5	57.5	18	19
8	1	2	9.4	58.8	18	59
8	1	∞	9.6	58.9	17	62
8	2	1	6.5	57.5	18	20
8	2	2	9.4	58.8	18	73
8	2	∞	9.6	58.9	18	46
10	1	1	7.3	57.9	21	23
10	1	2	9.5	58.8	21	73
10	1	∞	10.3	59.2	21	56
10	2	1	7.4	57.9	21	24
10	2	2	9.8	59.0	21	67
10	2	∞	10.3	59.2	21	70
12	2	∞	10.7	59.4	23	63
15	3	∞	10.6	59.3	27	76

Subproblem 8. (Nodes = 1 228) Original Cost = 4 764 838
Nearest Nbor Cost = 2 165 593
Nearest Nbor Improvement = 54.5%

Table 7.

k	m	l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6	1	1	0.0	36.1	0.02	0.03
6	1	2	0.0	36.1	0.02	0.03
6	1	∞	0.0	36.1	0.02	0.03
6	2	1	0.0	36.1	0.02	0.03
6	2	2	0.0	36.1	0.02	0.04
6	2	∞	0.0	36.1	0.02	0.04
8	1	1	0.0	36.1	0.02	0.04
8	1	2	0.0	36.1	0.02	0.05
8	1	∞	0.0	36.1	0.02	0.04
8	2	1	0.0	36.1	0.02	0.04
8	2	2	0.0	36.1	0.02	0.05
8	2	∞	0.0	36.1	0.02	0.05
10	1	1	0.0	36.1	0.03	0.05
10	1	2	0.0	36.1	0.03	0.06
10	1	∞	0.0	36.1	0.03	0.05
10	2	1	0.0	36.1	0.03	0.06
10	2	2	0.0	36.1	0.03	0.06
10	2	∞	0.0	36.1	0.03	0.05
12	2	∞	0.0	36.1	0.03	0.06
15	3	∞	0.0	36.1	0.03	0.07

Subprogram 9. (Nodes = 16) Original Cost = 97 763

Nearest Nbor Cost = 62 432

Nearest Nbor Improvement = 36.1%

Table 8.

k	m	l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6	1	1	0.6	39.5	0.07	0.08
6	1	2	0.6	39.5	0.07	0.10
6	1	∞	3.2	41.1	0.07	0.08
6	2	1	0.6	39.5	0.07	0.08
6	2	2	0.6	39.5	0.07	0.11
6	2	∞	3.2	41.1	0.07	0.09
8	1	1	0.6	39.5	0.07	0.11
8	1	2	3.2	41.1	0.07	0.12
8	1	∞	3.2	41.1	0.07	0.11
8	2	1	0.6	39.5	0.07	0.11
8	2	2	3.2	41.1	0.07	0.12
8	2	∞	3.2	41.1	0.07	0.11
10	1	1	0.6	39.5	0.11	0.14
10	1	2	3.9	41.4	0.11	0.24
10	1	∞	3.9	41.4	0.11	0.21
10	2	1	0.6	39.5	0.11	0.14
10	2	2	3.9	41.4	0.11	0.24
10	2	∞	3.9	41.4	0.11	0.22
12	2	∞	3.2	41.1	0.11	0.20
15	3	∞	5.3	42.3	0.10	0.49

Subproblem 10. (Nodes = 32)

Original Cost = 137 668
Nearest Nbor Cost = 83 843
Nearest Nbor Improvement = 39.0%

Table 9.

k	m	l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6	1	1	0.8	38.6	1.33	0.58
6	1	2	3.1	40.1	1.33	1.09
6	1	∞	3.1	40.1	1.33	1.04
6	2	1	0.8	38.6	1.33	0.62
6	2	2	3.1	40.1	1.33	1.20
6	2	∞	3.1	40.1	1.34	1.18
8	1	1	0.8	38.6	1.39	0.83
8	1	2	2.8	39.8	1.39	1.31
8	1	∞	2.8	39.8	1.39	1.22
8	2	1	0.8	38.6	1.39	0.88
8	2	2	2.8	39.8	1.39	1.39
8	2	∞	2.8	39.8	1.39	1.32
10	1	1	0.7	38.5	2.10	1.19
10	1	2	3.3	40.2	2.10	2.65
10	1	∞	3.3	40.2	2.10	2.43
10	2	1	0.7	38.5	2.10	1.23
10	2	2	3.3	40.2	2.10	2.79
10	2	∞	3.3	40.2	1.50	2.62
12	2	∞	3.6	40.4	2.20	3.60
15	3	∞	4.3	40.8	1.74	7.48

Subproblem 11. (Nodes = 151) Original Cost = 495 884

Nearest Nbor Cost = 306 755

Nearest Nbor Improvement = 38.1%

Table 10.

