AN INVESTIGATION INTO EFFICIENT MULTIPLE COMMAND ORDER PICKING IN HIGH BAY NARROW AISLE WAREHOUSES

Marin Dimitrov Guenov

Ph.D.  APPENDICES  1990
BEST COPY AVAILABLE.

VARIABLE PRINT QUALITY
CONTENTS

Appendix A: Software Developed for the Simulation Experiment in Chapter III 1

Appendix B: Additional Software Developed for the Simulation Experiment in Chapter IV 107
APPENDIX A

SOFTWARE DEVELOPED FOR THE SIMULATION EXPERIMENT IN CHAPTER III

Appendix A contains all functions, (except the ones for the graphic display) which have been used to conduct the simulation experiment in chapter III.
The majority of these functions have been used as well for the experiments conducted in chapters IV and V.
Functions that produce graphic display are not presented for the sake of brevity.
Prior to some of the more complicated functions a description and/or a pseudo code is given.
The functions are compiled following the hierarchy, shown in fig. A1.
Fig. A1. Hierarchy of the software functions used for the simulation experiment, described in chapter III.
/*****************************************************************************/
/*
* Header file-<externs.h> - contains external variables.
*
****************************************************************************/
/**************************** BAND ****************************/
extern int XCOL[PICKSIZE]; /* keeps X-coord of an address */
extern int YROW[PICKSIZE]; /* keeps Y-coord of an address */
extern int lowerx[PICKSIZE]; /* keeps X-coords in lower layer */
extern int lowery[PICKSIZE]; /* keeps Y-coords in lower layer */
extern int upperx[PICKSIZE]; /* keeps X-coords in upper layer */
extern int uppersy[PICKSIZE]; /* keeps Y-coords in upper layer */
extern int u_count; /* numb of addresses in upper layer */
extern int d_count; /* numb of addresses in lower layer */
extern float bcost; /* cost of band tour */
extern time_t band_start; /* start of Band heuristic */
extern time_t band_end; /* end of Band heuristic */
extern time_t band_2opt_end; /* end of Band plus two_opt phase */
extern double band_time; /* run time of Band heuristic */
extern double band_2opt_time; /* run time of Band & two_opt phase */

/**************************** CHEBHULL ****************************/
define ON_CONVEXHULL 1
#define NOT_ON_CONVEXHULL 0
extern struct coordinates convexpoint[PICKSIZE]; /* keeps coords of convex hull points */
exern l_convexhull[PICKSIZE]; /* logical array - keeps track of points already on convex hull */
exern struct hull_point
    {
        int xcoord;
        int ycoord;
    };
exern struct hull_point xmax_D,xmax_U,ymax_L,ymax_R,ymin_R,xmin_U; /* points with extrem. coords */
exern int convexcount; /* counts number points on convex hull */
extern struct coordinates cheb[PICKSIZE];
   /* keeps chebhull sequence */

extern int chebcount;
   /* counts number of points on the route during opt insertion phase */

extern struct coordinates cand[PICKSIZE];
   /* keeps points in a region (neighbourhood) during opt insertion phase */

extern float chebcost;
   /* cost of the chebhull tour */

extern time_t cheb_start;
   /* start of CHEB heuristic */
extern time_t cheb_end;
   /* end of CHEB heuristic */
extern time_t cheb_2opt_end; /* end of CHEB & TWO_OPT phase */
extern double cheb_time;
   /* run time of CHEB heuristic */
extern double cheb_2opt_time; /* run time of CHEB & TWO_OPT phase */

******************************************************************************
BABTSP **************************************************************************

extern int tweight;
   /* total cost(time) of the TSP tour */
extern int best[PICKSIZE]; /* keeps the best current solution */

extern int fwdptr[PICKSIZE]; /* keeps edges from a partial solution in a form fwdptr[i]=j */

extern int backptr[PICKSIZE]; /* keeps edges from a partial solution in a form backptr[j]=i */

extern int I,J; /* control indexes */

extern time_t tsp_start; /* start of BABTSP */
extern time_t tsp_end; /* end of BABTSP */
extern time_t tsp_limit; /* current run time */
extern double tsp_time; /* total run time for BABTSP */
extern double timelimit; /* elapsed run time, to be checked against the run time limit of twelve hours */
extern float bandcost;  /* cost of BAND with average velocities */
extern float band_two;  /* cost of BAND plus TWO_OPT with average velocities */
extern int BANDX[PICKSIZE];/* addr. X-coords of BAND tour */
extern int BANDY[PICKSIZE];/* addr. Y-coords of BAND tour */
extern float RBDCOST; /* recalculated cost of BAND */
extern float RBANDCOST; /* recalc. cost of BAND plus TWO_OPT */
extern float che;  /* cost of CHEBHULL with average velocities */
extern float che_two; /* costs of CHEBHULL plus TWO_OPT with average velocities */
extern int CROUTE[PICKSIZE];/* keeps CHEBHULL route for recalculation */
extern int CHEBROUTE[PICKSIZE];/* keeps CHEBHULL plus TWO_OPT route */
struct coordinates CHE[]; /* keeps CHEBHULL route for display */
struct coordinates CHE_TWO[]; /* keeps CHEBHULL plus TWO_OPT route for display */
extern float RCHEBCOST; /* recalculated cost of CHEBHULL */
extern float RCHEBCOST; /* recalc. cost of CHEBHULL plus TWO_OPT */
extern float tspcost; /* cost of BABTSP tour calculated with average velocities */
extern int TSPROUTE [PICKSIZE]; /* keeps the average BABTSP tour for recalculation */
extern float RTSPCOST;  /* recalculated cost of BABTSP tour */
extern double runtime; /* keeps total run time of BABTSP for calculations with average velocities */
#include <stdio.h>
#include <graph.h>
#include <time.h>

#define PICKSIZE 36
#define BIGNUMB 9999
#define REAL 2
#define AVERAGE 1

/* *** ALL DECLARATIONS THAT FOLLOW ARE FOR THE EXTERNAL VARIABLES. DESCRIPTION OF THEIR MEANING IS GIVEN IN THE HEADER FILE <externs.h>. ***/

int round;
int SEED;

int maxrows;
int maxcols;
int picks;
float vx,vy;
float avel;

struct coordinates
{
    int gennumb;
    int X;
    int Y;
}
struct coordinates address[PICKSIZE];

int BW[PICKSIZE][PICKSIZE];
int BROUTE[PICKSIZE];
float two_b_cost;
/************** BAND **************/
int XCOL[PICKSIZE];
int YROW[PICKSIZE];

int lowerx[PICKSIZE];
int lowery[PICKSIZE];
int upperx[PICKSIZE];
int uppersy[PICKSIZE];
int u_count;
int d_count;
float bcost;

time_t band_start, band_end, band_2opt_end;
double band_time, band_2opt_time;

/************** CHEBHUUL **************/

struct coordinates convexpoint[PICKSIZE];
l_convxhull[PICKSIZE];
struct hull_point
{
 int xcoord;
 int ycoord;
};
struct hull_point xmax_D, xmax_U, ymax_L, ymax_R, ymin_R, xmin_U;
int convexcount;
struct coordinates cheb[PICKSIZE];
int chebcount;
struct coordinates cand[PICKSIZE];
float chebcost;

time_t cheb_start, cheb_end, cheb_2opt_end;
double cheb_time, cheb_2opt_time;

/************** BABTSP **************/
int tweight;
int best[PICKSIZE];
int fwdptr[PICKSIZE];
int backptr[PICKSIZE];
int I, J;
time_t tsp_start, tsp_end, tsp_limit;
double tsp_time, runtime, timelimit;
float bandcost, band_two;
int BANDX[PICKSIZE], BANDY[PICKSIZE];
float RBDCOST = 0, RBANDCOST = 0;

float che, che_two;
int CHEBROUTE[PICKSIZE], CROUTE[PICKSIZE];
struct coordinates CHE[PICKSIZE], CHE_TWO[PICKSIZE];
float RCHBCOST = 0, RCHEBCOST = 0;

float tspcost;
int TSPROUTE [PICKSIZE];
float RTSPCOST = 0;

main()
{
    _clearscreen ( _GCLEARSCREEN );

    input();
    randgen();

    for(round = AVERAGE; round <= REAL; round++)
    {
        if(round == REAL)
        {
            /* fileinput(); */ /* used only when new data are stored */
        }

        band();
        _clearscreen ( _GCLEARSCREEN );

        chebhull();
        _clearscreen ( _GCLEARSCREEN );

        babtsp();
        _clearscreen ( _GCLEARSCREEN );
    }
}
#include "externs.h"

input()
{
    float height;

    printf("\n Enter number of picks (up to 35): ");
    scanf("%d",&picks);
    if ((picks > 35) || (picks <= 0))
    {
        printf("INCORRECT INPUT- NUMB OF PICKS SHOULD BE BETWEEN 1 AND 35\n");
        exit(1);
    }

    printf("\n Enter max number of columns (up to 100): ");
    scanf("%d",&maxcols);
    if ( (maxcols > 100) || (maxcols < 1) )
    {
        printf("INCORRECT INPUT- NUMB OF COLUMNS SHOULD BE BETWEEN 1 AND 100\n");
        exit(1);
    }

    printf("\n Enter horizontal velocity: ");
    scanf("%f",&vx);
    if (vx <= 0)
    {
        printf("INCORRECT INPUT-SHOULD BE POSITIVE NUMBER \n");
        exit(1);
    }

    printf("\n Enter vertical velocity: ");
scanf("%f", &vy);
if (vy <= 0)
{
    printf("INCORRECT INPUT—SHOULD BE POSITIVE NUMBER \n");
    exit(1);
}

printf("\n Enter velocity vector (between 0.5-2.0) a = ");
scanf("%f", &avel);
if ((avel < 0.5) || (avel > 2.0))
{
    printf("INCORRECT INPUT—SHOULD BE A NUMBER BETWEEN 0.5-2.0\n");
    exit(1);
}

height = (avel * maxcols * vy) / vx;
maxrows = height + 0.5;

if (height > 20)
{
    printf("RACK TOO HIGH! SHOULD BE LESS THAN 20\n");
    exit(1);
}

if (picks > (maxrows * maxcols))
{
    printf("INCORRECT INPUT—NUMB OF ADDRESSES GREATER THAN TOTAL NUMB OF CELLS\n");
    exit(1);
}

}
/********************************************************************
  *
  * source file <FILEINPUT.C>
  *
  * Function "fileinput" writes into files <hor> and <vert> travel
  * times in horizontal and vertical direction respectively.
  *
  *********************************************************************/

#include "externs.h"
#include <stdio.h>

fileinput()
{
  FILE *stream;
  int i;
  int hor[101], vert[21];
  int numwritten;

  for(i = 0; i <= 101; i++)
  {
    printf("Enter hor[%d]:", i);
    scanf("%d", &hor[i]);
  }

  stream = fopen("hor", "w");
  if(stream == (FILE *)NULL)
  {
    printf("Cannot open hor file");
    exit(1);
  }

  numwritten = fwrite((char*)hor, sizeof(int), 102, stream);
  printf("wrote in hor %d items\n", numwritten);
  fclose(stream);

  for(i = 0; i <= 21; i++)
  {
    printf("Enter vert[%d]:", i);
    scanf("%d", &vert[i]);
  }
}
stream = fopen("vert","w");
if(stream == (FILE *)NULL)
{
    printf("Cannot open vert file");
    exit(1);
}

numwritten = fwrite((char*)vert,sizeof(int),22,stream);
printf("wrote in vert %d items\n",numwritten);
fclose(stream);
Function "randgen"-description

Function "randgen" generates random addresses with uniform distribution within the rack boundaries. Coordinates of each address and its generation number are stored in arrays.

```plaintext
randgen()
begin
    set first address to be the I/O point with coordinates (1,1)

    for(i = 2 to picks)
    begin
        generate address X and Y coordinates
        while( address coincides with an already stored address)
        begin
            generate address X and Y coordinates
        end
        store generated address coords and its generation number,i
    end
end
```
/*********************************************************************/
* 
* source file < RANDGEN.C> 
* Function "randgen" generates random addresses with uniform 
* distribution within the boundaries of the rack. Coordinates of 
* each address and its generation number are stored in arrays. 
* 
*********************************************************************/

#include "externs.h"

#define UNIQUE 1
#define NOT_UNIQUE 0

randgen()
{
    unsigned int seed;
    int i,j;
    float a,b;
    float x,y;
    int state ;

    XCOL[1]=1; /* address with coordinates (1,1) is the I/O point */
    YROW[1]=1;

    printf("\n enter seed:");
    scanf("%d",&seed);
    srand(seed);

    for(i = 2;i <= picks;i++)
    {
        state = NOT_UNIQUE;
        while( state == NOT_UNIQUE)
        {
            state = UNIQUE;

            a = rand();
            b = rand();

            x = a;
            y = b;

            XCOL[i]=x;
            YROW[i]=y;
        }
    }
}
\[ x = a \cdot \text{maxcols}/33767 + 1; \]
\[ y = b \cdot \text{maxrows}/33767 + 1; \]

while \((x < 1) \lor (y < 1)\) {
\[ a = \text{rand}(); \]
\[ b = \text{rand}(); \]
\[ x = a \cdot \text{maxcols}/33767 + 1; \]
\[ y = b \cdot \text{maxrows}/33767 + 1; \]
}

for \(j = 1; j \leq i; j++\) {
if \((\text{YROW}[j] == \text{(int)}y) \land \text{XCOL}[j] == \text{(int)}x)\) {

printf("*******\n");
state = NOT_UNIQUE;
break;
}
}

YROW[i] = y;
XCOL[i] = x;

address[i].X = x;
address[i].Y = y;
address[i].gennumb = i;
}
/* end for */

} /* end */
Function "band"-description

Function "band" implements BAND heuristic for solving TSP

The function operates on the addresses generated by function "randgen". These addresses are split into two levels (layers)- upper and lower, according to their Y coordinate. If an address Y coordinate is larger than half of the rack height then the address is directed to upper layer, if not the address becomes part of the lower layer.

Addresses in each layer are sorted in ascending order of their X coordinate. If some addresses in the same layer happen to have common X coordinate they are sorted in ascending order of their Y coordinate.

Addresses are linked in a tour starting from the first address in the lower layer. The last address in the lower layer is linked to the last address in the upper layer. Addresses of upper layer are linked in reversed order. The last address of the upper sequence is linked to the first lower address and the tour is completed.

The cost (travel time) of the route is then calculated.

