Optimization of Car Park Design

Bachelor Thesis, Technical University of Denmark

Supervisor: Poul G. Hjorth, DTU Compute

Oscar Juul Andersen	s194316
Marcus Førby Petersen	s194343

May 31st, 2022

Oscar Juul Andersen

Marcus Førby Petersen

Abstract

In this paper, we aim to solve the problem of fitting the maximum amount of parking slots within a given car park. To do this, we first take a theoretical approach through operations research, in which we find that it is possible to model a simple version of the problem and solve it to optimality. However, we also find that the computational complexity for the theoretical approach increases exponentially with respect to either the number of parking slots or the number of nodes in a grid, due to binary variables. Therefore, we take an alternative approach to solve this problem.

We try using heuristics, where we place rows of parking slots in different ways. Together with a path out algorithm we developed, these heuristics can quickly give solutions, but since they are heuristics it isn't guaranteed that the solutions are optimal. We find that our heuristics work well on larger and more regular shapes of car parks, but are very bad at modeling smaller and more intricate shapes. We use our algorithms on three different existing car parks, where the algorithms are able to increase the number of parking slots in two out of the three car parks. Here it will also be illustrated, that the results of these algorithms should not be used without further human modeling.

The purpose of this paper is to give the reader an understanding of the complexity of this problem, and enlighten the reader with alternative approaches since this is a problem to which no one yet has found an optimal solution.

Contents

1 Introduction to the car park design problem

As cities grow larger and more densely populated, the need for parking slots for the citizens to park their cars grows too. However, it is very expensive to buy land inside the city, and also expensive to build large parking houses. Therefore it is very valuable to figure out a method to optimize the current car parks, in which there might be wasted space. This is the reason that ARUP, which is an architect firm, approached the European Study Group with Industry (ESGI), see [12], to solve this problem. This study group came up with some interesting preliminary results but found no algorithm or technique to solve the problem. Sidsel Klarskov Sørensen also explored this subject, see [13], in which she found a method to fill a rectangle with rows of parking slots at an angle she calculated as optimal. She also found a way to find the largest rectangle within an arbitrary convex polygon, which could then be filled, and then the algorithm could be called recursively on the unused area. While this generates a feasible solution to the problem, we believe that it is far from ideal to section the car park like this, and furthermore, the method is unable to handle non-intersecting polygons. Thus this is still an unsolved problem that we will research.

In this paper, we focus on different methods to optimize a car park design. First, we will introduce the reader to some basic geometric considerations of a parking slot that will be used throughout this paper. Then, we introduce the reader to basic operations research theory, which we afterward use to implement several linear programming models. We will introduce the reader to different greedy heuristics, which very quickly can solve the problem, but won't necessarily solve the problem to optimality. The reader will see that these heuristics generates solutions, where some parking slots can be confined. Therefore, we developed another algorithm named path out in which we remove parking slots in a certain order such that we end up with a solution, where the fewest parking slots are removed and where it's possible to leave the car park from all parking slots. Finally, we will test these algorithms on three different car parks from the real world.

The purpose of this paper is to research the complexity of this problem and enlighten the reader with approaches and techniques to obtain good solutions.

The code for this paper is written in MATLAB vR2021a and Julia v1.6.2. In Julia, we use the CBC MILP solver v2.10.5 and the GLPK solver v0.14.12. The code has been uploaded to GitHub, see [3], where everyone can use it for free.

2 Basic geometric considerations of a parking slot

Before we can start solving the actual problem, we want to introduce the reader to the basic rules we follow to place the parking slots. First, we will introduce our primary source on this subject which is a pamphlet written by The Danish Road Directorate, see [15], which gives guidelines on how to design a car park.

2.1 Car parks and parking slots

We define the car park as an arbitrary polygon, which can be more or less complex throughout this paper. It could be a rectangle, a convex polygon, or a non-self intersecting polygon.

Throughout this paper, we want to use variables to indicate which dimension of the parking slot we are referencing. We, therefore, introduce the measurement and definitions seen in figure 1.

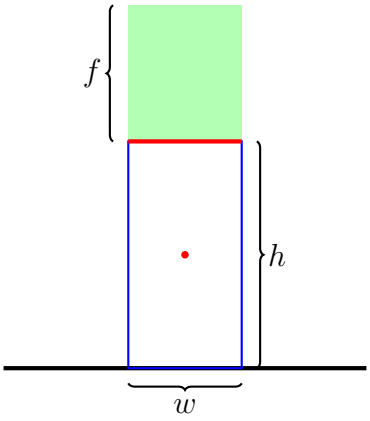


Figure 1: A figure illustrating the variable and name definitions we will use throughout this paper. w and h are the width and height of the parking slot, respectively. f is the height of the free space needed in front of the parking slot. The red dot is the center of a parking slot, and the red edge is the edge on which we drive in and out of the parking slot.

We will be consistent and use these variable names in figure 1 for the entirety of this paper. The measurements given in [15] is that the free space, f, is dependent on the angle of the parking slot. The angle of the parking slot will be defined later, see figure 2. The height of the parking slot is $h = 5$ meters and the width of the parking slot is $w = 2.5$ meters. However, we measured some existing car parks, in which we found that the width of those parking slots is approximately 2.4 meters, and as we would like to compare our solutions with existing solutions, we use $w = 2.4$ meters.

As mentioned, the length of the free space depends on the angle at which the parking slot is placed. The higher the angle, the less space is needed to get out. The exact measurements of the free space given by The Danish Road Directorate are listed in table 1.

Angle	0°	15°	30°	45°	60°
Free space needed	7m	5.5m	4.2m	3.2m	2.5m

Table 1: A table listing the free space needed at different angles of a parking slot. This is an official list from The Danish Road Directorate, see [15].

These are just five possible angles, but in reality, there exist infinitely many angles at which we can place the slots. Therefore we decide to interpolate between the points, such that we have a continuous function, $L(\alpha)$, which indicates the length of the free space needed, at an angle α.

Now we want to consider the angle of a parking slot, and how much free space we will need in front of a parking slot at a given angle. Before we do this, however, we want to consider the lowest possible angle we can place a parking slot in, while the parking slots in a row still touch each other. This situation is drawn on figure 2. This figure also illustrates

how we measure the rotation of parking slots for the rest of the paper. Hence, a rotation of $\alpha = 0$ degrees is equivalent to a vertically placed parking slot.

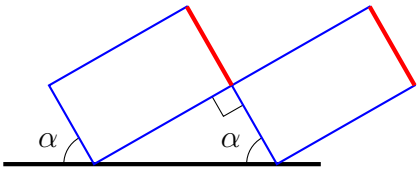


Figure 2: A figure illustrating the highest angle at which parking slots can be placed, while they still touching each other. Notice also that in the rest of the paper we will be measuring the rotation of a parking slot by the angle, α, drawn in this figure. Hence, a rotation of $\alpha = 0$ degrees is equivalent to a vertically placed parking slot.

Using figure 2 we can calculate the greatest possible angle, α_{max}, by the following:

$$\alpha_{max} = \arctan\left(\frac{h}{w}\right)$$

To be able to tell how well our different solutions perform relative to each other in different forms of car parks, we need to introduce a measure, other than simply the number of parking slots. This measure is the density of the car park, which can be calculated as the following:

$$\rho = \frac{\text{area occupied by parking slots}}{\text{area of the car park}} = \frac{h \cdot w \cdot \#\text{ parking slots}}{\text{area of the car park}}$$

How to calculate the area of the car park is further explained in the appendix, section 10.3.2. This section explains how to calculate the area of an arbitrary non-intersecting polygon.

2.2 The infinite corridor & the optimal angle

We want to find the optimal angle, which we define as the angle that allows us to find the highest density of parking slots. Since the optimal angle may depend on the polygon the slots are placed in, we will consider a car park of infinite size. We don't know what the optimal pattern of the parking slots in such a car park would be, but a good solution would be to place the slots in identical rows.

Since we want to place the parking slots in identical rows, we only need to look at one row and find the corresponding optimal angle. This problem is equivalent to placing parking slots in the infinite corridor, which is illustrated in figure 3. Here the corridor expands infinitely to the left and right with parking slots placed in the same pattern. Here, the height of the corridor depends on the angle, α, as it determines the vertical height of a parking slot, but also the height of the free space, $f = L(\alpha)$, which was described in section 2.1. As all the parking slots lie side-by-side, this corridor can be seen as infinitely many parallelograms marked with green lying side by side. This means that the density of the entire corridor must be equal to the density of one such green parallelogram.
Using trigonometric observations, we see that the area of the parallelogram is given by the following:

$$G(\alpha) = (\ell + h + k) \cdot w = \left(\frac{L(\alpha)}{\cos(\alpha)} + h + \tan(\alpha) \cdot w\right) \cdot w$$

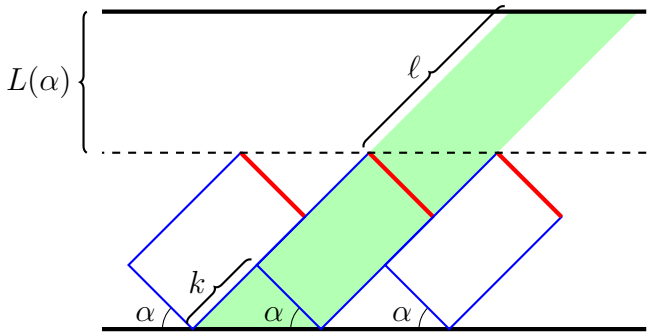


Figure 3: A figure illustrating the infinite corridor for a given α consisting of parking slots and the free space needed in front of it. It can be considered as infinitely many copies of the green parallelogram lying next to each other.

Using this expression of the green area, we can calculate the density of the parallelogram by the following:

$$\rho(\alpha) = \frac{h \cdot w}{G(\alpha)} = \frac{h \cdot w}{\left(\frac{L(\alpha)}{\cos(\alpha)} + h + \tan(\alpha) \cdot w\right) \cdot w} = \frac{h}{\frac{L(\alpha)}{\cos(\alpha)} + h + \tan(\alpha) \cdot w}$$

Here we see that maximizing the density corresponds to minimizing the green area in figure 3. To do this we calculate the first derivative of $G(\alpha)$:

$$\frac{G'(\alpha)}{w} = \left(\frac{a \cdot \alpha + b}{\cos(\alpha)} + h + \tan(\alpha) \cdot w\right)' = \frac{a}{\cos(\alpha)} + \frac{(a \cdot \alpha + b)\sin(\alpha)}{\cos^2(\alpha)} + \left(1 + \tan^2(\alpha)\right) \cdot w$$

Here a and b are parameters in the interpolation such that $L(\alpha) = a \cdot \alpha + b$. Usually, we would simply set this expression equal to zero, and solve for α such that we find all local extrema, and check which is the lowest. Here we see that it is not simple to solve this analytically, so instead, we solve it numerically using Newton's method. To use this method, we need the double derivative, which can be calculated as the following:

$$\frac{G''(\alpha)}{w} = \frac{2a\sin(\alpha)}{\cos^2(\alpha)} + \frac{2(a \cdot \alpha + b)\sin^2(\alpha)}{\cos^3(\alpha)} + \frac{a \cdot \alpha + b}{\cos(\alpha)} + 2\tan(\alpha)(1 + \tan^2(\alpha)) \cdot w$$

Implementing Newton's algorithm in MATLAB, we get that the optimal angle equals 26.99 degrees. To calculate the density of this infinite corridor (which is also the density of the entire infinite car park), we use a slightly altered calculation of ρ, see appendix, section 10.3.1. This calculation yields a density of $\rho = 0.616$.

3 Car parks as an operations research problem

It is easy to get a design for a car park, which fulfills all the requirements from section 2, but it is hard to find an optimal solution. In this section, we wish to investigate if we can solve our problem to optimality, by applying methods from operations research (OR). If we can transform our problem to a set of linear constraints, we have a guarantee that we can find an optimal solution using the simplex algorithm [8].

3.1 Operations research Theory

In this section, we introduce the reader to the OR theory, which we later will use to determine whether it is possible to solve our problem to optimality.

3.1.1 Linear programming

Linear programming (LP) is the discipline of optimizing (minimizing or maximizing) a linear combination of variables (the objective function), which are subject to several linear constraints. This can be done in many ways, but the most used method is probably the simplex algorithm. The reason why LP is so important is that we can model a lot of different optimization problems. For example, we can model a work schedule that covers all shifts, with minimal labor cost, see [4], or we can model the shortest path between a set of machines starting and ending on the same machine, see [7] (the Travelling Salesman Problem). These are just two examples, but the potential uses are limitless. For a more detailed description of LPs, see [8].

3.1.2 Branch and bound

Since LP is based upon continuous variables, we run into a problem when we want to model variables that must be integer or binary. This is a problem, because the simplex algorithm is not guaranteed to return integer solutions, and, in fact, often it won't return integer solutions. To ensure that we can solve the problem to optimality, we make use of the branch and bound algorithm.

The branch and bound algorithm work by repeatedly solving the LP-relaxation, which is the original integer or mixed-integer program, where we allow all variables to be continuous. If the solution to this problem contains a non-integer variable that is supposed to be an integer variable, we need to fix the problem. Consider that the variable x_i needs to be an integer, but is a fraction, s. Then we solve two new problems, namely one where we add the following constraint:

$$x_i \leq \lfloor s \rfloor$$

and one where we add the following constraint:

$$x_i \geq \lceil s \rceil$$

Then we solve the resulting problems and continue in this way. This example is illustrated on figure 4. Here we have solved the subsequent problem S_2 and found that the integer variable x_j is equal to t that is not an integer. We notice that each time we solve an LP-problem, we get an upper bound for the objective function in that branch. The algorithm ends when all branches have been closed. There are three ways to close a branch, namely if the problem is infeasible, if the problem yields an integer solution (a feasible solution), or if we have an integer solution in another branch with an objective value at least as good as the upper bound on the objective value of the mixed-integer program in this branch.

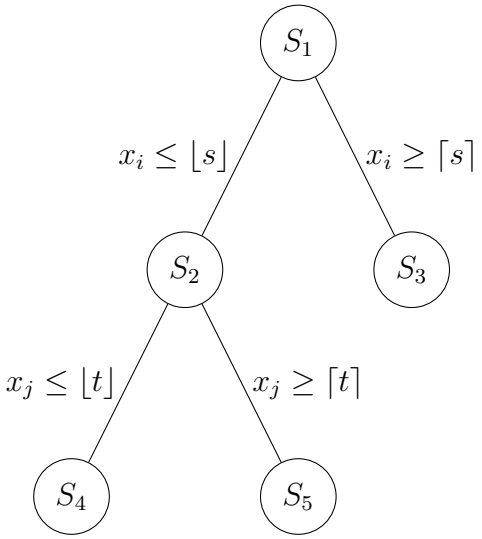


Figure 4: A figure illustrating an example of a branch and bound tree. Each node corresponds to an LP-problem. The constraints on each edge illustrate that all problems below have added that constraint. In node S_1 we branch on the x_i variable, which creates two new branches. In node S_2 we branch on the x_j variable, which again creates two new branches.

3.1.3 Running times

The running time of a single iteration of the simplex algorithm is in the worst-case exponential in the number of constraints. However, it is still quite a fast method and usually it solves problems in polynomial time, see [14]. The most time-consuming element when working with mixed-binary programs is the branch-and-bound algorithm. The worst-case for the branch and bound algorithm is that we have to consider all possible combinations of the binary variable, and run the simplex algorithm for each of these combinations. This means that in the worst-case scenario we run 2^n iterations of the simplex algorithm, where n is the total number of binary variables.

3.2 Operations research implementation

Now we want to see how we can solve the problem to optimality using OR. We have two approaches to solving the problem with OR, which results in two different LP-problems.

3.2.1 Discrete x- and y-variables

The first way to model this using OR is to discretize the problem. This is done by looking at the entire domain and placing many equidistant nodes. Each node represents the center of a parking slot, with a certain orientation. We place the nodes in a square pattern covering the entire polygon, and then remove all nodes, for which the corresponding parking slot cannot be placed. An example of this is shown in figure 5.

