1-1-1987

# Designing the optimal placement of spaces in a parking lot

R. Bingle

D. Meindertsma

W. Oostendorp

Gene A. Klaasen
*Calvin University*

*Mathl Modelling*, Vol. 9, No. 10, pp. 765–776, 1987
Printed in Great Britain. 

0270-0255/87 $3.00 + 0.00


# DESIGNING THE OPTIMAL PLACEMENT OF SPACES IN A PARKING LOT

*Team*: R. BINGLE, D. MEINDERTSMA and W. OOSTENDORP
*Faculty advisor*: G. KLAASEN
Mathematics Department, Calvin College, Grand Rapids, MI 49506, U.S.A.

**Abstract**—We develop a method for determining the optimal size and placement of parking spaces and approach aisles for an automobile parking lot. In particular, our solution concerns a parking lot of size 100′ × 200′ located at the corner of an intersection of two streets in a New England town. We begin by arguing the superiority of driver operation over attendant operation of vehicles to be parked. Then a statistical analysis is performed on a sampling of 160 1987 model automobiles to determine upper bounds and ideal values for the length and width of a parking space and for the turning radius required to navigate entrance into said space. Using this data, we show that the optimal degree for diagonal parking which minimizes necessary lot area for a particular space can be expressed as a function of the turning radius of the automobile and the width of the parking space. However, concerning our particular lot, right-angle parking is established as the method by which optimal use of space can be achieved by showing that to minimize wasted area using the diagonal parking scheme requires a greater number of spaces than can be accommodated in the given dimensions. Using these preliminary results, a computer program which utilizes a 12-way tree structure and recursion is employed to generate possible lot designs. Next, other considerations not so conducive to programming—such as snow removal, lot use fees and effects of our design on the adjacent intersection—are discussed.

## INTRODUCTION

We have been hired by the owner of a paved, 100′ × 200′ corner parking lot in New England to design the optimal size and location of the parking spaces to be painted on the lot. The lot owner has informed us that maximizing his revenue requires that his lot be able to accommodate as many automobiles as possible without compromising the convenience of his patrons and the accessibility of the establishment(s) that his lot serves.

What follows is our response to the lot owner. We begin by contrasting the use of a valet parking service with standard driver operation. Following an analysis of the size of parking spaces and approach aisles necessary is a discussion of right-angle parking vs diagonal parking. Then the computer program used to generate and analyze many of the diverse lot designs under consideration is described. Finally, an analysis of the strengths and weaknesses of the model and a section detailing other considerations not explicitly contained within the model is presented.

## ASSUMPTIONS AND HYPOTHESES

Our modelling of the optimal line placements on the parking lot is based on the following assumptions:

(1) No entrances to or exits from the lot exist other than those originating from or terminating at one of the two roads intersecting at the corner.
(2) Data gathered for 160 1987 model cars provides a reasonably reliable indication of the length, width and turning radius of those automobiles which will be using the lot [1]:

| | Length | Width | Turning radius |
|---|---|---|---|
| Mean | 14.7 | 5.63 | 17.5 |
| SD | 1.16 | 0.292 | 1.55 |
| Maximum | 17.3 | 6.58 | 21.0 |

(Where the dimensions are in feet and where turning radius is defined as the radius of the minimal circle required for a vehicle to complete a 360° turn.)
(3) A parked automobile will occupy one and only one parking space, and therefore the capacity of the lot is equivalent to the number of accessible spaces.

(4) A parking space is defined as the surface area on the lot bounded by two parallel line segments, called sides: one line segment in front; and one imaginary line segment parallel to that with terminal points at the endpoints of the side segments. Because these line segments—with infinitesimal width—cannot be painted, the painted stripes used to represent them are assumed to be painted in such a way that these line segments lie directly in the center of and parallel to the sides of the stripe.
(5) The "doubling up" of parked automobiles is not allowed, i.e. the removal of any vehicle in the lot is not dependent upon the moving of another vehicle.

## VALET PARKING vs DRIVER OPERATION

Our first concern in the development of the model is attendant operation vs driver operation of vehicles to be parked. Considering the goal of maximizing the number of automobiles which can be parked in the lot, attendant operation has the following advantages: first, this type of parking enables the lot owner to make more efficient use of available space by narrowing parking spaces and aisles with the realization that attendants will have greater driving abilities than that of the general public, and that attendants will gain a greater familiarity with the lot from experience, easing maneuverability in these narrower spaces and aisles; second, attendant parking serves as a buffer to regulate traffic in the lot—with a given number of attendants staffing a lot, the maximum number of cars traveling in the lot at a given time is equal to the number of employees, thus decreasing the risk of moving collisions.

