Methods for generating road networks in car parks

May 9, 2019

by Tim Roderick,
supervised by Dr Richard Porter

Mathematics Project
20 Credit Points at Level 6

School of Mathematics, University of Bristol

Contents

1 Introduction **4**
- 1.1 Background . . . 4
  - 1.1.1 Tile and Trim . . . 5
  - 1.1.2 Deterministic approach to optimising the road . . . 5
  - 1.1.3 Variational approach to optimising the road . . . 6
  - 1.1.4 Optimising over road networks . . . 6
- 1.2 Problem Statement . . . 7
  - 1.2.1 The Car Park . . . 7
  - 1.2.2 The Road Network . . . 7

2 The Deterministic Approach **7**
- 2.1 Details of the road network . . . 8
- 2.2 How the road network is created and expanded . . . 8
- 2.3 Attributes of the road network to consider . . . 9
- 2.4 Cost function for growing network . . . 10
- 2.5 Simplifying the approach . . . 11
  - 2.5.1 Boundary of the car park . . . 11
  - 2.5.2 How we grow the network . . . 12
  - 2.5.3 Representation of a car parking space . . . 12
- 2.6 Simplified cost function . . . 14
  - 2.6.1 Cost function for selecting the vertex to grow from . . . 14
  - 2.6.2 Cost function for deciding how we grow from a chosen vertex . . . 15
- 2.7 Growing the network . . . 17
  - 2.7.1 Choosing a vertex to grow from . . . 17
  - 2.7.2 Choosing the direction to grow from a chosen vertex . . . 18
- 2.8 Optimisation of parameters . . . 18
  - 2.8.1 The issue of optimisation . . . 19
  - 2.8.2 The brute force approach . . . 19
- 2.9 Explanation of code . . . 20
  - 2.9.1 Using “layers” to improve performance using memoisation . . . 20
  - 2.9.2 Implementation of the cost functions . . . 21
  - 2.9.3 Brute force minimisation of objective function . . . 22
- 2.10 Results . . . 22
  - 2.10.1 Square car park boundaries . . . 23
  - 2.10.2 Rectangular car park boundaries . . . 24
- 2.11 Result conclusions . . . 27
  - 2.11.1 The issue of parameters not being effective across different grid sizes 27
  - 2.11.2 Certain attributes being ignored . . . 27
  - 2.11.3 Overall approach conclusions . . . 28

3 The Stochastic Approach **28**
- 3.1 Explaining how we grow . . . 29
- 3.2 Formalising the rule sets for considerable new vertices . . . 29
  - 3.2.1 Completeness and relaxation of rules . . . 30
- 3.3 Filtering the candidate vertices . . . 31
- 3.4 Uncertainty and how we can use it in our filtering of potential vertices . . . 32
- 3.5 Growing the network . . . 33
  - 3.5.1 Pruning the road network . . . 34

3.6 Simulation . . . 35
3.7 Explanation of code . . . 36
3.7.1 Implementation of rule sets and relaxation of rules . . . 36
3.7.2 Ordering of filters in the filtering process . . . 38
3.7.3 Incorporating uncertainty into the filtering process . . . 39
3.7.4 Implementation of the cleaning process . . . 40
3.7.5 Brute force approach to optimising over our parameters . . . 42
3.8 Results . . . 42
3.8.1 Square car park boundaries . . . 43
3.8.2 Rectangular car park boundaries . . . 45
3.8.3 Result conclusions . . . 47
3.8.4 Performance in comparison to deterministic approach . . . 47
3.8.5 Parameter choice and general application . . . 48
3.8.6 Issue of how our rule choices negatively effect road network . . . 49
3.8.7 Overall approach conclusions . . . 49

4 Conclusions and possible extensions **50**
4.1 Testing both approaches with more complex car park configurations . . . 50
4.2 Using a generative model and the Fourier transform to construct the road network . . . 50
4.3 Using genetic algorithms to improve the road network structure . . . 51
4.4 Reinforcement learning as an approach to optimising the road network . . . 51

5 Appendices **51**
5.1 Python code for the Deterministic approach . . . 51
5.2 Python code for the Stochastic approach . . . 56

1 Introduction

As cities towns and villages expand, so does the scale of their urban developments. Supermarkets that are expected to serve thousands of people are being squeezed into spaces as small as a humble cal-de-sac. As you can imagine, this brings along many engineering problems that must be tackled for the development to function correctly. One of the main concerns is allocating enough car-parking space to serve the number of people.

The immediate question one may ask themselves upon entering a car park that is at capacity on a Tuesday evening is whether the design can be improved. Designers have constructed this car park with that goal in mind. Has the designer of this car park successfully maximised the number of car parking spaces? Rearranging the spaces to a different configuration may lead to more spaces being available. This project will focus on these configurations to find designs and layouts of car parks.

This project will not focus on bespoke car parks as with complete freedom to design a structure specifically for car parking, many issues can be mitigated. Instead this project will focus on what was described above, the allocating of car-parking spaces into an area that was not initially designed for such a task; retrofitting. The area in which these car-parking spaces are being allocated could have, in the ideal case, a rectangular or polygonal boundary that complements the shape of the car; however, in the more common case, the areas that are being chosen for these developments may be a collection of smaller developments from a time where accommodating for such large, structured spaces was unnecessary. These boundaries can be significantly complex. Consideration must also be taken for obstacles that may add complexity to the design in some way, such as supporting pillars in underground parking or trolley bays at a supermarket.

Although struggling to find an empty parking space is one unfortunate consequence of having a car park that does not adequately accommodate the number of cars it should, there is more significant motivation. For many types of developments, local authorities will not provide planning permission unless they can be assured that there will be adequate parking so as not to cause a "spill-over" where parking spaces elsewhere will be used instead due to demand. In the example of constructing a new block of flats, this directly affects the amount of potential profit from a planned estate as they will only be able to accommodate as many living spaces as they can the appropriate ratio of car parking spaces. Another example is that the cost of underground parking in the United Kingdom is typically £15,000 per space [12]. In areas that are much more costly to purchase land, such as London, this value could be much higher. These examples provide ample motivation for maximising the number of possible car parking spaces in a car park design.

1.1 Background

Currently the designers of a car park may not have an appropriate automated set of tools to assist them. Their intuition and experience will be used to configure the space to be satisfactory – not necessarily optimal. Because of this, researchers have given some initial consideration of how methods could be developed to aid the designer and optimise some basic configuration. The methods contained in [1] for example include the following:

1.1.1 Tile and Trim

This approach looked at packing cars into a space, allocating room for the roads. First, the "best" packing of cars in an infinitely long car park was found. This was done by considering the angle at which the parking spaces could be rotated, relative to the adjacent road, to reduce the amount of road needed to turn into a space. The spaces were also arranged so that they were stacked in pairs to form a *"rectilinear double-row parking pattern"* to reduce wasted space while still ensuring every space was accessible. Under these general assumptions, the most optimal packing of cars was found to be a rectilinear double-row parking pattern [1].

After considering this infinitely long car park, the Tile and Trim approach was then formalised for a bounded car park. The approach used the optimal pattern determined for the infinite unbounded case in the following steps to produce a rough estimate for the optimal solution:

> *"1. Overlap the given finite polygon with the best packing for the infinite plane.*
>
> *2. Turn and shift the given polygon to maximise the number of cars from the background packing falling inside the polygon.*
>
> *3. Trim away a small number of car spaces to provide connection between otherwise unconnected lanes, ensuring that all spaces are accessible.*
>
> *4. if there are pillars or obstacles inside the polygon, remove the conflicting car spaces. "*

This approach was considered similarly by [2] with the idea of generating the best possible pattern of parking spaces and packing this pattern into the space optimally. They used Mixed Integer Linear Programming (MILP)[7], Constraint Programming, and Stochastic Optimisation to optimise their objective function to determine the configuration of the patterns that was optimal.

1.1.2 Deterministic approach to optimising the road

This approach attempted to select a road network aiming for each part of the car park to be accessible from the road. The problem was then creating an optimal configuration of a road that lead to a maximal amount of car park area being accessible from the road inside a bounded area. This was a "road first" approach dissimilar to the packing problem tackled with the Tile and Trim approach.

This approach defined the road to be a closed loop, approximated by a piecewise linear curve connecting a set of nodes lying in a domain. The domain chosen was the unit square and the points in the domain were randomly distributed. The approach aimed to determine the position of the nodes that would minimise a cost function, designed so that at it's minimum,

> *"(i) as much of the domain as possible is within a distance l_r of one of the linear parts of the loop, (ii) as little of the domain as possible is is within a distance l_r of more than one non-contiguous segments of the loop, (iii) the angle at each node is between $\pi/2$ and $3\pi/2$,(iv) the loop is everywhere more than l_r from the boundary of the domain. Here, l_r is the half-width of the road plus the length of a parking space."*

The approach minimised the cost function using repeated passes of firstly a genetic algorithm and secondly the Nelder-Mead simplex algorithm, as implemented in MATLAB (*ga and fminsearch*) with default parameters. This converged to a local minima based off of an initial guess, but often not the global optimal solution.

1.1.3 Variational approach to optimising the road

This approach attempted to find a similar optimised network of roads, as in the previous deterministic approach. However, instead of looking to maximise the accessible space around nodes, it looked at how near any point was to a desired amount of road within the space. The road networks quality was determined by the spatial integral over a function that was defined as the absolute value of the difference between the actual and desired road length (at each point). Only non-branching continuous paths were considered. The idea behind this is that any point belonging to the space should have as little road around it as possible for an optimal solution. This approach produced a cost function as a spatial integral, the Euler-Lagrange equation was used in its optimisation to find the minimum of the difference between the actual and desired road length.

1.1.4 Optimising over road networks

The previous two approaches generally attempted to determine the global optimal road network by finding a minimum of a cost function. This approach instead considered that a small number of candidate car park configurations can be selected by the designer based on intuition, taking into account the entrance and all internal obstacles such as stairwells and lift shafts. These layouts can then be optimised and compared to determine the best layout of those candidates.

This approach developed an algorithm that quickly determines which candidate car park layout provides the best potential for maximum capacity. Instead of addressing the actual packing of cars distributed around the network, the algorithm instead considers areas of general parking space adjacent to the network. The algorithm still outputs a figure representative of total capacity, but not the actual maximum packing necessarily. This approach adopted a general rule that was defined in the Tile and Trim approach,

> *"that nose in parking (i.e.parking 90 degrees to the road) provides the most efficient packing of cars. Using this we assume that, along each linear segment of the road network, we always attempt to park nose in and make no account for other types of parking tiling patterns such as parallel parking or herring-bone."*

The road network for this approach was defined as N vertices with M line segments each connecting two vertices within the network. One vertex was attached to the entrance of the car park. The cost function for this approach was the weighted sum of the following:

1. The overlapping area of any of the rectangular parking stalls with one another.
2. The overlapping area of any of the rectangular parking stalls with the boundary of the car park.
3. The overlapping area of any of the rectangular parking stalls with the internal obstacles of the car park.
4. The total 'void' area defined as the sum of the above three areas.