A TWO OPTIMAL (local search) improvement procedure is applied to the tour created by BAND heuristic and its cost calculated.
Function "band" - structure

band()
begin
  split();        /* split addresses into two layers
                  according to their Y coord */

  bansor(lower layer);    /* sort addresses in lower layer in
                          ascending order of their X coord */

  colsort(lower layer);   /* sort addresses with same X coord in
                          ascending order of their Y coord */

  bansor(upper layer);    /* sort addresses in upper layer in
                          ascending order of their X coord */

  colsort(upper layer);   /* sort addresses with same X coord in
                          ascending order of their Y coord */

  rejoin();              /* link addresses of lower layer with
                          the addresses of upper layer, last ones
                          in reversed order */

  distmat();             /* fill distance matrix and calculate
                          the cost (travel time of the tour) */

  two_opt(route, dist matrix, cost);  
                          /* two optimal improvement procedure
                             on the tour created so far */

  brecalc();            /* recalculates cost (trav. time) of
                         the tour created by using const
                         velocities, with real times taken from
                         a manufacturer */
end
#include "externs.h"

band()
{
  int i;
time(&band_start);

  split();

  /*************** sort lower layer ***************/
  bandqsort(&lowerx[1],&lowerx[lowerx[d_count]],&lowery[1],&lowery[lowery[d_count]]);
  colsort(lowerx,d_count,lowery);

  /*************** sort upper layer ***************/
  bandqsort(&upperx[1],&upperx[upperx[u_count]],&uppery[1],&uppery[uppery[u_count]]);
  colsort(upperx,u_count,uppery);

  /*********************************************/

  rejoin();
distmat();

  if(round == AVERAGE) bandcost = bcost; /* for graph output */
two_b_cost = bcost; /* for graph output */
time(&band_end);

  two_opt(BW,BROUTE,bcost,&two_b_cost); /* 4th arg for graph output */
time(&band_2opt_end);

  band_time = difftime(band_end,band_start);
  band_2opt_time = difftime(band_2opt_end,band_start);

  if(round == AVERAGE) /* for graph display */
  {
    /* for graph display */
  }
}
band_two = two_b_cost;
for(i=1;i <= picks;i++)
{
    BANDX[i] = XCOL[BROUTE[i]];
    BANDY[i] = YROW[BROUTE[i]];
}
else
{
    brecalc(); /* recalculates average vel.tour with real times */
}
getchar();
getchar();
display(); /* graphic display, not presented here */
Function "split"-description

Function "split" separates previously generated addresses into upper and lower layer represented by corresponding arrays. Number of addresses in each layer is recorded.

split()
begin

splitlev = half of rack height;
counter of lower level = 0;
counter of upper level = 0;

for(i=1 to picks )
begin
    if(Y-coordr of i-th adress <= splitlev)
        begin
            increase counter of lower level by one;
            record the coordinates of the i-th adress in lower layer;
        end
    else
        begin
            increase counter of upper level by one;
            record the coordinates of the i-th adress in upper layer;
        end
end /* for */
end
/*********************************************************************/
/*
* source file <SPLIT.C>
* Function "split" separates generated addresses into upper and *
* lower level (layer) according to their Y-coordinate. *
* 
*********************************************************************/

#include "externs.h"

split()
{
  int splitlev;
  int i;

  splitlev = maxrows/2.0 + 0.5;
  u_count=d_count=0;

  for( i=1;i<=picks;i++)
  {
    if(YROW[i]<=splitlev)
    {
      d_count++;

      lowerx[d_count]= XCOL[i];
      lowery[d_count]= YROW[i];
    }
    else
    {
      u_count++;
      upperx[u_count]= XCOL[i];
      uppery[u_count]= YROW[i];
    }

  }/* end for */
}
Function "bandqsor"

Function "bandqsor" is a modified version of "qicksort" "C" procedure given by Hutchison and Just [1988] for sorting elements of an array in ascending order. The present function "bandqsor" accepts four arguments. The first two are the starting and the last address of the array to be sorted. This is the array that keeps the X-coordinates of rack locations of lower or upper layer. The next two arguments are the starting and the last address of the array that keeps the correspondent Y-coordinates of the locations. During the procedure the first array is sorted while the elements of the second one follow their counterparts in their relative positions. In the programme, lines that differ from the original "qicksort" are noted with an arrow.
Source file <BANDQSOR.C>

Function "bandqsor" sorts elements of an array in ascending order while the corresponding elements of the second array in the function follow the same procedure without being sorted.

#include "externs.h"

bandqsor( lower, upper, y_lower, y_upper )

int *lower, *upper;
int *y_lower, *y_upper; /* <- */

int partition;
int *iptr, *previous_low;
int *y_previous_low, *y_iptr; /* <- */

if (lower < upper)
{
    partition = *lower;
    previous_low = lower;

    y_previous_low = y_lower; /* <- */
    y_iptr = y_lower + 1; /* <- */

    for (iptr = lower + 1; iptr <= upper; iptr++)
    {
        if ( *iptr < partition )
        {
            previous_low++;
            swap( previous_low, iptr);

            y_previous_low++; /* <- */
            swap( y_previous_low, y_iptr); /* <- */
        }
        y_iptr++; /* <- */
    }
swap( lower, previous_low);
swap(y_lower, y_previous_low); /* <- */

bandqsor( lower, previous_low-1, y_lower, y_previous_low-1);
bandqsor( previous_low +1, upper, y_previous_low+1, y_upper);

} /* end if */

} /* end */
Function "colsort" accepts two arrays with same size. If at least two elements of the first array which has already been sorted in ascending order are the same, then their corresponding elements in the second array are sorted in ascending order.

```
#include "externs.h"

colsort(col_array, level, row_array)

int col_array[], row_array[];
int level; /* array size */
{
    int i, j;
    int *previous, *current;

    previous = &row_array[1];
    current = &row_array[1];

    for (i=1; i<=level; i++)
    {
        if (col_array[i] == col_array[i+1])
        {
            current++;
            quicksort(previous, current);
        }
        else
        {
            current++;
            previous = current;
        }
    }
}
**Function "quicksort"**

Function "quicksort" is a "C" procedure given by Hutchison and Just [1988] for sorting elements of an array in ascending order. The function accepts two arguments. They are the starting and the last memory address of the array to be sorted.

```c
quicksort(lower, upper)

lower- pointer to the first address of the array to be sorted
upper- pointer to the last address of the array to be sorted

begin
  if(lower < upper)
  begin
    for(pointer = lower+1 to upper)
      begin
        if(contents of pointer < contents of lower)
          begin
            search further up in the array until find current address with contents larger than the contents of lower;
            remember this as current address;
            search further up until find an address with contents less than the contents of lower;
            swap (contents of last found address and contents of current address);
          end
      end
    remember address of last swapped element of the array;
    swap(contents of lower and contents of last swapped address);
  now the array is divided in half by the new position of lower;
  apply recursively quicksort to these halves;
  end /* if */
end
```
#include "externs.h"

quicksort( lower, upper )

int *lower, *upper;
{
  int partition;
  int *iptr, *previous_low;

  if(lower<upper)
  {
    partition = *lower;
    previous_low = lower;

    for(iptr = lower +1; iptr <= upper; iptr++)
    {
      if( *iptr < partition )
      {
        previous_low++;
        swap( previous_low, iptr);
      }
    }

    swap( lower, previous_low);

    quicksort( lower, previous_low-1);
    quicksort( previous_low +1, upper);
  }/* end if */
}
swap (low, high)
int *low, *high;
{
  int temp;
  temp = *low;
  *low = *high;
  *high = temp;
}
source file <REJOIN.C>

Function "rejoin" rejoins addresses from lower and upper level after they have already been sorted in each layer. Thus they form the tour.

_ENTRIES

*************************************************************************/

#include"externs.h"

rejoin()
{
    int i,j,up;
    j=1;
    up = u_count;
    for(i = 1;i <= d_count;i++)
    {
        XCOL[i] = lowerx[i];
        YROW[i] = lowery[i];
        j++;
    }
    while(j <= picks)
    {
        XCOL[j] = upperx[up];
        YROW[j] = upery[up];
        up--;
        j++ ;
    }
}
/*
 * Function "distmat" calculates distance matrix for an already
 * constructed BAND tour. It then calculates the cost (travel time)
 * of that tour using the cost (travel time) matrix.
 */

#include "externs.h"

distmat()
{
    float hor, vert;
    float dX, dY;
    int dx, dy;
    int hori[101], verti[21];
    int horread, vertread;

    FILE *streamhor;
    FILE *streamvert;

    bcost=0.0;

    if(round == AVERAGE) /**< simulation with average velocities **/
    {
        for(I = 1; I <= picks; I++)
        {
            for(J = I+1; J <= picks; J++)
            {
                dX = (float)(XCOL[I] - XCOL[J]);
                dY = (float)(YROW[I] - YROW[J]);

                hor = (dX/vx)*100; /* travel time in horiz. direction */
                vert = (dY/vy)*100; /* travel time in vert. direction */

                if(hor < 0)
                {
                    hor = -hor;
                }
            }
        }
    }
}

31
if(vert < 0)
{
    vert = -vert;
}

/* distance(travel time) between any two locations is the larger distance (travel time) between the locations in horizontal or vertical direction */

if(hor >= vert)
{
    BW[I][J] = hor;
}
else
{
    BW[I][J] = vert;
}

BW[J][I] = BW[I][J];

}

for(I = 1;I <= picks-1;I++)
{
    bcost = bcost + BW[I][I+1];
}

bcost = bcost + BW[picks][1];

/* the sequence of addresses in the route is in an ascending order because the coordinates have already been sorted for Band */

for(I = 1;I <= picks;I++)
{
    BROUTE[I] = I;
}
} /* end if AVERAGE */
else  /*** simulation with real travel times ***/
{

streamhor = fopen("hor","r+"); /* read real horiz.travel times */
if(streamhor == (FILE *)NULL)
{
    printf("Cannot open hor file for reading\n");
    exit(1);
}
horread = fread((char *)hori,sizeof(int),102,streamhor);
printf("horead in BAND=%d\n",horread);
fclose(streamhor);

streamvert = fopen("vert","r+");/* read real vert. travel times */
if(streamvert == (FILE *)NULL)
{
    printf("Cannot open vert file for reading\n");
    exit(1);
}
vertread = fread((char *)verti,sizeof(int),22,streamvert);
printf("vertread in BAND=%d\n",vertread);
fclose(streamvert);

for(I = 1;I <= picks;I++)
{
    for(J = I+1;J <= picks;J++)
    {
        dx = (XCOL[I]-XCOL[J]);
        dy = (YROW[I]-YROW[J]);

        if(dx < 0)
        {
            dx = -dx;
        }

        if(dy < 0)
        {
            dy = -dy;
        }
    }
}
if(hori[dx] >= verti[dy])
{
    BW[I][J] = hori[dx];
}
else
{
    BW[I][J] = verti[dy];
}

BW[J][I] = BW[I][J];

}

for(I = 1;I <= picks-1;I++)
{
    bcost = bcost + BW[I][I+1];
}

bcost = bcost + BW[picks][1];

RBDCOST = bcost; /* FOR GRAPHIC OUTPUT */

/* the sequence of addresses in the route is in an ascending order because the coordinates have already been sorted for Band */

for(I = 1;I <= picks;I++)
{
    BROUTE[I] = I;
}

} /* end else */

} /* END */
source file <TWO_OPT.C>

Function "two_opt" is an implementation of the two optimal local search algorithm (see Syslo et al[1983]). The function takes distance matrix, initial tour and its cost. Then two links of the initial tour are replaced by two other links that have not yet been included to form a new tour. If the new tour is better it is stored. Procedure terminates at a local optimum when it is not possible to improve the tour by further link exchange.

/***************************************************************************/

#include "externs.h"

two_opt(TWO_DIST,ROUTE,TWEIGHT,TWEIGHT_GRAPH)

int TWO_DIST[PICKSIZE];
int ROUTE[];
float TWEIGHT[];
float *TWEIGHT_GRAPH;

{    
   int ahead;
   int i,I1,I2,index;
   int j,J1,J2,last;
   int limit,next;
   int max,max1;
   int s1,s2,t1,t2;
   int PTR[PICKSIZE];

   for(i = 1;i <= picks-1;i++)
   {
     PTR[ROUTE[i]] = ROUTE[i+1];
   }
   PTR[ROUTE[picks]] = ROUTE[1];

   do
   {
     max = 0;
     I1 = 1;

   }
for(i = 1; i <= picks-2; i++)
{
    if(i == 1)
    {
        limit = picks - 1;
    }
    else
    {
        limit = picks;
    }
    I2 = PTR[I];
    J1 = PTR[I2];
    for(j = i+2; j <= limit; j++)
    {
        J2 = PTR[J1];
        max1 = TWO_DIST[I][I2]+TWO_DIST[J1][J2] -
                (TWO_DIST[I][J1]+TWO_DIST[I2][J2]);
        if(max1 > max)
        {
            s1 = I1;
            s2 = I2;
            t1 = J1;
            t2 = J2;
            max = max1;
        }
        J1 = J2;
    }
    I1 = I2;
} /* end for */

if(max > 0)
{
    PTR[s1] = t1;
    next = s2;
    last = t2;
    do
    {
        ahead = PTR[next];
        PTR[next] = last;
        last = next;
        next = ahead;
    } while(next != t2);
TWEIGHT = TWEIGHT - max;
*TWEIGHT_GRAPH = TWEIGHT;   /* for graph purposes */
}
}
while(max != 0); /* end do */

index = 1;
for(i = 1; i <= picks; i++)
{
   ROUTE[i] = index;
   index = PTR[index];
}
}
#include "externs.h"

brecalc()
{
  int i;
  int dx,dy;
  int hor,vert;
  int hori[101],verti[21];
  int horread,vertread;

  FILE *streamhor;
  FILE *streamvert;

  streamhor=fopen("hor","r+");
  if(streamhor == (FILE *)NULL)
  {
    printf("Cannot open hor file for reading\n");
    exit(1);
  }
  horread=fread((char *)hori,sizeof(int),102,streamhor);
  printf("horread in BAND=%d\n",horread);
  fclose(streamhor);

  streamvert=fopen("vert","r+");
  if(streamvert == (FILE *)NULL)
  {
    printf("Cannot open vert file for reading\n");
    exit(1);
  }
}
vertread=fread((char *)&verti,sizeof(int),22,streamvert);
printf("vertread in BAND=%d\n",vertread);
fclose(streamvert);

for(i=1;i<=picks-1;i++)
{
    dx = XCOL[BROUTE[i]]-XCOL[BROUTE[i+1]];
    if(dx <0) dx=-dx ;

    dy=YROW[BROUTE[i]]-YROW[BROUTE[i+1]];
    if(dy <0) dy=-dy ;

    hor =hori[dx];
    vert=verti[dy];

    RBANDCOST=RBANDCOST + (int)max(hor,vert);
}

dx= XCOL[BROUTE[picks]]-XCOL[BROUTE[1]];
    if(dx <0) dx=-dx ;

    dy=YROW[BROUTE[picks]]-YROW[BROUTE[1]];
    if(dy <0) dy=-dy ;

    hor =hori[dx];
    vert=verti[dy];

    RBANDCOST=RBANDCOST + (int)max(hor,vert);
}
Function "chebhull"-description

Function "chebhull" is an implementation of the Convex hull approximate algorithm for solving TSP proposed by W.R. Stewart (see Golden et al [1980] and Allison and Noga [1984]). When Tchebyshev norm is applied, an intermediate insertion procedure proposed by Goetschalckx [1983,1985] is included. As with Band heuristic a two optimal improvement procedure is applied.