However, the disadvantages of attendant operation outweigh the advantages discussed above. The first disadvantage of attendant parking involves insurance considerations. Provided sufficient turning radii and parking space length and width are present, a lot owner would be relatively free from insurance claims filed as a result of lot mishaps where owners are operating their own vehicles. With attendant operation, on the other hand, a lot owner could most certainly be held liable for vehicle damage stemming from his employee's actions, resulting in higher insurance rates for the lot owner. Furthermore, attendant parking would dictate that all cars arrive at a central location, thus reducing the entrance possibilities for the lot and greatly reducing the general applicability of the model. The final disadvantage to attendant parking discussed here is the formality often connotated with "valets"; i.e. valet parking may limit the uses of the lot to a more socially formal establishment such as a place of fine dining: e.g. attendant parking would hardly seem appropriate at a supermarket or shopping mall.

## CHOICE OF SPACE SIZE AND AISLE WIDTH

There are many possible considerations to take into account when determining how large the parking spaces should be and how wide the aisle should be. Some of these are: What is the turnover? Is parking all day or do people come and go frequently? Will mostly small cars use the lot or will there be a mixture of small and large cars? Is it luxury use or elderly use where more room is desired? These are all things that should be considered. However, since this sort of information is not given in the problem, we chose to select the space size and aisle width based on what will accommodate the largest cars built today, perhaps a little on the tight side, and comfortably accommodate mid-sized cars. The largest vehicle in the *Road & Track Magazine* list is 17.3′ long [1]. So making the spaces 18′ long will accommodate that vehicle. A length of 18′ allows over 3′ extra for the "average car"; 95% of all vehicles in the list will have over $1\frac{1}{2}$′ extra. The maximum vehicle width in the list was 6.58′, so choosing a space width of 8′ will allow enough room for that vehicle although getting in and out of the car may be a bit tight. But the average width is 5.63′, allowing almost 2.4′ of extra width, plenty of room to get in and out of a car; 95% of the cars are under 6.1′ wide, allowing almost 2′ for entry and exit. In the next section it will be shown that aisle width depends on the turning radius of the vehicle. The turning radius must be less than the width of the aisle. The maximum in the list was 21′ so we made the aisles 22′ wide assuming right-angle parking (the width varies with diagonal parking but always depends on the

"maximum radius"). The average turning radius is 17.5′ and 95% fall within 20.3′, so 22′ is an acceptable choice.

## DIAGONAL vs RIGHT-ANGLE PARKING

Having decided upon driver operation over attendant operation, the next consideration is diagonal vs right-angle parking. Without attendant parking it may seem wise to make a driver's entrance into and exit from a space easier by placing the space at an angle $<90°$ off the aisle. In this section, we propose that this approach—referred to here as diagonal parking—does not make the most efficient use of the 20,000 ft$^2$ available in the given lot.

*Theorem*

Given an access lane of any width $R$, and a space size defined by $L$ and $W$ (used to denote length and width, respectively), the total lot area required for one space of angle $\theta$ (as pictured in the parallelogram dghl in Fig. 1) is given by

$$A(\theta) = W[L + R\text{cosec}(\theta) + (2W - R)\cot(\theta)].$$

*Proof.* Let $R$ be greater than or equal to the maximum turning radius required so that entry into a right-angle space is possible from the access lane. Consider an automobile traveling along the access lane (from left to right in the figure) such that the maximum extension of the passenger side of the vehicle lies directly on the line labeled as *aisle bottom*. Furthermore, say this automobile initiates a turn with radius $R$ at point d in an attempt to enter the right-angle space shown. With the passenger side following the arc as pictured, the automobile will clearly be able to enter the right-angle space. Then certainly the automobile would be able to enter any diagonal space of degree $\theta$, where $\theta \in (0, 90)$. Say the automobile attempts to enter such a space with $\theta$ given. With $\theta < 90$ the turning automobile will reach a point (before *aisle top*) at which turning is no longer necessary—the car can continue on a straight line into the space. This point is located where the extended right side of the diagonal space forms a tangent to the turning arc. Because turning farther is not necessary, the access lane width $R$ can be decreased (and space more efficiently utilized) by the distance $D$ expressed as a function of $\theta$. To calculate this distance $D$, let $L$ and $W$ be given as described above. Because opposite angles formed by the intersection of two line segments are congruent, we see that $\angle bce = \theta$. Also, because the radius drawn from point a through point e intersects $\overline{lh}$ at the point of tangency, and $\overline{dg}$ is parallel to $\overline{lh}$, $\angle aec$ must equal 90°. Then in $\triangle ace$, the measure of $\angle cae = 90° - \theta$. Finally, we have that the length of the radius from point e to $\overline{lh} = W$ so that the length of $\overline{ae} = R - W$, and the length of $\overline{be}$, i.e. $D$, is therefore given by