A crude estimate for the number of parking spaces for a configuration came from taking the total available non-void parking area and dividing it by the the area of a single parking space. This estimate was used to assess the performance of the different candidate car park layouts.

1.2 Problem Statement

Before we fully consider the problem, some initial assumptions can be made about how we may view the spaces we are considering and how they may be defined mathematically.

1.2.1 The Car Park

For the choice of the empty car park space, it is going to be assumed to be single storey with a single entrance for cars (that also functions as the exit). This assumption will simplify the problem, both computationally and analytically, without straying far from the setting of the problem; if there are multiple layers to a car park, then it is starting to resemble the multi-storey bespoke car park described before – not quite the retrofitting scenario at hand. The complexity of the car park boundary will be realistic, meaning not necessarily polygonal but also not significantly erratic. This definition is partially subjective and will surely not suffice to define some bound on how complex the boundary can be. Instead it is a guideline more so on how we choose the space to assess the algorithms performance, giving less significance in the results to the more erratic and less realistic parking lot boundaries. Allowance should be made for large internal obstacles that may significantly obstruct a potential parking space configuration.

1.2.2 The Road Network

The road network will be defined as a topology of the roads within the car park. The car park itself was assumed to have one entrance / exit, meaning that for our road network, there will be one assumed breach of the edge of the car park which is one point on this network. For the road, we do not consider how easy it is for the car to traverse the space as we only want to prioritise the amount of potential parking spaces available. This causes another assumption being that the road network should not contain any closed loops. This is because for any closed loop, by removing the appropriate amount of road on this loop, room for another space can be created while still ensuring that there is a path from any point on the network to the entrance. The above assumptions naturally define the road network to be a tree [3] that branches out into the car park space, filling it with road to access the space adjacent to the network.

Using the initial assumptions above, the problem can be formalised by the following: Given the boundary of a car park, with a defined entrance (*which also functions as an exit*) on the boundary, find a layout that maximises the number of car parking spaces with as little road as possible.

2 The Deterministic Approach

The approach that will initially be taken to explore this problem will be aligned with 1.1.4, with the idea of focusing on the road rather than the packing of the cars and how

they may be tiled or patterned. In 1.1.4, the assumption was that the designer would be able to select specific candidate car parks for consideration based on intuition. Instead, we will aim to develop an algorithm that generates these candidate car parks. The algorithm will aim to grow the network of roads itself from a starting point to fill the space. It will select a point to grow from and then decide how it will grow from that point. The network should stop growing once it has filled the space completely with either road or car parking space.

Motivation for this choice of approach can come from problems found in nature such as the root systems of plants which aims to grow the roots into a space in search of nutrients. There are also many man-made problems that are similar such as irrigation, and warehouse design. There is also some connection between these problems and the problem that comes from the field of robotics associated with how a moving robotic device plans its motion through a space populated with obstacles. Motivation for this also came from the described possible extensions to the approaches described in [1] where you could automate the design of the network.

2.1 Details of the road network

The network for the parking lot can be mathematically defined as a Graph $G = (V, E)$ with a set of Vertices V of order n and a set of edges E of order m, each connecting two vertices belonging to the network. Each edge is thought of as a straight length of road that is not intersected unless at an endpoint, which in this context is a road intersection, and a vertex in V. This naturally defines the graph as planar [4] which fits this scenario as we are considering topologies in a two dimensional plane. The boundary of the car park is $B \subset V$, which surrounds the rest of the graph and is a closed loop. The road network itself is the complement of the boundary subset $R = V \setminus B$. One edge connects a vertex belonging to B to a vertex belonging to R and is associated with the car park entrance and exit. R is defined as a tree meaning it doesn't contain any closed loops.The way we are going to arrange car parking spaces along these edges will be that they are arranged in double stacked rows of cars parked perpendicular to the road direction. This assumption is motivated similarly to 1.1.4:

> *"that nose in parking (i.e.parking 90 degrees to the road) provides the most efficient packing of cars. Using this we assume that, along each linear segment of the road network, we always attempt to park nose in and make no account for other types of parking tiling patterns such as parallel parking or herring-bone."*

2.2 How the road network is created and expanded

The algorithm this approach aims to create is one that grows the road network one vertex at a time from the starting vertex connected to the entrance. The way the algorithm grows this network is that it first chooses a vertex belonging to the road network R and then "branches" out from this vertex. The way the algorithm "branches" out from this vertex is by choosing some distance and some angle relative to the vertex and creating a new vertex at this location that is then added to the road network tree. This new vertex increases the roads length and adds a new edge to the edge set E. This process is iterated until some terminating condition is met, such as the car park being entirely accessible from some part of the road. We can break this process down into two main steps: 1. Choosing the vertex we should branch out from, and 2. How we branch out from the chosen vertex. For both of these steps, there will be a cost associated with expanding the

road network in a specific way, this will be what is optimised over to determine optimal configurations.

For this approach, we are choosing to branch out from any given vertex at some elemental length l. The motivation behind this is that l can represent the length of one parking spaces. This means that there should be little wasted space when branching as in between two vertices there is the exact amount of distance needed to fit a row of car parking spaces. For example, by considering the scenario where you may branch out from a vertex at 90 degrees relative to that vertex's connected edges, there would be no loss in the amount of potential car parking space if the distance between the vertex and any of its neighbouring vertices is double the elemental length l.

2.3 Attributes of the road network to consider

As we are not the designer of any specific car park, we do not have access to the specific details that may be associated with any individual endeavour. When we are considering how we should construct an algorithm that will penalise worse configurations of vertices, we will have to ensure that the attributes we consider can be general enough to apply to any car park. These ways of assessing the car park will be motivated by the general intuition any designer has as to what would be desired traits for the network.

Since we are growing the network vertex by vertex, we want to ensure that the way we assess a desirable configuration should be done at every iteration of grow, and for one individual vertex at a time. So the traits we would want to consider desirable for the network must be relative to any individual vertex.

One such trait when assessing which vertex to grow the network from could be the amount of road around the vertex. The motivation behind this would come from the inherent need to explore; an area with less road is an area less explored. This idea of using the amount of the network around a vertex to guide its exploration can be compared with the algorithm for the Rapidly Exploring Random Tree [5]. The metric for the amount of road could be thought of as a physical area of road within a bounded area around a vertex; but for our purposes counting the number of vertices within some bounded surrounding area will fulfill the same objective. This area would likely have a radius that is a multiple of the elemental length l as to encompass the adjacent road up to the last vertex.

Another trait that could be considered is whether moving into any given area of the car park provides any benefit; if there is no new parking space in a given direction, why go there? In a similar fashion to the above, we can consider an area that has a radius of some multiple of the elemental length l and determine how much space in that area is already accessible from any road. This area would be offset from the vertex being considered as we want to see the benefit of going in that offset direction. This offset would be of length l away as we will be growing by that distance from any vertex. (This leads to an area covering some radius that is a multiple of l around every point lying along the perimeter of the circle surrounding any given vertex with radius l).

An aspect that could be considered, that is inspired from the idea of the root of a plant as mentioned prior, is the distance from the entrance of the car park. You can imagine that as the root system expands, it will spread out in a combined front more-so than searching depth first. This can be taken into consideration also with the way we choose

to grow the road network. As the roads move further and further from the entrance of the car park, the distance that a road is from the entrance could reduce the likelihood of it being considered in comparison to a road closer to the entrance. This distance can be taken into consideration as the length of the shortest path from the entrance to any given vertex, but could also be a function of the amount of road currently in the car park or the dimensions of the car park.

A cost inspired more so by actual road networks is taking into consideration the number of intersections a vertex has. The more intersections any road network has, the more the road network branches creating wasted space. In some cases this may be ideal, but for the sake of creating a car park with as much space as possible, the less road the better. This means that we may want to consider the number of other vertices connected to a given vertex, and its surrounding vertices, to determine if we may branch from it; providing a preference to those vertices which are not at an intersection and are instead at the leaves of the road network tree.

How much overlapping area there is of car parking bays along any edge, taken into consideration in 1.1.4, can also be applied to our approach. Ideally, there would be as little overlap between the areas of car parking bays as possible as this would increase the amount of wasted space generated when branching out from a vertex. With this motivation, this area can be defined as the total area of car parking space from a potential branching edge that is overlapping the already placed parking space area. This loss is only present when branching out at an angle that is not at 90 degrees to the chosen vertex's other connected edges.

These five metrics for assessing any given vertex's "fitness" to be grown from can be summarised as:

1. The number of road network vertices around a vertex.
2. The amount of unfilled space accessed by moving in any given direction from a vertex.
3. The distance the vertex is from the entrance of the car park.
4. The number of vertices connected to a vertex and its surrounding vertices.
5. The amount of area wasted when branching out from a vertex at an angle not at 90 degrees to the vertex's other connected edges.

2.4 Cost function for growing network

From the above attributes, we can start to construct a cost function for growing the network. This cost function is actually a combination of two other cost functions, the cost of selecting the vertex to grow from and the cost of how you grow from said vertex.

For the former, the attributes that would make most sense to consider are (1), (3), and (4) as they are all relative to a given vertex that already belongs to the road network. (2) and (5) are related to the potential vertices that you could add to the road network by branching from any given vertex, and therefore make more sense to be considered for the second cost function that is related to how you choose to grow from a vertex.

The first cost function can then be constructed as a weighted sum of these three attributes:

$$f(a, b, c) = ax_1 + bx_3 + cx_4 \quad \text{for } a, b, c \geq 0$$

Where x_1, x_3, x_4 are the evaluated values of the attributes (1), (3), and (4) respectively for any vertex belonging to the road network R.

For the second cost function that determines how we branch out from the chosen vertex, we consider (2) and (5) as stated above. However, for any given way of growing the road network from a vertex, we also need to reconsider the number of road network vertices surrounding this new potential vertex; (1). The direction being considered for the new vertex may be branching into an area where there is already a piece of road nearby. It is easy to imagine that this may not be ideal which is why there is consideration for it. (3) and (4) can both be ignored for this potential new vertex as the distance should be equal for all potential options from a given vertex and similarly for the number of connections to the vertex.

This means the second cost function can be constructed as the weighted sum of three attributes:

$$g(d, e, f) = dy_1 + ey_2 + fy_5 \quad \text{for } d, e, f \geq 0$$

Where y_1, y_2, y_5 are the evaluated values of the attributes (1), (2), and (5) respectively for any potential vertex that is being considered from an already chosen vertex in R.