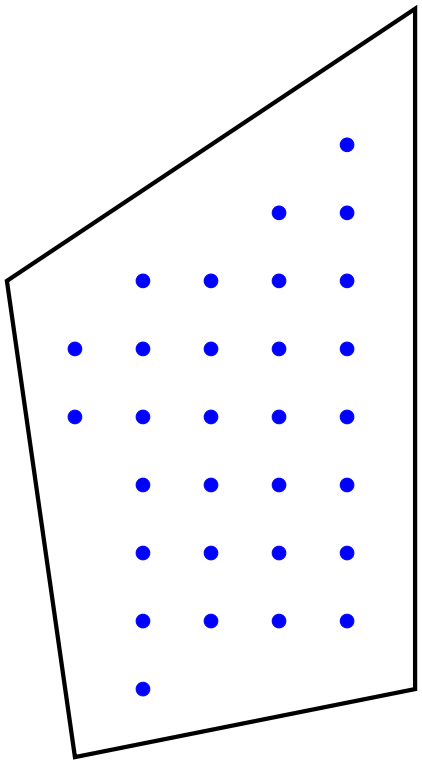


Figure 5: A figure illustrating an example of a polygon, which is covered by equidistant nodes. Each dot represents the center of a parking slot, and only parking slots which can be placed are included in the covering.

We then associate each node with a binary variable, $z_{i,j}$, which indicates whether the parking slot centered on node (i, j) is placed or not. The objective function which we want to maximize is the sum of all these binary variables. We then formulate constraints, such that if the parking slot in position (i, j) cannot be placed simultaneously as the slot in position (b, g), the sum of $z_{i,j}$ and $z_{b,g}$ must be less than or equal to 1. We formulate this condition as a function, k which is 1 if parking slot (i, j) cannot be placed simultaneously with parking slot (b, g), and else it is 0. The variables needed to create the LP-problem are all explained in table 2.

Variable name	**Meaning**
$z_{i,j}$	Binary variable determining if slot (i, j) is placed within the car park.
n_x	Number of points in the x-direction.
n_y	Number of points in the y-direction.
n	Number of points in mesh.

Table 2: A table explaining each variable used in the discrete LP-problem.

The LP-problem can then be written as the following:

$$\max \quad \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} z_{i,j}$$

s.t.

$$(1)\ z_{i,j} + z_{b,g} \leq 1 \qquad \forall \left\{ i, b \in [1, n_x]^2, j, g \in [1, n_y]^2 \mid g < j \wedge k(i,j,b,g) = 1 \right\}$$
$$(2)\ z_{i,j} \in \{0, 1\} \qquad \forall i \in [1, n_x], j \in [1, n_y]$$

This is an LP-problem with many constraints, but we can remove several of these constraints by creating dominating constraints. To do this, we define the sets $S_{(i,j)}$ as the following:

$$S_{(i,j)} = \{(b, g) \mid k(i, j, b, g) = 1\}$$

For each node, we can create a constraint saying that the sums of all $z_{b,g}$ in $S_{(i,j)}$ must be less than or equal to 1. Per definition 9.1 in [17], these constraints dominate the previous constraints, because, for each constraint on the form (1) in the first model, we have a corresponding constraint with more or as many variables on the left-hand side in the new constraints.

We can now formulate the LP-problem as the following:

$$\max \quad \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} z_{i,j}$$

s.t.

$$(1) \sum_{(b,g)\in S_{(i,j)}} z_{b,g} \leq 1 \qquad \forall i \in [1, n_x], j \in [1, n_y]$$
$$(2)\ z_{i,j} \in \{0, 1\} \qquad \forall i \in [1, n_x], j \in [1, n_y]$$

The difference is that with this formulation we only have one constraint per node, whereas the previous formulation could have an arbitrary number of constraints depending on how fine a mesh we use. This makes one simplex iteration faster, but unfortunately, it doesn't affect the branch-and-bound algorithm, which means that the problem's worst-case time complexity is still exponentially increasing with the number of nodes.

Examples of the output from the mixed-binary LP-problem can be seen in the appendix, section 10.1.1. Here we also see how to implement the program in Julia.

3.2.2 Continuous x- and y-variables

In the second method, we take a continuous approach to the x- and y-variables. This is because we actually limit the solution space of the first method to be based on how we initialize the grid, and therefore it actually can't be guaranteed that the solution is optimal. Here we instead take a continuous approach, where we take a set of nodes, and then move the nodes such that they are placed correctly.

Because we don't know how many slots we can fit in a given area, we first create too many nodes and then associate each node with a binary variable, z_i, which indicates if a

node is in the final solution or not. We also associate each node with a coordinate (x_i, y_i), which indicates the position of the node. To cut down on the number of constraints we implement a logic saying that if $i > j$ then $y_j \leq y_i$. We give the nodes the possibility to rotate 180 degrees, which is modeled by a binary variable, r_i, which indicates if slot i is pointed upwards or downwards, and another binary variable $b_{i,j}$ which is 1 if the slots i and j are placed back-to-back, such that there is no need for space to drive out between the nodes. All the variables are explained in table 3.

Variable name	**Meaning**
x_i	The x coordinate of point i.
y_i	The y coordinate of point i.
z_i	Binary variable determining if slot i is placed within the car park.
r_i	Binary variable determining the entrance of a slot is pointed upwards or downwards. It is equal to 1 if it is pointed upwards and 0 if pointed downwards.
$b_{i,j}$	Binary variable which determines if parking slot i and j can be placed back-to-back in the vertical direction.
$u_{i,j}$	Binary variable which makes sure that we only use either constraint 4 or 5 between points i and j.
n	Number of points we try to fit in the car park.

Table 3: A table explaining each variable used in the continuous LP-problem.

The entire mixed-integer program is given on the next page. Here we want to give a quick overview of the constraints. Constraints (1)-(3) are to ensure that all points remain within the car park. Constraints (4) and (5) ensure that none of the parking slots overlap with each other. Constraint (6) ensures that $y_i \geq y_j$ when $i > j$. Constraints (7) and (8) ensure that parking slots i and j are only marked as being back-to-back if slot i is oriented upwards and slot j is oriented downwards. Constraint (9) ensures that if we can only fit m slots, then we will use slot 1 to m, and discard all slots with an index greater than m. This ensures that the logic behind the other constraints holds.

$$\max \quad \sum_{i=1}^{n} z_i$$

s.t.

$$
\begin{aligned}
&(1)\ x_{max} - w/2 \geq x_i \geq x_{min} + w/2 && \forall i \in [1,n] \\
&(2)\ y_i \leq y_{max} - h/2 - r_i \cdot f && \forall i \in [1,n] \\
&(3)\ y_i \geq y_{min} + h/2 + (1-r_i) \cdot f && \forall i \in [1,n] \\
&(4)\ w \leq x_j - x_i + (2 - z_i - z_j) \cdot M + u_{i,j} \cdot M && \forall j \in [2,n], i \in [1, j-1] \\
&(5)\ h + f \cdot (1 - b_{i,j}) \leq y_j - y_i + (2 - z_i - z_j) \cdot M + (1 - u_{i,j}) \cdot M && \forall j \in [2,n], i \in [1, j-1] \\
&(6)\ y_i \leq y_{i+1} && \forall i \in [1, n-1] \\
&(7)\ b_{i,j} \leq r_j && \forall j \in [2,n], i \in [1, j-1] \\
&(8)\ b_{i,j} \leq 1 - r_i && \forall j \in [2,n], i \in [1, j-1] \\
&(9)\ z_i \geq z_{i+1} && \forall i \in [1, n-1] \\
&(10)\ z_i, r_i \in \{0,1\} && \forall i \in [1,n] \\
&(11)\ x_i, y_i \in \mathbb{R} && \forall i \in [1,n] \\
&(12)\ b_{i,j}, u_{i,j} \in \{0,1\} && \forall j \in [2,n], i \in [1, j-1]
\end{aligned}
$$

By looking at table 3, we see that we have 4 sets of binary variables. We need z_i to be binary since we want it to indicate whether we place parking slot i or not. The others are used to make the constraints. We have $\mathcal{O}(n^2)$ binary variables, which means that in the worst case, we have to solve $\mathcal{O}\left(2^{(n^2)}\right)$ LP-problems in the branch and bound tree.

Examples of the output from the mixed-binary LP-problem can be seen in the appendix, section 10.1.2. Here we also see how to implement the program in Julia.

3.2.3 Performance of OR methods

While both the OR models presented here give feasible solutions, they both have problems. For the model with continuous variables, we have only implemented that the slots can be turned in two directions, while they should be able to turn to whichever position is most optimal. That being said, this method guarantees an optimal solution when we can only use these two directions. We can turn the parking slots however we want in the method with discrete variables, but as we are discretizing the domain we don't have any guarantee that the solution from this model is optimal. This can be combated by splitting the domain into very small intervals, as that would allow us to create almost all solutions we could create with continuous variables. But splitting the domain comes at the cost of extra computation time, which brings us to the real problem with the OR models: they are very time expensive.

Referring to section 3.1.3, where we discussed the worst-case running time for the branch and bound algorithm, we see that for the method with discrete variables, we need to solve 2^n LP-problems, where n is the number of nodes in the grid. For the problem with continuous variables, the amount of LP-problems that need to be solved in a worst-case scenario is $O(2^{m^2})$, where m is the number of parking slots we try to fit. Remember that n is much

greater than m, so this is not necessarily an indication that one method is better than the other. This shows that the problems we try to solve quickly blow up, and therefore we can't solve problems on the scale that is needed for the methods to be usable in real-world applications.

3.3 Cutting and packing problem

The cutting and packing problem is the problem of placing a set of d-dimensional rectangular boxes within a rectangular container. This problem can be viewed in any dimension d. If we look at the problem in $d = 2$ dimensions, it corresponds to placing smaller rectangles within a larger rectangle. Since this is almost the problem we want to solve, we want to look further into this problem. The cutting and packing problem is described in greater detail in [2].

Since we are not the first who want to solve such a problem, there exist several methods to solve these problems effectively. We will look at one way to solve the cutting and packing problem, and how it applies to our problem.

First, we need to define the problem we are working with. We have several rectangular boxes which we want to place such that we have a *feasible packing*. We define a feasible packing as a packing with the following four properties described in [5]:

1. Each face of a box is parallel to a face of the container
2. All boxes are within the container
3. No boxes overlap
4. The boxes must not be rotated

We want to look at how we can solve this problem efficiently to give a better understanding of how we can solve our problem. More specifically, we will look into the orthogonal packing problem. To do this, we look at the method proposed by Fekete, Schepers and van der Veen in [6]. To help explain the logic behind this method, we have drawn a 2-dimensional example in figure 6. Here we have six rectangles that have been packed inside a rectangular container. We can reduce this to two one-dimensional drawings, namely one where we only look at the x-coordinates, and one where we only look at the y-coordinates. We then convert the problem into an undirected graph by representing each rectangle as a node, and if two rectangles v and u overlap, we add the edge vu. This results in two undirected graphs such as the ones in figure 6. We name these two undirected graphs G_x and G_y.
We now see that any packing can be represented as an undirected graph in the way illustrated in figure 6. Fekete and Schepers proves in [11] that for any feasible packing, the corresponding graphs, $G_i = (V, E_i),\ i = 1, \ldots, d$, fulfills the following three properties:

1. The graphs $G_i = (V, E_i)$ are in interval graphs
2. Each stable set of S of G_i is x_i-feasible
3. $\bigcap_{i=1}^{d} E_i = \emptyset$

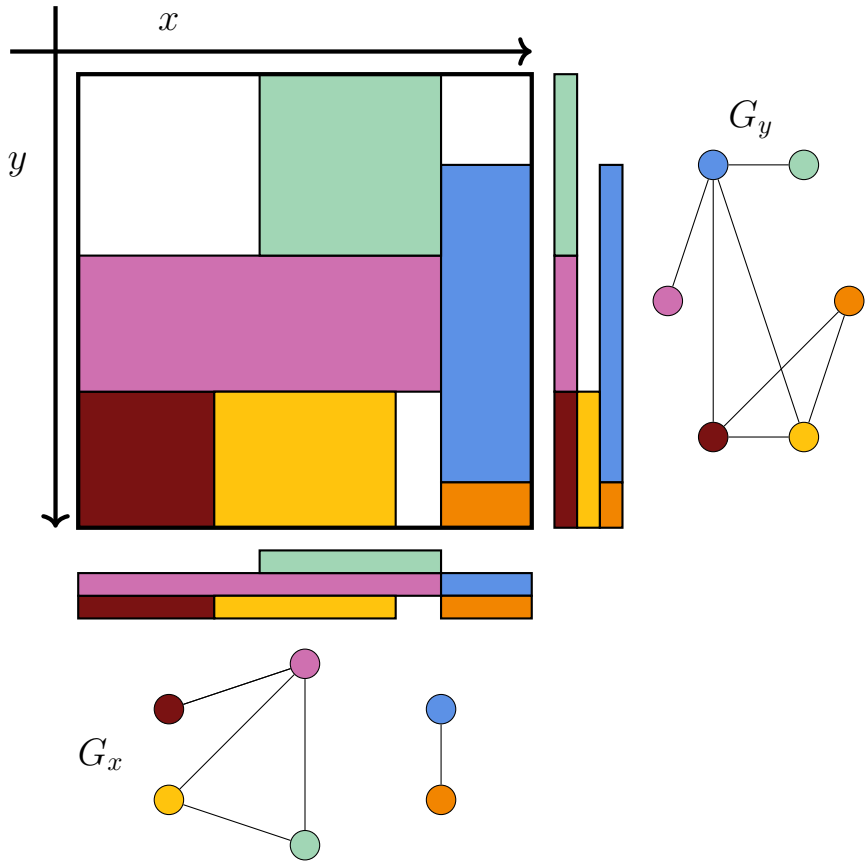


Figure 6: A figure illustrating a 2-dimensional packing of boxes in a container. The nodes in the undirected graphs G_x and G_y represents the boxes in the container, and the nodes share an edge if and only if they overlap in the x or y direction.

In our case $d = 2$. In [5] Fekete and Schepers also prove that these three properties are sufficient for determining if the graphs G_i belong to a feasible packing.

An *interval graph* is a graph where each node can be interpreted as an interval on the real line, and two nodes share an edge if and only if the two intervals overlap. A *stable set* is a subset of the nodes in the graph where no nodes share an edge. For a set of boxes, $S \subseteq V$, to be x_i-feasible means that it fulfills the following:

$$\sum_{b \in S} w_i(b) \leq W_i$$

Here $w_i(b)$ is the width of box, b, in the i direction, and W_i is the width of the container in the i direction. With this knowledge, property 2 can simply be interpreted as saying that the width of any set of non-overlapping boxes in the i direction must not be wider than the container. Property 3 is easily understood as no boxes can share an edge in all directions, because then they would overlap in all directions.

Now recall section 3.1.2 about the branch and bound algorithm. In this section, we describe branching by setting an upper and lower limit for an integer variable. But we can branch on anything as long as we ensure that we don't lose any feasible solutions in the

process. What Fekete, Schepers, and van der Veen propose in [6] is that we branch in the undirected graphs G_x and G_y, or in the general case, all the graphs G_i, by taking an edge in one of the graphs $vu \in E_i$, and excluding it in one branch and including it in the other branch.

By using the three properties described above we can extract more information from a given branch. For example, if we know that $\forall i \neq j : vu \in E_i$ then by property 3, $vu \notin E_j$.

In this way, we can determine if there exists a packing with X parking slots or not. This means that by using binary search we could determine the maximum amount of parking slots we can place. There are two problems with this approach in regards to our specific application. The first problem is that this approach can't handle the free space needed in front of the parking slot, since it assumes that a box either uses the space or not and, therefore, there is no way two boxes can share space.