```
chebhull()
begin
    find the convex hull of the set of addresses; /* function convexhull() */
    insert as many addresses as possible between all pairs of consecutive points on the convex hull without increasing the travel time between the convexhull points; /* function optinsrt() */
    insert the rest of the addresses one at a time between two consecutive points on the partial tour in a way to minimize the total length (cost) of the tour; /* function mininsrt() */
    calculate cost of the tour; /* function chebdist() */
    apply two optimal improvement procedure and calculate the cost of the improved tour; /* function two_opt() */
end
```
#include "externs.h"

chebhull()
{
    int i;

    if (round == AVERAGE)
    {
        for (i = 1; i <= picks; i++)
            {
                CROUTE[i] = i;        /* keeps norm. chebroute for recalc. */
                CHE[i] = cheb[i];
            }
    }

time(&cheb_start);
convexhull();
optinsrt();
mininsrt();
chebdist();

two_b_cost = chebcost;
time(&cheb_end);

    if (round == AVERAGE)
    {
        for (i = 1; i <= picks; i++)
            {   

CROUTE[i] = i; /* keeps norm chebroute for recalc. */
CHE[i] = cheb[i];
}
}
two_opt(BW,BROUTE, chebcost, &two_b_cost); /* 4th arg for graph
purposes only */
time(&cheb_2opt_end);
cheb_time = difftime(cheb_end, cheb_start);
cheb_2opt_time = difftime(cheb_2opt_end, cheb_start);

if(round == AVERAGE)
{
    che_two = two_b_cost; /* for display */

    for(i = 1;i <= picks;i++)
    {
        CHEBROUTE[i] = BROUTE[i]; /* save the two_opt route for
recalculation of tour cost */
        CHE_TWO[i] = cheb[i];
    }
    getchar();
    getchar();
    chebdisp();
}
else
{
    realmatr();

    RCHBCOST = RCHBCOST+ recalc(CROUTE,CHE);
    RCHEBCOST= RCHEBCOST+recalc(CHEBROUTE,CHE_TWO);

    getchar();
    getchar();
    chebdisp();
}
}
/*****source file < CONVEXHUL.C >*****

* Function "convexhull" initialises the logical array that keeps
  * track which addresses have been found to be on the convex hull.
  * Then it calls function "extrcoord" which finds addresses with
    * extreme coordinates and function "regions" which, in turn, finds
    * the rest of the addresses on the convex hull.

***************************************************************************/

#include "externs.h"

#define ON_CONVEXHULL 1
#define NOT_ON_CONVEXHULL 0

convexhull()
{
  int i;

  for(i = 1; i<= picks; i++)
  {
    l_convexhull[i] = NOT_ON_CONVEXHULL;
  }

  extrcoord();
  regions();
}
Function "extrcoor" description

Function "extrcoor" finds points (addresses) with extreme coordinates as shown in fig. A2. This is a first step for obtaining the convex hull of the set of generated addresses. Since some of the extreme points can coincide, the heptagon in fig. A2 can reduce to hexagon, pentagon, quadrilateral, triangle or even a line segment. The algorithm for finding addresses with extreme coordinates generated in a rectangular area (the rack) follows.

extrcoor()
begin

if (there are addresses on first row other than the I/O point )
begin
    find address with max x coordinate;
    set this point to be ymin_R;
end

if (there are addresses on first col. other than the I/O point )
begin
find address with max y coordinate;  
set this point to be xmin_U;
end

sort all points in ascending order of their x coordinate;

if (there are points with the same max x coordinate) begin
    sort these points in ascending order of their y coordinate;
    set point with min y coordinate to be xmax_D;
    set point with max y coordinate to be xmax_U;
end

sort all points in ascending order of their y coordinate;

if (there are points with the same max y coordinate) begin
    sort these points in ascending order of their x coordinate;
    set point with min x coordinate to be ymax_L;
    set point with max x coordinate to be ymax_R;
end

end
/*********************************************************************/
/*
* source file < EXTRCOOR.C >
* Function "extrcoor" finds addresses with extreme coordinates in
* a rectangular area.
* *
*********************************************************************/

#include "externs.h"
#define TEMPSIZE 15

extrcoor()
{
    int point_x[PICKSIZE];
    int point_y[PICKSIZE];

    int XM_X[TEMPSIZE];
    int XM_Y[TEMPSIZE];

    int YM_X[TEMPSIZE];
    int YM_Y[TEMPSIZE];

    int i,count,br;

    ymin_R.xcoord = 1;
    ymin_R.ycoord = 1;
    xmin_U.xcoord = 1;
    xmin_U.ycoord = 1;

    for(i = 1;i <= picks;i++)
    {
        point_x[i] = address[i].X;
        point_y[i] = address[i].Y;

        if((address[i].X ==1) && (address[i].Y >= xmin_U.ycoord))
        {
            xmin_U.xcoord = address[i].X;
            xmin_U.ycoord = address[i].Y;
        }
    }
if((address[i].Y == 1) && (address[i].X >= ymin_R.xcoord))
{
    ymin_R.xcoord = address[i].X;
    ymin_R.ycoord = address[i].Y;
}
} /* end for */

bandqsor(&point_x[1],&point_x[picks],&point_y[1],&point_y[picks]);
XM_X[1] = point_x[picks];
XM_Y[1] = point_y[picks];
count = picks;
br = 1;

while(point_x[count] == point_x[count-1])
{
    XM_X[br] = point_x[count];
    XM_Y[br] = point_y[count];
    br++;
    XM_X[br] = point_x[count-1];
    XM_Y[br] = point_y[count-1];
    count--;
}

bandqsor(&XM_Y[1],&XM_Y[br],&XM_X[1],&XM_X[br]);
xmax_D.xcoord = XM_X[1];
xmax_D.ycoord = XM_Y[1];

xmax_U.xcoord = XM_X[br];
xmax_U.ycoord = XM_Y[br];

bandqsor(&point_y[1],&point_y[picks],&point_x[1],&point_x[picks]);
YM_X[1] = point_x[picks];
YM_Y[1] = point_y[picks];
count = picks;
br = 1;

while(point_y[count] == point_y[count-1])
{
    YM_X[br] = point_x[count];
YM_Y[br] = point_y[count];
br++;
YM_X[br] = point_x[count-1];
YM_Y[br] = point_y[count-1];
count--;
}
bandqsort(&YM_X[l],&YM_X[br],&YM_Y[l],&YM_Y[br]);
ymax_L.xcoord = YM_X[l];
ymax_L.ycoord = YM_Y[l];
ymax_R.xcoord = YM_X[br];
ymax_R.ycoord = YM_Y[br];
}
Function "regions"-description

Function "regions" first finds all addresses (if any) which lie in regions I, II and III as denoted in fig. A2. These addresses are candidates to be on the convex hull. Each address is checked against the three equations of a straight line, each defined by two points: \( \{y_{min} \_R, x_{max} \_D\} \), \( \{x_{max} \_U, y_{max} \_R\} \) and \( \{y_{max} \_L, x_{min} \_U\} \), in order to determine if the address belongs to one of the regions. It should be noted that if some of the extreme points coincide, the number of regions can reduce to two, one or zero. The last case is possible when the extreme points happen to coincide with the corners of the rack.

Second, points in each region are sorted in ascending order of their x coordinate.

Third, using the vector cross-product rule proposed by Akl and Toussaint [1978], addresses that lie on the convex hull in each region are found.

```
regions()
begin
    struct coordinates REG1[ ], REG2[ ], REG3[ ]; /* arrays of structures which keep coordinates and generation numbers of addresses that lie in the corresponding regions */

    for(i = 1 to picks) begin
        if( address[i] is in region I) begin
            REG1[i] = address[i];
        end

        if( address[i] is in region II) begin
            REG2[i] = address[i];
        end

        if( address[i] is in region III) begin
```

49
REG3[i] = address[i];
end
end /*end for */

sort addresses in each region in ascending order of their x coordinate;

apply the vector crossproduct rule for the addresses in each region to find those ones that lie on the convex hull;

end
#include "externs.h"

regions()
{
    struct coordinates REG1[PICKSIZE], REG2[PICKSIZE], REG3[PICKSIZE];
    float grad1, grad2, grad3;
    int regcount1 = 0, regcount2 = 0, regcount3 = 0;
    int i;

    grad1 = (float) ((ymin_R.ycoord - xmax_D.ycoord) / ((ymin_R.xcoord -
    xmax_D.xcoord) + 0.0001));
    grad2 = (float) ((xmax_U.ycoord - ymax_R.ycoord) / ((xmax_U.xcoord -
    ymax_R.xcoord) + 0.0001));
    grad3 = (float) ((ymax_L.ycoord - xmin_U.ycoord) / ((ymax_L.xcoord -
    xmin_U.xcoord) + 0.0001));

    for(i = 1; i <= picks; i++)
    {
        if (((float) (grad1 * address[i].X - address[i].Y + 0.0001) >=
            (float) (-xmax_D.ycoord + grad1 * xmax_D.xcoord)) &&
            (address[i].X >= ymin_R.xcoord) && (address[i].Y <=
            xmax_D.ycoord))
        {
            regcount1++;
            REG1[regcount1] = address[i];
        }

        if (((float) (-grad2 * address[i].X + address[i].Y + 0.0001) >=
            (float) (ymax_R.ycoord - grad2 * ymax_R.xcoord)) &&
            (address[i].X >= ymax_R.xcoord) && (address[i].Y >=
            xmax_U.ycoord))
        {
            regcount2++;
            REG2[regcount2] = address[i];
if(((float)(grad3 * address[i].X - address[i].Y) <=
(float)(- xmin_U.ycoord + grad3 * xmin_U.xcoord + 0.0001)) &&
(address[i].X <= ymax_L.xcoord) && (address[i].Y >=
xmin_U.ycoord))
{
    regcount3++;
    REG3[regcount3]=address[i];
}

regqsort(&REG1[1], v&REG1[regcount1]);
regqsort(&REG2[1], &REG2[regcount2]);
regqsort(&REG3[1], &REG3[regcount3]);

crosprod1(REG1, regcount1);
crosprod2_n_3(REG2, regcount2);
crosprod2_n_3(REG3, regcount3);
source file < REGQSORT.C >

Function "regqsort" sorts addresses in ascending order of their x coordinate. It is an analogue of function "quicksort". The difference here is that addresses coordinates and generation numbers are handled by structure manipulation.

Function "descend_x" is an analogue of "regqsort". It sorts addresses in descending order of their x coordinate.

Function "ascend_y" is an analogue of "regqsort". It sorts addresses in ascending order of their y coordinate.

Function "descend_y" is an analogue of "regqsort". It sorts addresses in descending order of their y coordinate.

Function "sswap" exchanges contents of two structures.

#include "externs.h"

regqsort(p_lower, p_upper)
struct coordinates *p_lower, *p_upper;
{
    struct coordinates partition;
    struct coordinates *iptr, *previous_low;

    if(p_lower < p_upper)
    {
        partition = *p_lower;
        previous_low = p_lower;

        for(iptr = p_lower; iptr <= p_upper; iptr++)
        {
            if(iptr->x < partition.X)
            {
                previous_low ++;
                sswap(previous_low, iptr);
            }
        }
    }
}
sswap(p_lower, previous_low);
regqsort(p_lower, previous_low-1);
regqsort(previous_low+1, p_upper);
} /* end if */

/***
 * function "sswap"
 ***/

/***
 *** swaps contents of two structures ***/

sswap(low, high)
struct coordinates *low, *high:
{
    struct coordinates temp;
    temp = *low;
    *low = *high;
    *high = temp;
}
/***/ function "descend_x" /***/
/***/ sorts addresses in descending order of their x coordinate /***/

descend_x(p_lower, p_upper)
struct coordinates * p_lower, * p_upper;
{
    struct coordinates partition;
    struct coordinates * iptr, * previous_low;

    if(p_lower < p_upper)
    {
        partition = * p_lower;
        previous_low = p_lower;

        for(iptr = p_lower; iptr <= p_upper; iptr++)
        {
            if(iptr->X > partition.X)
            {
                previous_low ++;
                sswap(previous_low, iptr);
            }
        }
        sswap(p_lower, previous_low);
        descend_x(p_lower, previous_low-1);
        descend_x(previous_low+1, p_upper);
    }
}
/***/

```c
struct coordinates partition;
struct coordinates * iptr, * previous_low;

if(p_lower < p_upper)
{
    partition = * p_lower;
    previous_low = p_lower;

    for(iptr = p_lower; iptr <= p_upper; iptr++)
    {
        if(iptr->Y < partition.Y)
        {
            previous_low ++;
            sswap(previous_low, iptr);
        }
    }
    sswap(p_lower, previous_low);
    ascend_y(p_lower, previous_low-1);
    ascend_y(previous_low+1, p_upper);
}
```
/***/ function <descend_y> ***/
/***/ sorts addresses in descending order of their y coordinate /***/

descend_y(p_lower, p_upper)
struct coordinates * p_lower, * p_upper;
{
    struct coordinates partition;
    struct coordinates * iptr, * previous_low;

    if (p_lower < p_upper)
    {
        partition = * p_lower;
        previous_low = p_lower;

        for (iptr = p_lower; iptr <= p_upper; iptr++)
        {
            if (iptr->Y > partition.Y)
            {
                previous_low ++;
                sswap(previous_low, iptr);
            }
        }
        sswap(p_lower, previous_low);
        descend_y(p_lower, previous_low-1);
        descend_y(previous_low+1, p_upper);
    }
}
Function "crosprod1" is an implementation of the vector cross-product rule of an algorithm for finding points on the convex hull proposed by Akl and Toussaint [1978]. This function is applied only to the addresses in Region I (fig. A2). Function "crosprod" has two arguments: coordinates of the addresses in the region and number of addresses in the region.

Function "crosprod2_n_3" is an analogue of "crosprod1" and is applied to the addresses in Region II and Region III. This function applies the vector cross-product rule to the addresses in the region in a reversed order of their sort, in order to keep the sequence of points on the convex hull in anticlockwise direction.

```c
#include "externs.h"

/*** function "crosprod1" ***/

crosprod1(reg, points_in_region)
struct coordinates reg[];
int points_in_region;
{
    struct coordinates *first, *middle, *last;
    int k = 1, i;
    convexcount = 0;

    if((address[1].X != reg[1].X) || (address[1].Y != reg[1].Y))
    {
        convexcount = 1;
        convexpoint[convexcount] = address[convexcount];
        l_convexhull[convexcount] = ON_CONVEXHULL;
    }

    switch(points_in_region)
    {
```
case 1:
    convexcount++;
    convexpoint[convexcount] = reg[points_in_region];
    l_convexhull[reg[points_in_region].gennumb] = ON_CONVEXHULL;
    break;

case 2:
    convexcount++;
    convexpoint[convexcount] = reg[1];
    l_convexhull[reg[1].gennumb] = ON_CONVEXHULL;
    convexcount++;
    convexpoint[convexcount] = reg[2];
    l_convexhull[reg[2].gennumb] = ON_CONVEXHULL;
    break;

case 3:
    {
        for(i = 1;i <= points_in_region;i++)
        {
            convexcount++;
            convexpoint[convexcount] = reg[i];
            l_convexhull[reg[i].gennumb] = ON_CONVEXHULL;
        }
    }
    else
    {
        convexcount++;
        convexpoint[convexcount] = reg[1];
        l_convexhull[reg[1].gennumb] = ON_CONVEXHULL;
        convexcount++;
        convexpoint[convexcount] = reg[2];
        l_convexhull[reg[2].gennumb] = ON_CONVEXHULL;
    }
    break;

default:
    convexcount++;
    convexpoint[convexcount] = reg[1];
l_convexhull[reg[1].gennumb] = ON_CONVEXHULL;

first = (reg +1);
middle = &reg[2];
last = &reg[3];

while(last <= &reg[points_in_region])
{
  if(((float)(middle->Y - first->Y) * (last->X -
middle->X) + (float)(first->X - middle->X) *
(last->Y - middle->Y)) <= 0)
  {
    convexcount++;
    l_convexhull[middle->gennumb] = ON_CONVEXHULL;
    convexpoint[convexcount] = *middle;
    first = middle;
    middle++;
    last++;
  }
  else
  {
    middle++;
    last++;
  }
}
last--;
convexcount++;
l_convexhull[last->gennumb] = ON_CONVEXHULL;
convexpoint[convexcount] = *last;
break;
} /* end switch */

} /* end */
function <crosprod2_n_3> */

crosprod2_n_3(reg, points_in_region)
struct coordinates reg[];
int points_in_region;
{

struct coordinates *first, *middle, *last;
int i;

if (convexp0int[convexcount].x == reg[points_in_region].X) &&
    (convexp0int[convexcount].Y == reg[points_in_region].Y)
{
    convexcount--;
}

switch (points_in_region)
{
    case 1:
        convexcount++;
        convexpoint[convexcount] = reg[points_in_region];
        l_convexhull[reg[points_in_region].gennumb] = ON_CONVEXHULL;
        break;
    case 2:
        convexcount++;
        convexpoint[convexcount] = reg[2];
        l_convexhull[reg[2].gennumb] = ON_CONVEXHULL;
        convexcount++;
        convexpoint[convexcount] = reg[1];
        l_convexhull[reg[1].gennumb] = ON_CONVEXHULL;
        break;
    case 3:
        {
            for (i = points_in_region; i >= 1; i--)
            {
                convexcount++;
                convexpoint[convexcount] = reg[i];
            }
        }
}
else
{
    convexcount++;
    convexpoint[convexcount] = reg[3];
    l_convexhull[reg[3].gennumb] = ON_CONVEXHULL;
    convexcount++;
    convexpoint[convexcount] = reg[1];
    l_convexhull[reg[1].gennumb] = ON_CONVEXHULL;
}
break;

default:
    convexcount++;
    convexpoint[convexcount] = reg[points_in_region];
    l_convexhull[reg[points_in_region].gennumb] = ON_CONVEXHULL;

    first = (reg+points_in_region);
    middle = &reg[points_in_region-1];
    last = &reg[points_in_region-2];

while(last >= &reg[1])
{
    if{{(middle->Y - first->Y)*(last->X - middle->X) + (first->X - middle->X) * (last->Y - middle->Y)} <= 0}
    {
        cconvexcount++;
        l_convexhull[middle->gennumb] = ON_CONVEXHULL;
        convexpoint[convexcount] = * middle;
        first = middle;
        middle--;
        last--;
    }
    else
    {
        middle--;
        last--;
    }
}
last++;  
convexcount++;  
1_convexhull[last->gennumb] = ON_CONVEXHULL;  
convexpoint[convexcount] = *last;  
break;  
}/* end switch */

} /* end */
Function "optinsrt" description

Function "optinsrt" is an implementation of the Optimal insertion algorithm proposed by Goetschalckx [1985] as an intermediate stage of the Convex hull algorithm (see Golden et al [1980] and Allison and Noga [1984]) when Tchebyshev norm is applied. The purpose is to insert as many addresses as possible between two consecutive points (addresses) on the convex hull without increasing the travel time between the two convex hull points.

```plaintext
optinsrt()
begin
    for any two consecutive points on the convex hull begin
        find number of addresses in the region (neighbourhood) defined by the two convex hull points. Store this number as well as the addresses; /* function "neibhood" */

        find the longest path between the two consecutive convex hull points without increasing travel time between the two points; /* function "long_path" */
    end
end
```
Function "optinsrt" inserts as many addresses as possible between two consecutive points on the convex hull without increasing the travel time between the two convex hull points.