2.5 Simplifying the approach

For this approach, applying it completely as described above generally would be computationally and logistically challenging. So initially, to determine its efficacy and feasibility, the consideration of a simplified scenario could lead to promising results given the appropriate assumptions that represent the approach fairly.

The below assumptions simplify the problem while still ensuring that the approach is still representative of the original and not trivial. These assumptions reduce the computational and logistical complexity of the problem which suits our initial tackling of the problem.

2.5.1 Boundary of the car park

In the more general case, the boundary is only restricted to be a subset of the Graph G that is a closed loop. The assumption made here to simplify the problem is that this boundary is now a rectilinear approximation. The boundary is composed of straight lines that are connected at right angles to each other where each lines length is a multiple of the elemental length l. This assumption can reasonably be made as more simplistic car parks will take this shape and for the purposes of this simplification, we are willing to sacrifice some of the models complexity while still allowing it to apply to many realistic car park configurations. The car park boundary then becomes a more simplified approximation of a more complex boundary that will have a length and width that is a multiple of l.

2.5.2 How we grow the network

In the more general case defined above, vertices in the plane can be connected by an edge belonging to the edge set. The angle of this edge relative to some fixed euclidean axis has no restrictions. The assumption taken by this simple configuration is that for the rectilinear boundary defined above, branching out at an angle that is not 90 degrees relative to one of the boundaries is almost exclusively non-optimal and will lead to wasted space. This means that we can presume that we are only branching out from four possible directions from any given vertex – North, South, East, and West. In actuality, as we are branching out from an already existing vertex, branching back in the direction of that vertex is impossible. This means that for any given vertex we would be branching out from, there would be at most three potential directions to branch out.

This combined with the prior assumption effectively simplifies the graph to be a rectilinear grid, as shown in 1. A cell in this grid is filled if it contains a vertex. This vertex can belong to the road network or the boundary. Two vertices can then be connected via an edge if their cells are immediately adjacent to each other in the grid and belong to the same subset of the graph; R or B. We will also relax our definition of the road network being a tree. We are going to be optimising a cost function that entirely decides how we may choose to grow the network, restricting it to not branch in a way that may make a loop is not the goal of this approach. Instead, we aim to optimise over the parameters of our cost functions so that our end result does not contain any cycles in the graph as that would be non-optimal. This means that our optimal solution should be an acyclic graph, but for completeness it is not a requirement.

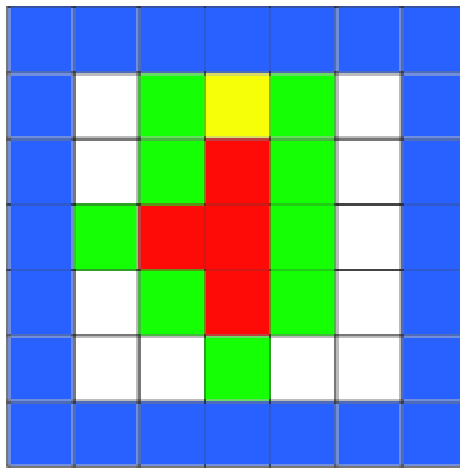


Figure 1: This is an example of a rectilinear grid that represents our car park. The blue grid cells represent cells that are filled with boundary vertices. The red cells represent the vertices belonging to the road network, where the yellow vertex is also the entrance to the car park. The green cells represent the car parking spaces that are accessible from the road network.

2.5.3 Representation of a car parking space

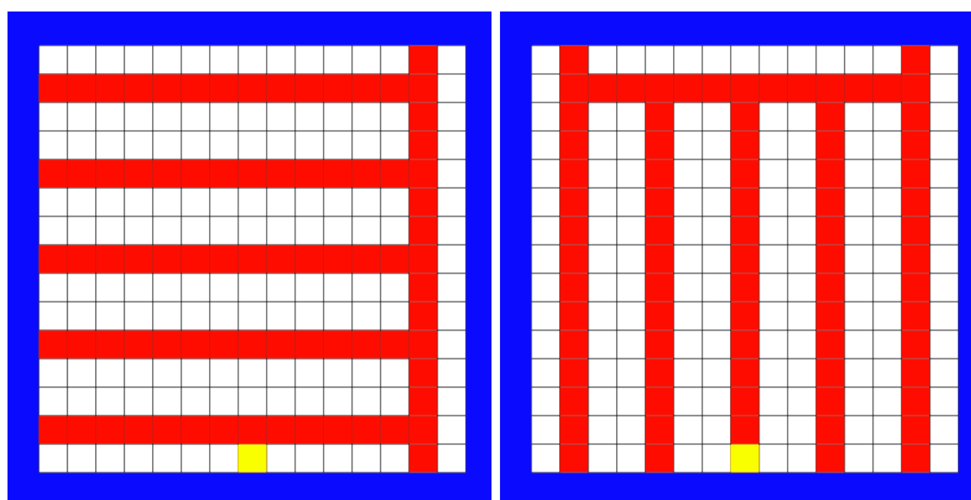
The size of car parking spaces in the more general case has a defined width and height for each car parking space. We then have the elemental length l that represents the length of a parking space to help reduce wasted space when

branching. The assumption made here is that every cell in the grid mentioned above has a fixed width and height of l. This means that any cell represents the length of one car parking space. The idea behind this assumption is that as we are attempting to generate a car park topology, we more so care about the arrangement of the road network vertices in the plane. This simplified configuration can be viewed more so as an approximation of this topology, similar to the boundary of the car park. The fixed length and width of a car parking space is then not an incredibly important factor when generating these configuration.

The generalisation is then that any car parking space can be represented by an empty cell in the grid adjacent to a cell that contains a vertex belonging to the road network. This allows us to view the entire grid as a countable amount of road cells and car parking cells that can then be used to assess performance and determine approximate road network structures.

The above assumptions, coupled with choosing an appropriate car park boundary size, will allows us to tackle this problem effectively while being less computationally complex. The entire car park can be represented as a matrix where each element represents a cell; a structure very easy to implement. The assumptions made do reduce some of the models application, but can still be generally applied to determining the efficacy of our approach of generating a car park configuration. With some effort, this abstraction could be applied to an approach that considers more realistic or representative scenarios.

It is also useful for us to define in this simplified approach what we may consider to be optimal. As this approach is relatively simple, finding a construction that we can define as the most optimal can be done by eye. For our purposes, we will be testing our algorithms in a variety of circumstances, but specifically the case where we have a 15×15 grid. The following results can be viewed as optimal as they can contain no more parking spaces with their exact configuration of the boundary and entrance. They do not represent the only optimal result, but two very different results that achieve the optimal.



(a) The entrance to the car park is placed at $(0, 7)$. This result has 81 road network vertices and 144 car parking spaces.

(b) The entrance to the car park is placed at $(0, 7)$. This result has 81 road network vertices and 144 car parking spaces.

Figure 2: These are the optimal results defined for a 15×15 grid.

2.6 Simplified cost function

Now that we have defined our new simplified approach, we need to adjust some of the definitions we made earlier. The cost functions we defined earlier can still be applied to this new simplified approach in much the same way:

1. The number of road network vertices around a vertex can be the number of adjacent road network cells in the grid for a given vertex.
2. The amount of unfilled space accessed by moving in any given direction from a vertex can be the number of unfilled spaces now accessible by placing a road in a given cell.
3. The distance the vertex is from the entrance of the car park can now be measured as the shortest path between the vertex and the entrance along vertices of the road network.
4. The number of vertices connected to a vertex and its surrounding vertices is then becomes: the number of the cells that are filled with road that are adjacent to a vertex or its surrounding vertices.

The main change now is in (5) in our list of attributes from the more general approach; the amount of area wasted when branching out from a vertex at an angle not at 90 degrees to the vertex's other connected edges. As described earlier, in this simplified configuration we are only branching out at an angle at 90 degrees to the vertex's other connected edges. This means that we can effectively ignore this attribute. This only effects the second cost function relating to how we branch.

2.6.1 Cost function for selecting the vertex to grow from

Now that we have our new consolidated list of attributes that we will use to construct the two cost functions, similar to how we did before, we need to define them. For these attributes, we want to consider the values of individual grid elements that represent an empty space, a car parking space, or a space filled with road. A vertex belonging to the road network now has an associated grid position indexed by $i, j \in \mathbb{Z}, \quad r_{ij} \in R$. We define the neighbourhood of a point indexed by i, j to be the set of all indexes that would be adjacent to the current position:

$$N(i,j) = \{(i,j+1),(i,j-1),(i+1,j),(i-1,j)\}$$

We also define all the elements surrounding a vertex indexed by i, j to be the set of all indexes that would be adjacent to the current position including the diagonally adjacent elements and (i, j):

$$S(i,j) = N(i,j) \ \cup \ \{(i,j),(i-1,j-1),(i+1,j+1),(i-1,j+1),(i+1,j-1)\}.$$

First we will consider the first cost function relating to how we select the vertex to branch from. This cost function considered three attributes x_1, x_3, x_4, the evaluated values of the attributes (1), (3), and (4) respectively for any vertex belonging to the road network.

We described the attribute (1) under this new simplified configuration before as the number of adjacent road network cells in the grid for a given vertex. This means that for a

given vertex belonging to the road network, $r_{ij} \in R$, we want to count the number of adjacent cells that are also vertices of the road network R. We can define this as:

$$x_1(i,j) = \sum_{p \in N(i,j)} h(r_p)$$

Where h is the piecewise function defined as:

$$h(p) \begin{cases} 1 & r_p \in R \\ 0 & \text{otherwise} \end{cases}$$

The next attribute to consider is (3). This has been defined as the length of the shortest path between the vertex r_{ij} and the entrance to the car park. For a bit of simplification, the way we calculate the shortest path can be left to the implementation in code. Instead we will define $s(r_{i,j})$ to be the shortest path from r_{ij} to the entrance of the car park. We can then define the evaluated value of this attribute as:

$$x_3(i,j) = |s(r_{i,j})|,$$

The length of the shortest path from the given vertex to the entrance.