The other problem is also pointed out by Fekete, Schepers, and van der Veen in [6]. The problem is the combinatorial complexity when all our parking slots have the same size. If we have a feasible packing we can obtain another feasible packing by simply switching two parking slots. This means that using the method described here would require us to do the same calculation many times. This could easily be avoided by looking at the graphs in the different branches and identifying isomorphic graphs as we would only need to look at one of the isomorphic graphs, and could discard the others. Unfortunately, there is no fast way to determine if two graphs are isomorphic, and, thereby, the complexity of this approach would explode exponentially, like the methods in section 3, as the combinatorial complexity increases exponentially with the number of parking slots.

This is of course only one method to solve the cutting and packing problem and there exist other methods which also solve the problem. However, there has yet to be found a method that can solve the cutting and packing problem effectively, that is, in polynomial time. Therefore we have chosen to look at different methods such that we can optimize car parks that could appear in real-life applications.

4 Heuristics in arbitrary polygons

We just found out in the previous section that the LP-problems are hard to solve due to the computational complexity. Therefore, we choose to make several different heuristics such that we can generate solutions. The heuristics can solve the problem fast but are not guaranteed to be optimal. The way we have developed these algorithms was by first implementing them on a straight rectangle, then for arbitrary convex polygons, and lastly for non-intersecting polygons. Some of our algorithms proved themselves poorly in simpler cases and for these, we didn't use resources to develop a version able to handle more complex cases.

4.1 Algorithm for finding placement interval

Before actually introducing the heuristics, we will introduce the function for finding a placement x-interval that is used frequently in our heuristics. We find these placement intervals in arbitrary non-intersecting polygons, which means that we can have a case, where there

are one or more indents in the polygon. Therefore, we can get several intervals at which it is possible to place parking slots.

We find the intervals by considering a certain y-value at the center of a parking slot and then search for x-intervals at which the center of a parking slot can be placed legally. We see that three situations could block the possibility of placing a parking slot at an x-value:

- If the parking slot hits the polygon in a top corner-point
- If the parking slot hits the polygon in a bottom corner-point
- If a corner-point in the polygon has y-value between the highest and lowest y-value of the parking slot, including the free space

Let y_{low} denote the lowest y-value of the parking slot and let y_{high} be the highest. We can then create a list of all x-values at which the polygon either intersects at y_{low} or y_{high}, and the x-values of the corner-points of the polygon which have y-value between y_{low} and y_{high}. We sort this list and traverse it from the beginning. When looking at points i and $i+1$ we see that they are an interval if and only if we can draw a line parallel to the y-axis from y_{low} to y_{high} without leaving the polygon.

The pseudocode can be found in algorithm 2 in the appendix, section 10.4.1.

By calling the function given by the pseudocode in algorithm 2, we get a list of all x-value intervals along a given y-value. E.g. $\{1, 2, 5, 7, 9, 11\}$ denotes that we have three legal intervals, namely $[1, 2], [5, 7]$ and $[9, 11]$.

Given this function, we are now able to present our heuristics.

4.2 Parking slots in arbitrary polygons

The first heuristic we want to implement is the simplest of the five we will present in this paper. First, we simply want to place rows of straight parking slots. We furthermore want to be able to do this in an arbitrary non-intersecting polygon. The heuristic works like this:

We rotate and move the given polygon such that one of the sides lies on the x-axis. We will place parking slots along this side's direction. We then find x-intervals in which we can place parking slots (the method to do this is described in section 4.1). We then iteratively take each interval and from the lowest x-value in the interval, we place parking slots orthogonally along a line parallel to the x-axis. The parking slots are placed side-by-side. This continues until the next parking slot we would place would leave the interval.

Now we iteratively place more rows. These rows are placed in pairs, such that the lower row points downwards and the upper row point upwards. The distance between the opposite rows is given by the free space described in section 2.1.

This method creates parking slots that are all placed legally. An example of this algorithm can be seen in figure 7, which clearly illustrates how we avoid placing slots outside the car park. The pseudocode for this algorithm can be seen in algorithm 3 in the appendix, section 10.4.1.

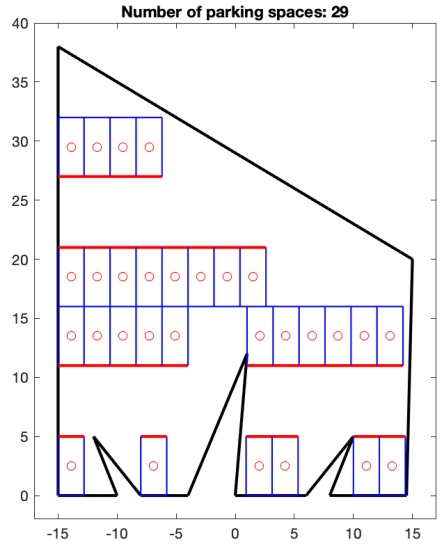


Figure 7: A figure illustrating an example of a non-intersecting car park filled with rows of straight parking slots. This car park has three spikes to illustrate how we place the parking slots such that they are placed within the car park, and there is enough space for a car to leave a parking slot. In this figure, we use the bottom side to place all rows.

4.3 Parking slots with angles in arbitrary polygons

This heuristic works on the same principles as the previous heuristic, but with the difference that the parking slots have been rotated by a given angle. This adds some complications to the algorithm. First, we have to ensure that when we place different rows the parking slots are placed back-to-back to minimize wasted area. This means that when we place two rows they can no longer be placed independently from each other as we have to ensure that the rows are aligned. The way we ensure this is by placing the first parking slot freely. Afterward, we place the next parking slots by moving the center with a whole width of a rotated parking slot. When placing the next row, we do the same, only that the first parking slot is placed by using the back-side of the first parking slot of the previous row.

Another complication is that a lot of measurements are constant in the previous heuristic but now depend on the angle. The length of the free space in front of the parking slot can be interpolated as described in section 2.1. Furthermore, we use a rotation matrix to determine the height and width of a parking slot when it is rotated.

The pseudocode for this greedy algorithm can be seen in algorithm 4 in the appendix, section 10.4.1.

An illustration of the algorithm can be seen in figure 8.

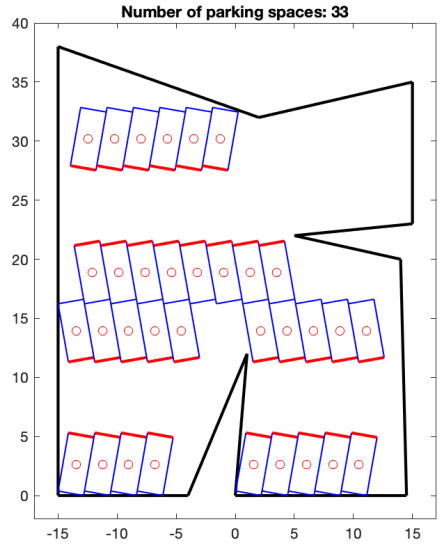


Figure 8: A figure illustrating an example of a concave car park filled with rows of angled parking slots. Here we also see an illustration of how the parking slots are placed back-to-back. In this figure, we use the bottom side to place all rows. All the parking slots have an angle of $\alpha = 10$ degrees.

4.4 Parking slots with angles in arbitrary polygons using a first side

From figure 8 we see that since we limited the algorithm to place all parking slots from one side then there can be some empty spaces along some other edges, where we do not fill the car park optimally. Therefore, we introduce an algorithm such that we can index two different sides, namely one called *first side* and one called *side*. Those two sides can also have independent angles. We now first fill one row along the *first side*, whereas we afterward try filling the rest of the car park along the *side*. The pseudocode for the algorithm can be seen in algorithm 5 in the appendix, section 10.4.1.

The big difference between algorithm 4 and 5 is that we first place a row on one side, and afterward use algorithm 4 on another side. To do this, we define a box containing all parking slots from the first side, and before we insert a new parking slot using the algorithm 4 on the other side, we check that the parking slot is not inside this box.

We tried using the algorithm on the same car park used in figure 8. This gives us the solution seen in figure 9.

We see that we now are able to place more parking slots inside the same area, which also was expected. This strategy could be further expanded to place on multiple sides independent of each other, but we did not investigate this further.

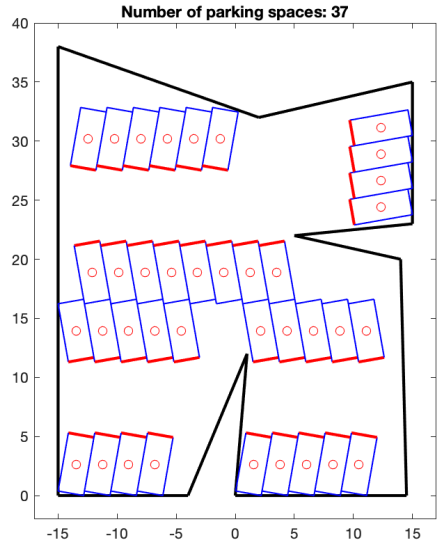


Figure 9: A figure illustrating an example of a non-intersecting car park filled by rows of angled parking slots using a specific first side. In this figure, we use side 8 to be the first side, and the bottom side to place all remaining rows. The angle of the parking slots on the first side is $\alpha = -10$ degrees, and the angle for the remaining parking slots is $\alpha = 10$ degrees.

4.5 Parking slots around the boundary

We have also created two different heuristics that try to fill the car park by placing parking slots along the boundary. However, we quickly discovered that these methods were far inferior to the methods described above, where we place parking slots in rows. For this reason, we have only implemented these algorithms for convex polygons. We have chosen to include a short description of these algorithms anyway to give an idea of what we have tried.

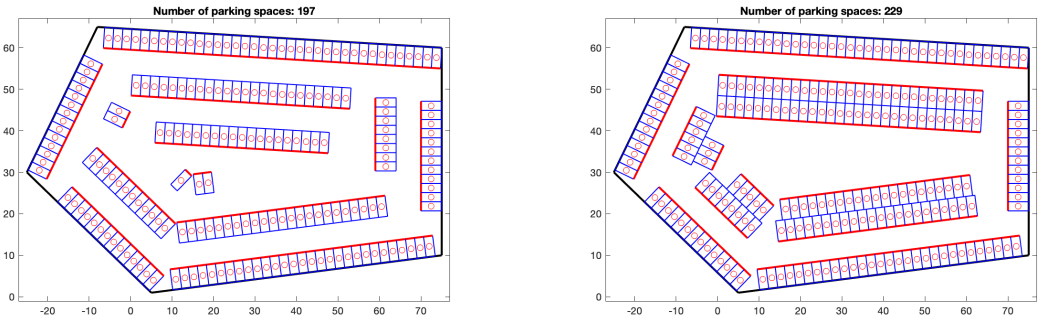
To fill out the boundary, we start by rotating the entire car park such that a chosen side lies on the x-axis. Then we place as many parking slots as possible along this side. Then we move on to the next side and, by using the angle between the sides and the point where the sides intersect, we can easily calculate the placement of the next parking slot. We then do the same for the remaining sides until we have filled up all sides.

The geometric calculations and arguments have only been described very superficially here, but for the interested reader, we have included a more detailed description in the appendix, section 10.2.

We know how much free space there is needed in front of each parking slot, so we can calculate a new area inside the original car park. Afterward, we can use the same algorithm to place more parking slots. We do this recursively until the area is empty.

The difference between the two methods using the boundary is that one only places a single row along the boundary of any given car park while the other places two rows back-to-back when we call the heuristic recursively. Figure 10a shows an example of the first heuristics, while figure 10b shows an example of the second.

The pseudocode for the two heuristics can be found in the appendix, section 10.4.1.



(a) Example of a convex car park filled by parking slots placed in a single rows along the boundary of the car park.

(b) Example of a convex car park filled by parking slots placed in a single rows along the boundary of the car park and double rows inside.

Figure 10: A figure illustrating examples of two different heuristics placing parking slots along the boundary of a car park.

4.6 Performance of heuristic algorithms

We have mentioned before that the heuristic algorithms should be faster than the OR method described in section 3. Therefore, it is interesting to look at some timings for our heuristic algorithms from the previous sections. When we have chosen all the parameters for each of the algorithms, we would expect that the time to place the parking slots would be close to the same for all the algorithms. This is because the complexity of the algorithms doesn't come from the placement algorithm, but rather from how many times the placement algorithm is called.

This is tested for two arbitrary car parks of different sizes. We call each algorithm with the same parameters and record the time for 100 calls. The result of the test has been summarized in table 4. We see that algorithm 3 takes the most time, which also was expected since we make more operations in this algorithm. Furthermore, algorithm 4 and 5 is the fastest, which also was expected because the algorithm only fills the boundary when we make one call. Though, we overall see that all the algorithms perform fast.

We now have an idea of how much time each algorithm takes to execute one call. We, therefore, want to look at how many times each algorithm should be called to test all combinations to generate the best solution. This has been summarized for each algorithm in table 5.

Heuristic	Time (s)	Average time (s)
1	1.6386	0.0164
2	2.1017	0.0210
3	1.9907	0.0199
4	0.3913	0.0039
5	0.3237	0.0032

(a) Small car park that is 1770.5 square meters.

Heuristic	Time (s)	Average time (s)
1	5.6176	0.0562
2	15.7877	0.1579
3	16.5265	0.1653
4	1.0711	0.0107
5	1.3113	0.0131

(b) Large car park that is 38379.13 square meters.

Table 4: A table listing the testing time calling each algorithm 100 times. Heuristic 1, 2 and 3 correspond to the algorithm described in section 4.2, 4.3 and 4.4, respectively. Heuristic 4 and 5 is the single and double boundary heuristic, respectively, described in section 4.5.

Heuristic	# of calls
1	m
2	$\alpha \cdot m$
3	$\alpha^2 \cdot m^2$
4	m
5	m

Table 5: A table illustrating the number of calls for each heuristic algorithm. α is the number of angles we try in the given algorithm, and m is the number of edges in the user-defined car park. Heuristics 1, 2 and 3 correspond to the algorithm described in section 4.2, 4.3 and 4.4, respectively. Heuristics 4 and 5 are the single and double boundary heuristic, respectively, described in section 4.5.

In table 5 we see that the number of calls is largest in Heuristic 3. We, therefore, expect this algorithm to take the longest time if we set the same parameters for the four algorithms.

5 Path out algorithm

We have now seen how we can create solutions in which the parking slots are placed legally with regards to each other. It is not guaranteed that these solutions are completely feasible, however, as we are not guaranteed that it is possible to exit the car park from all parking slots.

Therefore, we first need to be able to define an edge, wherethrough a car inside the car park can exit the area. We call this an *out edge* and it is defined to be an interval where a car can hit the main road. We now want an algorithm that given a solution, where it's not possible to hit the main road from all parking slots, we remove the minimum amount of parking slots such that we can get out from all parking slots. We name this algorithm the *path out algorithm*.

First, we create a grid of points in the unused areas of the car park. These points are

placed with a user-defined interval in both directions. From this grid, we remove all points which lie within 2 meters of a parking slot or the boundary of the car park. This is done to ensure that the road out is wide enough for a car to drive through. Furthermore, we create several points on the out edge.

For all parking slots, we define an individual *first point*. The *first point* is defined to be one random point in the grid that lay in front of the parking slot. The idea is that if it's possible to get from the first point to an out edge, then we say that it's possible to get from the parking slot to the main road.

We now create an edge between all points where the distance in either direction is precisely the user-defined interval. We also create edges between the points in the grid and a point on the out edge if the distance is less than or equal to the user-defined interval.

We can now use a modified version of the Depth-First Search (DFS) algorithm to check whether a first point sometimes hits a point on the out edge. The modified DFS algorithm pseudocode can be seen in algorithm 1.