```c
optinsrt()
{
  int i;
  chebcount = 1; /* counts all points on the tour during opt insertion phase */
  cheb[chebcount] = convexpoint[1]; /* I/O point */
  l_convexhull[chebcount] = ON_CONVEXHULL;

  /* for any two consecutive points from the convex hull determine the points laying in the region confined by the lines through the two convex hull points with gradient +e and -e */
  for(i = 2; i <= convexcount; i++)
  {
    neibhood(convexpoint[i-1], convexpoint[i]);
  }
```
chebcount++;
cheb[chebcount] = convexpoint[1];

}  
neibhood(convexpoint[convexcount],convexpoint[1]);
} /* end */
Function "neibhood"- description

Function "neibhood" finds addresses in a region (neighbourhood) defined by two consecutive convex hull points. The first of these points, assuming a certain direction of traversing the convex hull, is called "local" and the second one "settler".

"Settler" as shown in fig. A3 can be EAST, WEST, NORTH or SOUTH of "local".

After addresses in the neighborhood have been found, they are sorted according to their x or y coordinate depending on the relative position of "settler" to "local".

Finally the longest path between "local" and "settler" is found.

![Diagram](image)

Fig. A3. Relative position between two consecutive points on the convex hull.

```
neibhood(local, settler)
begin
  int asm;   /* indicates relative position between local and settler */
  find relative position of settler to local
  asm = {EAST, WEST, NORTH, SOUTH};
```
for( i = 2 to picks) 
begin
  find all addresses in the neighbourhood defined by local 
  and settler;
end

according to value of asm:
begin
  case EAST: sort addresses in the neighbourhood in 
  ascending order of their x coordinate;

  case WEST: sort addresses in the neighbourhood in 
  descending order of their x coordinate;

  case NORTH: sort addresses in the neighbourhood in 
  ascending order of their y coordinate;

  case SOUTH: sort addresses in the neighbourhood in 
  descending order of their y coordinate;
end

find longest path between local and settler /* function 
"long_path" */
end

Note: Functions for sorting addresses in ascending (descending) order of their x 
or y coordinate are regqsrt, descend_x, ascend_y and descend_y. These 
are presented in the source file REGQSORT.C.
Function "neibhood" finds addresses in a region (neighbourhood) defined by two consecutive points. Then these addresses are sorted. Finally the longest path between the two convex hull points is found.

neibhood(local, settler)
struct coordinates local, settler:
{
    struct coordinates neib[PICKSIZE-SUBTRACT];
    /* keeps addresses (points) in the region defined by the two convex hull points */
    int asm, j, ncount;
    float e;
    e = vy/vx; /* gradient */

    /*** for any two consecutive points from the convex hull determine the position of the second point (settler) relatively to the first point (local) ***/
        asm = EAST;
        asm = WEST;
        asm = NORTH;
    else
        asm = SOUTH;
}
/*** if a point(address) is not already on the convex hull it is checked if it is in the region (neighborhood) defined by the two convexhull points ***/

ncount = 0;
for(j = 2; j <= picks; j++)
{
    if(l_convexhull[j] == NOT_ON_CONVEXHULL)
    {
        switch(asm)
        {
            case EAST:
            {
                if(((float)(-address[j].Y + e*address[j].X) >=
                (float)(-local.Y + e*local.X)) &&
                ((float)(address[j].Y + e*address[j].X) >=
                (float)(local.Y + e*local.X)) &&
                ((float)(-address[j].Y + e*address[j].X) <=
                (float)(-settler.Y + e*settler.X)) &&
                ((float)(address[j].Y + e*address[j].X) <=
                (float)(settler.Y + e*settler.X))))
                {
                    ncount++;
                    neib[ncount] = address[j];
                }
                break;
            }
            case WEST:
            {
                if(((float)(-address[j].Y + e*address[j].X) <=
                (float)(-local.Y + e*local.X)) &&
                ((float)(address[j].Y + e*address[j].X) <=
                (float)(local.Y + e*local.X)) &&
                ((float)(-address[j].Y + e*address[j].X) >=
                (float)(-settler.Y + e*settler.X)) &&
                ((float)(address[j].Y + e*address[j].X) >=
                (float)(settler.Y + e*settler.X))))
                {
                    ncount++;
                    neib[ncount] = address[j];
                }
                break;
            }
case NORTH:
{
    if(((float)(-address[j].Y + e*address[j].X) <=
        (float)(-local.Y + e*local.X)) &&
    ((float)(address[j].Y + e*address[j].X) >=
        (float)(local.Y + e*local.X))) &&
    (((float)(-address[j].Y + e*address[j].X) >=
        (float)(-settler.Y + e*settler.X)) &&
    ((float)(address[j].Y + e*address[j].X) <=
        (float)(settler.Y + e*settler.X))))
    {
        ncount++;  
        neib[ncount] = address[j];  
    }
    break;
}

default: /* SOUTH */
{
    if(((float)(-address[j1].Y + e*address[j1].X) >=
        (float)(-local.Y + e*local.X)) &&
    ((float)(address[j1].Y + e*address[j1].X) <=
        (float)(local.Y + e*local.X))) &&
    (((float)(-address[j1].Y + e*address[j1].X) <=
        (float)(-settler.Y + e*settler.X)) &&
    ((float)(address[j1].Y + e*address[j1].X) >=
        (float)(settler.Y + e*settler.X))))
    {
        ncount++;  
        neib[ncount] = address[j];  
    }
    break;
}
} /* end switch */

} /*end if*/

} /*end for*/
switch(asm)
{
    case EAST:
        regqsort(&neib[1], &neib[ncount]);
    break;
    case WEST:
        descend_x(&neib[1], &neib[ncount]);
    break;
    case NORTH:
        ascend_y(&neib[1], &neib[ncount]);
    break;
    default:  /** SOUTH **/
        descend_y(&neib[1], &neib[ncount]);
    break;
}
if(ncount == 0)
;
else if(ncount == 1)
{
    chebcount++;
    cheb[chebcount] = neib[ncount];
    l_convexhull[neib[ncount].gennmb] = ON_CONVEXHULL;
}
else if(ncount > 1)
    long_path(neib, settler, asm, ncount);
} /* end */
**Function "long_path" - description**

Function "long_path" finds the longest path amongst addresses in a region (neighbourhood) defined by two consecutive points on the convex hull without increasing the travel time between the two convex hull points (fig. A4). Function long path accepts four arguments from function "neibhood". These are: an array of the addresses found in the neighbourhood; the second of the two convex hull points defining the neighbourhood; relative position of the second convex hull point to the first one; number addresses in the neighbourhood.

![Diagram](image)

**Fig. A4. Recursive procedure for finding the longest path.**

```plaintext
long_path()
begin
for (each address in the neighbourhood)
begin
    form a new neighbourhood defined by the address itself and the second point (settler) of the neighbourhood;
    find number addresses in this new neighbourhood;
end
```
find the address whose new neighbourhood contains maximum addresses;

apply recursively long_path to the neighbourhood containing maximum addresses;

end
/*** Function "long_path" finds the longest path amongst addresses in a region (neighborhood) defined by two consecutive points on the convex hull without increasing the travel time between the points

*/

long_path(neibmemb, second, azimuth, pcount)
struct coordinates neibmemb[];
struct coordinates second;
int azimuth, pcount;
{
    struct coordinates new_memb;
    struct coordinates * indicator;
    struct coordinates candidate[PICKSIZE-SUBTRACT];
    int j,locmax=0,count;
    float e;
    e = vy/vx;

    indicator = neibmemb+1;
    while(indicator <= (neibmemb+pcount))
    {
        count = 0;

        for(j = 1; j <= pcount; j++)
        {

            switch (azimuth)
            {
            case EAST:
                {
                    if(((float) (-neibmemb[j].Y + e*neibmemb[j].X + 0.0001) >= (float) (-indicator->Y + e*indicator->X))
                        && ((float) (neibmemb[j].Y + e*neibmemb[j].X+0.0001) >=
                        (float) (indicator->Y + e*indicator->X))
                        && ((float) (-neibmemb[j].Y + e*neibmemb[j].X)
                        <= (float) (-second.Y + e*second.X + 0.0001))
                        && ((float) (neibmemb[j].Y + e*neibmemb[j].X)
                        <= (float) (second.Y + e*second.X + 0.0001)))
                        
                        count++;
cand[count] = neibmemb[j];
}
break;
}
case WEST:
{
if(((float)(-neibmemb[j].Y + e*neibmemb[j].X) <=
(float)(-indicator->Y + e*indicator->X +
0.0001)) && ((float)(neibmemb[j].Y +
e*neibmemb[j].X) <= (float)(indicator->Y +
e*indicator->X + 0.0001))) &&
(((float)(-neibmemb[j].Y + e*neibmemb[j].X +
0.0001) >= (float)(-second.Y + e*second.X))
&& ((float)(neibmemb[j].Y + e*neibmemb[j].X +
0.0001) >= (float)(second.Y + e*second.X)))
{
  count++;  
cand[count] = neibmemb[j];
}
break;
}
case NORTH:
{
if(((float)(-neibmemb[j].Y + e*neibmemb[j].X) <=
(float)(-indicator->Y + e*indicator->X +
0.0001)) && ((float)(neibmemb[j].Y +
e*neibmemb[j].X) <= (float)(indicator->Y +
e*indicator->X + 0.0001))) &&
(((float)(-neibmemb[j].Y + e*neibmemb[j].X +
0.0001) >= (float)(-second.Y + e*second.X))
&& ((float)(neibmemb[j].Y + e*neibmemb[j].X +
0.0001) >= (float)(second.Y + e*second.X)))
{
  count++;  
cand[count] = neibmemb[j];
}
break;
}
default: /* SOUTH */
{
    if(((float)(-neibmemb[j].Y + e*neibmemb[j].X) +
        0.0001) >= (float)(-indicator->Y +
        e*indicator->X)) && ((float)(neibmemb[j].Y +
        e*neibmemb[j].X) <= (float)(indicator->Y +
        e*indicator->X + 0.0001)) &&
    (((float)(-neibmemb[j].Y + e*neibmemb[j].X)
        <= (float)(-second.Y +
        e*second.X + 0.0001))
    && ((float)(neibmemb[j].Y +
        e*neibmemb[j].X) +
        0.0001) >= (float)(second.Y +
        e*second.X)))
    {
        count++;
        cand[count] = neibmemb[j];
    }
    break;
}
} /* end switch */
} /*end for*/

if(locmax <= count)
{
    locmax = count;
    new_memb = *indicator;
    for(j=2;j <= locmax;j++)
    {
        candidate[j-1] = cand[j];
    }
    indicator++;
}
} /* end while */

if(locmax>0)
{
    chebcount++;
    cheb[chebcount] = new_memb;
    l_convexhull[new_memb.gennumb] = ON_CONVEXHULL;
    long_path(candidate, second, azimuth, locmax-1);
}
} /* end */
source file < MININSRT.C >

Function "mininsrt" is an implementation of the insertion phase (step 2 to 5) of the Convex hull algorithm proposed by W.R. Stewart (see Golden et al [1980] and Allison and Noga [1984]). The insertion procedure is applied twice in the present function, first for insertions when average velocities are used and second for insertions when real times are used, real times being read from files "hor" and "vert".

#include "externs.h"

mininsrt()
{
    struct insert
    {
        struct coordinates point_i;
        struct coordinates point_j;
        struct coordinates point_k;
        float mindrt;
        float mindst;
        int place_for_insert;
    }

    struct insert ins[PICKSIZE];
    struct insert insert;
    struct coordinates new;
    float mindrt, mindst, findrt;
    float dik, dkj, dij;
    float dikl, dkjl, dijl;
    float dik2, dkj2, dij2;
    float md, mr;
    int k, i, in_k;
    int virtual_end, new_insert;
}
int d1, d2;
int horread, vertread;
FILE *streamhor;
FILE *streamvert;
int hori[101], verti[21];

if(cheb[chebcount].gennumb == cheb[1].gennumb) /* could happen */
    chebcount--;
cheb[chebcount+1] = cheb[1]; /* to close the route */

if (round == AVERAGE)
{
    while(chebcount < picks)
    {
        in_k = 0;
        for(k = 1; k <= picks; k++)
        {
            mindrt = mindst = BIGNUMB;
            if(l_convexhull[address[k].gennumb] == NOT_ON_CONVEXHULL)
            {
                for(i = 1; i <= chebcount; i++)
                {
                    dikl = (cheb[i].X - address[k].X)/vx + 0.0001;
                    if(dikl < 0) dikl = -dikl;

                    dik2 = (cheb[i].Y - address[k].Y)/vy;
                    if(dik2 < 0) dik2 = -dik2;

                    dik = (float) max(dikl, dik2);

                    dkj1 = (cheb[i+1].X - address[k].X)/vx + 0.0001;
                    if(dkj1 < 0) dkj1 = -dkj1;

                    dkj2 = (cheb[i+1].Y - address[k].Y)/vy;
                    if(dkj2 < 0) dkj2 = -dkj2;

                    dkj = (float) max(dkj1, dkj2);

                    dijl = (cheb[i].X - cheb[i+1].X)/vx + 0.0001;
                    if(dijl < 0) dijl = -dijl;

                    dij2 = (cheb[i].Y - cheb[i+1].Y)/vy;
                    if(dij2 < 0) dij2 = -dij2;

                    dij = (float) max(dij1, dij2);

                    }
dij2 = (cheb[i].Y - cheb[i+1].Y)/vy;
if(dij2 < 0) dij2 = -dij2;

dij = (float)max(dij1, dij2);

md = dik + dkj - dij;
mr = (dik + dkj)/dij;

if(mindst > md)
{
if(cheb[i+1].gennumb != 1)
{
    mindst = md;
mindrt = mr;
insert.point_i = cheb[i];
insert.point_j = cheb[i+1];
insert.point_k = address[k];
insert.mindst = mindst;
insert.mindrt = mindrt;
insert.place_for_insert = i+1;
}
else /* insertion before I/O point */
{
    mindst = md;
mindrt = mr;
insert.point_i = cheb[i];
insert.point_j = cheb[i];
insert.point_k = address[k];
insert.mindst = mindst;
insert.mindrt = mindrt;
insert.place_for_insert = i;
}
}

} /* end for i */

in_k++;
ins[in_k]= insert;

} /* end if */

} /*end for k */
findrt = BIGNUMB;
for(k = 1; k <= in_k; k++)
{
    if(findrt > ins[k].mindrt)
    {
        findrt = ins[k].mindrt;
        new = ins[k].point_k;
        new_insert = ins[k].place_for_insert;
    }
}
l_convexhull[new.entrada] = ON_CONVEXHULL;
virtual_end = chebcount+1; /* because cheb[chebcount+1] = cheb[1] */
for(i = virtual_end; i > new_insert; i--)
{
    cheb[i] = cheb[i-1];
}

cheb[new_insert] = new;
chebcount++;
cheb[chebcount+1]=cheb[1];

} /* end while */

} /* end if round = average */

else /* round = real */
{
    streamhor = fopen("hor", "r++");
    if(streamhor == (FILE *)NULL)
    {
        printf("Cannot open hor file for reading
");
        exit(1);
    }
    horread = fread((char *)hori, sizeof(int), 102, streamhor);
    printf(" IN MININSRT horread = %d\n", horread);
    fclose(streamhor);

}
streamvert = fopen("vert","r+");
if(streamvert == (FILE *)NULL)
{
    printf("Cannot open vert file for reading\n");
    exit(1);
}

vertread = fread((char *)verti, sizeof(int), 22, streamvert);
printf(" IN MININSRT vertread = %d\n", vertread);
fclose(streamvert);

while(chebcount < picks)
{
    in_k = 0;
    for(k = 1; k <= picks; k++)
    {
        mindrt = mindst = BIGNUMB;
        if(l_convexhull[address[k].gennumb] == NOT_ON_CONVEXHULL)
        {
            for(i = 1; i <= chebcount; i++)
            {
                if(cheb[i].gennumb != cheb[i+1].gennumb)
                {
                    d1 = (cheb[i].X - address[k].X);
                    if(d1 < 0)  d1 = -d1;
                    dik1 = hori[d1];

                    d2 = (cheb[i].Y - address[k].Y);
                    if(d2 < 0)  d2 = -d2;
                    dik2 = verti[d2];

                    dik = (float)max(dik1, dik2);

                    d1 = (cheb[i+1].X - address[k].X);
                    if(d1 < 0)  d1 = -d1;
                    dkj1 = hori[d1];

                    d2 = (cheb[i+1].Y - address[k].Y);
                    if(d2 < 0)  d2 = -d2;
                    dkj2 = verti[d2];
    }
dkj = (float)max(dkj1, dkj2);

dl = (cheb[i].X - cheb[i+1].X);
if (dl < 0) dl = -dl;
dij1 = hori[dl];

d2 = (cheb[i].Y - cheb[i+1].Y);
if (d2 < 0) d2 = -d2;
dij2 = vert1[d2];

dij = (float)max(dij1, dij2);

md = dik + dkj - dij;
mr = (dik + dkj)/dij;

if (mindst > md)
{
    if (cheb[i+1].gennumb != 1)
    {
        mindst = md;
mindrt = mr;
insert.point_i = cheb[i];
insert.point_j = cheb[i+1];
insert.point_k = address[k];
insert.mindst = mindst;
insert.mindrt = mindrt;
insert.place_for_insert = i+1;
    }
else /* insertion before I/O point */
    {
    mindst = md;
mindrt = mr;
insert.point_i = cheb[i];
insert.point_j = cheb[i];
insert.point_k = address[k];
insert.mindst = mindst;
insert.mindrt = mindrt;
insert.place_for_insert = i;
    }
}
} /* end if cheb[i] != cheb[i+1] */
} /* end for i */

in_k++;
ins[in_k] = insert;

} /* end if */
}

/* end for k */

findrt = BIGNUMB;
for(k = l;k <= in_k;k++)
{
    if(findrt > ins[k].mindrt)
    {
        findrt = ins[k].mindrt;
        new = ins[k].point_k;
        new_insert = ins[k].place_for_insert;
    }
}

l_convexhull[new.gennumb] = ON_CONVEXHULL;

virtual_end = chebcount + l; /* because cheb[chebcount+1] = cheb[1] */

for(i = virtual_end;i > new_insert;i--)
{
    cheb[i] = cheb[i-1];
}
cheb[new_insert] = new;

chebcount++;
cheb[chebcount+1] = cheb[1];
} /* end while */

} /* end else ( round = real) */

} /* end */
/***************************************************************
* 
* source file <CHEBDIST.C> 
* Function "chebdist" calculates distance matrix for an already 
* constructed CHEBHULL tour. It also calculates the cost (time) 
* of that tour. Calculations are made for both average velocities 
* and real times, real times being read from files "hor" and "vert".
* 
***************************************************************

#include "externs.h"

chebdist()
{
    float hor, vert;
    float dX, dY;
    int dx, dy;
    int horread, vertread;

    FILE *streamhor;
    FILE *streamvert;

    int hori[101], verti[21];

    if(round == AVERAGE)
    {
        chebcost = 0.0;
        for(I = 1; I <= picks - 1; I++)
        {
            for(J = I+1; J <= picks; J++)
            {
                dX = (float)(cheb[I].X - cheb[J].X);
                dY = (float)(cheb[I].Y - cheb[J].Y);
                hor = (dX/vx)*100;
                vert = (dY/vy)*100;

                if(hor < 0 )
                {
                    hor = -hor;
                }
                if(vert < 0)
vert = -vert;

if(hor >= vert)
{
    BW[I][J] = hor;
}
else
{
    BW[I][J] = vert;
}

BW[J][I] = BW[I][J];

for(I = 1; I <= picks-1; I++)
{
    chebcost = chebcost + BW[I][I+1];
}

chebcost = chebcost + BW[picks][1];
che = chebcost; /* for graph display */

for(I=1; I<=picks; I++) /* the sequence here is normal because the coordinates have already been sorted for chebhull */
{
    BROUTE[I] = I;
}

} /* end if round = average */

else
{
    streamhor = fopen("hor","r+"晩);    
    if(streamhor == (FILE *)NULL)
    {
        printf("Cannot open hor file for reading\n");
        exit(1);
    }
}
horread = fread((char *)hori, sizeof(int), 102, streamhor);
printf("horread = %d\n", horread);
fclose(streamhor);

streamvert = fopen("vert", "r+");
if(streamvert == (FILE *)NULL)
{
    printf("Cannot open vert file for reading\n");
    exit(1);
}

vertread = fread((char *)verti, sizeof(int), 22, streamvert);
printf("vertread = %d\n", vertread);
fclose(streamvert);

for(I = 1; I <= picks-1; I++)
{
    for(J = I+1; J <= picks; J++)
    {
        dx = (cheb[I].X - cheb[J].X);
        dy = (cheb[I].Y - cheb[J].Y);
        if(dx < 0)
        {
            dx = -dx;
        }
        if(dy<0)
        {
            dy = -dy;
        }
        if(hori[dx] >= verti[dy])
        {
            BW[I][J] = hori[dx];
        }
        else
        {
            BW[I][J] = verti[dy];
        }
        BW[J][I] = BW[I][J];
    }
}
chebcost = 0;
for(I = 1; I <= picks - 1; I++)
{
    chebcost = chebcost + BW[I][I+1];
}

chebcost = chebcost + BW[picks][1];

for(I = 1; I <= picks; I++)
{
    BROUTE[I] = I;
} /* the sequence for BROUTE here is normal because the coordinates have already been sorted for chebhull*/

} /* end else */

} /* end */
**source file <REALMATR.C>**

Function "realmatr" creates a distance matrix whose entries are the real travel times between any two locations of the rack. Real times are read from the files "hor" and "vert".

---