$$\begin{aligned} D &= (R - W)\sin(90 - \theta) \\ &= (R - W)\cos(\theta). \end{aligned}$$

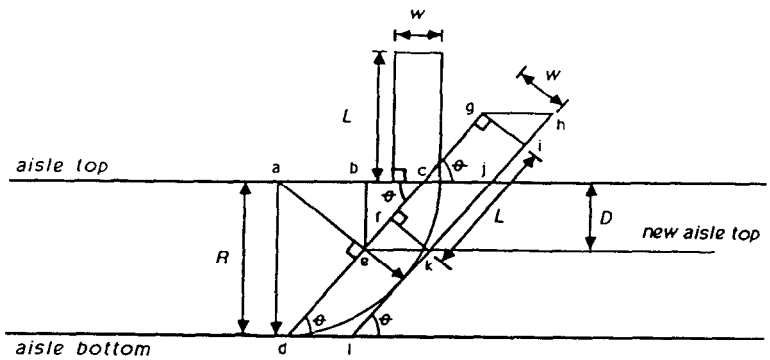



Fig. 1

So a saving of $D$ units is achieved if the aisle width is dependent on the angle $\theta$. To assess the area required for both the space and the aisle necessary for entrance into said space, consider the area of the parallelogram dghl as a function of $\theta$. Clearly, the "height" of the parallelogram dghl = $W$. To find its width, we need the sum of the lengths of $\overline{lj}$, $\overline{ji}$, $\overline{ih}$. Simple trigonometry yields $\sin(\theta) = R/\overline{lj}$ so that $\overline{lj} = R/\sin(\theta)$. Because $\angle$ghi is opposite $\angle$gcj in the parallelogram cghj, and $\angle$gcj = $\theta$, we have $\angle$ghi = $\theta$ and $\tan(\theta) = W/\text{ih}$ so that $\text{ih} = W/\tan(\theta)$. Finally, observe that the length of $\overline{ji} = L - \overline{kj}$, and that the angle formed by the intersection of $\overline{jk}$ with *new aisle top* is $\theta$. Then $\sin(\theta) = D/\overline{kj}$ so that $\overline{kj} = D/\sin(\theta)$, and the length of $\overline{ji} = L - \overline{kj} = L - (D/\sin(\theta))$. Then the total area, $A$, as a function of $\theta$ is given by

$$A(\theta) = W\left[\frac{R}{\sin(\theta)} + L - \frac{D}{\sin(\theta)} + \frac{W}{\tan(\theta)}\right];$$

substituting $D = (R - W)\cos(\theta)$,

$$A(\theta) = W[L + R\operatorname{cosec}(\theta) + (2W - R)\cot(\theta)].$$ ■

Now we want to minimize $A$. Taking the derivative yields

$$A'(\theta) = W\left\{R - \frac{\cos(\theta)}{\sin^2(\theta)} + (2W - R)\left[-\frac{1}{\sin^2(\theta)}\right]\right\},$$

setting it equal to zero and solving for $\theta$ yields

$$\theta = \arccos\left(\frac{R - 2W}{R}\right).$$

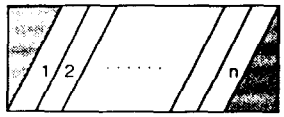
Using our dimensions of $R = 22'$ and $W = 8'$, this formula tells us that to minimize the area for a space, $\theta = 74.2°$ should be chosen. So,

$$A(74.2°) = 313.3$$

and

$$A(90°) = 320.0,$$

giving a saving of 6.7 ft$^2$/space. However, this does not take into consideration the amount of unusable space at the end of each angled row (see Fig. 2). With $\theta = 74.2°$ this amounts to 402.4 ft$^2$ wasted. At a saving of only 6.7 ft$^2$/space over right-angle parking, this means there would have to be over 60 parking spaces in a row before any space is really saved. In a 100′ × 200′ lot with 8′ parking spaces, the maximum number of spaces in a row is 25. So, right-angle parking is more efficient than our supposed minimum, in the small lot. For this reason, we chose to allow only right-angle parking in the lot.