Finally we want to consider the attribute (4) which was described as the number of the cells that are filled with road that are adjacent to a vertex or its surrounding vertices. This means that for a vertex we are considering, we want to take the sum over all surrounding vertices and itself, the number of road vertices adjacent to each vertex. We can define this as:

$$x_4(i,j) = \sum_{p \in S(i,j)} \left(\sum_{q \in N(p)} h(r_q) \right) = \sum_{p \in S(i,j)} x_1(p).$$

With these attributes now defined explicitly, we can construct the cost function in terms of these attributes for a given vertex r_{ij}. This cost function will be used to choose the vertex to grow from as we did with the general approach:

$$\begin{aligned} f(a,b,c) &= ax_1 + bx_3 + cx_4 \\ &= a \left(\sum_{p \in N(i,j)} h(r_p) \right) + b|s(r_{i,j})| + c \left(\sum_{p \in S(i,j)} x_1(p) \right) \\ &= a \left(\sum_{p \in N(i,j)} h(r_p) \right) + b|s(r_{i,j})| + c \left(\sum_{p \in S(i,j)} \left(\sum_{q \in N(p)} h(r_q) \right) \right) \quad \text{for } a,b,c \geq 0 \end{aligned}$$

2.6.2 Cost function for deciding how we grow from a chosen vertex

Similar to how we redefined the first cost function in this new simplified configuration, we shall redefine the second cost function for how we grow from a chosen vertex. The attributes we will be considering for this cost function will be the same as the general approach, with the exclusion of attribute (5) as explained prior. For these attributes, we want to consider the values of individual grid elements that represent an empty space, a car parking space, or a space filled with road as above. We take the definition for the neighbourhood and surrounding vertices from the previous section:

$$N(i,j) = \{(i,j+1),(i,j-1),(i+1,j),(i-1,j)\},$$

$$S(i,j) = N(i,j) \ \cup \ \{(i,j),(i-1,j-1),(i+1,j+1),(i-1,j+1),(i+1,j-1)\}.$$

This second cost function considered three attributes y_1, y_2, y_5; the evaluated values of the attributes (1), (2), and (5) respectively for any vertex belonging to the road network. We will be ignoring y_5.

We defined attribute x_1 in the previous section which was a count of the number of adjacent cells that are also vertices of the road network R. y_1 is much the same except that we are now considering a potential new vertex adjacent to an element of the road network instead of an already existing $r_{ij} \in R$. This however does not effect the way we will calculate the evaluated value of this attribute, meaning:

$$y_1(i,j) = x_1(i,j) = \sum_{p \in N(i,j)} h(r_p)$$

Where h is the piecewise function defined as:

$$h(p) \begin{cases} 1 & r_p \in R \\ 0 & \text{otherwise} \end{cases}$$

The next and final attribute to consider is (2). This was described as the number of unfilled spaces now accessible by placing a road in a given cell. However, as we are creating a cost function that we aim to minimise, this definition does not suffice. This is because we would under this description want to maximise the amount of empty space we are now able to access. Instead, we can change the definition to be the number of filled spaces surrounding a vertex before moving in that direction. This gives us a measure of how many spaces are neither accessible from a road, nor a vertex belonging to the road network. This is intuitively something we would aim to minimise, as this gives a natural tendency to move into unexplored areas of the car park.

With this updated definition, we can define the following to reflect an evaluation of this attribute of the network:

$$y_2(i,j) = \sum_{p \in N(i,j)} k(r_p)$$

Where k is the piecewise function defined as:

$$k(p) \begin{cases} 1 & r_p \in R \\ 1 & \exists\, t \in \{r_q \mid q \in N(p)\} \text{ such that } t \in R \\ 0 & \text{otherwise} \end{cases}$$

With these attributes now defined explicitly, we can construct the cost function in terms of these attributes for a potential new vertex at index i, j. This cost function will be used to choose how we grow from a vertex that has been chosen through the use of the previous cost function. Due to the simplified approach we now take, this is one of at most three

potential directions. The cost function is then defined as:

$$\begin{aligned} g(d,e) &= dy_1 + ey_2 \\ &= d\left(\sum_{p\in N(i,j)} h(r_p)\right) + e\left(\sum_{p\in N(i,j)} k(r_p)\right) \\ &= \sum_{p\in N(i,j)} dh(r_p) + \sum_{p\in N(i,j)} ek(r_p) \\ &= \sum_{p\in N(i,j)} dh(r_p) + ek(r_p) \quad \text{for } d,e \geq 0 \end{aligned}$$

2.7 Growing the network

With the cost functions for choosing the next vertex constructed, we can now formalise how we will grow the road network. The road network will be expanded one vertex at a time, iterating periodically. This process will be done in two parts as described prior; first choosing a vertex to grow from and then deciding in what direction we grow. Below we will explain the process for both steps algorithmically.

We also must consider when we will decide to stop growing the network. The terminating condition for the algorithm, and subsequently when the car park will be considered filled, is when *all grid cells are either a vertex belonging to the road network R or are grid cells in the neighbourhood of a vertex belonging to R.* These conditions will be checked at the beginning of every iteration and if met will cause the algorithm to terminate. Once we have terminated, we will output the entire car park grid as our finished configuration.

2.7.1 Choosing a vertex to grow from

The algorithm for growing the network will at every time step first select a vertex to branch from. This will be done using the first cost function we defined with a given set of parameters a, b, c. This first cost function considers attributes that we have chosen so that a smaller cost function value would seemingly lead to a more optimal choice of vertex to branch from. This means that for determining the best vertex across the whole car park, we will have to minimise this cost function over all vertices $r \in R$. An explanation of how this will be done practically will be explained in section 2.9. This will then give us a position i, j in the car park grid that we can then pass onto the next step of the algorithm;"branching out" from the chosen vertex in one of three directions.

Importantly, there are a few cases that we should consider that would cause issues with this algorithm. The main issue being that suppose the vertex that minimises the cost function cannot be expanded from. This could happen if all of the grid cells in the neighbourhood of a vertex are vertices that belong to the graph G. In this situation, the solution we will take is to take the vertex that has the next smallest evaluated cost value. If this vertex *also* falls into the same situation of being surrounded by graph vertices, then we repeat the steps prior. This will either lead to the next least costly vertex being selected to branch from, or will be a situation which is a terminating condition of the algorithm and would have already been checked.

Another issue to consider is that of two vertices having the exact same minimum cost

value. We want this approach to be deterministic meaning that we must have a consistent approach to tackling this situation. The approach we will take is to simply choose the first vertex found with this minimum cost. This keeps this approach deterministic without any change to the resulting car park grid.

2.7.2 Choosing the direction to grow from a chosen vertex

After the first cost function has been utilised to determine the vertex that should be used as the "branching" vertex, we want to utilise the second cost function we defined to choose the *direction* in which we grow the road network. This takes place at the same iteration step as the first cost function. The first cost function will have produced a position in the grid i, j. This position is what will be given to the second cost function which will calculate the cost for each branching direction. Similar to the first cost function, we have designed this function specifically to consider attributes of the road network so that the smaller the evaluated value, the more optimal the branching direction. We minimise over all of the potential branching directions to determine the most optimal choice. For this step of the algorithm we now only need to consider three potential vertices at most and is therefore far less computationally demanding. This will lead to an implementation that is explained further in section 2.9. Once we have chosen the direction in which we wish to grow the road network, we then add a vertex to the road network R at the appropriate position in the neighbourhood of the previously chosen vertex. Edges are then also implicitly added to the edge set connecting the newly added vertex to all adjacent road network vertices. After this step the algorithm then repeats until the terminating condition is met.

Similar to the previous part of the algorithm, there are cases that we must consider that may cause an issue and are not obviously dealt with. This comes in the form of what branching directions we consider. We only want to consider cells that are able to be filled with a vertex and be added to the road network. This means that the cells adjacent to the chosen vertex that are considered must not be vertices belonging to the road network R. However, this does not cover situations that occur at the boundary of the car park. This means that we must be more restrictive and say that the cells in the neighbourhood of the chosen vertex must not be vertices belonging to the graph G. We also must consider if there are *any* possible branching opportunities from a given vertex. This issue is dealt with due to the conditions met by the previous part of this algorithm; the neighbourhood of the chosen vertex must have an empty cell that is not filled with vertices belonging to G. If this condition is met then there will always be a potential branching direction.

2.8 Optimisation of parameters

With a formalised and simplified approach to how we attempt to generate configurations of car parks, we now need to formalise our approach to optimising. Previously we explained that the simplified configuration now gives us a countable amount of cells that can be either a car parking space, or a piece of road. When we are optimising, we want to maximise or minimise over one of these attributes. This is intuitive as a car park with as many car parking spaces as possible is what we are aiming to achieve, but is also equivalent to minimising the amount of road in the car park. This is equivalent due to our terminating condition being that the car park must be filled with car parking spaces or with road. The inclusion of obstacles would not change this as long as the obstacles were kept in the same consistent locations across different executions of our algorithm.

For our approach, we have chosen to maximise the number of car parking cells. This

means that we are wanting to maximise over the integer number of cells in the car park grid that are adjacent to a vertex belonging to the road network R. We could have alternatively minimised over the number of cells that contained vertices belonging to the road network. These two approaches would achieve the exact same objective, but may be preferred depending on the chosen method of optimisation.

With a calculable value that we can use to assess the performance of our car parking grid, we can use this to optimise over our parameters. For this approach we have two cost functions that take in total four parameters that we need to optimise over to determine an optimal configuration. These four parameters are continuous real numbers that are only required to be non negative.

2.8.1 The issue of optimisation

With the details of our optimisation problem considered, we now encounter several issues. In most optimisation problems we have an objective function, here this function is the number of car parking spaces. This objective function is what we want to minimise or maximise. Objective functions are usually continuous differentiable functions that are taken advantage of by several different optimisation schemes and algorithms to reach a minimum or maximum value. The issue with our objective function is that it is integer valued and non continuous. This means that several optimisation schemes that have been used in previous studies [1] such as the *Nelder-Mead simplex algorithm*[1] or *genetic algorithms* do not effectively tackle this problem. This is because they quickly find themselves in local minimum values and terminate without considering a significant amount of the parameter space. This is because the integer valued objective function may sometimes change based on a small change in the value of the parameters, or may not change until a large change is made to the parameters. This quickly leads to minimum values that are not close to or representative of a global minimum value.

Other schemes were considered that were used by other studies such as [2], mainly MILP (Mixed integer linear programming) [7]. MILP however more so applies to integer valued parameters instead of being focused on integer valued objective functions. This means that in our approach, where we define all the parameters to be non-negative real values, this optimisation scheme does not seem as applicable.

This leads to a situation where our optimisation schemes seem limited or would require extensive manipulation of already existing schemes to accommodate our situation. We could also potentially change our objective function to attempt to find something that may be more easily optimisable at the cost of our simplicity.

2.8.2 The brute force approach

We want to be focusing on the algorithm we implement more so than the optimisation problem it may represent. The issues above lead to a very obvious but seemingly inefficient approach encompassed by the idea of brute force. This optimisation scheme would simply iterate over the entire parameter space to find a minimum value. This approach in most scenarios is incredibly inefficient and would be unrealistic in most application, with the only main benefit being assured of a global minimum value at termination. However, as we have simplified our problem quite dramatically, the computational complexity of such a scheme is not too unreasonable. The only relaxation we would practically have to make

[1] Implemented using the MATLAB inbuilt function "fmin"[6].

is to bound the maximum values for certain parameters as the actual parameter space is infinite. This is because we defined all the parameters to be non-negative and bounded below by 0, but none are bounded above. So this scheme would instead iterate over an appropriate percentage of the parameter space to find a global minimum. In our actual application of this, we will be finding the minimum over the amount of road rather than the maximum over the amount of parking space as described prior. Under this scheme the two different approaches to evaluating our objective function are equivalent.