Algorithm 1 DFS algorithms

```
1: function DFS_INIT(points)
2:     for all points p in points do
3:         DFS_ALGORITHM(p)
1: function DFS_ALGORITHM(v)
2:     if v ∈ endpoints then
3:         p can get out
4:     else
5:         mark v
6:         for each unmarked neighbour u do
7:             DFS_ALGORITHM(u)
```

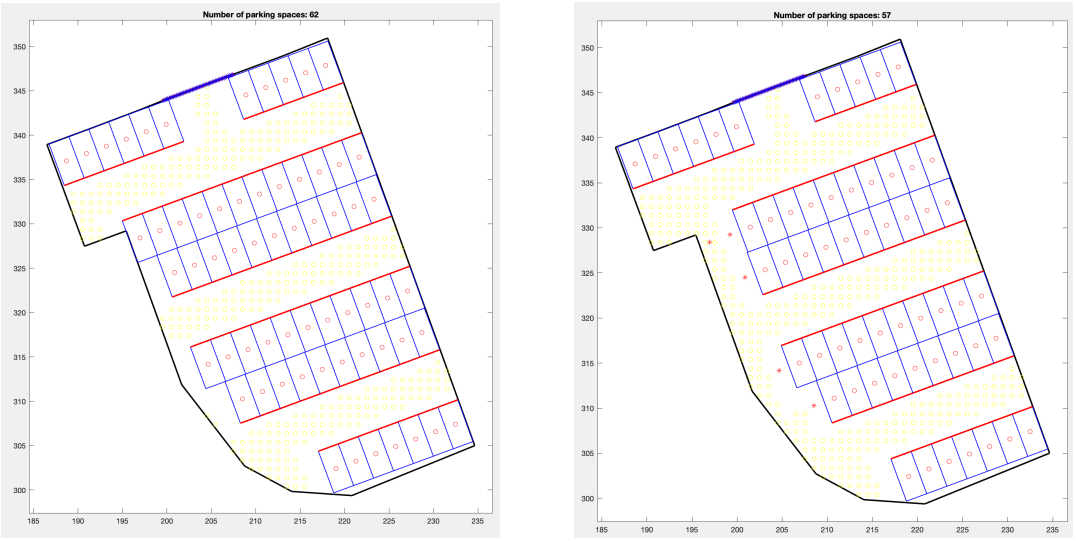
After running the DFS method to check whether all first points can reach a point on the out edge, we are in two scenarios, namely a scenario where all parking slots can get out and one where they can't. If all parking slots can get out we simply stop our algorithm. If all parking slots can't get out we first create fictive edges between all first points and the center of its parking slot. Afterward, we create fictive edges between the center of a parking slot that can get out and the center of a parking slot that can't get out if and only if they are less than or equal to a user-defined distance. All the edges that we create using the center of the parking slots we name *fictive edges*.

We can now run the modified DFS algorithm again to check whether there exist some first points that by using the fictive edges going through the center of a parking slot now can reach a point on the out edge. If there exist those extra first points we try removing parking slots in a certain order. The order can be seen in table 6. Note that a *old parking slot* is defined as a parking slot that could get out before, and a *new parking slot* is defined as a parking slot that can get out now using fictive edges but not before.
In each iteration, we check whether more first points can reach a point on the out edge without using the fictive edges. If this happens, we choose to remove these parking slots

from the solution and start the algorithm over. The pseudocode for the whole path out algorithm can be found in algorithm 8 in the appendix, section 10.4.2.

Order	# parking slots	Description
1	1	Remove 1 new parking slot
2	1	Remove 1 old parking slot
3	2	Remove 2 new parking slots
4	2	Remove 1 new parking slot together with 1 old parking slots
6	2	Remove 2 old parking slots
5	3	Remove 2 new parking slots together with 1 old parking slot
7	3	Remove 1 new parking slot together with 2 old parking slots
8	3	Remove 3 new parking slots
9	3	Remove 3 old parking slots
10	4	Remove 2 new parking slots and 2 old parking slots.

Table 6: A table listing the order in which we try to remove parking slots. Note that an *old parking slot* is defined as a point that could get out before, and a *new parking slot* is defined as a point that can get out now but not before. The goal is to remove the fewest possible parking slots and, therefore, the order is as stated in this table.



(a) The initial infeasible solution before running the path out algorithm. Here there are 62 parking slots within the car park, where only 23 can get out.

(b) The car park after running the path out algorithm. Here there are 57 parking slots within the car park. We remove 5 parking slots with the algorithm.

Figure 11: A figure illustrating the path out algorithm. The blue stars are the points on the out edge. The yellow stars are the grid. The red circles are the centers of the parking slots, and the red edge is the exit edge of a given parking slot. The red stars are the parking slots that we remove with the path out algorithm such that every remaining parking slot in the car park can come out.

An example of the path out algorithm can be seen in figure 11. Here we see in figure 11a that we start with a solution where 23 out of 62 parking slots can get out. We see in figure 11b, that after running the path out algorithm all parking slots can get out. We remove 5 parking slots in the algorithm, which results in a car park with 57 parking slots.

Please note, that in this algorithm we assume that being able to access the out edge from the parking slot means that we can access the parking slot from the out edge. This is unfortunately not always the case since the free space, f, describes in section 2.1 can be small enough such that we need to make a one-way road.

6 Physical car parks

We have now introduced our heuristics and the path out algorithm that together can create a feasible placement of parking slots in a car park. Of course, we want to test how well our algorithms perform in different scenarios, so we decided to look at three different physical car parks. The reason why we choose to look at physical car parks is to avoid any bias we might create by constructing our own areas. Furthermore, we want to see if we can improve on an already existing solution.

To generate the corner-points of a car park, we first take a snapshot of the car park in Google Earth v7.3.4.8573. Afterward, we import this snapshot into Logger Pro v3.15, where we can set the corner-points of the car park. We can furthermore define a certain length in the snapshot to be a certain length in the metric scale. In this way, Logger Pro generates all the corner-points, whereas we have defined the car park. The corner-points for the three car parks have been listed in the appendix, section 10.5.

Because we are also interested in measuring the performance of our algorithms, we use DTU's High Performance Computing (HPC). Using the same CPU type for the different car parks gives us precise and comparable time results. We ran the algorithms on four cores of the CPU type Intel® Xeon® Processor E5-2660 v3.

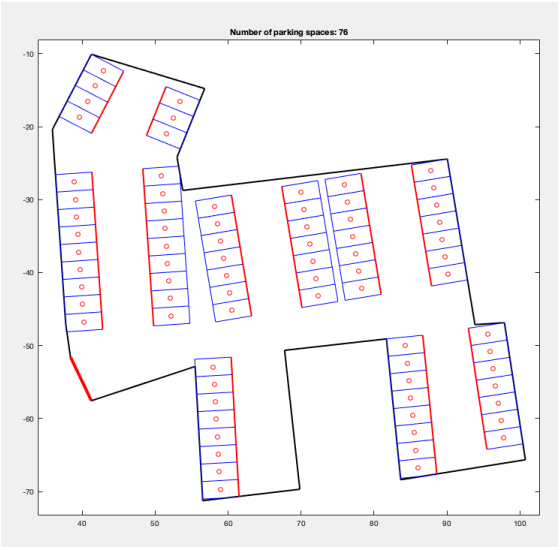
6.1 Lyngby-Taarbæk municipality car parks

We wrote to Lyngby-Taarbæk municipality, see [9], and they told us that there is no space for more car parks within the municipality, whereas they are interested in whether it is possible to optimize their existing car parks. They suggested that we tried optimizing two different car parks located near the station. We name those two car parks for *Stades Krog* and *Engelsborgvej*.

6.1.1 Stades Krog

Stades Krog is the smallest of the two car parks on which we tried our algorithms in Lyngby-Taarbæk municipality. There are currently 76 parking slots in the car park, and there is one road in and out of the car park. An air photo of the car park and the current solution can be seen in figure 12. We see that there are 18 corner-points, which results in 18 edges. The red edge in figure 12b is the access to the main road in and out of the car park.

(a) An air photo from Google Earth of the car park named Stades Krog. This car park has 18 edges.



(b) The current solution of the car park named Stades Krog generated in our visual design.

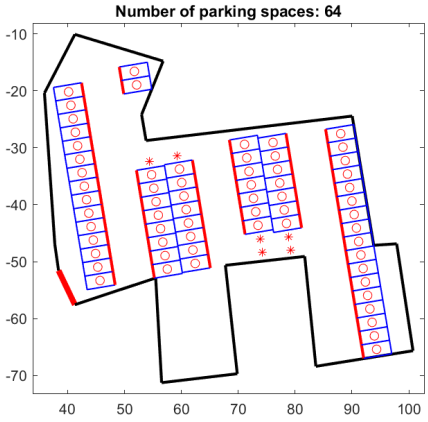
Figure 12: A figure illustrating an overview of the car park named Stades Krog. We see that the car park has 18 edges, and the current solution has 76 parking slots in the car park. The red edge on the boundary of the car park illustrates the one way in and out of the car park.

Even though the area of the car park isn't that big, we see that the car park has many edges. Therefore, we see that the number of calls explained in table 5 for each algorithm becomes large. This is also confirmed by table 7. In table 7, we also see the timing of the algorithms using HPC, as described at the beginning of this section.

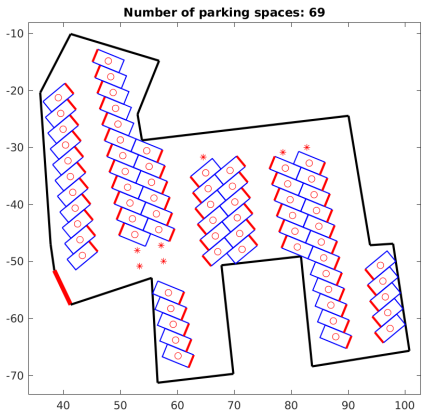
Heuristic	# of calls	Time to run algorithm (s)	Average time (s)
1	18	1.67	0.1856
2	1620	96.14	0.0593
3	2624400	161223.27	0.0614

Table 7: A table listing the running time for heuristic algorithms on the Stades Krog car park. $\alpha = 90$ is the number of angles we try in the given algorithm, and $m = 18$ is the number of edges in the car park. The third column is measured by timing all the calls of a given algorithm, using HPC as described at the beginning of this section. The last column is generated by dividing the third column by the second.

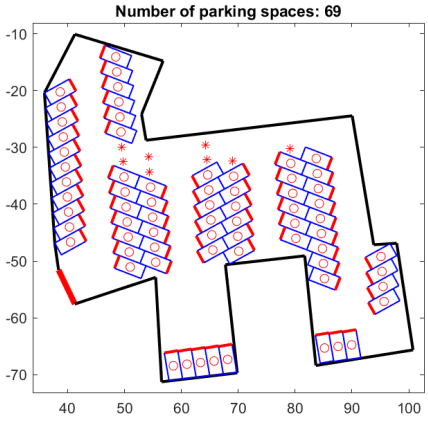
For each algorithm, we get an output which is shown in figure 13. We see that all the algorithms are worse than the current solution listed in figure 12b. Therefore, this is a case where our generated solutions from the algorithms would not optimize the current solution, and we really can't use it for further implementation in the physical car park. This is discussed further in section 7.2.



(a) The output of algorithm 1. Here we have 64 parking slots within the car park. We have removed 6 parking slots along the path out algorithm. We fail to remove some parking slots from the leftmost column, because we only create fictive edges between slots, and not between parking slots and points in the grid.



(b) The output of algorithm 2. Here we have 69 parking slots within the car park. We have removed 7 parking slots along the path out algorithm. All parking slots are placed with an angle of -31 degrees. Here we see that all parking slots can get out after using the path out algorithm.



(c) The output of algorithm 3. Here we have 69 parking slots within the car park. We have removed 8 parking slots along the path out algorithm. All parking slots on the first side are placed with an angle of -3 degrees, and the rest is placed with an angle of -25 degrees. Here we see that all parking slots can get out after using the path out algorithm.

Figure 13: A figure illustrating our optimized car parks with algorithms 1, 2, and 3. The red stars are the parking slots that we remove along the path out algorithm such that every remaining parking slot in the car park can get to the red out edge. The algorithms are run on all sides and tried at all integer angles where $\alpha \in [-45, 45]$. The grid of the path out method is set to be 0.25.

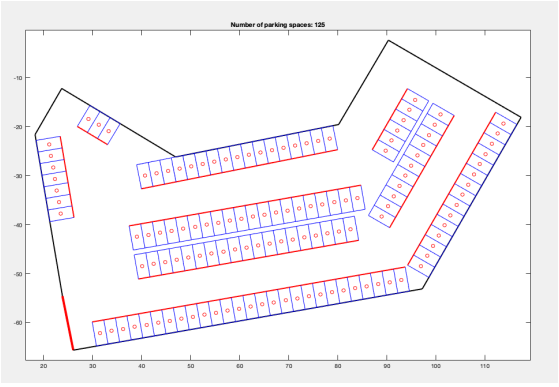
We want to calculate the density from section 2.1 of the best solution in figure 13. We use the solution from either algorithms 2 or 3 since they have the solution with the most parking slots. The density is calculated in the appendix, section 10.3.3. The density becomes $\rho = 0.377$. This is a decrease of 9.16% from the current solution, which has a density of $\rho = 0.415$.

As we see that none of the algorithms create a feasible solution, we want to modify the solutions to create a feasible one. To do this, we either need to make the roads in the car park one-way in the solution from algorithms 2 and 3 or remove further parking slots from the solution from algorithm 1. We see that in this car park there exists no combination of one-way roads or removing parking slots in any of the solutions in figure 13 that results in a better and legal solution. We can make a legal solution using the solution from algorithm 1, but this results in a solution with far fewer parking slots than the current one. Therefore, we, unfortunately, must accept the fact that our algorithms can't improve the current solution.

6.1.2 Engelsborgvej

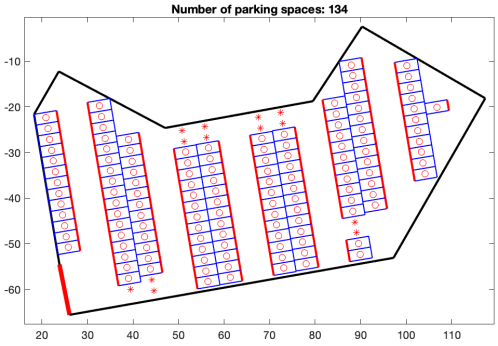
As mentioned before, we also try using our algorithms on another car park named Engelsborgvej. There are currently 120 parking slots in the car park, but it has just been confirmed in a meeting record from Lyngby-Taarbæk municipality, see [10], that they intend to take over a piece of the neighboring land, whereas they can fit 5 more parking slots in the existing car park. We, therefore, investigate the car park with 125 parking slots. Again, we only have one road in and out of the car park. The air photo and current solution can be seen in figure 14. We see that there are 9 corner-points, which results in 9 edges. The red edge in figure 14b is the access to the main road in and out of the car park.

(a) An air photo from Google Earth of the car park named Engelsborgvej. This car park has 9 edges.

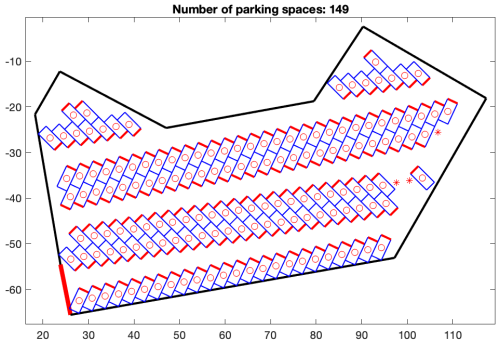


(b) The current solution of the car park named Engelsborgvej generated in our visual design.

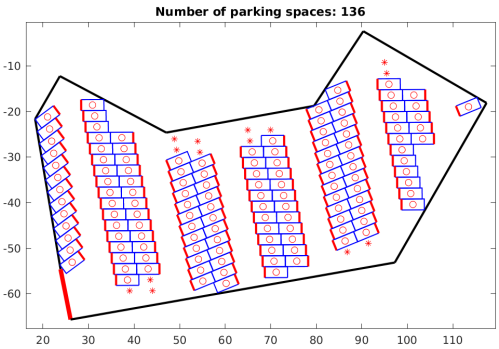
Figure 14: A figure illustrating the car park named Engelsborgvej. We see that the car park has 9 edges, and the current solution has 125 parking slots in the car park. The red edge on the boundary of the car park illustrates the one way in and out of the car park.