```c
#include "externs.h"

realmatr()
{
    int i, j;
    int hor, vert;
    int dX, dY;
    int horread, vertread;

    FILE *streamhor;
    FILE *streamvert;
    int hori[101], verti[21];

    streamhor = fopen("hor", "r+");  
    if(streamhor == (FILE *)NULL)   
    {   
        printf("Cannot open hor file for reading\n");
        exit(1);
    }

    horread = fread((char *)hori, sizeof(int), 102, streamhor);
    printf(" IN MATR horread = %d\n", horread);
    fclose(streamhor);

    streamvert = fopen("vert", "r+");
    if(streamvert == (FILE *)NULL)   
    {   
        printf("Cannot open vert file for reading\n");
        exit(1);
    }

    vertread = fread((char *)verti, sizeof(int), 22, streamvert);

    // Further processing...
}
```

---

89
printf(" IN MATR vertread = %d\n", vertread);
fclose(streamvert);

for(i = 1; i <= picks; i++)
{
    for(j = i; j <= picks; j++)
    {
        if(i == j)
        {
            BW[i][j] = BIGNUMB;
        }
        else
        {
            dX = (address[i].X - address[j].X);
            dY = (address[i].Y - address[j].Y);
            if(dX < 0)
            {
                dX = -dX;
            }
            if(dY < 0)
            {
                dY = -dY;
            }
            if(hori[dX] >= verti[dY])
            {
                BW[i][j] = hori[dX];
            }
            else
            {
                BW[i][j] = verti[dY];
            }

            BW[j][i] = BW[i][j];
        }
    }
}
source file <RECALC.C>

Function "recalc" recalculates cost (time) of the CHEBHULL or CHEBHULL plus TWO_OPT tour obtained previously with average velocities. The function accepts as a first argument the tour sequence and an array of the address coordinates as a second argument. Then the real travel time times between any two consecutive addresses on the tour are obtained using the cost (real travel time) matrix.

#include "externs.h"

recalc(route, ch)
int route[];
struct coordinates ch[];
{
    int i;
    int cost = 0;

    for(i=1;i<=picks-1;i++)
    {
        cost = cost + BW[ch[route[i]].gennumb][ch[route[i+1]].gennumb];
    }

    cost = cost + BW[ch[route[picks]].gennumb][ch[route[1]].gennumb];
    return(cost);
}
Function "babtsp" - description

Function "babtsp" is an implementation of Little's Branch and Bound algorithm (see Little, J. et al [1963]) for solving TSP. The actual algorithm used here was taken from Syslo et al [1983], where a detailed description is given.

Function "babtsp" finds an exact solution of TSP in a network of $N$ addresses given as $N \times N$ weight (cost or distance) matrix. If the calculations are performed with average velocities the distance matrix is constructed by calling function "wmatrix" first. If the calculations are carried out with real times the distance matrix has already been created by function "realmatr" during the execution of function "chebhull".

Next, the function "babtsp" calls the recursive function "explore" which considers a given partial solution and searches for a better solution through the entire solution space (the decision tree) in a depth-first fashion. Function "explore", in turn, calls function "reduce" which reduces the associated matrix and computes the amounts to be subtracted from the corresponding row and column. "Explore" also calls the function "bestedge" which finds the best edge on which to branch next and the value by which the lower bounds of the branches differ.
/**  
* source file <BABTSP.C>  
*  
* Function "babtsp" is an implementation of Little's Branch and Bound algorithm (see Little, J. et al [1963]) for solving the TSP.  
* The actual algorithm used here was taken from Syslo, M. et al [1983] and translated into C language.  
*/

#include "externs.h"

babtsp()
{
    void explore();
    void wmatrx();

    int edges;       /* number of edges in the partial tour */
    int cost;

    int row[PICKSIZE]; /* keeps record which rows from orig. dist. matrix are in the current (reduced) matrix */

    int col[PICKSIZE]; /* keeps record which columns from orig. dist. matrix are in the current (reduced) matrix */

    int i, index;

    if(round == AVERAGE)
    {
        wmatrx();   /* cost matrix */
    }
    else
    {
        RTSPCOST = RTSPCOST + tsprcalc(TSPROUTE); /* recalculation of the const solution */
    }
for(i = 1; i <= picks; i++)
{
    row[i] = i;
    col[i] = i;
    fwdptr[i] = 0;
    backptr[i] = 0;
}
tweight = BIGNUMB;
edges = 0;
cost = 0.0;
__clearscreen(_GCLEARSCREEN);
printf("\t\t _________________\n");
printf("\t\t PLEASE, WAIT A MINUTE ! \n");
printf("\t\t ___________________
");
time(&tsp_start);
explore(edges, cost, row, col);
__clearscreen(_GCLEARSCREEN);
index = 1;
for(i = 1; i <= picks; i++)
{
    BROUTE[i] = index;
    index = best[index];
}
time(&tsp_end);
tsp_time = difftime(tsp_end, tsp_start);

if(round == AVERAGE)
{
    for(i = 1; i <= picks; i++)
    {
        TSPROUTE[i] = BROUTE[i]; /* save TSP-route for recalculation */
    }
tspcost = tweight; /* for display */
    runtime = tsp_time;
}
printf("\n\n\n\n\n			 PRESS ENTER ");
getchar();
tspdisp();
getchar();
} /* end */
/***************************************************/
* *
* source file <WMATRX.C>
* Function "wmatrx" creates cost (distance) matrix whose entries
* are travel times, between addresses calculated with average
* velocities.
* *
***************************************************/

#include "externs.h"

wmatrx()
{
    int i, j;
    float hor, vert;

    for(i = 1; i <= picks; i++)
    {
        for(j = i; j <= picks; j++)
        {
            if(i == j)
            {
                BW[i][j] = BIGNUMB;
            }
            else
            {
                hor = (((float) (address[i].X - address[j].X)) / vx)*100;
                vert = (((float) (address[i].Y - address[j].Y)) / vy)*100;

                if(hor < 0) hor = -hor;
                if(vert < 0) vert = -vert;

                if(hor >= vert) BW[i][j] = hor;
                else BW[i][j] = vert;

                BW[j][i] = BW[i][j];
            }
        }
    }
}

95
Function "explore" considers a given partial solution and searches for a better solution. It maintains a global copy of the best solution obtained so far, together with its weight (cost). For more details see Syslo, M. et al [1983].

#include "externs.h"

void explore(e_edges, e_cost, e_row, e_col)
int e_edges;
int e_cost;
int e_row[], e_col[];
{
    void reduce();
    void bestedge();

    int rowred[ PICKSIZE ];
    int colred[ PICKSIZE ];
    int newcol[ PICKSIZE ];
    int newrow[ PICKSIZE ];

    int size; /* current size of the reduced weight matrix */
    int r, c; /* row r and column c indicating the best edge for inclusion (exclusion) */
    int most;
    int avoid;
    int i, j;
    int first, last;
    int val, incl;
    int colrowval, lowerbound;

    int index;
size = picks - e_edges;
reduce(e_row, e_col, rowred, colred, size, &val);
e_cost = e_cost + val;

if(e_cost < tweight)
{
    if(e_edges == (picks-2))
    {
        for(i = 1; i <= picks; i++)
        {
            best[i] = fwdptr[i];
        }
        if(BW[e_row[1]][e_col[1]] == BIGNUMB)
            avoid = 1;
        else
            avoid = 2;
        best[e_row[1]] = e_col[3 - avoid];
        best[e_row[2]] = e_col[avoid];
        tweight = e_cost;
    }
    else
    {
        bestedge(e_row, e_col, &r, &c, &most, size);
        lowerbound = e_cost + most;

        incl = BW[e_col[c]][e_row[r]]; /* prevent reverse link */
        BW[e_col[c]][e_row[r]] = BIGNUMB; /* prevent reverse link */

        fwdptr[e_row[r]] = e_col[c]; /* record chosen edge */
        backptr[e_col[c]] = e_row[r]; /* prevent cycles */

        while(fwdptr[last] != 0)
        {
            last = fwdptr[last];
        }
        first = e_row[r];

        while(backptr[first] != 0)
        {
            first = backptr[first];
        }
    }
}
colrowval = BW[last][first];
BW[last][first] = BIGNUMB;

for(i = 1; i <= r - 1; i++)
{
    newrow[i] = e_row[i];
}
for(i = r; i <= size - 1; i++)
{
    newrow[i] = e_row[i+1];
}
for(i = 1; i <= c - 1; i++)
{
    newcol[i] = e_col[i];
}
for(i = c; i <= size - 1; i++)
{
    newcol[i] = e_col[i+1];
}
explore(e_edges+1, e_cost, newrow, newcol);

BW[last][first] = colrowval; /* restore previous values */
BW[e_col[c]][e_row[r]] = incl; /* restore reverse link */
backptr[e_col[c]] = 0;
fwdptr[e_row[r]] = 0;
if(lowerbound < tweight)
{
    BW[e_row[r]][e_col[c]] = BIGNUMB; /* exclude edge already chosen */
    explore(e_edges, e_cost, e_row, e_col);
    BW[e_row[r]][e_col[c]] = 0; /* restore excluded edge */
}
} /* end else */

} /* end if  cost < tweight */
for(i = 1; i <= size; i++) /* unreduce matrix */
{
    for(j = 1; j <= size; j++)
    {
        if(BW[e_row[i]][e_col[j]] < BIGNUMB)
            BW[e_row[i]][e_col[j]] = BW[e_row[i]][e_col[j]] + rowred[i] + colred[j];
    }
}

/* run time constraints */

time(&tsp_limit);
timelimit = difftime(tsp_limit, tsp_start);
if(timelimit > 36000) /* twelve hours */
    {
        printf("TIME - LIMIT EXCEEDED");
        exit(1);
    }
} /* end */
/*********************************************************************/
/*
  *  source file <REDUCMAT.C>
  */

Function "reduce" reduces the distance matrix, thus obtaining a new lower bound for the TSP. For more details see Syslo, M. et al [1983].

*********************************************************************/

#include "externs.h"

reduce(r_row, r_col, r_rowred, r_colred, size, rvalue)
int size;
int r_row[], r_col[];
int r_rowred[], r_colred[];
int *rvalue;
{
    int temp;
    *rvalue = 0;

    for(I = 1; I <= size; I++)
    {
        temp = BIGNUMB;
        for(J = 1; J <= size; J++)
        {
            temp = (int)min(temp, BW[r_row[I]][r_col[J]]);
        }
        if(temp > 0 )
        {
            for(J=1;J<=size;++J)
            {
                if(BW[r_row[I]][r_col[J]] < BIGNUMB)
                {
                    BW[r_row[I]][r_col[J]] = BW[r_row[I]][r_col[J]] - temp;
                }
                *rvalue = (*rvalue + temp);
            }
            r_rowred[I]=temp;
        }
    }
for(J = 1; J <= size; J++)
{
    temp = BIGNUMB;
    for(I = 1; I <= size; I++)
    {
        temp = (int)min(temp, BW[r_row[I]][r_col[J]]);
    }
    if(temp > 0 )
    {
        for(I = 1; I <= size; I++)
        {
            if(BW[r_row[I]][r_col[J]] < BIGNUMB)
            {
                BW[r_row[I]][r_col[J]] = BW[r_row[I]][r_col[J]] - temp;
            }
            *rvalue=( *rvalue+temp);
        }
        r_colred[J]=temp;
    }
} /* end */
source file <BESTEDGE.C>

Function "bestedge" determines the best edge to be included (excluded) next in (from) the tour. For more details see Syslo, M. et al [1983].

#include "externs.h"

void bestedge(b_row, b_col, rr, cc, mmost, size)
{
    int size;
    int b_row[], b_col[];
    int *rr;
    int *cc;
    int *mmost;

    *mmost = -BIGNUM;
    for(i = 1; i <= size ;i++)
    {
        for(j = 1; j <= size; j++)
        {
            if(BW[b_row[i]][b_col[j]] == 0)
            {
                minrowelt = BIGNUM;
                zeroes = 0;
                for( k = 1; k <= size; k++)
                {
                    if(BW[b_row[i]][b_col[k]] == 0)
                        zeroes++;
                    else
                        minrowelt = (int)min(minrowelt, BW[b_row[i]][b_col[k]]);
                }
            }
        }
    }
}
if (zeroes > 1) minrowelt = 0;

mincolelt = BIGNUMB;
zeroes = 0;

for (k = 1; k <= size; k++)
{
    if (BW[b_row[k]][b_col[j]] == 0)
        zeroes++;
    else
        mincolelt = (int)min(mincolelt, BW[b_row[k]][b_col[j]]);
}

if (zeroes > 1) mincolelt = 0;

if ((minrowelt + mincolelt) > *mmost) /* a better edge has been found */
{
    *mmost = minrowelt + mincolelt;
    *rr = i;
    *cc = j;
}

} /* end if */

} /* end for i */
} /* end for j */

} /* end */
/* source file <TSPRCALC.C> */

Function "tsprcalc" recalculates the cost (travel time) of the TSP tour with real times which has been previously obtained using average velocities. The function "babtsp" accepts as an argument the tour sequence obtained with average velocities and using the corresponding entries of the cost matrix, recalculates the tour total travel time.

********************************************************************

#include "externs.h"

tsprcalc(route)
int route[];
{
int i;
int cost;
cost = 0;

for(i = 1; i <= picks-1; i++)
{  
cost = cost + BW[route[i]][route[i + 1]];  
}

cost = cost + BW[route[picks]][route[1]];
return(cost);
}
source file <output.c>

Function "output" displays on the screen input parameters and 
results of simulation.

#include "externs.h"

output()
{
    printf("********************************************************\n");
    printf("PICKS=%d \n",picks);
    printf("H=%d \n",maxrows,avel);
    printf("L=%d \n",maxcols,vx,vy);
    printf("SEED=%d \n", SEED);
    printf("VX=%4.1f \n",pickS):
    printf("VY=%4.1f \n", a=%4.1

    printf("BAND(const)=%4.2f \n",RBDCOST/10,RBANDCOST/10);
    printf("BAND(real)= %4.2f \n",bcost/10,TBANDCOST/10);
    printf("BANDruntime= %4.1f \n",band_time,band_2opt_time);
    printf("CHEB(const)=%4.2f \n",RCHBCOST/10,RCHEBCOST/10);
    printf("CHEB(real)= %4.2f \n",chebcost/10,two_b_cost/10);
    printf("CHEBruntime=%4.1f \n",cheb_time,cheb_2opt_time);
}
printf("* -----------------------------------------------------\n");
printf("* TSP(const)=%4.2f TSP(real) = %4.2f
\n",RTSPCOST/10,DTSPCOST/10);
printf("* TSP(const) runtime=%5.1f TSP(real) runtime=%5.1f
\n",runtime,tsp_time);
printf("********************************************************\n");
APPENDIX B

ADDITIONAL SOFTWARE DEVELOPED FOR THE SIMULATION EXPERIMENT IN CHAPTER IV

Appendix B contains all functions (except those producing a graphic display) that were needed, in addition to those ones presented in Appendix A for carrying out the simulation experiment in chapter IV. Hierarchy of the functions, included in the simulation software for each of the modelled zone configurations is presented in fig. B1, fig. B3 and fig. B5 respectively.
Fig. B1. Hierarchy of the functions involved in the simulation of the BAND1 zone configurations.
#include <stdio.h>
#include <graph.h>
#include <time.h>

#define PICKSIZE 36