Another important factor for this brute force approach is that the way our cost functions are designed means that the individual attributes are weighted based on the parameters we provide. These attributes are summed to provide the value that we use to assess a vertex. This leads to the individual values of each attribute not being incredibly relevant to the end result of the algorithm. Instead, the relative weighting between parameters is what will provide difference in results. This means that we may also consider less of the parameter space as we can already guarantee certain results will be identical. For example, say we provided the values a, b, c, d as the parameters to our two cost functions. We can guarantee that the result of this execution of the algorithm will be identical to the algorithm executed with parameters $2a, 2b, 2c, 2d$ as the relative waiting is identical.

The amount of change we make to each parameter when iterating over the parameter space is also non-trivial. We cannot change each parameter an infinitely small amount so we must provide an amount which we should change a parameter at each iteration. This will be the main increase in execution time as the smaller the amount we change each parameter by during iterations, the more times we execute the algorithm. When practically applying this scheme, multiple different amounts that we change the parameters by will have to be considered.

With this in mind, we can practically apply this approach to determine the parameters that maximise the number of car parking spaces generated using our algorithm. There is no guarantee that we will be finding the global maximum through this approach, but for the purpose of analysing the efficacy of this type of algorithmic approach, this optimisation scheme should suffice.

2.9 Explanation of code

With the simplified approach and our approach to optimising over our objective function defined, we can now execute our algorithm. The program written that executes this approach is in the appendix of this report. The program was written in python version 3.6.3 [8]. Python was chosen as I personally have experience using the development language and are familiar with its libraries. The libraries that proved must useful where NumPy [9] and SciPy [10] which both provided useful utilities to assist with creating our program. The program outputs its results as an image representing the car park grid.

2.9.1 Using "layers" to improve performance using memoisation

In the program, I made the car park itself a class. This class has several attributes that assist in keeping track of the current state of the car park at any iteration. Its member variables include the car park grid, implemented as a matrix, and the width and height of the car park grid. This class also has methods that update the car park when a new vertex is added to the road network.

Previously mentioned attributes that are computationally demanding to calculate at every iteration have been dealt with by incorporating in the idea of “layers”; a use of memoisation[2] . The different layers are all copies of the car park grid implemented as a matrix. The elements of the matrix are different depending on the layer. For example, there is a distance layer where every cell that contains a vertex belonging to the road network is an integer value representing the distance of the shortest path from the position to the entrance of the car park. This allows us to easily determine the distance of a new vertex added to the road network by simply taking the largest distance layer value in the vertex's neighbourhood, increasing the distance by one.

```
def updateDistance(self, i, j):
    adjacentCount = 0
    minDistance = math.inf
    for m in neighbourhood:
        if (i+m[0] >= 0) and (j+m[1] >= 0) and (i+m[0] < self.w) and (j+m[1] < self.h):
            if self.road[i + 1 + m[0], j + 1 + m[1]] > 0:
                if self.distance[i + m[0], j + m[1]] < minDistance:
                    minDistance = self.distance[i + m[0], j + m[1]]
    self.distance[i, j] = minDistance + 1
```

Listing 1: Function for updating distance layer. It also uses the road layer which contains the information associated with x_4 and is implemented in a very similar way to the distance layer.

When assessing the cost of any vertex, we can simply look at the value at that vertex's position in the distance layer to find the value of x_3. This method of memoisation will improve performance.

2.9.2 Implementation of the cost functions

Previously we have described our process for implementing our cost functions. Specifically, we separated out the process into two separate algorithms that use both cost functions individually to select a vertex to branch from, and then in what direction we branch from that vertex. However, in this implementation of the algorithm, the two have been combined. Instead of choosing to separate the cost functions, we calculate their sum and use it as a cost for every branching direction for every potential vertex. This was implemented as such because the additional computation of calculating the preferred branching direction is very minimal. This means that on top of calculating the cost of choosing a vertex to branch from, the cost calculation of each individual branching direction becomes more insignificant. Combining these two processes simplifies the code without a significant performance deficit in our implementation. However, choosing to scale up the size of the potential car park grid would lead to a more significant effect on performance. This issue however would not seem incredibly relevant as the execution of our algorithm to create a car park configuration should produce similar results irrespective of the car park size under the same set of parameters.

Along with this combination of the two cost functions, we are now only considering branching directions. This however creates some redundancy in our calculation by considering x_1 and y_1, both the number of road vertices in their neighbourhood. They are almost entirely the same evaluation meaning that it shouldn't cause significant change in the way we choose to select our new vertex. Because of this we instead only consider the attribute y_1.

The one cost function is then implemented as the weighted sum of the four parameters.

[2]Although this looks to be a misspelling of memorisation, *memoisation* is in fact a specific optimisation technique that I would urge you to look into further https://en.wikipedia.org/wiki/Memoization

```
def cost(self, i, j, k):
    return (self.gridSum(i+1, j+1) * k[0] +
            self.roadSum(i+1, j+1) * k[1] +
            self.getDistance(i, j) * k[3] +
            self.getFilledSpaces(i,j) * k[2])
```

Listing 2: Function for calculating the cost of a potential new vertex. The individual functions inside this function calculate the attributes y_1, x_4, x_3, y_2 respectively

2.9.3 Brute force minimisation of objective function

As discussed before, our approach to determining parameters that maximise our objective function will be a brute force approach. This approach will iterate at some set interval over the parameter space, keeping track of the minimum value achieved so far. The way this has been implemented is to create a new instance of the car park class with the same dimensions with every new set of parameters. The algorithm is then run over this car park with the parameters provided, which then produces a deterministic result. This result is the number of cells that contain vertices that belong to the car park; what will be minimised.

In the part of the program shown below, the parameters are iterated over where "beginning" refers to the lower bound on the values of the car park; in most cases 0. The variable "end" refers to the upper bound. The variable "sample_n" refers to the number of values to sample within the range specified, this value if made too large is the most computationally demanding part of this process. The other variables given to this function specify the dimension of the car park grid to iterate over and where to position the entrance on said grid.

```
def bruteForceMaximisation(beginning, end, sample_n, car_park_size, entrance):
    minimum_road = math.inf
    results = [[]]
    for q in np.linspace(beginning,end,sample_n):
        print("progress: ", (q/end)*100, "%\n")
        for w in np.linspace(beginning,end,sample_n):
            for e in np.linspace(beginning,end,sample_n):
                for r in np.linspace(beginning,end,sample_n):
                    CarPark = Grid(car_park_size[0], car_park_size[1],
                                   entrance[0], entrance[1])
                    k = [q,w,e,r]
                    valu = CarPark.iterate(k)
                    if valu == minimum_road:
                        results.append(k)
                    if valu < minimum_road:
                        results =[[]]
                        results.append(k)
                        minimum_road = valu
```

Listing 3: Function that iterates over the parameter space to find the miminum of our objective function

2.10 Results

The below figures and results will show how this deterministic approach performed under multiple different situations. The program outputs an image which represents the car park grid. Every vertex that is coloured blue represents the boundary of the car park. Any cell coloured red represents a vertex belonging to the road network, where the square red vertex denotes the entrance of the car park. Edges between vertices belonging to the road network are drawn as black lines. All green vertices represent a car parking space. The program used to generate these results can be found in the appendix.

2.10.1 Square car park boundaries

Initially the car park grids that were considered were those with a square boundary shape. For these car park shapes, the entrance was initially placed as central to one axis of the car park as possible, and was required to be adjacent to one of the boundary vertices. The brute force approach was run with parameters where the value of each parameter fell in the interval $[0, 1]$. The number of samples taken within this interval varied. The more samples chosen, the longer the time the program took to return a successful result.

An example of one of these results is the following figure. This result was run with a set of parameters found through the brute force approach. The car park grid size was 15×15 and and the entrance to the car park was placed centrally against the bottom edge of the boundary.

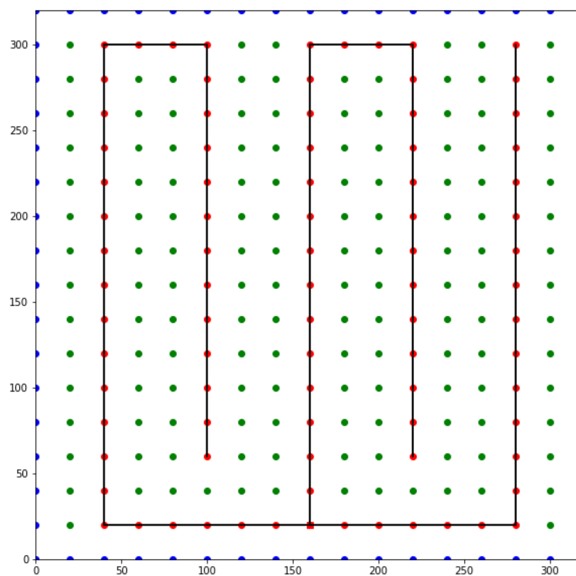


Figure 3: This is the result output from the program listed in the appendix. The car park grid size was 15×15 and and the entrance to the car park was placed centrally against the bottom edge of the boundary, marked by a square. The parameters used were $(0.45, 0, 1, 0)$ This result has 85 road network vertices and 140 car parking spaces.

This results merits are visible immediately based on our own intuition of how a car park may be designed. As described prior, we did not restrict the algorithm to generating a network that was a tree as that would restrict the logistic freedom of our potential car park network. The idea behind this was that an optimal car park will be an acyclic graph inherently. The resulting road network in this example is an acyclic graph. The number of road vertices is 85 against 140 car parking spaces leading to a car parking space to road vertex ratio of 1.65. There are lots of straight lines that reduce the amount of wasted space from having to create an intersection of multiple roads. There are few cases where a car parking space is accessible from two different road segments that aren't at a corner. However, through some mild rearrangement of road network vertices it is clearly possible to improve and is therefore not the global optimum.

This example also highlights one of the issues of this approach that was found from the results garnered; optimal configurations completely disregarded certain attributes by setting certain parameters to 0. This issue is discussed more thoroughly in the following sections.

These exact same parameters were used on the exact same sized car park grid, but with the entrance position changed. The idea behind testing this was to determine whether we had found parameters that were not only optimal for a specific configuration, but could be applied to others. The following shows this test, with the entrance moved several space to the left.