(a) The output of algorithm 1. Here we have 134 parking slots within the car park. We have removed 13 parking slots along the path out algorithm. The grid of the path out method is set to be 0.25.



(b) The output of algorithm 2. Here we have 149 parking slots within the car park. We have removed 3 parking slots along the path out algorithm. All parking slots are placed with an angle of 35 degrees. The grid of the path out method is set to be 0.085.



(c) The output of algorithm 3. Here we have 136 parking slots within the car park. We have removed 14 parking slots along the path out algorithm. Parking slots along the first side have been placed with an angle of -27 degrees, and the rest have been placed with an angle of -11 degrees. The grid of the path out method is set to be 0.25.

Figure 15: A figure illustrating the output of algorithm 1, 2 and 3 using the car park named Engelsborgvej. The red stars are the parking slots that we remove along the path out algorithm such that every remaining parking slot in the car park can get to the red out edge. The algorithms are run on all sides and tried at all integer angles where $\alpha \in [-45, 45]$.

Since there are fewer edges in the car park of Engelsborgvej than in Stades Krog, then

the number of calls explained in table 5 for each algorithm becomes smaller. This is also confirmed in table 8. In table 8, the time column is again measured by HPC as described at the beginning of this section.

Heuristic	# of calls	Time to run algorithm (s)	Average time (s)
1	9	0.98	0.1089
2	81	34.38	0.4244
3	656100	29127.85	0.0444

Table 8: A table listing the running time for heuristic algorithms on the car park named Engelsborgvej. $\alpha = 90$ is the number of angles we try in the given algorithm, and $m = 9$ is the number of edges in the car park. The third column is measured by timing all the calls of a given algorithm, using HPC as described at the beginning of this section. The last column is generated by dividing the third column by the second.

For each algorithm, we get an output which is shown in figure 15. We see that after running the path out method algorithm 2 in figure 15b has the best solution.

In figure 15 we see that all the solution contain more parking slots than the current solution listed in figure 14b.

Again, we want to calculate the density from section 2.1 of the best solution. We use algorithm 2 since it has the solution with most parking slots in it. The density is calculated in the appendix, section 10.3.4. The density becomes $\rho = 0.487$. This is an improvement of 19.07% from the current solution, which has a density of $\rho = 0.409$.

The solution from algorithm 2 in figure 15b is a huge improvement to the current solution, but unfortunately, it is not completely legal. To make it legal, we have to remove the slots which can't come out at the top. We also have to make sure that one can access and exit from all parking slots without driving in the wrong direction on a one-way street. We have done this in figure 16, where we have marked the directions of the streets and which extra slots we have removed. The density of our solution becomes $\rho = 0.454$, which is an improvement of 11.00% from the current solution. The solution improves the existing car park with 14 parking slots and is, therefore, a desirable alternative to the existing car park.

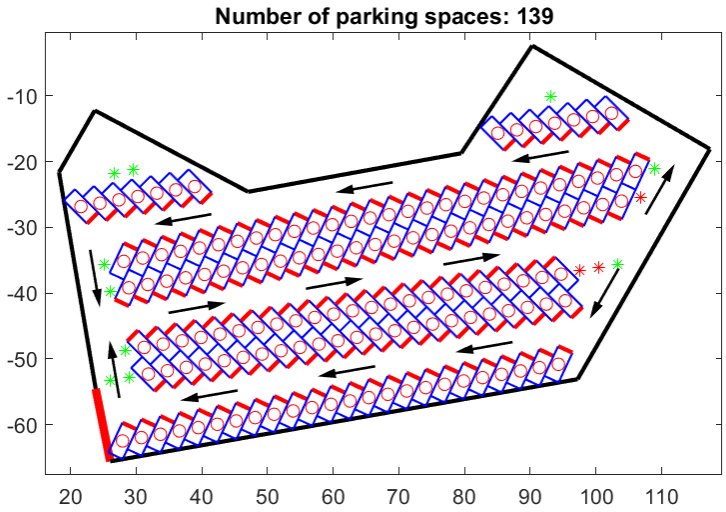
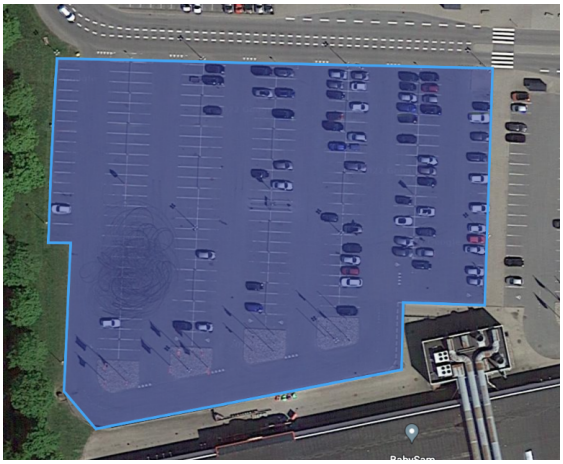


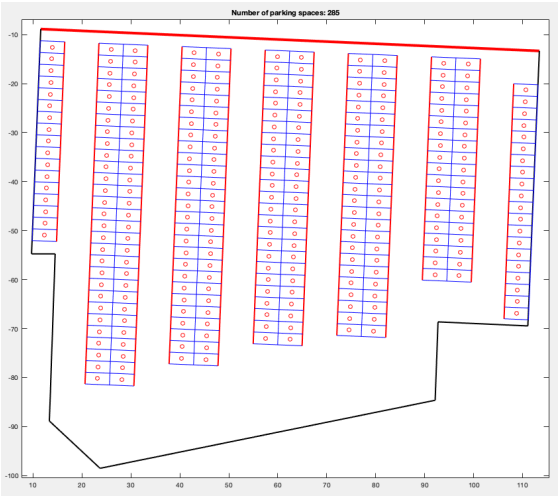
Figure 16: A figure illustrating our preferred solution for the car park using inspiration from the solution of algorithm 2 in figure 15b. This solution has 149 parking slots within the car park. The red stars are the parking slots that we remove along the path out method, and the green stars are the ones that we remove from the solution in figure 15b. The black arrows represent the direction in which the cars in the car park should drive. We see that we have succeeded to create a new solution which improves upon the current one with 14 parking slots, and this is, therefore, a desirable alternative to the existing car park.

6.2 IKEA Taastrup car park

While the car parks in Lyngby-Taarbæk are interesting examples that one can easily relate to, they are also examples of relatively small car parks. To investigate how well the methods we have developed work on larger car parks we decided to have a look at the car park in front of IKEA Taastrup. The parking space outside of IKEA Taastrup is huge and separated by roads and streets and we, therefore, only look at $1/3$ of the area. There are currently 285 parking slots in the car park and there are multiple roads in and out of the car park. Since all the roads in and out of the car park is along one edge, we can settle with only one out edge. The air photo and current solution can be seen in figure 17. We see that there are 9 corner-points, which results in 9 edges. The red edge in figure 17b is the access to the main road in and out of the car park.



(a) An air photo from Google Earth of the car park at IKEA. This car park has 9 edges.



(b) The current solution of the car park at IKEA generated in our visual design.

Figure 17: A figure illustrating an overview of the car park named IKEA. We see that the car park has 9 edges, and the current solution has 285 parking slots in the car park. The red edge on the car park boundary illustrates the one way in and out of the car park.

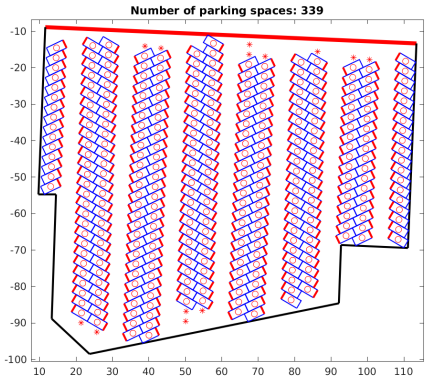
Since there are the same amount of edges in the car park of IKEA as in Engelsborgvej, then the number of calls explained in table 5 for each algorithm is the same. This is also confirmed by table 9. In this table, the time column is again measured by using HPC as described at the beginning of the section.

Heuristic	# of calls	Time to run algorithm (s)	Average time (s)
1	9	1.01	0.1122
2	810	59.09	0.0730
3	656100	46776.93	0.0713

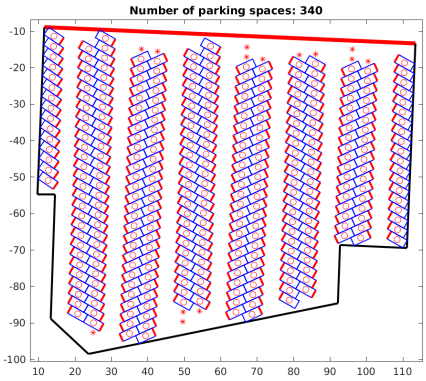
Table 9: A table listing the running time for heuristic algorithms on the car park named IKEA. $\alpha = 90$ is the number of angles we try in the given algorithm, and $m = 9$ is the number of edges in the car park. The third column is measured by timing all the calls of a given algorithm, using HPC as described at the beginning of this section. The last column is generated by dividing the third column by the second.

For each algorithm, we get an output which is shown in figure 18. Notice that all the solutions are better than the current solution listed in figure 17b. Furthermore, notice that the angles in the solutions from algorithms 2 and 3 are close the the optimal angle described in section 2.2. We see that after running the path out method algorithm 3 in figure 18c has the best solution. Though, this solution is not legal because of the angle on the first side. Therefore, the best legal solution is from algorithm 2.

(a) The output of algorithm 1. Here we have 286 parking slots within the car park. We have removed 11 parking slots along the path out algorithm.



(b) The output of algorithm 2. Here we have 339 parking slots within the car park. We have removed 13 parking slots along the path out algorithm. All parking slots are placed with an angle of 28 degrees.



(c) The output of algorithm 3. Here we have 340 parking slots within the car park. We have removed 14 parking slots along the path out algorithm. The parking slot on the first side, which is the left most side, is placed with an angle of 32 degrees. The rest of the parking slots are placed with an angle of 27 degrees.

Figure 18: A figure illustrating our optimized car parks with algorithms 1, 2, and 3. The red stars are the parking slots that we remove along the path out algorithm such that every remaining parking slot in the car park can get to the red out edge. The algorithms are run on all sides and tried at all integer angles where $\alpha \in [-45, 45]$. The grid of the path out method is set to be 0.25.

Again, we want to calculate the density from section 2.1 of the best solution. We use algorithm 2. The density is calculated in the appendix, section 10.3.5. The density becomes $\rho = 0.532$. This is an improvement of 18.75% from the current solution, which has a density

of $\rho = 0.448$.

Since the best solution from algorithm 2 in figure 18b uses angled parking slots, the streets need to be one-way only. Currently, it is not possible to come from any parking slot to the out edge if the streets become one-way. We, therefore, modify our solution, such that this is possible. The modified solution can be seen in figure 19. Notice also that this modified solution allows us to reinsert a parking slot that was previously removed by the path out algorithm. The slot is marked with a black star. The density for this solution becomes $\rho = 0.523$, which is an improvement of 16.8% from the current solution. The solution improves the existing car park with 48 parking slots and is, therefore, a desirable alternative to the existing car park.

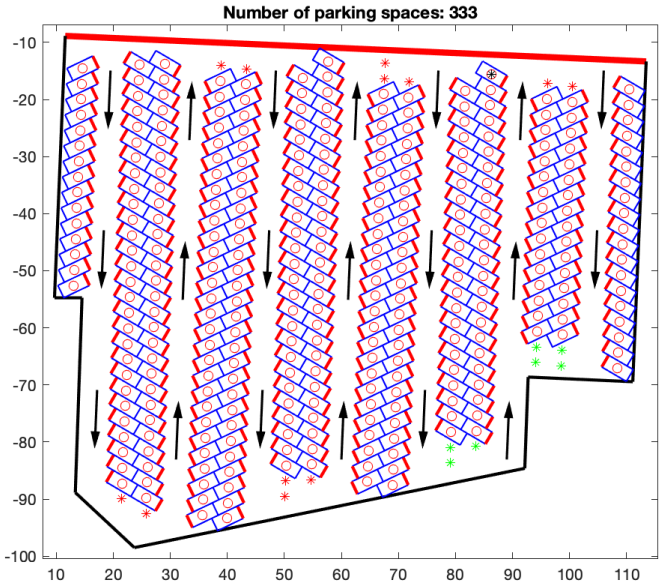


Figure 19: A figure illustrating our preferred solution for the car park using inspiration from the solution of algorithm 2 in figure 18b. This solution has 333 parking slots within the car park. The red stars are the parking slots that we remove along the path out method, the green stars are the ones that we remove from the solution in figure 18b and the black star represents a parking slot that we have reinserted after the path out algorithm. The black arrows represent the direction in which the cars in the car park should drive. We see that we have succeeded to create a new solution which improves upon the current one with 48 parking slots, and this is, therefore, a desirable alternative to the existing car park.

7 Discussion

7.1 Theoretical approach

In this paper, we have shown that one can use linear programming and operations research to model our problem. This is a theoretical approach, which is nice because we can prove that

our solution is optimal. The problem with this approach is that it is very computationally heavy, so it is not usable to model a full-scale problem. As we argued in section 3.2.3 the time complexity increases exponentially for both our methods.

Our models aren't complete since they still have problems that we have not fixed. One of the big holes in our approach is that it only accounts for the placement of parking slots such that they don't overlap with each other. We haven't introduced a way to ensure that there exists a path from any given parking slot to the out edge, which is needed for a completely legal solution. Another problem is that the discrete approach doesn't ensure an optimal solution, and the continuous approach can only orient the slots in two ways.

The reason why we haven't looked into and fixed these problems is that even with the simple models, we can't solve full-scale problems within a reasonable time. If there are some breakthroughs within the theoretical approaches (E.g. a way to recognize if two graphs are isomorphic in polynomial time), it might be worth looking further into these approaches.

7.2 Results

As mentioned in the previous section, the analyses of the theoretical approach using operational research turned out to be rather computationally expensive. Therefore, we had to approach the problem in another way to obtain a solution. We made up three greedy heuristics. These heuristics perform well in terms of the running time even for large car parks, and, therefore, the big advantage is that we can generate a solution in less than 0.2 seconds, as seen in table 4. Furthermore, those heuristics work for all non-intersecting car parks, which generally speaking includes all car parks from the real world. The disadvantages of these algorithms are unfortunately many. One of the biggest disadvantages is that we are not guaranteed any optimal conditions. Furthermore, the algorithms only save one solution which has the highest number of parking slots inside the boundary of the car park. If the algorithms generate a different solution with the same number of parking slots, this is not saved since we only change the solution if it is better. This could be a huge mistake since the new solution could be better in terms of other criteria such as comfort and complexity, but the algorithms do not handle these aspects. The fact that comfort is important was also pointed out in a mail by an employee from NCC Danmark A/S, see [1]. This was also why we quickly stopped the development of the algorithms 4 and 5 in section 4.5 since we realized that a car located in the center of the car park had a long and complicated road to the out edge.

In section 6 we always use the path out algorithm on the best solution from each heuristic algorithm. This is a fairly intuitive approach, but this might not be the smartest if we want to end up with the solution with the most parking slots after the path out algorithm. We could have cases where the path out method would remove fewer parking slots on another heuristic solution, that initially may not be as good as the best, but after the path out algorithm might be better. The most ideal and easy fix to this would be to run the path out algorithms on all generated solutions, but as we see in table 5 the amount of solutions is large. Therefore, this was simply not possible to do due to the time it would take to use the path out algorithm on all solutions generated in the section 6.

To test our code, we tried to optimize three different existing car parks in section 6. Stades

Krog is the smallest and most intricate of the three. Our heuristics are not designed with this kind of car park in mind, which was highlighted when we not only failed to optimize the existing solution, but the outputs of our algorithms were very far from being legal.