#define BIGNUMB 9999
#define REAL 2
#define AVERAGE 1

/*** ALL DECLARATIONS THAT FOLLOW ARE FOR THE EXTERNAL VARIABLES ***/

int round;
unsigned int seedADR; /* seed for addresses */
unsigned int seedZON; /* seed for zones */

int maxrows;
int maxcols;
int picks;
int avpicks; /* average numb. of picks/cycle */

float vx, vy;
float e; /* e = v_y/v_x */
float avel;
float two_b_cost;

struct coordinates
{
    int gennumb;
    int X;
    int Y;
}

struct coordinates address[PICKSIZE];
int BW[PICKSIZE][PICKSIZE];
int BROUTE[PICKSIZE];

int XCOL[PICKSIZE];
int YROW[PICKSIZE];

int lowerx[PICKSIZE];
int lowery[PICKSIZE];
int upperx[PICKSIZE];
int upperry[PICKSIZE];
int u_count;
int d_count;
int OAH, OBH, OAV, OBV;    /* boundaries of zone A and zone B */
float bcost1;

time_t band_start1, band_end1, band_2opt_end1;
double band_timel, band_2opt_timel;

/**************************** BABTSP ******************************/
int tweight;
int best[PICKSIZE];
int fwdptr[PICKSIZE];
int backptr[PICKSIZE];
int I, J;
time_t tsp_start, tsp_end, tsp_limit;
double tsp_time, runtime, timelimit;

/**************************** AVERAGE ******************************/

/* ARRAYS AND VARIABLES WHICH KEEP TOURS AND TOUR COSTS OBTAINED WITH
AVERAGE VELOCITIES FOR OUTPUT FILE, GRAPHIC DISPLAY OR RECALCULATION
WITH REAL TRAVEL TIMES */

float bandcost1, band_twol;
int BANDX[PICKSIZE], BANDY[PICKSIZE];
float RBANDCOST1 = 0;
float RBDCOST1 = 0;
float tspcost;
int TSPROUTE [PICKSIZE];
float RTSPCOST = 0;

/*********************************************************************************/
main()
{
    _clearscreen(_GCLEARSCREEN);
    input();

    for(round = AVERAGE; round <= REAL; round++)
    {
        if(round == REAL)
        {
            /* fileinput(); */
        }

        if(round == AVERAGE)
        {
            layout1();
            randgen1();
        }
        bandl();
        babtsp();
        _clearscreen(_GCLEARSCREEN);

        if(round == REAL)
        {
            output1();
            getchar();
            getchar();
        }
        _clearscreen(_GCLEARSCREEN);
    } /* end for */
} /* end */
/*********************************************************************/

source file <BAND1.C>

Function "bandl" operates in the same way as function "band",
with the exception that it is only applied to the layout
designed for the heuristic BAND1.

*********************************************************************/

#include "externs.h"

bandl()
{
  int i;
  time(&band_startl);
  split1();

  bandqsor(&lowerx[l], &lowerx[d_count], &lowery[l], &lowery[d_count]);
  colsort(lowerx, d_count, lowery)

  bandqsor(&upperx[l], &upperx[u_count], &uppery[l], &uppery[u_count]);
  colsort(upperx, u_count, uppery);

  rejoin();
  distmat(&bcostl, &RBDCOSTl);

  if(round == AVERAGE)
    bandcostl = bcostl; /* for graph purposes */

  two_b_cost = bcostl; /* for graph purposes */
  time(&band_endl);

  two_opt(BW, BROUTE, bcost1, &two_b_cost); /* 4th arg. for graph purposes */
  time(&band_2opt_endl);
  band_timel = difftime(band_endl, band_startl);
  band_2opt_timel = difftime(band_2opt_endl, band_startl);

112
if (round == AVERAGE) /* for graph disp */
{
    band_twol = two_b_cost;
    for (i = 1; i <= picks; i++)
    {
        BANDX[i] = XCOL[BROUTE[i]];
        BANDY[i] = YROW[BROUTE[i]];
    }
}
else
{
    brecalc(&RBANDCOST1);
}

getchar();
gechar();
display1();

} /* end */
Function "layout1" - description

Function "layout1" calculates the dimensions of zones A and B for BAND1 heuristic as it is shown on fig. B2. During the calculations the areas of zone A and zone B are kept as close as possible to 10 and 20 percent of the rack area respectively.

The function "layout1" starts iterations for zone A by setting $OAH = L/3$ and $OAV = e \cdot OAH$, where $e = V_y/V_x$.

After the dimensions of zone A have been determined, OBH is set $OBH = 2 \cdot OAH$, $OBV = e \cdot OBH$ and the iterations for zone B are performed.

Because of the rack's cellular structure $OAH$, $OAV$, $OBH$ and $OBV$ can be increased (decreased) only by a discrete integer value. Thus the accuracy achieved is in the range of one column of locations ($OAV$ or $OBV$) by violating the value of $e$ by no more than 25%.

Fig. B2. Zone boundaries for the heuristic BAND1.
/*******************************************************************************/
/*
* source file <LAYOUT1.C>
* Function "layoutl" determines boundaries of zone A and zone B for
* the heuristic BAND1.
*
*******************************************************************************/

#include "externs.h"
#include <math.h>

layout1()
{
    int AA, BB, aa bb; /* absolute and current areas of zones A and B respectively. */
    float ecur, ecurb;
    double correct;
    correct = sqrt(avel); /* correction coefficient; correct=1 if rack is square in time */

    OAH = maxcols * correct / 3;
    OAV = OAH * e + 0.6;
    ecur = (float)OAV / OAH;

    AA = 0.1 * maxcols * maxrows;
    BB = 0.2 * maxcols * maxrows;

    aa = OAH * OAV;
    while( ((AA - aa) > OAV) || ((AA - aa) < -OAV) )
    {
        if( (AA - aa) > OAV)
        {
            if(0.755 * e > ecur) /* prevent no more than 25% violation of squareness */
            {
                break;
            }

        OAH = OAH + 1;
        OAV = (OAH - 1) * e;
    }
ecur = (float)OAV/OAH;
aa = OAH * OAV;
}

if ( (AA - aa) < -OAV)
{
    if(0.755 * ecur > e)
    {
        break;
    }
    OAH = OAH -1;
    OAV=(OAH + 1) * e +0.5;
    ecur = (float)OAV/OAH;
aa = OAH * OAV;
}
}/* end while */

OBH = 2 * OAH/correct;
OBV = OBH * e+0.5;
ecurb = (float)OBV/OBH;

while(((OBH*OBV - aa - BB) > OBV) || ((OBH*OBV - aa - BB) < -OBV))
{
    if((aa + BB - OBH*OBV) > OBV)
    {
        if((0.755 * e > ecurb) && (OAV < OBV)) /* prevent more than 25% violation of squareness */
        {
            break;
        }
    }
    OBH = OBH +1;
    if((aa + BB - OBH*OBV) <= OBV)
    {
        break;
    }
    OBV = (OBH - 1) * e;
    ecurb = (float)OBV/OBH;
}
if((aa + BB - OBH*OBV) < -OBV)
{
    if((0.76 * ecurb > e) && (CAV < OBV))
    {
        break;
    }

    OBH = OBH - 1;
    if((aa + BB - OBH*OBV) >= -OBV)
    {
        break;
    }

    OBV = (OBH + 1) * e + 0.5;
    ecurb = (float)OBV/OBH;
}

} /* end while */

if(0.8*picks > OAH*OAV)
{
    printf("\n ZONE 'A' IS NOT LARGE ENOUGH FOR THIS NUMBER OF PICKS");
    exit(1);
}

} /* end */
**source file <RANDGEN1.C>**

Function "randgen1" generates pseudo random addresses in zones A, B and C. The addresses are distributed such that there is a probability of 80% that a generated address will fall in zone A, 15% in zone B and 5% in zone C.

```
#include "externs.h"
#define UNIQUE 1
#define NOT_UNIQUE 0

randgen1()
{
    int i, j;
    float a, b;
    float x, y;
    int state;
    int ZR, AR;

    XCOL[1] = 1; /* address with coord. (1,1) is the I/O point */
    YROW[1] = 1;

    address[1].X = 1;
    address[1].Y = 1;
    address[1].gennumb = 1;

    srand(seedZON);
    for(i = 2; i <= picks; i++)
    {
        state = NOT_UNIQUE;
        ZR = rand();

        /* The probability of an address falling in a particular zone is obtained by dividing the interval [0,32767] into three intervals, whose lengths as percentage of the whole length correspond to the required probabilities */
```
if(ZR <= 26213)  /* 80% chance in zone A */
{
    AR = 1;
}
else if((ZR > 26213) && (ZR <= 31128)) /* 15% chance in zone B */
{
    AR = 2;
}
else
{
    AR = 3;
}

if (AR == 1)
{
    seedADR = rand();
    srand(seedADR);
    while( state == NOT_UNIQUE)
    {
        state = UNIQUE;
        a = rand();
        b = rand();
        x = a*maxcols/32767 ;
        y = b*maxrows/32767 ;
        while( x<1 || y<1)
        {
            a = rand();
            b = rand();
            x = a*maxcols/32767 ;
            y = b*maxrows/32767 ;
        }
        for(j = 1; j <=i ; j++)
        {
            if( YROW[j] == (int)y) && (XCOL[j] == (int)x))
            {
                printf("*******\n");
                state = NOT_UNIQUE;
                break;
            }
        }
    }
}

119
if(((int)y > OAV) || ((int)x > OAH))
{
    printf("*******\n");
    state = NOT_UNIQUE;
    break;
}
}

printf("*******\n");
state = NOT_UNIQUE;
break;

} /* end while */
YROW[i] = y;
XCOL[i] = x;

address[i].X = x;
address[i].Y = y;
address[i].gennumb = i;
} /* end if AR == 1 */

if(AR == 2)
{
    seedADR = rand();
srand(seedADR);
    while( state == NOT_UNIQUE)
    {
        state = UNIQUE;
        a = rand();
        b = rand();
        x = a*maxcols/32767 ;
        y = b*maxrows/32767 ;
        while( x < 1 || y < 1)
        {
            a = rand();
            b = rand();
            x = a*maxcols/32767 ;
            y = b*maxrows/32767 ;
        }
        for(j=1;j<i;j++)
        {
            if(( YROW[j] == (int)y) && (XCOL[j] == (int)x))
            {
                printf("*******\n");
                state = NOT_UNIQUE;
                break;
            }
        }
    }
}
else if(((int)x <= OAH) && ((int)y <= OAV))
{
    printf("******\n");
    state = NOT_UNIQUE;
    break;
}

else if(((int)x > OBH) || ((int)y > OBV))
{
    printf("******\n");
    state = NOT_UNIQUE;
    break;
}
}
/* end while */
YROW[i] = y;
XCOL[i] = x;

address[i].x = x;
address[i].y = y;
address[i].gennumb = i;
} /* end if AR == 2 */

if( AR == 3)
{
    seedADR = rand();
srand(seedADR);
    while( state == NOT_UNIQUE)
    {
        state = UNIQUE;
        a = rand();
        b = rand();
        x = a*maxcols/32767 ;
        y = b*maxrows/32767 ;
        while( x < 1 || y < 1)
        {
            a = rand();
            b = rand();
            x = a*maxcols/32767 ;
            y = b*maxrows/32767 ;
        }
    }
for(j = 1; j <= i; j++)
{
    if(( YROW[j] == (int)y) && (XCOL[j] == (int)x))
    {
        printf("******\n");
        state = NOT_UNIQUE;
        break;
    }

    else if(( (int)x <= OBH) && ( (int)y <= OBV))
    {
        printf("******\n");
        state = NOT_UNIQUE;
        break;
    }
}

YROW[i] = y;
XCOL[i] = x;

address[i].X = x;
address[i].Y = y;
address[i].gennumb = i;

} /* end if AR == 3 */

} /* end for i */

} /* end */
Function "splitl" works in the same way as function "split", with the exception that the dividing line between the lower and the upper layer is situated at half of the height of zone B.

#include "externs.h"

splitl()
{
    int splitlev;
    int i;

    splitlev = OV/2.0;  /* dividing line */
    u_count = d_count = 0;

    for( i = 1; i<= picks; i++)
    {
        if(YROW[i] <= splitlev)
        {
            d_count++;
            lowerx[d_count] = XCOL[i];
            lowery[d_count] = YROW[i];
        }
        else
        {
            u_count++;
            upperx[u_count] = XCOL[i];
            uppery[u_count] = YROW[i];
        }
    }
}
Source file <OUTPUT1.C>

Function "output1" is a screen display of the results from the heuristic BAND1.

#include "externs.h"

output1()
{
    float rtweight;
    rtweight = tweight/1.0; /* converts int tweight into float */
    printf("_________________________
")
    printf("PICKS = %d
", picks, avpicks);
    printf(" vel.vector = %1.1f
", maxrows, avel);
    printf(" L = %d
", maxcols, vx, vy);
    printf(" SEED = %d
", seedZON);
    printf("_________________________
")
    printf("BAND1
")
    printf(" BAND1(const) = %4.2f %4.2f
", RBDCOST1/10, RBANDCOST1/10);
    printf(" BAND1(real) = %4.2f
", bcost1/10, two_b_cost/10);
    printf(" BAND1runtime = %2.1f %2.1f
", band_time1, band_2opt_time1);
    printf("_________________________
")
    printf("TSP
")
    printf(" TSP(const) = %4.2f %4.2f
", RTSPCOST/10, rtweight/10);
    printf(" TSP(real) runtime = %5.1f %5.1f
", runtime, tsp_time);
}

124
Fig. B3. Hierarchy of the functions involved in the simulation of the BAND2 zone configurations.
Function "main" is the main function for the heuristic BAND2.

#include <stdio.h>
#include <graph.h>
#include <time.h>

#define PICKSIZE 36
#define BIGNUMB 9999
#define REAL 2
#define AVERAGE 1

/*** ALL DECLARATIONS THAT FOLLOW ARE FOR THE EXTERNAL VARIABLES ***/

int round;
unsigned int seedADR; /* seed for addresses */
unsigned int seedZON; /* seed for zones */

int maxrows;
int maxcols;
int picks;
int avpicks; /* average numb. of picks/cycle */

float vx, vy;
float e; /* \( e = \frac{v_y}{v_x} \) */
float avel;
float two_b_cost;

struct coordinates {
    int gennumb;
    int X;
    int Y;
}

struct coordinates address[PICKSIZE];

int BW[PICKSIZE][PICKSIZE];
int BROUTE[PICKSIZE];

int XCOL[PICKSIZE];
int YROW[PICKSIZE];

int lowerx[PICKSIZE];
int lowery[PICKSIZE];
int upperx[PICKSIZE];
int uppery[PICKSIZE];
int u_count;
int d_count;
int w1, w2; /* widths of zones A and B in BAND2 and BAND3 */
float bcost2;

time_t band_start2, band_end2, band_2opt_end2;
double band_time2, band_2opt_time2;

/****************************** BABTSP ******************************/

int tweight;
int best[PICKSIZE];
int fwdptr[PICKSIZE];
int backptr[PICKSIZE];
int I, J;
time_t tsp_start, tsp_end, tsp_limit;
double tsp_time, runtime, timelimit;

/****************************** AVERAGE ******************************/

/* ARRAYS AND VARIABLES WHICH KEEP TOURS AND TOUR COSTS OBTAINED WITH
AVERAGE VELOCITIES FOR OUTPUT FILE, GRAPHIC DISPLAY OR RECALCULATION
WITH REAL TRAVEL TIMES */

float bandcost2, band_two2;
int BANDX[PICKSIZE], BANDY[PICKSIZE];
float RBANDCOST2 = 0;
float RDCCOST2 = 0;
float tspcost;
int TSPROUTE [PICKSIZE];
float RTSPCOST = 0;

/****************************** */
main()
{
    _clearscreen( _GCLEARSCREEN );
    input();

    for(round = AVERAGE; round <= REAL; round++)
    {
        if(round == REAL)
        {
            /* fileinput(); */
        }

        if (round == AVERAGE)
        {
            strwidth();
            layout2();
            randgen2();
        }
        band2();
        babtsp();
        _clearscreen( _GCLEARSCREEN );
    }
    if(round == REAL)
    {
        output2();
        getchar();
        getchar();
    }
    _clearscreen( _GCLEARSCREEN );
}
  /* end for */
}  /* end */
Function "strwidth" calculates widths $w_1$ and $w_2$ of the strips which form zones A and B for the heuristics BAND2 and BAND3. The analytical derivation of the strip widths is presented in chapter IV.