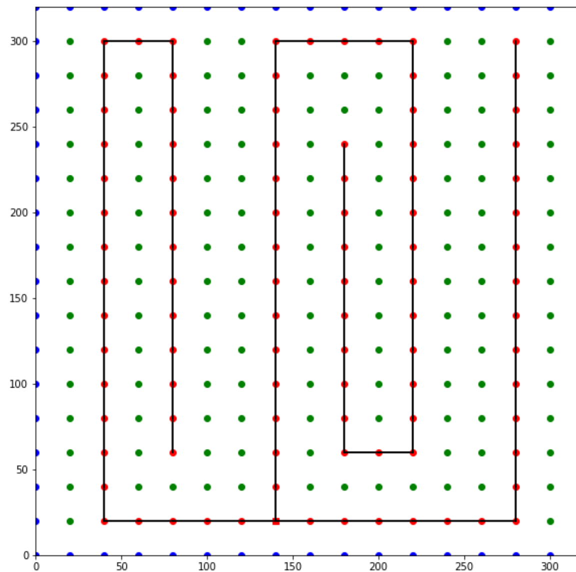


Figure 4: This is the result output from the program listed in the appendix. The car park grid size was 15×15 and and the entrance to the car park was placed slightly off center against the bottom edge of the boundary, marked by a square. The parameters used were $(0.45, 0, 1, 0)$ This result has 96 road network vertices and 129 car parking spaces.

This result is less effective than the previous result with 96 road vertices being present, leading to a car parking space to road vertex ratio of 1.34. The same characteristics are visible from the previous result with many straight lines of road segments, but the road network is much more wasteful by allowing many more car parking spaces to be accessible in two or more ways. This highlights another issue with this approach; the lack of consistent results for a given parameter set over different car park configurations. This issue will also be discussed in section 2.11.1 .

2.10.2 Rectangular car park boundaries

After considering square shaped car park grids, the same process was performed providing a rectangular car park. Initially the entrance was placed as central to one axis of the car park as possible and was required to be adjacent to one of the boundary vertices, much the same as the previous example. The brute force approach was run in the same way where the value of each parameter fell in the interval $[0, 1]$. The number of samples taken

within this interval varied.

The following example highlights some of the merits and issues of this approach with the above setup. The parameters were obtained through the brute force approach to maximisation. The car park grid size was chosen as 10×21 and the entrance to the car park was placed centrally against the bottom edge of the boundary.

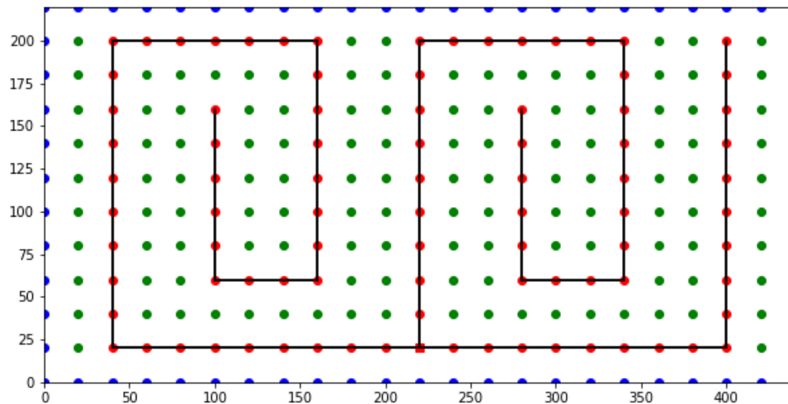
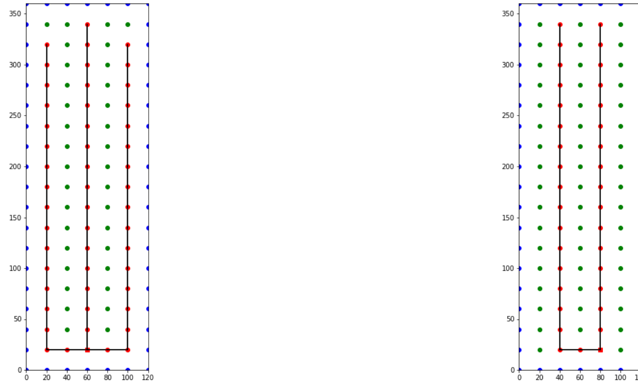


Figure 5: This is the result output from the program listed in the appendix. The car park grid size was 10×21 and and the entrance to the car park was placed centrally against the bottom edge of the boundary, marked by a square. The parameters used were $(0.25, 0, 0.75, 0)$ This result has 88 road network vertices and 122 car parking spaces.

This result illustrates similar merits to the previously shown example in the square car park grid. It shares almost all characteristics in its long straight lines of road with few intersections. It is also a tree. The car park contains a total of 88 road vertices and 122 car parking space, leading to a car parking space to road vertex ratio of 1.39. This is comparable to the previous examples and illustrates that this change of shape has little effect on the end result of our car park grid after optimising over our parameters.

Another note about this result is that it is almost the exact same. The main change is that the parameters found were not exactly the same. However, changing the parameters to be the same as the parameters used in the square grid case from prior shows that this is the same local minima. This highlights the same issue as before relating to the lack of consistency in the way certain parameters behave over different shapes. For the square

and rectangular car park they were virtually the same and both as effective, but moving the entrance only two places in one direction caused a massive change in the behaviour of the growth of the car park. This can similarly be illustrated in this example by moving the car park entrance a few spaces in one direction with the same parameters.

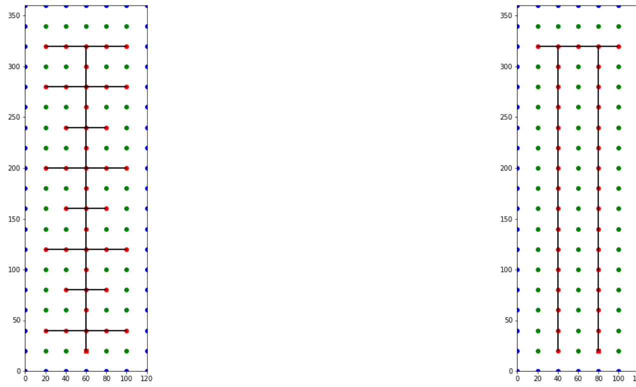


(a) The entrance to the car park was placed centrally against the bottom edge of the boundary, marked by a square. This result has 51 road network vertices and 34 car parking spaces

(b) The entrance to the car park was placed slightly off center against the bottom edge of the boundary, marked by a square. This result has 35 road network vertices and 50 car parking spaces

Figure 6: This is the result output from the program listed in the appendix. The car park grid size was 17×5 and and the entrance to the car park was placed at two different locations. The parameters used were $(0.45, 0.0, 1, 0.0)$

We then also ran the optimisation approach on this car park and found the appropriate parameters. The results of running our algorithm with these parameters are shown below. This also illustrates the exact same issue with the movement of the entrance by a small amount.



(a) The entrance to the car park was placed centrally against the bottom edge of the boundary, marked by a square. This result has 42 road network vertices and 43 car parking spaces

(b) The entrance to the car park was placed slightly off center against the bottom edge of the boundary, marked by a square. This result has 35 road network vertices and 50 car parking spaces

Figure 7: This is the result output from the program listed in the appendix. The car park grid size was 17×5 and and the entrance to the car park was placed at two different locations. The parameters used were $(0.75, 0.0, 0.25, 0.0)$

This issue will be explored more in the following sections.

2.11 Result conclusions

2.11.1 The issue of parameters not being effective across different grid sizes

This issue was mentioned in the above two example results. An issue this approach suffers from is that a mild change to the size of the grid or the location of the entrance to the car park can lead to very different results. This is illustrated in both examples showing that the issue is not isolated. By running the parameter optimisation again on this new configuration, more optimal parameters can be found. The issue with this approach is that we are ideally trying to generate candidate car park configurations, meaning we want to find many potential candidates that would be optimal for a given car park topology. This means that if some mild change to any part of the configuration causes a need to re-optimise over our parameters, this approach becomes wildly inefficient as it is too volatile. If however a few sets of parameters could be used to effectively generate multiple optimal solutions in a variety of cases then this approach would become increasingly effective.

2.11.2 Certain attributes being ignored

This issue was also highlighted by all the example results above. The parameter sets generated from our optimisation approach had several of the parameters being assigned the value of 0. This effectively means that most optimal results took no consideration of several attributes we hope would effectively lead to some form of optimal configuration. Mainly, the attributes referring to the distance a new vertex is from the entrance of the car park and the number of road network vertices connected to a vertex were ignored. This means that our approach does not appropriately take them into consideration when assessing the cost, and a reevaluation of the structure of our cost function may lead to improved results. It could also mean that the attributes we were considering in fact have very little effect on the design of an optimal car park. If the later was to be true, removing

the calculations associated with these attributes would lead to improved performance with little to no change to our end result.

2.11.3 Overall approach conclusions

The car park grids produced from this approach describe a set of potential car park configurations that could be optimised further to achieve an optimal configuration. The car park grids produced from our optimisation were close to being optimal, where the optimal could be visibly observed from our own intuition as being a few car parking spaces different from our result. This means that this approach does generate candidate car parks that could be optimised through the approach of optimising over road networks designed in [1]. However, the issues described above lead this result into being too volatile and inefficient to be applicable to larger car parks or car parks with more complex boundary network topologies; mainly due to having to reevaluate the parameters sets for any change made to the car park configuration. Significant changes to the way the cost function is evaluated for this approach to not be wasteful, due to certain attributes of the car park grid not being effective at finding an optimal configuration, may be necessary to improve upon this approach.

3 The Stochastic Approach

This approach will now be taken to explore the same problem tackled by the deterministic approach and will also be aligned with 1.1.4, with the idea of focusing on the road rather than the packing of the cars and how they may be tiled or patterned. In 1.1.4, the assumption was that the designer would be able to select specific candidate car parks for consideration based on intuition. Instead, we will aim to develop an algorithm that develops these candidate car parks. The algorithm will aim to grow the network of roads itself from a starting point to fill the space.

Unlike the previous deterministic approach, this approach will be stochastic. The idea being that instead of adding roads that minimise a cost function, we will instead define a set of rules that will be used to decide whether a road should be placed in a particular area. Similarly to the previous approach, two separate sets of rules will be used for this process; one for choosing the road we branch from and one for determining how we branch. This then produces a subset of the potential ways of placing a new piece of road, which is then used to randomly select a piece of road to add to the car park. The network should stop growing once it has filled the space completely with either road or car parking space.

This approach is motivated similarly to the previous approach, with the added motivation of our own intuition. The previous approach did not encode our knowledge of how a car park should look from our perspective, to provide it with more freedom. This approach will instead have less freedom to choose but, through random sampling and a set of rules that encode our beliefs, should produce potentially optimal candidate car parks.