Unusable area

Fig. 2

## THE APPROACH

We have already shown that for a single parking space to make the most efficient use of area in a 100′ × 200′ lot, it should be positioned perpendicular to the aisle. Now, to make that even more efficient, put another perpendicular parking space on the other side of the aisle so that they share the aisle area. In effect, this reduces the aisle area per parking space by a factor of 2. Since we have disallowed "doubling up" this is the most efficient way to position two parking spaces. We have called these two spaces and the aisle between them our "tile":

| SPACE | AISLE | SPACE |
|---|---|---|

Each tile in this case is 8′ × 58′. Now, if you want to get as many spaces as possible into a parking lot, find an arrangement of tiles in the lot that will put as many tiles as possible in the lot while still leaving aisles to make all spaces easily accessible, allowing entrances on two adjacent sides. The question then arises, how do we find these arrangements? Our first approach was to design an algorithm which would construct possible tiling arrangements. We first noticed that given a tile, there are 12 basic ways to position a tile adjacent to it (see Fig. 3). The program, given a tile parallel to a side of the rectangular lot, will systematically attempt to place a subsequent tile in each of these 12 positions, allowing only tiles that will stay within the boundaries of the parking lot without covering any previously positioned tiles. The process starts over using this new tile, and continues until no more tiles can be placed. If the number of tiles placed exceeds a certain minimum, the layout is sent to an output device. At this point, the last tile placed is "erased" from the layout and the process continues recursively from the parent tile. This process continues until all 12 possible positionings have been attempted. This obviously leads to a 12-way tree which even in our relatively small parking lot with an 8′ × 58′ tile has a height of over 40 in the worst case. This being impractical and, therefore, undesirable, leads to a modification of this approach. Instead of positioning individual tiles, we position blocks of tiles. This is accomplished changing a few constants within the program.

To choose the block size, we first noticed that a block of 5 tiles should be the maximum, since in our 100′ × 200′ lot, a block of 6 tiles is clearly too limiting. Running the program with a block size of 5 tiles works well. The program runs in a reasonable amount of time, < 15 min, and produces a feasible layout for the parking lot. Running the program with blocks of 4 or 3 tiles fails to produce any superior designs, to run it with blocks of 2 tiles would cause the tree to become too large. Figure 4 shows a layout, produced using blocks of 5 tiles, consisting of 80 parking spaces. From this layout, the owner can customize spaces to accommodate handicapped patrons or possibly move a few around to allow for easier access.