k	m	l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6	1	1	0.9	33.9	0.07	0.12
6	1	2	2.2	34.8	0.07	0.14
6	1	∞	2.2	34.8	0.07	0.12
6	2	1	0.9	33.9	0.07	0.12
6	2	2	2.2	34.8	0.07	0.14
6	2	∞	2.2	34.8	0.07	0.13
8	1	1	2.5	35.0	0.07	0.11
8	1	2	2.6	35.0	0.07	0.16
8	1	∞	2.6	35.0	0.07	0.14
8	2	1	2.5	35.0	0.07	0.12
8	2	2	2.6	35.0	0.07	0.16
8	2	∞	2.6	35.0	0.07	0.14
10	1	1	2.5	35.0	0.11	0.21
10	1	2	2.7	35.1	0.11	0.25
10	1	∞	2.7	35.1	0.11	0.22
10	2	1	2.5	35.0	0.11	0.21
10	2	2	2.7	35.1	0.11	0.26
10	2	∞	2.7	35.1	0.11	0.23
12	2	∞	2.7	35.1	0.12	0.28
15	3	∞	2.7	35.1	0.11	0.34

Subproblem 12. (Nodes = 32) Original Cost = 133 272
Nearest Nbor Cost = 88 844
Nearest Nbor Improvement = 33.3%

Table 11.

k	m	l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6	1	1	0.4	38.5	3.55	0.97
6	1	2	1.3	39.1	3.53	4.41
6	1	∞	2.2	39.6	3.53	4.26
6	2	1	0.4	38.5	3.54	1.03
6	2	2	1.3	39.1	3.53	4.88
6	2	∞	2.2	39.6	3.55	4.95
8	1	1	0.5	38.6	3.68	1.17
8	1	2	2.8	40.0	3.68	6.45
8	1	∞	1.9	39.5	3.68	2.30
8	2	1	0.5	38.6	3.69	1.23
8	2	2	2.8	40.0	3.68	7.00
8	2	∞	1.9	39.5	3.69	2.58
10	1	1	1.0	38.9	5.57	3.52
10	1	2	2.0	39.5	5.57	5.31
10	1	∞	3.0	40.1	5.57	4.34
10	2	1	1.0	38.9	5.58	3.66
10	2	2	2.0	39.5	5.58	5.67
10	2	∞	3.0	40.1	5.57	4.79
12	2	∞	3.9	40.7	5.80	7.41
15	3	∞	3.6	40.5	4.54	12.27

Subproblem 13. (Nodes = 247) Original Cost = 754 507

Nearest Nbor Cost = 465 616

Nearest Nbor Improvement = 38.2%

Table 12.

k	m	l	2 - opt Improve(%)	Total Improve(%)	Nearest Points CPU Time(sec)	2 - opt CPU Time(sec)
6	1	1	9.9	52.8	261	4784
6	1	2	17.3	56.6	262	6607
6	1	∞	17.6	56.8	263	6111
6	2	1	9.8	52.7	265	6461
6	2	2	17.4	56.7	266	10310
6	2	∞	17.6	56.8	267	7954
8	1	1	10.7	53.2	286	4040
8	1	2	18.3	57.2	290	8057
8	1	∞	18.6	57.3	288	8950
8	2	1	11.0	53.3	292	6372
8	2	2	18.2	57.1	292	9717
8	2	∞	18.5	57.3	292	11714
10	1	1	11.6	53.6	328	5694
10	1	2	19.0	57.5	328	13118
10	1	∞	19.0	57.5	328	9454
10	2	1	11.9	53.8	332	9145
10	2	2	19.0	57.5	332	13189
10	2	∞	19.3	57.7	329	14533
12	2	∞	19.8	57.9	368	16822
15	3	∞	17.9	57.0	433	11813

Sum of 10 subproblems. (Nodes = 46 669) Original Cost = 37 432 824
Nearest Nbor Cost = 19 615 617
Nearest Nbor Improvement = 47.5%

Table 13.

References

[1] M. R. Garey and D. S. Johnson "Computers and intractability: A guide to the theory of NP-completeness," *Freeman, San Francisco*, **1979**.

[2] W. Schwärzler "(private communication)," **1989**.

[3] B. L. Golden and W. R. Stewart "Empirical analysis of heuristics," in The Travelling Salesman Problem *edited by* E. L. Lawler, J. K. Lenstra, A. H. G. Rinnoy Kan and D. B. Shmoys *John Wiley & Sons, Chichester*, **1986**, pp. 207–249.

[4] T. Asano, M. Edahiro, H. Imai, M. Iri and K. Murota "Practical use of bucketing techniques in computational geometry," *in* Computational Geometry *Elsevier, North Holland*, **1985**, pp. 153–195.

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

[6] R. E. Tarjan "Data structures and network algorithms," *CBMS-NSF Regional Conference Series in Applied Mathematics* **44**(1983).

[7] L. Hárs "Random search in the Travelling Salesman Problem," *Manuscript* **1989**.

[8] D. J. Rosenkrantz, R. E. Stearns, P .M . Lewis "An analysis of several heuristics for the traveling salesman problem," *SIAM J. Comput.* **6**(1977), pp. 563–581.