This approach will define the road network in the exact same way as the previous approach. This approach will also be considered in the context of the simplified configuration from the previous approach. This means that we already define the car park as a grid of cells, filled with vertices belonging to the graph G. Two vertices are connected via an edge if they are adjacent to each other and belong to the same subset of the graph, R or B. There are differences in terms of how we choose to grow the network, which is what

will be described below. However, the structure of our simplified car park is identical. We do make the added requirement that the road network must be an acyclic graph; a tree. This added requirement is motivated similarly to the previous approach, but will explained in the following sections.

3.1 Explaining how we grow

The way we choose to grow the road network in the deterministic approach is very similar to how we will grow the network with this approach. The only difference between them is that we are not incorporating a cost function into this approach that we will optimise over.

This means that we can presume that we are only branching out from four possible directions from any given vertex North, South, East, and West. In actuality, as we are branching out from an already existing vertex, branching back in the direction of that vertex is impossible. This means that for any given vertex we would be branching out from, there would be at most three potential directions to branch out. This aligns with the previous approach.

The process that will replace the cost function minimisation is the idea of using a rule set that must be followed for a given vertex to be considered as a potential point to be branched out from, with another rule set for determining *how* we branch. This effectively creates a subset of the potential branching opportunities. A random value is then chosen from this set of vertices to determine the next vertex added to the road network. The random way in which this vertex is selected is not trivial and will be explored in section 3.5 .

3.2 Formalising the rule sets for considerable new vertices

With a definition of our simplified car park for this approach, we can move onto determining what factors could be used to help construct our rules. These rules should disclude certain vertices from consideration as the next vertex to be added to the road network.

Initially, there are more obvious factors that should be avoided for a car park to successfully grow. We need to ensure that the cell we are choosing to branch from is filled with a vertex belonging to the road network. We also need to ensure similarly to the previous approach that this vertex can be branched from, and is not surrounded by other road vertices.

Now, the more difficult set of rules that need to encode our intuition are defined in relation to how we branch from a vertex that follows the rules above. One of the more obvious factors should be whether the branching direction is already a vertex belonging to the graph; we can't add a vertex at a position where there is already a vertex belonging to the graph.

Another useful rule may incorporate the idea of only adding vertices that may access potentially inaccessible areas of the car park. This makes sense as if a newly added road accesses no new areas of the car park, then it is adding no benefit. Since this is now a rule that discludes certain vertices from consideration as the next potential vertex to be added to the road network, this attribute cannot be maximised. We instead can formalise this as a potential new vertex must access at least one previously inaccessible cell in the car park.

When we consider optimal car parks, we consider those with ample space in between

roads that can be filled with car parking spaces. How can this kind of behaviour be encoded into rules that we can apply to how we branch? One attempt at doing this is to require that no potential new vertex being added to the road network can be adjacent to a road vertex, excluding the vertex it is branching from. This would then effectively create a buffer of empty car parking spaces around any new potential vertex. This condition would also ensure that the road network would be a tree which is both useful and necessary.

Finally, another rule that could be considered is one that is based on our analysis of the previous approaches results and our own intuition. A car parking space should only be accessible from one road. The results of the previous approach were improved the fewer spaces were accessible in multiple ways. However, this is sometimes unavoidable due to corners and other such intersections. What we can guarantee is that no space should be accessible in more than two ways, as there is no possible benefit or necessity for this. This can be encompassed into another rule that forbids branching out from a vertex in a given direction if doing so would make a car parking space accessible in more than two ways.

These ideas can now be consolidated into the sets of rules that must be followed for a given potential vertex to be considered. We define the neighbourhood of a point indexed by i, j to be the set of all indexes that would be adjacent to the current position:

$$N(i, j) = \{(i, j + 1), (i, j - 1), (i + 1, j), (i - 1, j)\}$$

With the above definition, we can formally define the rule sets as follows:

For any vertex $v \in V$ that is being considered as one that could be branched out from:

1. $v = r_{ij} \in R$ for some i, j.
2. There exists $p \in N(i, j)$ such that $r_p \notin R$.

For any potential vertex to be added at position i, j to the road network, adjacent to a vertex that follows the above rules at position m, n:

1. $r_{ij} \notin R$.
2. There exists $p \in N(i, j)$ such that for all $q \in N(p),\ q \notin R$.
3. For all $p \in N(i, j),\ p \notin V$ unless $p = (m, n)$.
4. For all $p \in N(i, j),\ |\{x \mid x \in N(p),\ r_x \in R\}| \leq 1$.

3.2.1 Completeness and relaxation of rules

An issue that has unfortunately arisen from these definitions is that of completeness. The rules defined above can lead to a situation where the car park may find itself in a situation where it can no longer fill any more unfilled spaces, even though the car park itself is not full. There are two rules that cause this, rules (3) and (4). The car park may find itself in a situation where the only possible way to fill a space may involve placing road adjacent to an already existing road network vertex, or choosing a road network vertex that would cause a car parking space to be accessible in more than two ways.

Because of this, we choose to let these specific rules be relaxed under these circumstances so that the car park may be filled. This shouldn't have any major effect on the end result of the algorithm as the more optimal results will not suffer from this problem.

The relaxed versions of rule (3) will allow the new potential vertex to be adjacent to the boundary, formalised as the following:

$$\text{For all } p \in N(i,j),\ p \notin R \text{ unless } p = (m,n).$$

This definition may still need to be further relaxed for completeness in certain situations, meaning that the even more relaxed version of this rule is to ignore it completely. The relaxed version of rule (4) is much the same; ignore the rule for completeness.

3.3 Filtering the candidate vertices

With our rules now properly defined, we can generate a set of road network vertices that should reflect more of our beliefs in what a car park configuration may be. However, these rules alone would lead to many potential vertices at any iteration of our algorithm to choose from. Many of these potential vertices may also not be the most optimal choice when considering our beliefs. This motivates the idea of filtering this subset of potential vertices based off of some attributes relating to the current state of the road network. Vertices would be removed from consideration if they don't minimise certain attributes, filtering our set of vertices.

A particularly useful attribute could be the amount of filled space in the neighbourhood of a potential vertex, as we are accessing fewer new parking spaces. The more filled space, the less the potential benefit from selecting said vertex. This means that we can filter our subset of potential vertices by selecting those that minimise the amount of filled space in their neighbourhood. This can be formally defined as:

$$m(i,j) \;=\; \sum_{p \in N(i,j)} k(r_p),$$

where i, j is the position of the cell being considered as a potential vertex, k is the piecewise function defined as:

$$k(p) \begin{cases} 1 & r_p \in R \\ 1 & \exists\ t \in \{r_q \mid q \in N(p)\} \text{ such that } t \in R \\ 0 & \text{otherwise} \end{cases}$$

Another attribute that may be useful for similarly assessing a potential new vertex's benefit could be the distance that vertex is from the entrance of the car park. This caries less immediate benefit, but similarly to the deterministic approach this attribute is inspired by systems such as the root system of a plant. As the root system expands, it will spread out in a combined front more-so than searching depth first. This can be taken into consideration also with the way we choose potential vertices. As the roads move further and further from the entrance of the car park, the distance that a road is from the entrance could reduce the likelihood of it being considered in comparison to a road closer to the entrance. This can be formally defined as:

$$n(i,j) \;=\; s(r_{i,j}) + 1 \text{ where } r_{i,j} \text{ is the vertex being branched from.}$$

Where s is the function that returns the shortest path from the given vertex to the entrance of the car park.

With these two attributes filtering our potential vertices, the subset of potential vertices should now be more optimal. Often there would only be one remaining vertex in a smaller car park configuration, but in larger grid sizes this may not be the case. If the vertex set after filtering contains more than one element, a vertex shall be chosen from this subset probabilistic with a uniform distribution, making each potential vertex left equally likely.

3.4 Uncertainty and how we can use it in our filtering of potential vertices

We have now defined the rules that produce potential vertices to be added to the road network. Along with this we have used several attributes, that would seemingly be beneficial to have for a potential new vertex, to whittle down the number of vertices further. Both of these processes were fuelled based on our beliefs of what a car park should be to be optimal. But are these beliefs fact? We cannot be certain. This is why in our filtering of our potential vertices, we should ideally try to incorporate some uncertainty in our decision. We can do this in a variety of ways, but a simple approach to adding uncertainty into our process would be to randomly allow vertices that would be considered less optimal by our filtering to "pass through" the filter and still be considered as a potential vertex. This adds more diverse options to our set of potential vertices for the algorithm to choose from.

The way this filtering could be controlled is with some level of randomness that is associated with both the distance attribute and the filled space attribute considered in our filtering. This then encodes the uncertainty in both of these attributes. The values that should be associated with how random we make this process are not trivial. However, with some idea of how much uncertainty we may want to encode into each step of the filtering process, we can give every vertex that is not found to be most optimal some amount of probability to pass through the filtering steps.

Another way to implement some uncertainty could be in our preference over a direction in which the car park chooses to grow. For example, we may prefer our car parks to grow horizontally more than vertically. This preference can be implemented with some level of randomness that would guide the growth of our road network. This could be done by associating some probability with all vertices that are branched out in specific directions – determining how likely the vertex is to be selected as a potential vertex.

These considerations should then give the algorithm the "freedom" to be more stochastic while also allowing us to control the process. This can be implemented by providing parameters to our algorithm that specify the way in which we choose to randomise our growth. This can be formalised with the following:

1. $s \in \mathbb{R},\ 0 \leq s \leq 1$ is the parameter that represents the probability of a vertex that does not minimise the function m being allowed to pass through the filled space filter described above.

2. $d \in \mathbb{R},\ 0 \leq d \leq 1$ is the parameter that represents the probability of a vertex that does not minimise the function n being allowed to pass through the distance filter described above.

3. $h \in \mathbb{R},\ 0 \leq h \leq 1$ is the parameter that represents the probability of the set of potential vertices that are horizontal in their branching direction being chosen

instead of those that have a vertical branching direction. This would split the set of potential vertices.

These parameters will then need to be provided for this approach to be executed. This then gives us a set of parameters, similar to the previous approach, that we can attempt to optimise over. This will be discussed further in section 3.7.4 .

3.5 Growing the network

The road network will be expanded one vertex at a time, iterating periodically as the previous approach did. The main difference to this approach comes however in the process of analysing the next most optimal vertex to add to the road network. This approach will, at every iteration step, determine all of the potential vertices that could be based that follow the rules we defined prior. This set of potential vertices will be a set of positions of cells in the car park grid defined as:

$$\{(i, j) \mid (i, j) \in N(r), \text{ where } r \in R, r \text{ and } (i, j) \text{ follow the rules defined prior}\}$$

This subset of the road network R is a set of all possibilities without consideration for other attributes that may benefit an optimal configuration. First we split the set of potential vertices into two subsets based on their branching direction. One of these subsets is randomly chosen and is kept as our set of potential vertices. This subset is then filtered by removing potential vertices that are less optimal through the attributes we defined above. Every potential vertex that would have been discluded through this process has a probability of passing through the filter that is parameterised by s, d, h defined in the previous section. After this filtering process, a position is randomly selected from the remaining potential vertices with a uniform distribution. A vertex is then added to the road network at this position, adding edges to the graph that connect the new vertex to all adjacent road network vertices. This process is then repeated.