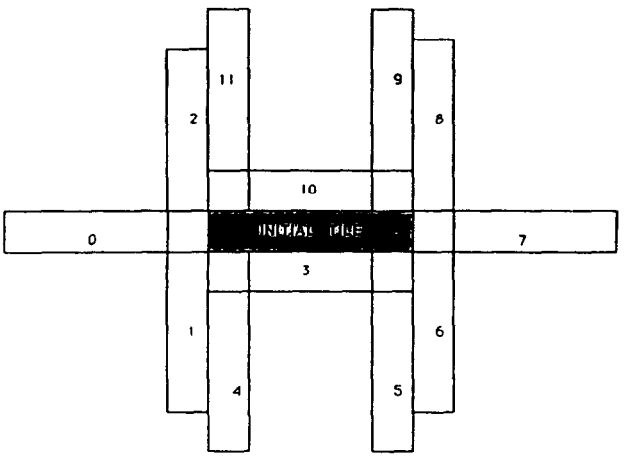



Fig. 3

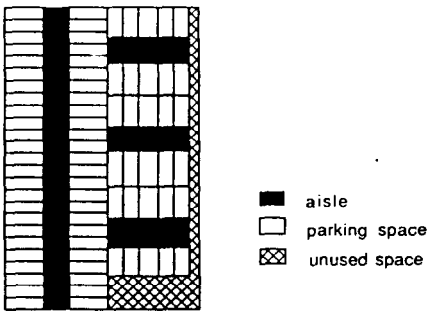



Fig. 4

## CRITIQUE

The algorithm is quite flexible. It can easily be adapted to any size rectangular lot using any size block of any size tile. One should be careful, however, in choice of block size. Since the algorithm is $O(n^x)$, where $x$ is the maximum number of tiles that can be placed in the lot, a small block size should not be chosen. It is unfortunate that using smaller block sizes will cause such an undesirable increase in run time, but using blocks with more tiles produces layouts that are more aesthetically pleasing and easier to use because each aisle is longer. Also, too large a block size will severely limit the possibilities. Unfortunately, our algorithm does not find all the possibilities and it sometimes leaves holes where a tile would fit. One reason for this defect is because it can only build in one direction at a time from a given tile. This is tolerated because adapting the algorithm to find all possibilities and fill all holes would significantly increase the run time without producing significantly better results and because viewer inspection of a layout can easily detect such problems and correct them.

Another reason that the algorithm does not produce all possible layouts is that we allow only 12 different placements of the next tile. This number could be increased, but this would increase run time unnecessarily. Our 12 choices are the only transformations which align at least one vertex and are adjacent to the original tile.

Another problem with the algorithm is that it does not check if the layout produced is feasible. However, again viewer inspection of the layouts produced can easily eliminate those that have inaccessible spaces and keep the ones that are feasible. This also suggests the use of larger blocks to cut down on the number of layouts that must be checked.

The algorithm also has an inherent drawback in that only rectangular lots can be filled with it and only by perpendicular spaces. The owner of a large rectangular lot which would make the use of angled parking feasible would probably not want to use this algorithm. Similarly, the owner of a nonrectangular lot would not get very efficient layouts.

## ADDITIONAL CONSIDERATIONS

As yet, no consideration of the geographical location of the lot has been made. This is one of the few topics which we will consider in closing our discussion. The information presented here is provided not in the form of technical analysis, but rather in an informal or "further discussion" context.

First, consider the geographical location of our parking lot. The problem states that the lot is located in a New England town. Then, some provision for snow removal should be brought into the design of the lot. Without proper snow and ice removal, the parking lines will not be visible, and all the work put into the design of the layout will have been in vain. Perhaps then, some other demarcation of spaces should be used, such as concrete bumpers. With this scheme, however, snow removal would be more difficult because the plowing vehicles would be required to maneuver around these barriers.

A consideration should also be made of the hours of operation of the lot. If it will be vacant during the night, then snow removal will be easier than if the plows are required to work around vehicles parked overnight. The frequency of traffic in the lot during daytime hours is also a factor. Is traffic regular, or is traffic related to work hours so that the lot is very busy for entrance during mid-morning, and very busy for exits in the early evening? If the traffic is not regularly dispersed, but rather congested at certain times, then another goal of the lot design should concern the proximity of the entrances and exits in relation to the intersection. Also, is the traffic through the intersection congested or light? Is this traffic regulated by signs or signals? If the traffic is light or regulated by signals, then entering or exiting the lot near the intersection will be easier and openings in the lot can be positioned nearer the crossing. If not, then an effort should be made to position these entrances and exits as far from the intersection as possible.

Finally, how is revenue to be generated by the lot owner? If toll-booths or guard houses are to be used, then space for these must be allocated in the lot design. Also, where should these parking fees be collected: upon entrance or exit? In a busy lot, fee collecting upon entrance would greatly slow the rate of traffic entering the lot and thus cause congestion in the street from which entrances