Engelsborgvej on the other hand is more regular in its shape, and also quite a lot larger (it holds almost twice as many parking slots as Stades Krog). It was also easier to optimize it and we got a quite good solution from algorithm 2 that could be modified to a legal solution relatively easily. That being said, there is still some wasted space in the top right and left corners of the modified solution, see figure 16, which would indicate that our solution is good, but there exists an even better optimal solution. This is also backed up by the density of the modified solution, which was equal to 0.454 while the upper limit of the density when placing rows was calculated as 0.616.

The last car park was the one in front of IKEA Taastrup. As this is a large car park without any intricate corners, we would assume that optimizing this would be easier for our algorithms than the other two car parks. This was also the case when we looked at our solution, which has 16.8% more parking slots than the current solution, and an impressive density of 0.523. However, the reason for the high improvement is not just because our algorithms are smart, but also that the existing car park takes other factors into account, e.g. it has a large road going through the bottom of the car park for easier entry and exit. It is also worth noting that the density of 0.523 is the highest of all the car parks, and is quite close to the theoretical maximum of 0.616. This is also what we would expect as we can get the theoretical maximum in the infinite car park, and the larger the car park is, the closer it is to the infinite car park. The size of the car park also means that the angle of the parking slots is very close to what we calculated as the optimal angle in section 2.2.

If we should conclude from these three cases, it would be this: We can easily utilize the interior of a car park effectively but we have trouble dealing with edges and corners.

7.3 Applications

We believe that the methods, heuristics, and code that comes with the paper could be useful to other people. Among others, we think that the heuristic approach is a fairly good way to generate a lot of ideas, and could therefore work as an inspirational tool. The architects that design car parks now-a-day get the boundary of the car park, and start designing it from scratch. With our code and heuristics we see in table 7, 8 and 9 that they would be able to generate 2626038 solutions in less than 45 hours, 656190 solutions in less than 9 hours and 656190 solutions on a larger car park in approximately 13 hours. With even more heuristics we would be able to create even more solutions. Of course, many of these solutions are not worth much, but many of them also are. As mentioned before, the code could be modified such that we saved top X solutions from each algorithm such that the amount of solutions we get is X times the number of algorithms we use. These solutions could all be shown to the architect and, thereby, the architect could quickly, depending on the number of edges and size of the car park, have many solutions to look at.

Furthermore, we also think that the algorithms in this paper could be used to solve other problems than optimizing car park designs. Similar problems could be optimization of warehouses such that the number of storage racks is optimized or optimization of a marina (almost the same problem as the car park design problem). Furthermore, the algorithm

could also be used in the cutting and packing problem described in section 3.3. The cutting and packing problem is similar to the car park design problem but simpler since we have no demands of roads from parking slots to an out edge and no free space in front of the parking slot. This problem could be solved with our heuristics by simply letting $f = 0$ and using our algorithms.

7.4 Further perspectives

To develop further upon this project an obvious approach would be to consider other heuristics or variations of the current heuristics, that might solve the problem better or as well but with a different design. An example of a variation of our heuristics one could consider implementing a heuristic that places more than one row of parking slots along the first side. But before implementing new heuristics, it is worth considering if the current heuristics should be optimized. Currently, we solve the same problem multiple times after having placed almost identical first sides with only a few angles difference. The path out algorithm could also be optimized to find a smarter way to delete parking slots, than simply using brute force and considering all combinations. Besides boosting the performance of the algorithm, the algorithms could also be made much more robust. Currently, we have algorithms that work fine in the general case, but can easily be broken in edge cases.

It would also be interesting to take another approach to the problem as a whole. An example of such an approach we would be excited about trying is a machine learning approach. Here we would discretize the problem, like in the discrete OR model described in section 3.2.1, such that we have a grid of nodes. Here we would build a model which can choose which of these points to use and which to discard. As each car park is unique, reinforcement learning might be a good machine learning approach to the problem. It is not certain that machine learning is a good approach, but it would be interesting to research the possibility.

8 Conclusion

Throughout this paper, we have seen that it is very easy to find a feasible design for a car park but much harder to find a good one. Through a theoretical approach, we saw that there is a potential to model this very complex problem using operations research, and through such an approach achieve an optimal solution. However, this approach is computationally very expensive and thus it is not possible to solve cases that might be applicable in a real-life scenario.

Our main focus in this paper has been to develop heuristics that can generate good and feasible solutions for a given car park. We developed a variety of heuristics but chose to dedicate ourselves to three variants, which we tested on three existing car parks. Through these three car parks, we saw that the heuristic which places the parking slots with an angle created the best solutions. The algorithm that first filled a side with angled parking slots, outperformed the other two in the number of parking slots. Unfortunately, the solutions this algorithm gave were infeasible, so we ended up using the algorithm that places angled parking slots without considering a first side for our modified solutions. From Stades Krog we learned that our heuristics are bad at handling small and intricate car parks. From

IKEA's car park, we see that when we have a more regular shape and a larger interior our heuristics perform remarkably better. Here we were able to get a density of 0.523 which is quite close to the theoretical maximum for placing angled slots which is 0.616.

Although our algorithms gave quite useful results, none of them were completely legal, meaning that we had to make modifications to get a design that could be implemented in the real world. We would need to work further with the algorithms if we wanted to get legal solutions directly from them. The current versions of the heuristics still have their use in creating a solution that an architect could work from, but as they are now, they are only a tool and not the entire solution.

9 Bibliography

[1] NCC Danmark A/S and Marcus Førby Petersen. *Private e-mail conversation with NCC*. Mar. 2022.

[2] Ramón Alvarez-Valdes, Maria Antónia Carravilla, and José Fernando Oliveira. "Cutting and packing". eng. In: *Handbook of Heuristics* 2-2 (2018). Ed. by Mauricio G. C. Resende, Panos M. Pardalos, and Rafael Martí, pp. 931–977. DOI: 10.1007/978-3-319-07124-4_43.

[3] Oscar Juul Andersen and Marcus Førby Petersen. *The MatLab and Julia code for this project posted on GitHub*. 2022. URL: https://github.com/marcusforby/Optimization-of-Car-Park-Designs.

[4] Alain Billionnet. "Integer programming to schedule a hierarchical workforce with variable demands". In: *European Journal of Operational Research* 114.1 (1999), pp. 105–114. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(98)00182-9. URL: https://www.sciencedirect.com/science/article/pii/S0377221798001829.

[5] Sándor P. Fekete and Jörg Schepers. "A combinatorial characterization of higher-dimensional orthogonal packing". In: (2003). URL: http://arxiv.org/abs/cs/0310032.

[6] Sándor P. Fekete, Jörg Schepers, and Jan C. van der Veen. "An Exact Algorithm for Higher-Dimensional Orthogonal Packing". In: *Operations Research* 55.3 (2007), pp. 569–587. DOI: 10.1287/opre.1060.0369.

[7] David Grace, Alessandro Waldron, and Tahir Ahmad. "On the Travelling Salesman Algorithm: An Application". eng. In: (2013). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.294.8422.

[8] Frederick S. Hillier & Gerald J. Lieberman. *Introduction to operations research*. Eleventh. McGraw-Hill, 2021. ISBN: 9781260575873, 1259872998.

[9] Trafik og Mobilitet Lyngby-Taarbæk and Marcus Førby Petersen. *Private e-mail conversation with Lyngby-Taarbæk kommune*. Apr. 2022.

[10] Kommunalbestyrelsen 2022-2025, Lyngby-Taarbæk municipality. *REFERAT 31. MARTS 2022 KL. 17.00, punkt 22*. Mar. 2022. URL: https://dagsordener.ltk.dk/vis?id=59bc3fb6-9486-4fa8-8f07-5da6d7166c9b.

[11] Sandor P. Fekete and Jörg Schepers. "On more-dimensional packing I: Modeling". eng. In: (2000). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.9486.

[12] Richard Porter et al. "Optimisation of Car Park Designs". In: *Report from the 91st European Study Group with Industry, Bristol University* (2013).

[13] Sidsel Klarskov Sørensen. "The Parking Space Problem, Parkeringspladsproblemet". dan. 2014.

[14] Daniel A. Spielman and Shang-Hua Teng. "Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time". eng. In: (2003).

[15] Vejdirektoratet. *Udformning af parkeringsanlæg til personbiler*. Vejdirektoratet. Jan. 2019. URL: http://vejregler.lovportaler.dk/static/MayflowerImageCache.aspx?blobid=vd20190008.pdf.

[16] Wikipedia contributors. *Shoelace formula — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Shoelace_formula&oldid=1085850819. [Online; accessed 24-May-2022]. 2022.

[17] Laurence A. Wolsey. *Integer programming*. eng. Wiley, 1998, p. 168. ISBN: 0471283665, 9780471283669.

10 Appendix

10.1 Result from the operations research problems

Generally, the Julia code has only been developed to work for rectangles. This was not further expanded to arbitrary convex polygons since we saw the performance was poor for a rectangle.

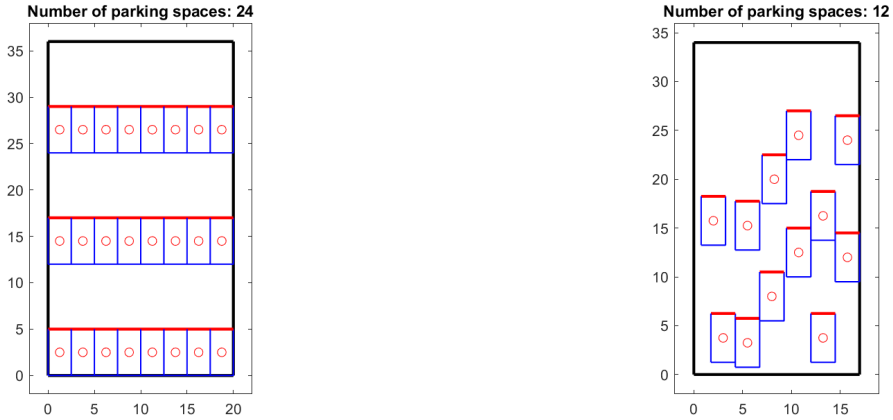
10.1.1 Discrete x- and y-variables

Given that the user defines $x_{min}, x_{max}, y_{min}, y_{max}, interval$ and parking_size, we can use the following code from Julia, see [3] to set up and solve the mixed-binary LP-program:

```
using JuMP, Cbc, DelimitedFiles

######## USER DEFINED VARIABLES ########
xmin = ?
xmax = ?
ymin = ?
ymax = ?
interval = ?
parking_size = [? ? ?]
########################################


## LOADING VARIABLES
m = Model(Cbc.Optimizer)
w = parking_size[1]
h = parking_size[2]
f = parking_size[3]
area = [xmin ymin; xmax ymin; xmax ymax; xmin ymax];
ny = Int(floor((ymax-ymin-h-f)/interval))+1
nx = Int(floor((xmax-xmin-w)/interval))+1
n = ny*nx
M = max(ymax-ymin, xmax-xmin)
xs = collect(range(xmin+w/2, xmax-w/2, length=nx))
ys = collect(range(ymin+h/2, ymax-h/2-f, length=ny))

X = []
for i=1:ny
    append!(X, xs)
end

Y = []
for i=1:ny
    append!(Y, ys[i]*ones(nx))
end

## DEFINING VARIABLES
@variable(m, z[1:nx,1:ny], Bin)

## DEFINING OBJECTIVE FUNCTION
@objective(m, Max, sum(z[i, j] for i=1:nx for j=1:ny))

## DEFINING CONSTRAINTS
for i=1:nx
    for j=1:ny
        x_coordinates = [i]
        y_coordinates = [j]
        for b=i:nx
            for g=j:ny
```

```
                if b != i || g != j
                    if w > abs(xs[b] - xs[i])
                        if h + f > abs(ys[j] - ys[g])
                            append!(x_coordinates, b)
                            append!(y_coordinates, g)
                        end
                    end
                end
            end
        end
        @constraint(m, sum(z[x_coordinates[l], y_coordinates[l]] for ...
            l=1:length(x_coordinates)) <= 1)
    end
end

## OPTIMIZE
optimize!(m)
```

The results of two examples using this Julia code can be seen in figure 20.



(a) An illustration of how a car park is filled, when the parameters we run the program with allows a tight packing. Here $x_{min} = 0$, $x_{max} = 20$, $y_{min} = 0$ and $y_{max} = 36$.

(b) An illustration of how a car park is filled when the parameters we run the program with do not allow for tight packing. Here $x_{min} = 0$, $x_{max} = 17$, $y_{min} = 0$ and $y_{max} = 34$. Even though the placement may not seem optimal, we can't fit any more parking slots, since the width only allows six slots, and the height only allows two slots.

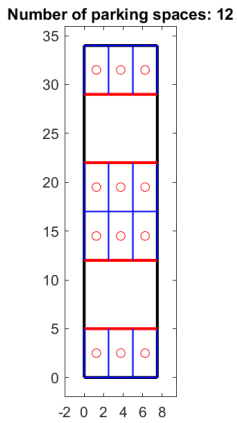
Figure 20: A figure illustrating how a car park is filled when we use the discrete LP-problem from section 3.2.1. The car park is simply a rectangle defined by minimum and maximum x and y values. The interval size of the grid is set to be 0.25 in both figures, and the parking sizes is set to be $w = 2.5$, $h = 5$ and $f = 7$.

10.1.2 Continues x- and y-variables

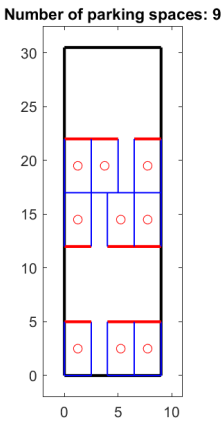
Given that the user defines $x_{min}, x_{max}, y_{min}, y_{max}, n$ and parking_size, we can use the following code from Julia, see [3], to set up and solve the mixed-binary LP-problem:

```
using JuMP, GLPK, DelimitedFiles

######## USER DEFINED VARIABLES ########
xmin = ?
xmax = ?
ymin = ?
ymax = ?
n = ?
parking_size = [? ? ?]
########################################

## LOADING VARIABLES
m = Model(GLPK.Optimizer)
w = parking_size[1]
h = parking_size[2]
f = parking_size[3]
area = [xmin ymin; xmax ymin; xmax ymax; xmin ymax]
M = max(ymax-ymin, xmax-xmin)

## DEFINING VARIABLES
@variable(m, x[1:n])
@variable(m, y[1:n])
@variable(m, z[1:n], Bin)
@variable(m, 0<=r[1:n]<=1, Bin)
@variable(m, 0<=b[1:n, 1:n]<=1, Bin)
@variable(m, 0<=uxy[1:n, 1:n]<=1, Bin)

## DEFINING OBJECTIVE FUNCTION
@objective(m, Max, sum(z[i] for i=1:n))

## DEFINING CONSTRAINTS
@constraint(m, [i=1:n], xmax - w/2 >= x[i] >= w/2 + xmin) # (1)
@constraint(m, [i=1:n], ymax - h/2 - r[i]*f >= y[i]) # (2)
@constraint(m, [i=1:n], y[i] >= h/2 + (1-r[i])*f + ymin) # (3)
@constraint(m, [j=2:n, i=1:j-1], w <= x[j] - x[i] + (2 - z[i] - z[j])*M + uxy[i, j]*M) # (4)
@constraint(m, [j=2:n, i=1:j-1], h + f*(1-b[i, j]) <= y[j] - y[i] + (2 - z[i] - z[j])*M + (...
    1-uxy[i, j])*M) # (5)
@constraint(m, [i=1:n-1], 0 <= y[i+1] - y[i]) # (6)
@constraint(m, [j=2:n, i=1:j-1], b[i, j] <= r[j]) # (7)
@constraint(m, [j=2:n, i=1:j-1], b[i, j] <= (1 - r[i])) # (8)
@constraint(m, [i=1:n-1], z[i] >= z[i+1]) # (9)

## OPTIMIZE
println("Optmizing")
optimize!(m)
```

The results of two examples can be seen in figure 20.