```
#include <stdio.h>
#include <math.h>

stridth()
{
    float P1 = 0.8, P2 = 0.2;
    double d1, d2;
    double a, b, c, d, w, ww;

    d1 = (0.8*avpicks*vx*vy) / (0.1*maxcols*maxrows*3600); /* density in zone A */
    d2 = (0.15*avpicks*vx*vy) / (0.2*maxcols*maxrows*3600); /* density in zone B */

    w = sqrt(3/d1);
    w = 0.67*w;
    a = (3*P1*P2 + P2*P2)*d2;
    b = ((P1*P1 + 3*P1*P2)*d2*w) + (3*P1*P2 + P2*P2)*d1*w;
    c = (P1*P1 + 3*P1*P2)*d1*w*w - 3;
    d = sqrt(b*b - 4*a*c);
    ww = (-b + d)/(2*a); /* only the positive real root */

    w2 = ww*vy/60 + 0.5;
    w1 = w*vy/60 + 0.5;

    if(w1 < 1) w1 = 1;
    if(w2 < 1) w2 = 1;
    if(2*(w1 + w2) > maxrows)
    {
        printf("\nRACK TOO LOW TO ACCOMMODATE ZONE 'B'\n");
        exit(1);
    }
}
```
/***********************/

* source file <BAND2.C>*
* Function "band2" operates in the same way as function "band", *
* with the exception that it is only applied to the layout *
* designed for the heuristic BAND2. *
*
***********************************************************/

#include "externs.h"

band2()
{
    int i;
    time(&band_start2);
    split2();

    bandqsor(&lowerx[1], &lowerx[d_count], &lowery[1], &lowery[d_count]);
    colsort(lowerx, d_count, lowery)

    bandqsor(&upperx[1], &upperx[u_count], &uppery[1], &uppery[u_count]);
    colsort(upperx, u_count, uppery);

    rejoin();
    distmat(&bcost2, &RBDCOST2);

    if(round == AVERAGE)
        bandcost2 = bcost2; /* for graph purposes */

    two_b_cost = bcost2; /* for graph purposes */
    time(&band_end2);

    two_opt(BW, BROUTE, bcost2, &two_b_cost); /* 4th arg. for graph purposes */
    time(&band_2opt_end2);
    band_time2 = difftime(band_end2, band_start2);
    band_2opt_time2 = difftime(band_2opt_end2, band_start2);
if (round == AVERAGE) /* for graph disp */
{
    band_two2 = two_b_cost;
    for (i = 1; i <= picks; i++)
    {
        BANDX[i] = XCOL[BROUTE[i]];
        BANDY[i] = YROW[BROUTE[i]];
    }
}
else
{
    brecalc(&RBANDCOST2);
}
getchar();
getchar();
display2();
} /* end */
Function "layout2" - description

Function "layout2" calculates the dimensions of zones A and B for the BAND2 heuristic as shown on fig. B4. Initial values of w1 and w2 have been calculated by function "strwidth". During the calculations, the areas of zone A and zone B are kept as close as possible to 10 and 20 percent of the rack area respectively. If the length OAH or OBH happen to exceed the length of the rack L (maxcols in the function), w1 or w2 respectively is increased by one and the length recalculated.

---

Fig. B4. Zone boundaries for the heuristic BAND2.
#include "externs.h"
#include <math.h>

layout2()
{
    int AA, BB, bb;

    AA = 0.1*maxcols*maxrows;
    BB = 0.2*maxcols*maxrows;

    OAH = AA/(2*wl);

    while( OAH > maxcols )
    {
        w1 = w1 + 1;
        OAH = AA/(2*w1);
    }

    bb = 2*w2*OAH;
    OBH = OAH + (BB - bb)/(2*(wl + w2));

    while(OBH > maxcols)
    {
        w2 = w2 + 1;
        bb = 2*w2*OAH;
        OBH = OAH + (BB-bb)/(2*(w1 + w2));
        if( 2*(w1 + w2) > maxrows)
        {
            printf("\n RACK TOO LOW TO ACCOMMODATE ZONE 'B'");
            exit(1);
        }
    }
}
if(0.8*picks > OAH*2*w1)
{
    printf("\n ZONE 'A' IS NOT LARGE ENOUGH FOR THIS NUMB. OF PICKS");
    exit(1);
}
}
/**************************
source file <RANDGEN2.C>
Function "randgen2" generates pseudo random addresses in zones A, B and C. The addresses are generated such that there is a probability of 80% that a generated address will fall in zone A, 15% in zone B and 5% in zone C.

#include "externs.h"
#define UNIQUE 1
#define NOT_UNIQUE 0

randgen2()
{
  int i, j;
  float a, b;
  float x, y;

  int state;
  int ZR, AR;

  XCOL[l] = 1; /* address with coord. (1,1) is the I/O point */
  YROW[l] = 1;

  address[l].X = 1;
  address[l].Y = 1;
  address[l].gennumb = 1;

  srand(seedZON);
  for(i = 2; i <= picks; i++)
  {
    state = NOT_UNIQUE;
    ZR = rand();

    /* The probability of an address falling in a particular zone is obtained by dividing the interval [0,32767] into three intervals, whose lengths as percentage of the whole length correspond to the required probabilities */
  }
if(ZR <= 26213)  /* 80% chance in zone A */
{
    AR = 1;
}
else if((ZR > 26213) && (ZR <= 31128)) /* 15% chance in zone B */
{
    AR = 2;
}
else
{
    AR = 3;
}

if (AR==1)
{
    seedADR = rand();
    srand(seedADR);
    while( state == NOT_UNIQUE)
    {
        state = UNIQUE;
        a = rand();
        b = rand();
        x = a*maxcols/32767 ;
        y = b*maxrows/32767 ;
        while( x<1 || y<1)
        {
            a = rand();
            b = rand();
            x = a*maxcols/32767 ;
            y = b*maxrows/32767 ;
        }
        for(j = 1; j <= i; j++)
        {
            if(( YROW[j] == (int)y) && (XCOL[j] == (int)x))
            {
                printf("******\n");
                state = NOT_UNIQUE;
                break;
            }
        }
    }
}
if(((int)y > (diff + w2 + 2*w1)) || ((int)y <= 
(diff + w2))
{
    printf("*******\n");
    state = NOT_UNIQUE;
    break;
}
if(((int)y <= (diff + w2 + 2*w1) && ((int)y > 
(diff + w2)))
{
    if((int)x > CAH)
    {
        printf("*******\n");
        state = NOT_UNIQUE;
        break;
    }
}
/* end while */
YROW[i] = y;
XCOL[i] = x;
address[i].X = x;
address[i].Y = y;
address[i].gennumb = i;
} /* end if */

if(AR == 2)
{
    seedADR = rand();
srand(seedADR);
    while( state == NOT_UNIQUE)
    {
        state = UNIQUE;
        a = rand();
        b = rand();
        x = a*maxcols/32767 ;
        y = b*maxrows/32767 ;
        while( x<l || y<l)
        {
            a = rand();
        }
b = rand();
x = a*maxcols/32767;
y = b*maxrows/32767;
}

for(j = 1; j <= i ; j++)
{
  if(( YROW[j] == (int)y) && (XCOL[j] == (int)x))
  {
    printf("*******\n");
    state = NOT_UNIQUE;
    break;
  }
  if(((int)y <= diff) || ((int)y > (diff + 2*wl + 2*w2)))
  {
    printf("*******\n");
    state = NOT_UNIQUE;
    break;
  }
  if(((int)y > diff) && ((int)y <= (diff + 2*wl + 2*w2)))
  {
    if((int)x > OBH)
    {
      printf("*******\n");
      state = NOT_UNIQUE;
      break;
    }
  }
  if(((int)y > (diff + w2)) && ((int)y <= (diff + w2 + 2*wl )))
  {
    if(((int)x < OAH) || ((int)x > OBH))
    {
      printf("*******\n");
      state = NOT_UNIQUE;
      break;
    }
  }
} /* end for */
} /* end while */
if (AR == 3)
{
    seedADR = rand();
srand(seedADR);
    while (state == NOT_UNIQUE)
    {
        state = UNIQUE;
        a = rand();
        b = rand();
        x = a*maxcols/32767;
        y = b*maxrows/32767;
        while (x < l || y < l)
        {
            a = rand();
            b = rand();
            x = a*maxcols/32767;
            y = b*maxrows/32767;
        }
        for (j = 1; j <= i; j++)
        {
            if (!((YROW[j] == (int)y) && (XCOL[j] == (int)x))
            {
                printf("******\n");
                state = NOT_UNIQUE;
                break;
            }
            if (((int)y > diff) && ((int)y <= (diff + 2*w1 + 2*w2)))
            {
                if ((int)x < OBH)
                {
                    printf("******\n");
                    state = NOT_UNIQUE;
                    break;
                }
            }
        }
    }
} // end if AR == 2 */
YROW[i] = y;
XCOL[i] = x;

address[i].X = x;
address[i].Y = y;
address[i].gennumb = i;

} /* end if AR == 3 */

} /* end for i */

} /* end */
/***/

* source file <SPLIT2.C>*

* Function "split2" works in the same way as function "split", with *
* the exception that the dividing line between the lower and the *
* upper layer is the axis of symmetry of the zones A and B. *

***************************************************************************/

#include "externs.h"

split2()
{
    int splitlev, diff;
    int i;
    diff = (maxrows - 2*(w1 + w2))/2;
    splitlev = diff + w1 + w2;
    u_count = d_count = 0;

    for( i = 1; i<= picks; i++)
    {
        if(YROW[i] <= splitlev)
        {
            d_count++;
            lowerx[d_count] = XCOL[i];
            lowery[d_count] = YROW[i];
        }
        else
        {
            u_count++;
            upperx[u_count] = XCOL[i];
            uppery[u_count] = YROW[i];
        }
    }
}
/*********************************************************************/
/*
 * Source file <OUTPUT2.C>
 * Function "output2" is a screen display of the results from the 
 * heuristic BAND2.
 */
*********************************************************************/

#include "externs.h"

output2 ()
{
float rtweight;
rtweight = tweight; /* converts int tweight into float */

printf(" 
");
printf("** PICKS = %dAVRPICKS = %d
\n", picks, avpicks);
printf("** H = %dvel.vector = %4.1f
\n", maxrows, avel);
printf("** L = %dVX = %4.1fVY = %4.1f
\n", rnaxcols, vx, vy);
printf("** SEED = %dSEEDZON);
printf(" 
");
printf("** BAND2BAND_202 = const = %4.2f BAND_202 = real = %4.2f
\n", RBDCOST2/10, RBANDCOST2/10);
printf("** BAND2(real) = %4.2f
\n", bcost2/10, two_b_cost/10);
printf("** BAND2runtime = %4.1f
\n", band_time2, band_2opt_time2);
printf(" 
");
printf("** TSTP(const) = %4.2fRTSP = const = %4.2f
\n", RTSPCOST/10, rtweight/10);
printf("** TSP(real) = %4.2fTSP(real) runtime = %5.1f
\n", RTSPCOST/10, rtweight/10);
printf("** TSP(const) runtime = %5.1fTSP(real) runtime = %s
\n", runtime, tsp_time);
printf(" 
");
}

Fig. B5. Hierarchy of the functions involved in the simulation of the BAND3 zone configurations.
#include <stdio.h>
#include <graph.h>
#include <time.h>

#define PICKSIZE 36
#define BIGNUMB 9999
#define REAL 2
#define AVERAGE 1

/*** ALL DECLARATIONS THAT FOLLOW ARE FOR THE EXTERNAL VARIABLES ***/

int round;
unsigned int seedADR; /* seed for addresses */
unsigned int seedZON; /* seed for zones */

int maxrows;
int maxcols;
int picks;
int avpicks; /* average numb. of picks/cycle */

float vx, vy;
float e; /* e = v_y/v_x */
float avel;
float two_b_cost;

struct coordinates {
    int gennumb;
    int X;
    int Y;
}

struct coordinates address[PICKSIZE];
int BW[PICKSIZE][PICKSIZE];
int BROUTE[PICKSIZE];

int XCOL[PICKSIZE];
int YROW[PICKSIZE];

int lowerx[PICKSIZE];
int lowery[PICKSIZE];
int upperx[PICKSIZE];
int uppery[PICKSIZE];
int u_count;
int d_count;
int w1, w2; /* widths of zones A and B in BAND2 and BAND3 */
float bcost3;

time_t band_start3, band_end3, band_2opt_end3;
double band_time3, band_2opt_time3;

/******************************* BABTSP **************************************/
int tweight;
int best[PICKSIZE];
int fwdptr[PICKSIZE];
int backptr[PICKSIZE];
int I,J;
time_t tsp_start, tsp_end, tsp_limit;
double tsp_time, runtime, timelimit;

/******************************* AVERAGE **************************************/

/* ARRAYS AND VARIABLES WHICH KEEP TOURS AND TOUR COSTS OBTAINED WITH AVERAGE VELOCITIES FOR OUTPUT FILE, GRAPHIC DISPLAY OR RECALCULATION WITH REAL TRAVEL TIMES */

float bandcost3, band_two3;
int BANDX[PICKSIZE], BANDY[PICKSIZE];
float RBANDCOST3 = 0;
float RBDCOST3 = 0;
float tspcost;
int TSPROUTE [PICKSIZE];
float RTSPCOST = 0;

/******************************* */
main()
{
    _clearscreen(_GCLEARSCREEN);
    input();

    for(round = AVERAGE; round <= REAL; round++)
    {
        if(round == REAL)
        {
            /* fileinput(); */
        }

        if(round == AVERAGE)
        {
            strwidth();
            layout3();
            randgen3();
        }
        band3();
        bactsp();
        _clearscreen(_GCLEARSCREEN);

        if(round == REAL)
        {
            output3();
            getchar();
            getchar();
        }
        _clearscreen(_GCLEARSCREEN);
    } /* end for */
} /* end */
/*******************************/
*/
*/ source file <BAND3.C>*/
*/ Function "band3" operates in the same way as function "band",*/