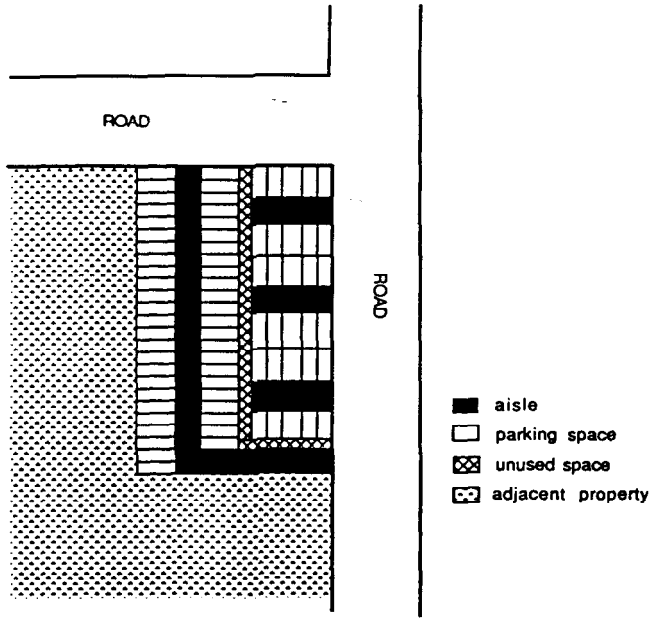



Fig. 5

are made. However, lot users may not wish to wait in a long exit line where fees are collected if many are leaving the lot at a certain time. With this in mind, the lot owner may wish to sell parking stickers for automobiles and allow lot use by permit only. But this would entail budgeting wages for employees to periodically patrol the lot, checking if all users have purchased a permit. Perhaps revenue in the form of parking fees is not feasible as in the case for establishments such as supermarkets or retail stores. In these instances, the lot owner is often concerned not with lot revenue but rather with things such as the upkeep or aesthetics of the parking lot.

## CONCLUSION

Our program produced a layout that fit 80 spaces into the 100′ × 200′ lot. This was the most the algorithm ever produced for this lot. This output should not be looked at as necessariiy the best layout. A little common sense should be used in deciding upon a design. Evaluate the layout and possibly modify it based on some of the "additional considerations" previously mentioned. Look at the dead space to see if a parking space can be widened or possibly remove 1 or 2 spaces to form a new aisle for a more convenient entrance or exit. In conclusion, we present one possible modification of the computer designed layout (see Fig. 5): two spaces have been removed to form another entrance, and the dead space was moved from the edge to the middle of the lot to allow for a curb or guardrail along the back of the row.

## REFERENCE

1. *Road & Track Magazine, Buyer's Guide '87*. CBS Magazines, New York (1986).