(a) An illustration of how the car park is filled, when the parameters we run the program with allows a tight packing. Here $x_{min} = 0$, $x_{max} = 7.5$, $y_{min} = 0$ and $y_{max} = 35$.



(b) An illustration of how a car park is filled, when the parameters we run the program with does not allow a tight packing.Here $x_{min} = 0$, $x_{max} = 9$, $y_{min} = 0$ and $y_{max} = 30$.

Figure 21: A figure illustrating how a car park is filled when we use the continuous LP-problem from section 3.2.2. The car park is simply a rectangle defined by minimum and maximum x and y values. The number of parking slots we try to fit is set to be $n = 13$ in both figures, and the parking sizes is set to be $w = 2.5$, $h = 5$ and $f = 7$.

10.2 Boundary heuristic illustration

10.2.1 Placement of a single row of parking slots on the boundary

When looking at an arbitrary polygon and we want to place parking slots along all the edges, we need to know where to start placing parking slots when we move from side to side. This can be calculated by looking at two scenarios, namely one where the angle between the two sides is greater than 90 degrees and one where the angle is lesser than 90 degrees.

First, we look at the case where the angle is less than 90 degrees, which can be seen in figure 22. Here α is the angle that is known, and the problem is to find the coordinate A, as that, together with the angle, is enough to determine the placement of the next parking slot. The blue rectangles mark the parking slots, and the green area marks the free space needed to get out of the parking slot.

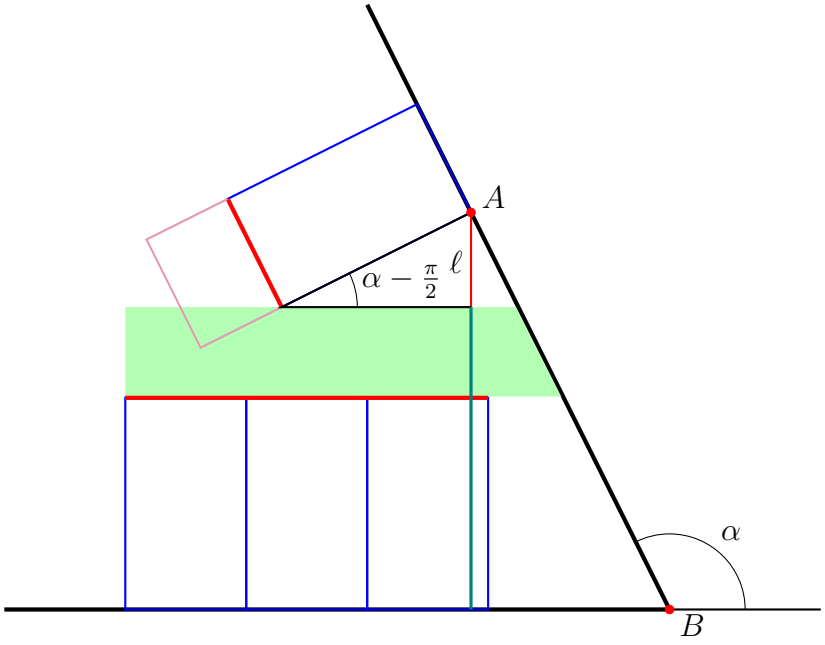


Figure 22: A figure illustrating the lowest possible position of point A when α is greater than 90 degrees. The green area is the free space needed from the previous row in which we are not allowed to place parking slots. The red box in front of the new parking slot we want to place is the new parking slots free space. Furthermore, some other lengths and polygons have been drawn that are used to determine the value of A.

To find the coordinates of A, we first look at the small triangle containing ℓ. Here we want to find the length of the side ℓ. This can be calculated by the following:

$$\ell = \sin\left(\alpha - \frac{\pi}{2}\right) \cdot h$$

Here h is the height of a parking slot. Now we can calculate the length of the side $|AB|$ by the following:

$$|AB| = (\ell + h + f) / \cos\left(\alpha - \frac{\pi}{2}\right)$$

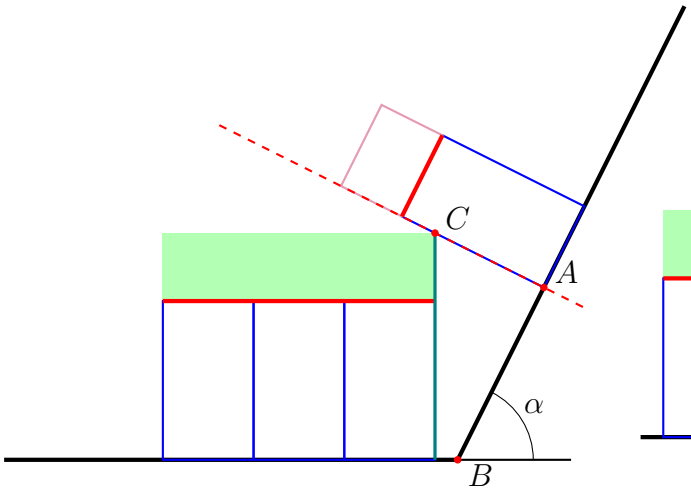
Here f denotes the length of the free space needed in front of the parking slot. The length of the side $|AB|$ however is not enough to determine the coordinates of A. The length of $|AB|$ can be used together with the vector v which is a direction vector between B and A, to calculate A as the following:

$$A = v \cdot |AB| + B$$

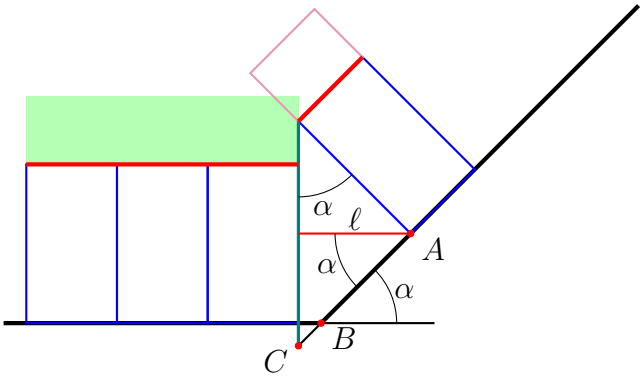
Here v can be found by simply rotating the vector $[1, 0]^T$ by α degrees in a positive direction. Since the coordinates of B are known, we have now derived a method to get the coordinates of A in this case.

Now we have covered the case where α is greater than 90 degrees, and want to look at the case where α is less than 90 degrees. Here we have three cases that are illustrated in figure 23. In figure 23a we have illustrated the case where the new parking slot hits the corner of

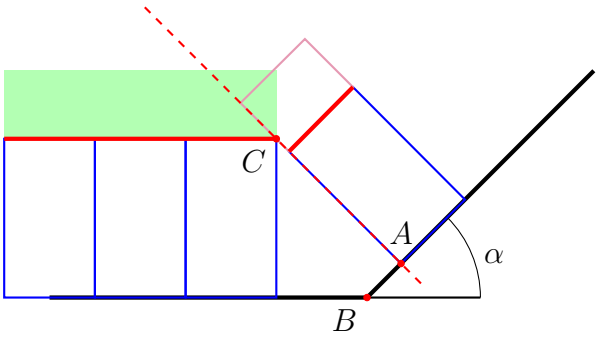
the free space from the previous parking slots. In this case, the coordinate of A is simply given by the intersection of the red dashed line in figure 23a, and the line passing through A and B. This can be calculated easily as we know all direction vectors of both lines and the points B and C.



(a) Here the corner of the free space from the previous parking slots hits the side of the new parking slot.



(b) Here the corner of the new parking slot hits the side of the free space of previous parking slots.



(c) Here the corner of the free space of the previous parking slots hits the side of the free space of the new parking slot.

Figure 23: A figure illustrating the lowest possible position of point A when α is less than 90 degrees. The green area is the free space needed from the previous row in which we are not allowed to place parking slots. The red box in front of the new parking slot we want to place is the new parking slots free space. Furthermore, some other lengths and polygons have been drawn which are used to determine the value of A.

Figure 23b illustrates the case where the new parking slot hits the side of the free space

of the previous parking slots. In this case, the side length can be calculated as the following:

$$\ell = \sin(\alpha) \cdot h \tag{1}$$

$$|AB| = \frac{\ell}{\cos(\alpha)} - |BC| \tag{2}$$

Here we can find the length of the side BC when we know both points. B is already known, and C can easily be found as the intersection between two lines. The one line has a direction orthogonal to the previous side containing the points on the rightmost side of the last parking slot. The other line is parallel to the next side and contains the point B.

The last case is illustrated in figure 23c. In this case, the corner of the new parking slot hits the side of the free space of the previous parking slots. Here we see that finding the coordinates to A corresponds to finding the intersection between two straight lines: the line which passes through C and A, and the line which passes through B and A. Since we know the coordinates of B and C, we only need to find the slope of the lines. We do this by using the fact that we know the angle α. This gives a slope of $\tan(\alpha)$ for the line passing through B and A, and a slope of $-\tan(\pi/2 - \alpha)$ for the line passing through C and A. Having determined these two lines we can easily find the intersection and thereby find A.

Between the three cases in figure 23 we take the minimum of the outputs.

10.2.2 Placement of double rows of parking slots on the boundary

Placing new lower row

We choose to look at the rows independent of each other. We first look at where the lower row should start according to the previous lower row. When α is greater than 90 degrees we almost have the same situation as in figure 22. The only difference is that the green area doesn't exist and, therefore, the green side is shorter, since it is equal to h. We, therefore, calculate ℓ in the same way as previously, but the following is different:

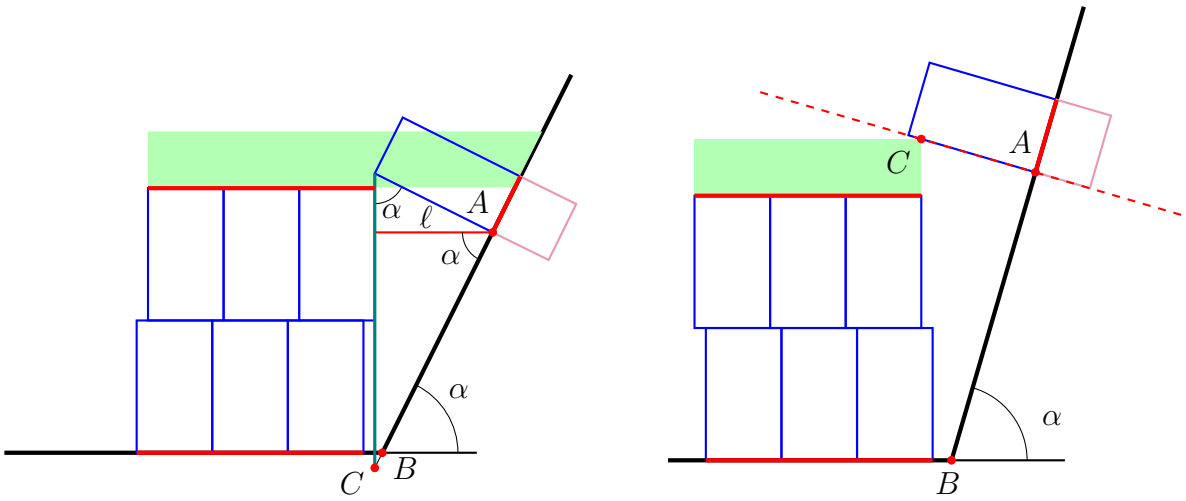
$$|AB| = (\ell + h)/\cos\left(\alpha - \frac{\pi}{2}\right)$$

When α is less than 90 degrees, we have the exact situations from figure 23b and 23c. Therefore, the only difference from previous algorithms is that the green area is gone, and the case in figure 23a therefore disappears.

We now consider how to place the first row with respect to the previous second row. When α is greater than 90 degrees, we also have a similar case as in figure 22. The only difference is that the green side now is larger since it is equal to $2 \cdot h + f$. We, therefore, calculate ℓ in the same way as previously, but the following is different:

$$|AB| = (\ell + 2 \cdot h + f)/\cos\left(\alpha - \frac{\pi}{2}\right)$$

The case where α is less than 90 degrees is illustration in figure 24.



(a) Here the corner of the new parking slot hits the side of the free space from the previous parking slots.

(b) Here the side of the new parking slot hits the corner of the free space from the previous parking slots.

Figure 24: A figure illustrating the lowest possible position of point A when α is greater than 90 degrees, and we want to place the first row on the double boundary heuristic. The green area is the free space needed from the previous row in which we are not allowed to place parking slots. The red box in front of the new parking slot we want to place is the new parking slots free space. Furthermore, some other lengths and polygons have been drawn which are used to determine the value of A.

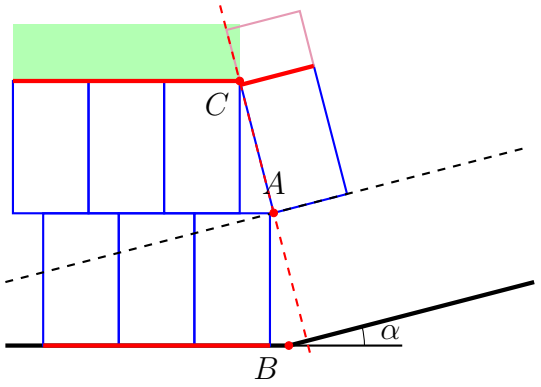
Those look a lot like previous cases, and the geometry is also pretty much the same. Between those two cases in figure 24 we take the minimum of the outputs.

Between the two results, which are the outputs of checking each previous row independently, we take the maximum of those values, and this is the length we need to move from point B to point A.

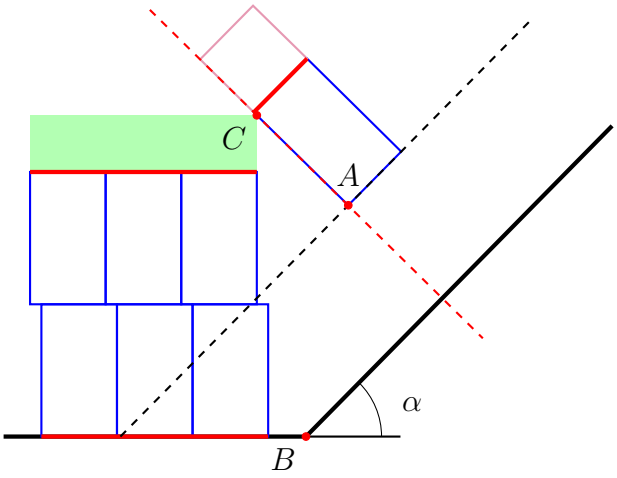
Placing new upper row

When placing the new upper row, we again look at the previous two rows separately. First, we look at how to place the second row when we look at the previous lower row. When α is greater than 90 degrees, then there are no cases for placing parking slots according to the first row. This is because that parking slot needs to be above $2h + f$, and, therefore, it is irrelevant to calculate anything according to the first row. This also mean that the situation is almost the same as in figure 22 with the small modification that the green side in figure 22 is equal to $2h + f$.

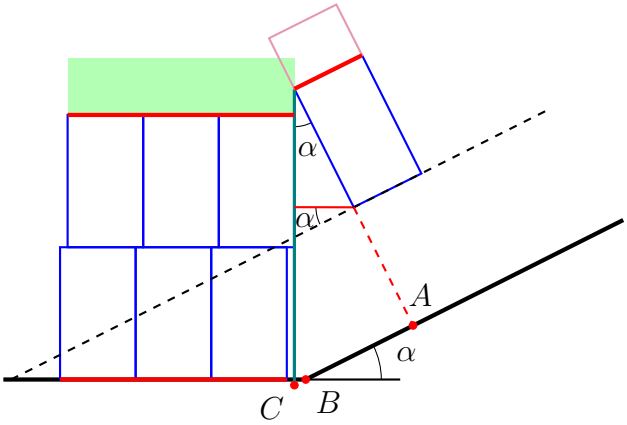
When α is less than 90 degrees and we want to look at how to place a new parking slot in regard to the previous upper row, we have the same three situations as in figure 23. Though, we have small modifications, see figure 25.