*/ with the exception that it is only applied to the layout*/
*/ designed for the heuristic BAND3.*/
*/
*********************************************************************/

#include "externs.h"

band3()
{
int i;
time(&band_start3);
split3();

bandqsor(&lowerx[1], &lowerx[d_count], &lowery[1], &lowery[d_count]);
colsort(lowerx, d_count, lowery)

bandqsor(&upperx[1], &upperx[u_count], &uppery[1], &uppery[u_count]);
colsort(upperx, u_count, uppery);

rejoin();
distmat(&bcost3, &RBDCOST3);

if(round == AVERAGE)
bandcost3 = bcost3; /* for graph purposes */
two_b_cost = bcost3; /* for graph purposes */
time(&band_end3);

two_opt(BW, BROUTE, bcost3, &two_b_cost); /* 4th arg. for graph purposes */
time(&band_2opt_end3);
bond_time3 = difftime(band_end3, band_start3);
bond_2opt_time3 = difftime(band_2opt_end3, band_start3);
if (round == AVERAGE) /* for graph disp */
{
    band_two3 = two_b_cost;
    for (i = 1; i <= picks; i++)
    {
        BANDX[i] = XCOL[BROUTE[i]];
        BANDY[i] = YROW[BROUTE[i]];
    }
}
else
{
    brecalc(&RBANDCOST3);
}
getchar();
getchar();
display3();

} /* end */
Function "layout3" - description

Function "layout3" calculates the dimensions of zones A and B for the BAND3 heuristic as shown on fig. B6. Initial values of w1 and w2 have been calculated by function "strwidth". During the calculations, the areas of zone A and zone B are kept as close as possible to 10 and 20 percent of the rack area respectively. If we note 'L' to be the rack length and 'a' the velocity vector, then if the length OAH or OBH happen to exceed a.L/sin(e), w1 or w2 respectively is increased by one and the length recalculated.

Fig. B6. Zone boundaries for the heuristic BAND3.

From fig. B6 it is seen that due to the inclined zones some areas fall outside the extremes of the rack. For zone A in total this is the area of triangle Omt and for zone B triangle OMT. Function "layout3" takes this into account by calculating during each iteration the areas of each of the above triangles and adding them to the area of the corresponding zone for accurate
calculation of length OAH and OBH respectively. Using the notations \( Oq = w_1 \) and \( qQ = w_2 \) from the figure, the area of triangle Omt is:

\[ S_{Otm} = \frac{w_1^2}{2\sin(e)\cos(e)} \]

and the area of triangle OMT is:

\[ S_{OMT} = \frac{(w_1 + w_2)^2}{2\sin(e)\cos(e)} \]

Letters in brackets on the figure indicate some of the possible shapes of zone B depending on the rack size.
source file <LAYOUT3.C >
Function "layout3" determines boundaries of zone A and zone B for
the heuristic BAND3.

#include "externs.h"
#include <math.h>

layout3()
{
    int AA, BB, bb;
    int addl, add2;   /* areas lost because of the inclined zones */
    double el, sine, cose, tane;
    double vyl, vx1;
    float velvector;

    vyl = vy/1.0;   /* convert into float */
    vx1 = vx/1.0;

    el = (vy1)/(vx1);
    sine = sin(el);
    cose = cos(el);
    tane = tan(el);

    add1 = (w1*w1)/(2*cose*sine) + 0.5;   /* triangle Omt */
    add2 = ((w1+w2)*(w1+w2))/(2*sine*cose);   /* triangle OMT */

    AA = 0.1*maxcols*maxrows;
    BB = 0.2*maxcols*maxrows;

    if (avel < 1)
    {
        velvector = (1/avel);
    }
    else
    {
        velvector = avel;
    }
OAH = AA/(2*wl) + 0.5;
OAH = OAH + add1/(2*wl) + 0.5;

while( OAH > (maxrows/(sine*velvector)))
{
    wl = wl + 1;
    OAH = AA/(2*wl) + 0.5;
    OAH = OAH + add1/(2*wl) + 0.5;
}

bb = 2*w2*OAH - 2*((2*wl+w2)*w2)/(2*tane) ; /* -2trapeziun QMmq */
OBH = OAH + (BB-bb)/(2*(wl + w2));
OBH = OBH + add2/(2*(wl + w2));

while(OBH > (maxrows/(sine*velvector)))
{
    w2 = w2+1;
    bb = 2*w2*OAH - ((2*wl+w2)*w2)/(2*tane);
    OBH = OAH + (BB-bb)/(2*(wl+w2));
    OBH = OBH + add2/(2*(wl+w2));
    if( 2*(w1 + w2) > maxrows)
    {
        printf("\n RACK TOO LOW TO ACCOMMODATE ZONE 'B' ");
        exit(1);
    }
}

if(0.8*picks > OAH*2*wl)
{
    printf("\n ZONE 'A' IS NOT LARGE ENOUGH FOR THIS NUMB OF PICKS");
    exit(1);
}

} /* end */
Function "randgen3" generates pseudo random addresses in zones A, B and C for the heuristic BAND3. The distribution of addresses is such that there is a probability of 80% that a generated address will fall in zone A, 15% in zone B and 5% in zone C.

#include "externs.h"

#define UNIQUE 1
#define NOT_UNIQUE 0

randgen3()
{
    int i, j;
    float a, b;
    float x, y;

    int state;
    int ZR, AR;
    double e1, sine, cose, tane;
    double vyl, vxl;

    /* We denote zone A and zone B as two pentagons A{O,a1,a2,a3,a4} and B{O,b1,b2,b3,b4}. The points of these pentagons correspond to the points of pentagons A{O,m,r,p,n} and B{O,M,R,P,N} on fig.(***) */

    float alx, a2x, a4x;
    float aly, a2y, a4y;
    float blx, b2x, b4x;
    float bly, b2y, b4y;
    float grad1, grad2;

    vyl = vy/1.00; /* convert into float */
    vxl = vx/1.00;
\[ e_1 = \frac{v_y}{v_x}; \]

\[ \text{sine} = \sin(e_1); \]

\[ \text{cose} = \cos(e_1); \]

\[ \text{tane} = \tan(e_1); \]

\[ \text{grad}1 = e_1; \]

\[ \text{grad}2 = -\frac{1}{e_1}; \quad /* e + 90\text{deg} */ \]

\[ a_{1x} = \frac{w_1}{\text{sine}}; \]

\[ a_{1y} = 0; \]

\[ a_{2x} = \frac{w_1}{\text{sine}} + ((\text{OA} - (\frac{w_1}{\text{tane}})\text{cose})\]

\[ a_{2y} = (\text{OA} - (\frac{w_1}{\text{tane}})\text{sine}; \]

\[ a_{4x} = 0; \]

\[ a_{4y} = \frac{w_1}{\text{cose}}; \]

\[ b_{1x} = \frac{(w_1 + w_2)}{\text{sine}}; \]

\[ b_{1y} = 0; \]

\[ b_{2x} = \frac{(w_1 + w_2)}{\text{sine}} + ((\text{OB} - ((w_1 + w_2)/\text{tane})\text{cose})\]

\[ b_{2y} = (\text{OB} - ((w_1 + w_2)/\text{tane})\text{sine}; \]

\[ b_{4x} = 0; \]

\[ b_{4y} = \frac{(w_1 + w_2)}{\text{cose}}; \]

\[ \text{XCOL}[1] = 1; \quad /* \text{address with coordinates (1,1) is the I/O point */} \]

\[ \text{YROW}[1] = 1; \]

\[ \text{address}[1].X = 1; \]

\[ \text{address}[1].Y = 1; \]

\[ \text{address}[1].\text{gennumb} = 1; \]

\[ \text{srand}(\text{seedZON}); \]

\[ \text{for}(i = 2; i <= \text{picks}; i++) \]

\[ \quad \{ \]

\[ \quad \text{state} = \text{NOT}\_\text{UNIQUE}; \]

\[ \quad \text{ZR} = \text{rand}(); \]

\[ /* \text{The probability of an address falling in a particular zone is obtained by dividing the interval [0,32767] into three intervals whose lengths as percentage of the whole length correspond to the required probabilities */} \]

154
if(ZR <= 26213) /* 80% chance in zone A */
{
    AR = 1;
}
else if((ZR > 26213) && (ZR <= 31128)) /* 15% chance in zone B */
{
    AR = 2;
}
else /* 5% chance in zone C */
{
    AR = 3;
}
if (AR == 1)
{
    seedADR = rand();
    srand(seedADR);
    while( state == NOT_UNIQUE)
    {
        state = UNIQUE;
        a = rand();
        b = rand();
        x = a*maxcols/32767 ;
        y = b*maxrows/32767 ;
        while( x<1 || y<1)
        {
            a = rand();
            b = rand();
            x = a*maxcols/32767 ;
            y = b*maxrows/32767 ;
        }
        for(j = 1; j <= i; j++)
        {
            if(( YROW[j] == (int)y) && (XCOL[j] == (int)x))
            {
                printf("********
");
                state = NOT_UNIQUE;
                break;
            }
        }
    }
}
else if(((int)y -(int)(grad1*x)) <= ((int)a4y -
(int)(grad1*a4x)) & & ((int)y -
(int)(grad1*x)) > ((int)a4y -
(int)(grad1*a4x)))
{
    if(( (int)y -(int)(grad2*x)) > (int)a2y -
(int)(grad2*a2x))
    {
        printf("*******\n");
        state = NOT_UNIQUE;
        break;
    }
}

else if(((int)y -(int)(grad1*x)) > ((int)a4y -
(int)(grad1*a4x)) & & ((int)y -(int)
(grad1*x)) <= ((int)a4y - (int)(grad1*a4x)))
{
    printf("*******\n");
    state = NOT_UNIQUE;
    break;
}

} /* end for */
} /* end while */

YROW[i]=y;
XCOL[i]=x;

address[i].x=x;
address[i].y=y;
address[i].gennumb=i;

} /* end if AR == 1 */

if(AR == 2)
{
    seedADR = rand();
srand(seedADR);
    while( state == NOT_UNIQUE)
    {
        state = UNIQUE;
    }
a = rand();
b = rand();
x = a*maxcols/32767 ;
y = b*maxrows/32767 ;
while( x<1 || y<1)
{
    a = rand();
b = rand();
x = a*maxcols/32767 ;
y = b*maxrows/32767 ;
}

for(j = 1; j <=i; j++)
{
    if(( YROW[j] == (int)y) && (XCOL[j] == (int)x))
    {
        printf("*******\n");
        state = NOT_UNIQUE;
        break;
    }

    else if(((int)y - (int)(grad1*x)) <= ((int)a4y - (int)(grad1*a4x))) && (((int)y - (int)(grad1*x)) > ((int)a4y - (int)(grad1*a4x)))
    {
        if(((int)y - (int)(grad2*x)) < ((int)a2y - (int)(grad2*a2x)) || (((int)y - (int)(grad2*x)) > ((int)b2y - (int)(grad2*b2x))))
        {
            printf("*******\n");
            state = NOT_UNIQUE;
            break;
        }
    }

    else if(((int)y - (int)(grad1*x)) < ((int)b4y - (int)(grad1*b4x)) && (((int)y - (int)(grad1*x)) >= ((int)bly - (int)(grad1*blx))))
    {

if(((int)y - (int)(grad2*x)) > ((int)b2y -
(int)(grad2*b2x)))
{
  printf("*******\n");
  state = NOT_UNIQUE;
  break;
}
else if(((int)y - (int)(grad1*x)) > ((int)b4y -
(int)(grad1*b4x)) || ((int)y -
(int)(grad1*x)) <= ((int)bly -
(int)(grad1*blx)))
{
  printf("*******\n");
  state = NOT_UNIQUE;
  break;
}
} /* end for*/
} /* end while */

YROW[i] = y;
XCOL[i] = x;
address[i].X = x;
address[i].Y = y;
address[i].gennumb = i;
} /* end if AR == 2 */

if( AR == 3)
{
  seedADR = rand();
srand(seedADR);

  while( state == NOT_UNIQUE)
  {
    state = UNIQUE;
    a = rand();
    b = rand();
    x = a*maxcols/32767;
    y = b*maxrows/32767;
  }
while( x<1 || y<1)
{
    a = rand();
    b = rand();
    x = a*maxcols/32767 ;
    y = b*maxrows/32767 ;
}

for(j = 1; j <= i; j++)
{
    if(( YROW[j] == (int)y) && (XCOL[j] == (int)x))
    {
        printf("*******\n");
        state = NOT_UNIQUE;
        break;
    }
    else if(((int)y - (int)(grad1*x)) <=
             ((int)b4y-(int)(grad1*b4x)) && (((int)y -
             (int)(grad1*x)) > ((int)bly-(
             int)(grad1*blx)))
    {
        if(((int)y - (int)(grad2*x)) <= ((int)b2y -
             (int)(grad2*b2x)))
        {
            printf("*******\n");
            state = NOT_UNIQUE;
            break;
        }
    }
}
} /* end for */
} /* end while */

YROW[i] = y;
XCOL[i] = x;

address[i].X = x;
address[i].Y = y;
address[i].gennumb = i;

} /* end if AR == 3 */
} /* end for i */
} /* end */
/********************************************************************
* 
* source file <SPLIT3.C> 
* Function "split3" separates the generated addresses into a lower 
* and upper layer. The dividing line starts from the bottom left 
* corner (I/O point) of the rack and is inclined at an angle equal 
* to atan(v_y/v_x). 
* 
********************************************************************/

#include "externs.h"

split3()
{
  int splitlev;
  int i;

  u_count = 0;
  d_count = 1;
  lowerx[d_count] = XCOL[1];
  lowery[d_count] = YROW[1];

  for( i = 2; i <= picks; i++)
  {
    if(YROW[i] - e*XCOL[i] <= 0.5 )
    {
      d_count++;
      lowerx[d_count] = XCOL[i];
      lowery[d_count] = YROW[i];
    }
    else
    {
      u_count++;
      upperx[u_count] = XCOL[i];
      uppery[u_count] = YROW[i];
    }
  }
}
/*********************************************************************/
/*
* Source file <OUTPUT3.C)
* Function "output3" is a screen display of the results from the
* heuristic BAND3.
*
*********************************************************************/

#include "externs.h"

output3()
{
    float rtweight;
    rtweight = tweight; /* converts int tweight into float */
    printf("PICKS = %d
\n",picks,avpicks);
    printf("H = %d
\n",maxrows,avel);
    printf("L = %d
\n",maxcols,vx,vy);
    printf("SEED= %d
\n",seedZON);
    printf("BAND3
\n");
    printf("BAND3(const) = %4.2f\n",RBDCOST3/10,RBANDCOST3/10);
    printf("BAND3(real) = %4.2f\n",bcost3/10,two_b_cost/10);
    printf("BAND3runtime = %4.1f\n",band_time3,band_2opt_t1me3);
    printf("TSP
\n");
    printf("TSP(const) = %4.2f\n",RTSPCOST/10,rtweight/10);
    printf("TSP(real)runtime = %5.1f\n", runtime,tsp_time);
}

161