*See Overleaf for the Appendix*

## APPENDIX

```
PROGRAM Parking_Lot (input,output,datafile,parking);

(*************************************************************************
   Program using a block of five tiles
*************************************************************************)

CONST
   Width = 49;
   Height = 99;
   CutOff = 7;
   Block_Width = 20;
   Space_Length = 9;
   Aisle_Width = 11;

TYPE
   Coord = RECORD
      x,
      y: INTEGER;
      END;

   TreeNodePtr = ^TreeNode;

   ChildType = ARRAY [0..11] OF TreeNodePtr;

   TreeNode = RECORD
      LT,
      RT,
      RB,
      LB: Coord;
      Tile_No: INTEGER;
      CHILDREN: ChildType;
      END;

   LotType = PACKED ARRAY [0..Width, 0..Height] OF CHAR;

VAR
   Lot: LotType;
   Root: TreeNodePtr;
   Block_Length,i,j: INTEGER;
   parking: TEXT;
   datafile: TEXT;
   LB,RT: Coord;
   Letter: CHAR;
   a: integer;

FUNCTION Fill (RightT, LeftB: Coord; Tile: INTEGER): TreeNodePtr; forward;

PROCEDURE Add_A_Tile (Current: TreeNodePtr);

VAR
   i,j: INTEGER;
   LeftB,
   RightT: Coord;
   None: BOOLEAN;

BEGIN
   WITH Current^ DO
      IF (RB.x - LB.x) > (BLOCK_WIDTH + 1) THEN
         FOR i := 0 TO 11 DO
            BEGIN
               CASE i OF
                  0 :BEGIN
                        LeftB.x := LB.x - Block_Length;
                        LeftB.y := LB.y;
                        RightT.x := LB.x - 1;
                        RightT.y := LT.y;
                     END;
                  1 :BEGIN
                        LeftB.x := LT.x - (Block_Width);
                        LeftB.y := LT.y - (Block_Length - 1);
                        RightT.x := LT.x - 1;
                        RightT.y := LT.y;
                     END;
```

```
            2 :BEGIN
                 LeftB.x := LB.x - (Block_Width);
                 LeftB.y := LB.y;
                 RightT.x := LB.x - 1;
                 RightT.y := LB.y + (Block_Length - 1);
               END;
            3 :BEGIN
                 LeftB.x := LB.x;
                 LeftB.y := LB.y - (Block_Width);
                 RightT.x := RB.x;
                 RightT.y := RB.y - 1;
               END;
            4 :BEGIN
                 LeftB.x := LB.x;
                 LeftB.y := LB.y - Block_Length;
                 RightT.x := LB.x + (Block_Width - 1);
                 RightT.y := LB.y - 1;
               END;
            5 :BEGIN
                 LeftB.x := RB.x - (Block_Width - 1);
                 LeftB.y := RB.y - Block_Length;
                 RightT.x := RB.x;
                 RightT.y := RB.y - 1;
               END;
            6 :BEGIN
                 LeftB.x := RB.x + 1;
                 LeftB.y := Rt.y - (Block_Length - 1);
                 RightT.x := RT.x + (Block_Width);
                 RightT.y := RT.y;
               END;
            7 :BEGIN
                 LeftB.x := RB.x + 1;
                 LeftB.y := RB.y;
                 RightT.x := RT.x + Block_Length;
                 RightT.y := RT.y;
               END;
            8 :BEGIN
                 LeftB.x := RB.x + 1;
                 LeftB.y := RB.y;
                 RightT.x := RB.x + (Block_Width);
                 RightT.y := RB.y + (Block_Length - 1);
               END;
            9 :BEGIN
                 LeftB.x := RB.x - (Block_Width - 1);
                 LeftB.y := RT.y + 1;
                 RightT.x := RB.x;
                 RightT.y := RT.y + Block_Length;
               END;
           10 :BEGIN
                 LeftB.x := LB.x;
                 LeftB.y := LT.y + 1;
                 RightT.x := RT.x;
                 RightT.y := RT.y + (Block_Width);
               END;
           11 :BEGIN
                 LeftB.x := LB.x;
                 LeftB.y := LT.y + 1;
                 RightT.x := LB.x + (Block_Width - 1);
                 RightT.y := RT.y + Block_Length;
               END;
         END; (* case *)
         Current^.Children[i] := Fill(RightT, LeftB, Current^.Tile_No);
      END  (* for *)
ELSE
   FOR i := 0 TO 11 DO
      BEGIN
         CASE i OF
            0 :BEGIN
                 LeftB.x := LB.x;
                 LeftB.y := LT.y + 1;
                 RightT.x := RT.x;
                 RightT.y := RT.y + Block_Length;
               END;
```

```
                1 :BEGIN
                      LeftB.x := RT.x - (Block_Length - 1);
                      LeftB.y := RT.y + 1;
                      RightT.x := RT.x;
                      RightT.y := RT.y + (Block_Width);
                  END;
                2 :BEGIN
                      LeftB.x := LB.x;
                      LeftB.y := LT.y + 1;
                      RightT.x := LB.x + (Block_Length - 1);
                      RightT.y := RT.y + (Block_Width);
                  END;
                3 :BEGIN
                      LeftB.x := LB.x - (Block_Width);
                      LeftB.y := LB.y;
                      RightT.x := LB.x - 1;
                      RightT.y := RT.y;
                  END;
               4 :BEGIN
                      LeftB.x := LB.x - Block_Length;
                      LeftB.y := LT.y - (Block_Width - 1);
                      RightT.x := LB.x - 1;
                      RightT.y := RT.y;
                  END;
                5 :BEGIN
                      LeftB.x := LB.x - Block_Length;