(a) Here the side of the free space from the new parking slot hits the corner of the the previous parking slots.



(b) Here the corner of the free space from the previous parking slots hits the side of the new parking slot.



(c) Here the corner of the new parking slot hits the side of the free space from the previous parking slots.

Figure 25: A figure illustrating the lowest possible position of point A when α is greater than 90 degrees, and we want to place the second row on the double boundary heuristic. The green area is the free space needed from the previous row in which we are not allowed to place parking slots. The red box in front of the new parking slot we want to place is the new parking slots free space. Furthermore, some other lengths and polygons have been drawn which are used to determine the value of A.

The math here is nearly the same as previously stated since we use interpolation in cases 1 and 2, and trigonometry in case 3. Some of the sides have a changed size in case 3, but the method is the same as stated before.

10.3 The density function

10.3.1 Density of the infinite car park

As mentioned in section 2, we used Newton's algorithm to get that the optimal angle is given by 26.99 degrees. We now want to calculate the density of the infinite corridor. One might think that we can use the expression for ρ stated in section 2, but this is unfortunately not true. This is because figure 3 doesn't take into account that two parking slots can share the free space if they are placed opposite of each other, see figure 26. Therefore, we see that the green area increases in size, and the number of parking slots are doubled. Therefore, we can calculate the green area in the following way:

$$G(\alpha) = (\ell + 2 \cdot h + k) \cdot w = \left(\frac{L(\alpha)}{\cos(\alpha)} + 2 \cdot h + \tan(\alpha) \cdot w \right) \cdot w$$

Hence, we can calculate the density in the following way:

$$\rho(\alpha) = \frac{2 \cdot h \cdot w}{G(\alpha)} = \frac{2 \cdot h \cdot w}{\left(\frac{L(\alpha)}{\cos(\alpha)} + 2 \cdot h + \tan(\alpha) \cdot w \right) \cdot w} = \frac{2 \cdot h}{\frac{L(\alpha)}{\cos(\alpha)} + 2 \cdot h + \tan(\alpha) \cdot w}$$

When interpolating between the values in table 1, we get the following:

$$L(\alpha) = \left(\frac{4.2 - 5.5}{30 - 15} \right) \alpha + \left(4.2 - \left(\frac{4.2 - 5.5}{30 - 15} \right) \cdot 30 \right) = -0.0867\alpha + 6.8$$

when $\alpha = 26.99$ degrees, i.e. $\alpha \in [15, 30]$. Hence, $L(26.99) = 4.461$. Therefore, we get the following using $w = 2.4$ and $h = 5$:

$$\rho(\alpha) = \frac{2 \cdot 5}{\frac{4.461}{\cos(26.99°)} + 2 \cdot 5 + \tan(26.99°) \cdot 2.4} = 0.616$$

Please note that one could also calculate the optimum angle from figure 26, but this yield the exact same angle as from the illustration in figure 3.

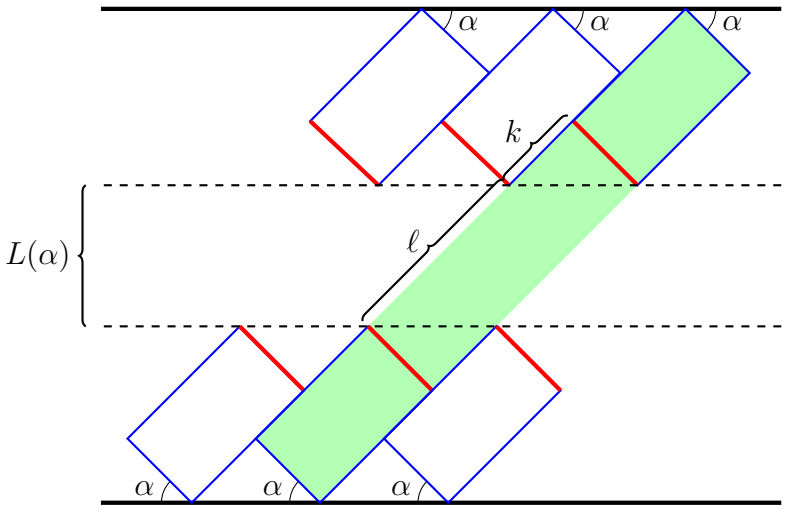


Figure 26: A figure illustrating the infinite corridor for a given α consisting of two rows of parking slots, and the free space needed in front of them. It can be seen as infinitely many copies of the green rectangle lying next to and back-to-back of each other.

10.3.2 Shoelace formula

We want to calculate the density of the solution we achieve by our algorithms. To calculate the density, we use the following formula:

$$\rho = \frac{h \cdot w \cdot n}{\text{area of the parking area}}$$

w and h are simply user-defined constants illustrated in figure 1, and n is the number of parking slots in our solution. We, therefore, need a formula for calculating the area of an arbitrary non-intersecting polygon. To do this, we use Shoelace formula[16]. Shoelace formula is the following given a car park with m corner-points:

$$Area = \frac{1}{2}\left|\sum_{i=1}^{m} x_i y_{i+1} - \sum_{i=1}^{m} x_{i+1} y_i\right|$$

Please note that $x_{m+1} = x_1$ and $y_{m+1} = y_1$, and the $|\cdot|$ around the expression is the absolute value.

Since we define a parking area to be a set of corner-points (x_i, y_i), we simply have a list of $\mathbf{x} = [x_1, \ldots, x_m]$ and $\mathbf{y} = [y_1, \ldots, y_m]$. Therefore, we see that we can calculate the area of the car park in the following way:

$$Area = \frac{1}{2}\left|\mathbf{x}^{\mathrm{T}} \cdot [\mathbf{y}(2:\text{end}), y_1] - [\mathbf{x}(2:\text{end}), x_1]^{\mathrm{T}} \cdot \mathbf{y}\right|$$

Here $\mathbf{x}(2:\text{end})$ denotes $[x_2, \ldots, x_m]$.

10.3.3 Stades Krog

The best solution for Stades Krog is the solutions from either algorithms 2 or 3. The solutions have $n = 69$ and we used $w = 2.4$ and $h = 5$. We calculate the area of the car park with the Shoelace formula described in section 10.3.2. This gives us the following:

$$Area = 2195.8$$

Therefore, the density of our solution becomes the following:

$$\rho = \frac{2.4 \cdot 5 \cdot 69}{2195.8} = 0.377$$

The density of the current solution with 76 parking slots has the following density:

$$\rho = \frac{2.4 \cdot 5 \cdot 76}{2195.8} = 0.415$$

10.3.4 Engelsborgvej

The best solution for Engelsborgvej is the solution from algorithm 2. The solution has $n = 149$ and we used $w = 2.4$ and $h = 5$. We calculate the area of the car park with the Shoelace formula described in section 10.3.2. This gives us the following:

$$Area = 3670.62$$

Therefore, the density of our solution becomes the following:

$$\rho = \frac{2.4 \cdot 5 \cdot 149}{3670.62} = 0.487$$

The density of the current solution with 125 parking slots has the following density:

$$\rho = \frac{2.4 \cdot 5 \cdot 125}{3670.62} = 0.409$$

The density for our modified solution with 139 parking slots is equal to the following:

$$\rho = \frac{2.4 \cdot 5 \cdot 139}{3670.62} = 0.454$$

10.3.5 IKEA

The best solution for IKEA is the solution from algorithm 2. The solution has $n = 339$ and we used $w = 2.4$ and $h = 5$. We calculate the area of the car park with the Shoelace formula described in section 10.3.2. This gives us the following:

$$Area = 7637.2$$

Therefore, the density of our solution becomes the following:

$$\rho = \frac{2.4 \cdot 5 \cdot 339}{7637.2} = 0.532$$

The density of the current solution with 285 parking slots has the following density:

$$\rho = \frac{2.4 \cdot 5 \cdot 285}{7637.2} = 0.448$$

The density for our modified solution with 333 parking slots is equal to the following:

$$\rho = \frac{2.4 \cdot 5 \cdot 333}{7637.2} = 0.523$$

10.4 Pseudocodes

10.4.1 Heuristic

Algorithm 2 Finding x-value intervals in a row

function FIND_X_INTERVALS(y_{low}, y_{high}, area)
    n ← number of edges
    x ← all x-values in area
    y ← all y-values in area
    x_{vals} ← empty list
    for $i = 1$ to $n - 1$ **do**
        if y_{low} or $y_{\text{high}} \in [y_{\text{i}}, y_{\text{i+1}}]$ **or** $y_{\text{i}} \in [y_{\text{low}}, y_{\text{high}}]$ **then**
            append interpolation between x_{i} and $x_{\text{i+1}}$ to x_{vals}
    Sort x_{vals} in increasing order
    $x_{\text{intervals}}$ ← empty list
    found_interval_beginning ← False
    n_{xvals} ← length of x_{vals}
    bot ← 1, top ← 2
    while bot $< n_{\text{xvals}}$ and top $\leq n_{\text{xvals}}$ **do**
        if we can draw a vertical line at $(x_{\text{vals}}[i] + x_{\text{vals}}[i+1])/2$ from y_{low} to y_{high} without leaving the area **then**
            top ← top + 1
            found_interval_beginning ← True
        else if found_interval_beginning **then**
            append x_{vals}[bot] and x_{vals}[top − 1] to $x_{\text{intervals}}$
            bot ← top
            top ← top + 1
            found_interval_beginning ← False
        else
            bot ← bot + 1
            top ← top + 1
    if found_interval_beginning **then**
        append x_{vals}[bot] and x_{vals}[top − 1] to $x_{\text{intervals}}$

Algorithm 3 Heuristic 1 - Polygon straight

```
1: function HEURISTIC_POLYGON_STRAIGHT(parking_size, area, index)
2:     area ← rotate area such that side index lies on the x-axis
3:     h, f ← load sizes from parking_size
4:     y_max ← find highest y-value in area
5:     y_low ← 0
6:     y_high ← h+f
7:     x_interval ← find_x_intervals(y_low, y_high, area)
8:     for interval ∈ x_interval do
9:         insert parking slots faced up in interval
10:    while y_high < y_max do
11:        update y_low and y_high
12:        x_interval ← find_x_intervals(y_low, y_high, area)
13:        for interval ∈ x_interval do
14:            insert parking slots faced down in interval
15:        update y_low and y_high
16:        x_interval ← find_x_intervals(y_low, y_high, area)
17:        for interval ∈ x_interval do
18:            insert parking slots faced up in interval
```

Algorithm 4 Heuristic 2 - Polygon straight with arbitrary angle

```
1: function HEURISTIC_POLYGON_ANGLE(parking_size, area, index, angle, box)
2:     area ← rotate area such that side index lies on the x-axis
3:     y_max ← find highest y-value in area
4:     y_low ← 0
5:     y_high ← highest corner of a rotated parking slot
6:     x_interval ← find_x_intervals(y_low, y_high, area)
7:     for interval ∈ x_interval do
8:         insert parking slots faced up rotated with angle in interval and not in box
9:     while y_high < y_max do
10:        update y_low and y_high
11:        x_interval ← find_x_intervals(y_low, y_high, area)
12:        first_point ← safe first parking slot that can be inserted
13:        for interval ∈ x_interval do
14:            insert parking slots faced down rotated with 180 + angle in interval and not in box
15:        update y_low and y_high
16:        x_interval ← find_x_intervals(y_low, y_high, area)
17:        build first parking slot in new row from first_point s.t. the two rows are aligned
18:        for interval ∈ x_interval do
19:            insert parking slots faced up rotated with angle in interval and not in box
```

Algorithm 5 Heuristic 3 - Polygon straight with arbitrary angle with specific first side

```
1: function HEURISTIC_POLYGON_ANGLE_FS(parking_size, area, indexes, angles)
2:     index, index_fs ← load from indexes
3:     angle, angle_fs ← load from angles
4:     // Insert 1 row of parking slots using index_fs as bottom side
5:     area_fs ← rotate area such that side index_fs lies on the x-axis
6:     y_low ← 0
7:     y_high ← highest corner of a rotated parking slot using angle_fs
8:     x_interval ← find_x_intervals(y_low, y_high, area_fs)
9:     for interval ∈ x_interval do
10:        insert parking slots faced up rotated with angle_fs in interval
11:    box ← define a rectangle where all parking slots from the first side is inside
12:    // Insert many rows of parking slots using index as bottom side
13:    heuristic_polygon_angle(parking_size, area, index, angle, box)
```

Algorithm 6 Heuristic 4 - Polygon single slots in boundary

```
1: function HEURISTIC_BOUNDARY_SINGLE(parking_size, area, index)
2:     if area is too small then
3:         return [ ]
4:     area ← rotate area such that side index lies on the x-axis
5:     for sides in area do
6:         Place parking slots faced up on side
7:     new_area ← area not used within the current area
```

Algorithm 7 Heuristic 5 - Polygon double slots in boundary

```
1: function HEURISTIC_BOUNDARY_DOUBLE(parking_size, area, index)
2:     if area is too small then
3:         return []
4:     area ← rotate area such that side index lies on the x-axis
5:     for sides in area do
6:         Place parking slots faced down on side
7:         Place parking slots faced down down side on top of previous row
8:     new_area ← area not used within the current area
```

10.4.2 The path out algorithm

Algorithm 8 Path out algorithm

```
1: function PATH_OUT_ALGORITHM(init_solution, out_edges, area, parking_size)
2:     endpoints ← create point on out-edges
3:     V ← create grid
4:     E ← create edges
5:     first_points ← pick first-points
6:     old_points ← DFS_INIT(first_points, endpoints, G = {V, E})
7:     while all first_points can't get out do
8:         E ← add edges from first_point to center of slot
9:         E ← add edges between center of slots
10:        new_points ← DFS_INIT(first_points, endpoints, G = {V, E})
11:        remove_points ← empty
12:        for i in 1 to 10 do
13:            remove_points ← remove slots at order i, and check if more can come out
14:            if remove_points is not empty then
15:                break
16:        if remove_points is empty then
17:            return
18:        V ← update grid
19:        E ← update edges
20:        first_points ← update first-points
21:        old_points ← new_points
22:        new_points ← DFS_INIT(first_points, endpoints, G = {V, E})
```

10.5 Car parks corner-points

10.5.1 Stades Krog

x	y
41.2588655423	−10.085152642
35.9109231876	−20.389376676
37.7369928846	−46.9978208318
38.3891606335	−51.5629950742
41.2586987287	−57.5629383642
55.4759556551	−52.867330572
56.5194240533	−71.2584610915
69.8236461312	−69.6932584941
67.7367093347	−50.6499602257
81.6930991615	−49.0847576283
83.6496024083	−68.3889229962
100.73639743	−65.6498184508
97.8668593346	−46.867387282
93.8234192913	−47.1282543816
90.0408463476	−24.4328167193
53.7803195079	−28.7371238621
52.9977182092	−24.1719496197
56.7802911529	−14.7807340353

Table 10: A table listing the 18 corner-points for Stades Krog used throughout this paper.

10.5.2 Engelsborgvej

x	y
23.693477153	−12.2231989881
18.2524477671	−21.6347092772
23.8405484142	−54.5749298445
26.0463547253	−65.6041089094
97.2208233772	−53.1043832521
117.367311694	−18.1053743519
90.3092543927	−2.37052581277
79.499729116	−18.7441577853
47.0491283011	−24.6052795184

Table 11: A table listing the 9 corner-points for Engelsborgvej used throughout this paper.

10.5.3 IKEA

x	y
113.441561409	−13.3308421882
11.5662584181	−8.86056330403
9.68403573006	−54.7397413254
14.5072313682	−54.7397413254
13.3308421882	−88.8550275465
23.6830669725	−98.5014188228
92.1489172506	−84.6200264984
92.7371118406	−68.6211336499
111.088783049	−69.4446060759

Table 12: A table listing the 9 corner-points for IKEA used throughout this paper.