                      LeftB.y := LB.y;
                      RightT.x := LB.x - 1;
                      RightT.y := LB.y + (Block_Width - 1);
                  END;
                6 :BEGIN
                      LeftB.x := RB.x - (Block_Length - 1);
                      LeftB.y := RB.y - (Block_Width);
                      RightT.x := RB.x;
                      RightT.y := RB.y - 1;
                  END;
                7 :BEGIN
                      LeftB.x := LB.x;
                      LeftB.y := LB.y - Block_Length;
                      RightT.x := RB.x;
                      RightT.y := RB.y - 1;
                  END;
                8 :BEGIN
                      LeftB.x := LB.x;
                      Leftb.y := LB.y - (Block_Width);
                      RightT.x := LB.x + (Block_Length - 1);
                      RightT.y := LB.y - 1;
                  END;
                9 :BEGIN
                      LeftB.x := RB.x + 1;
                      LeftB.y := RB.y;
                      RightT.x := RB.x + Block_Length;
                      RightT.y := RB.y + (Block_Width - 1);
                  END;
               10 :BEGIN
                      LeftB.x := RB.x + 1;
                      LeftB.y := RB.y;
                      RightT.x := RB.x + (Block_Width);
                      RightT.y := RT.y;
                  END;
               11 :BEGIN
                      LeftB.x := RT.x + 1;
                      LeftB.y := RT.y - (Block_Width - 1);
                      RightT.x := RT.x + Block_Length;
                      RightT.y := RT.y;
                  END;
              END; (* case *)
            Current^.Children[i] := Fill(RightT, LeftB, Current^.Tile_No);
          END; (* for *)

None := true;
FOR i := 0 TO 11 DO
   IF Current^.Children[i] <> nil THEN
      None := false;
IF None THEN
```

```
      BEGIN
IF Current^.Tile_NO > Cutoff THEN
         IF Current^.Tile_No > CutOff THEN
            BEGIN
               FOR j := Height DOWNTO 0 DO
                  BEGIN
                     FOR i := 0 TO Width DO
                        BEGIN
                           write(parking,Lot[i,j]);
                        END;
                     writeln(parking);
                  END;
               writeln(parking);
               writeln(parking);
            END; (* if > CutOff *)
      END;
   FOR i := Current^.LB.x TO Current^.RT.x DO
      FOR j := Current^.LB.y TO Current^.RT.y DO
         Lot [i,j] := '+';
   dispose (Current);
END; (* Add_A_Tile *)

FUNCTION Fill;

VAR
   Temp: TreeNodePtr;
   Filled: BOOLEAN;
   i,j: INTEGER;

BEGIN
   Filled := false;
   IF (LeftB.x < 0) OR (RightT.x > Width) OR
      (LeftB.y < 0) OR (RightT.y > Height) THEN
         Filled := true;
   i := LeftB.x;
   WHILE (i <= RightT.x) AND (NOT Filled) DO
      BEGIN
         j := LeftB.y;
         WHILE (j <= RightT.y) AND (NOT Filled) DO
            BEGIN
               IF Lot [i,j] <> '+' THEN

                  Filled := true;
               j := j + 1;
            END;
         i := i + 1;
      END;
   IF NOT Filled THEN
      BEGIN
         Letter := chr(Tile + 96);
         IF RightT.x - LeftB.x > 21 THEN
            BEGIN
               FOR i := 0 TO (Space_length - 1) DO
                  FOR j := 0 TO (Block_Width - 1) DO
                     Lot[LeftB.x+i, LeftB.y+j] := Letter;
               FOR i := (Space_Length) TO (Space_Length + Aisle_Width - 1) DO
                  FOR j := 0 TO (Block_Width - 1) DO
                     Lot[LeftB.x+i, LeftB.y+j] := ' ';
               FOR i := (Space_Length + Aisle_Width) TO (Block_Length - 1) DO
                  FOR  j := 0 TO (Block_Width - 1) DO
                     Lot[LeftB.x+i, LeftB.y+j] := Letter;
            END
         ELSE
            BEGIN
               FOR i := 0 TO (Block_Width - 1) DO
                  FOR j := 0 TO (Space_length - 1) DO
                     Lot[LeftB.x+i, LeftB.y+j] := Letter;
               FOR i := 0 TO (Block_Width - 1) DO
                  FOR j := (Space_Length) TO (Space_Length + Aisle_Width - 1) DO
                     Lot[LeftB.x+i, LeftB.y+j] := ' ';
               FOR i := 0 TO (Block_Width - 1) DO
                  FOR j := (Space_Length + Aisle_Width) TO (Block_Length - 1) DO
                     Lot[LeftB.x+i, LeftB.y+j] := Letter;
            END;
         new (Temp);
         WITH Temp^ DO
            BEGIN
               LT.x := LeftB.x;
               LT.y := RightT.y;
```

```
                RT := RightT;
                RB.x := RightT.x;
                RB.y := LeftB.y;
                LB := LeftB;
                Tile_No := Tile + 1;
             END;
          Fill := Temp;
          Add_A_Tile(Temp);
       END (* if *)
    ELSE (* filled *)
       Fill := nil;
END; (* Fill *)

BEGIN (* MAIN *)
   Block_Length := 2 * Space_Length + Aisle_Width;
   a := 1;
   reset(datafile);
   rewrite(parking);
   REPEAT
      FOR i := 0 TO Width DO
         FOR j := 0 TO Height DO
            Lot[i,j] := '+';

      readln(datafile,LB.x, LB.y);
      readln(datafile,RT.x, RT.y);
      Root := Fill (RT, LB, 0);
      writeln(parking);
      writeln(parking,'NEXT RUN');
      writeln(parking);









      writeln;
   UNTIL a = 2;
END.   (* MAIN *)
```