This repetition of the algorithm must then also have a terminating condition. Similar to the previous approach, the terminating condition for the algorithm that grows the network will be when the car park is considered full. The car park is considered full when all cells in the car park grid are either filled with road network vertices, or are in the neighbourhood of a road network vertex. This condition will be checked at the beginning of each iteration of the algorithm and will stop the execution if found to be the case. After terminating, the car park grid will be returned as our finished configuration.

One issue that is important to highlight is in our splitting of the set of potential vertices based on branching direction. The issue with this is that there may be a situation where we have no potential vertices for branching in a given direction, meaning that one of the subsets will be empty. This subset should never be chosen as a possibility. Therefore our solution to this issue is to default to choosing the non-empty subset if one is empty.

In this process, we also need to consider the relaxation of rules for completeness as described prior. There are two specific rules that were defined that may need to be relaxed for the algorithm that grows the road network to successfully terminate; rules (3) *and* (4). Importantly, we should formalise when we should be relaxing these rules. This can be illustrated rather simply; if the set of potential road network vertices is empty, reacquire the set of potential vertices with rule (3) relaxed and rule (4) ignored. This first step of relaxing our rule set changes the above rules in this way as they are the less significant rule changes that would allow for the car park to terminate in certain situations. If the

above relaxation of the rule set still leads to a set of potential vertices that is empty, then reacquire the set of potential vertices with both rules (3) and (4) ignored. This final relaxation is more extreme as it now breaks our definition of the road network being a tree, but this problem will be dealt with in section 3.7.3 . These two steps for relaxing our rule set will allow the algorithm to always terminate.

3.5.1 Pruning the road network

Fortunately due to our choice of rules in how we choose specific cells as potential new vertex location, we have already dealt with many issues that would require a specific solution to be defined. However, an issue that would not stop the algorithm from terminating, but would lead to less optimal results consistently, is caused by our inclusion of uncertainty. This inclusion of uncertainty into our process should ideally allow the algorithm to be less constrained, but still guided by our beliefs. This, however, may lead to many small changes to the road network which may prove unnecessary after the car park grid is finished. For example, in the following, we can see that the algorithm randomly chooses to add a road network vertex to the side of a straight line of road network vertices, creating an intersection. This path however is never continued and is ignored. As the car park grid is filled, this vertex is completely unnecessary and could easily be removed without changing the number of accessible car parking spaces.

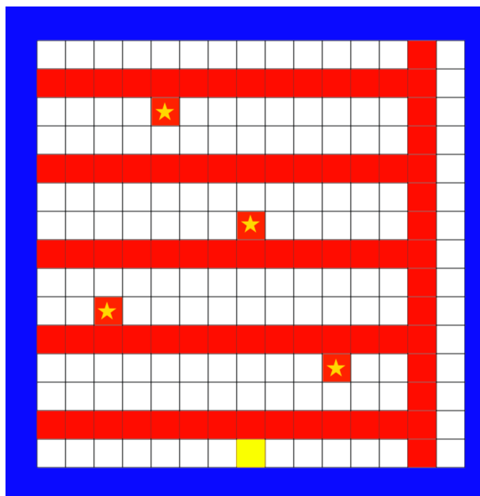


Figure 8: This is an example of an optimal car park, that has multiple small changes to the road network that would make it less so. These road vertices are marked with a star. These vertices would be the vertices removed through the cleaning process.

This is where we can introduce the idea of "cleaning" our road network after it has been filled. After the algorithm terminates, as the car park grid has been filled, we can run a separate algorithm that removes unnecessary road network vertices. Choosing these vertices however is not trivial as we do not want to remove access to car parking spaces, or break off a road segment so that an area of the road network has no direct path to the entrance. Vertices that are able to be removed have to meet the following criteria:

For a vertex $r_{ij} \in R$:

1. All car parking spaces in $N(i, j)$ are adjacent to other road network vertices.
2. removing r_{ij} from the road network must not cause any other vertex to become inaccessible from the car park entrance.

How both of these attributes should be considered practically is a choice we must make, and will be explored in section 3.7.3 .

3.6 Simulation

Unlike the previous approach, we don't have a cost function that we are trying to provide parameters to optimise over our objective function. Our objective function is the same as defined in the previous approach and will be practically implemented in much the same way; minimising the number of road vertices. With this approach now being stochastic, the results we obtain from any given set of parameters may be different. This means that to determine the best result from a given parameter set, we must generate the car park through the above algorithm multiple times. We would then return the result that minimised our objective function. This process can be described as simulation. This simulation is what will be executed whenever a candidate car park is to be generated with a given set of parameters. It is important to note also that any simulation may find multiple car parks with the same objective function value that are the minimum. In this scenario, we will default to taking the first car park grid we find. This should have little change on the end result of our algorithm as the process is stochastic; any execution of the algorithm with the same parameters may not return the same car park.

The number of times we choose to simulate the car park grid with a particular parameter set is another choice we must make. We would like to ideally choose a number of simulations that will find an optimal result in the majority of cases, while also being as small a number as possible to reduce execution time. This is not easily determined and may require simple trial and error to determine an adequate value.

With an ideal number of simulations decided, we would then have to define how we are going to consider our parameter set and how we may determine our best choice of parameters. We cannot strictly optimise over our objective function with a set of parameters as the algorithm is now stochastic. Now we must instead try to determine whether a specific set of parameters is optimal by simulating as described prior. Similar to the previous approach, we can implement a brute force approach to considering our parameter space; returning the parameters that minimised the objective function through simulation after iterating over a significant number of parameters in our parameter space. This approach however no longer has as many guarantees for determining a global minimum. This is because we no longer have a guarantee that the simulated performance accurately reflects the effectiveness of the chosen parameter set – it is only an implication. The number of times we would have to simulate our car park grid would also mean we may have to reduce the number of simulations for this process to be computationally feasible. This entire process above of iterating over the parameter space however could itself be simulated multiple times to mitigate some of the issues described above – providing greater accuracy in our resulting choice of parameters that produce the best car parks.

The underlying issue with all of the above is that more simulation is our best attempt at increasing the accuracy of our assessment of a given set of parameters. This directly increases our computational complexity however and may not be practical if the number of simulations is greater than required. With this considered, the above approach to determining optimal parameters will be considered and implemented in our program.

3.7 Explanation of code

Using the simplified configuration similar to that used in the deterministic approach, we can now create our new algorithm. The program written that executes this approach is in the appendix of this report. The program was written in python version 3.6.3 [8]. Python was chosen as I personally have experience using the development language and are familiar with its libraries. The libraries that proved must useful where NumPy [9] and SciPy [10] which both provided useful utilities to assist with creating our program. The program outputs its results as an image representing the car park grid.

Some aspects of the approach are very similar to that of the previous. The idea of “layers” that was implemented in the previous approach is done in virtually the same way. We only consider two of the attributes now however meaning that some parts of the previous approaches structure are redundant when applied here. These layers provide the same benefit to performance in the idea of memoisation.

3.7.1 Implementation of rule sets and relaxation of rules

We defined our rule sets prior to be in relation to two separate entities; the vertex we are branching from, and the potential new vertex branching out from a vertex. These are implemented in the function shown below.

```
def generateCandidates(self, relaxed=False, veryRelaxed=False):
    minimum = math.inf
    candidates = [[]]
    for i in range(0, self.w):
        for j in range(0, self.h):

            ############################
            # Initial vertex selection #
            ############################
            if not self.road[i+1, j+1] > 0: #Has to be a road
                continue

            if not (i >= 0) and (j >= 0) and (i < self.w) and (j < self.h):
            # Has to be within boundary
                continue

            count = 0
            for m in neighbourhood:
                if self.road[int(i + 1 + m[0]), int(j + 1 + m[1])] > 0:
                    count = count + 1
            if count == 4: # Has to not be surrounded
                continue

            ##################################################
            # Now individual branching opportunity consideration #
            ##################################################
            for m in neighbourhood:
                flag = True

                if self.road[int(i + 1 + m[0]), int(j + 1 + m[1])] > 0:
                # Has to not already be a road
                    continue

                if self.neighbourEmptyCount(int(i + m[0]), int(j + m[1])) < 1:
                    continue

                for n in neighbourhood:
                    if self.road[int(i + 1 + m[0] + n[0]), int(j + 1 + m[1] + n[1])] > 0:
                    # Don't check roads
                        if [int(i+m[0]+n[0]), int(j+m[1]+n[1])] == [int(i), int(j)]:
                            continue
                        else:
                            if not veryRelaxed:
                                if relaxed:
```

```python
                                    if self.road[int(i+m[0]+n[0]+1),
                                                 int(j+m[1]+n[1]+1)] == 1:
                                        continue
                                    else:
                                        flag = False
                                else:
                                    flag = False


                    if self.neighbourRoadCount(i+m[0]+n[0], j+m[1]+n[1]) > 1 and not relaxed:
                    # Check if the road is already accessible in 2 ways
                        flag = False

                if flag:
                    candidates.append([int(i + m[0]), int(j + m[1])])

    return candidates
```

Listing 4: Function for generating the set of potential new vertices to be added to the road network

The above part of the program separates out the two rule sets under the headings "Initial vertex selection", which is the consideration for which vertices we consider the neighbourhood of, and "Individual branching opportunity consideration", which is where we consider branching directions which follow the respective rule set. The above function also incorporates the idea of the relaxation of the rules defined prior. It has two levels of relaxation, as required, and enables this with the toggle of the parameters "relaxed" and "veryRelaxed". These parameters are set to true in the following function when the set of potential vertices is found to be empty.

```python
def iterate(self, probV=1/2, threshRoad=1, threshDist=1):
    while True:
        if self.getAmountEmpty() == 0:
            break

        chosen = self.getNextVertex(False, probV, threshRoad, threshDist)

        if chosen:
            nextRoad = chosen
        else:
            chosen = self.getNextVertex(True, probV, threshRoad, threshDist)

            if chosen:
                nextRoad = chosen
            else:
                chosen = self.getNextVertex(True, probV, threshRoad, threshDist, True)

                if chosen:
                    nextRoad = chosen
                else:
                    print('Error - could not iterate as no potential vertices')
                    self.show()
                    break

        self.update(nextRoad[0], nextRoad[1])

    ret = self.roadCount()

    return(ret)
```

Listing 5: Function for iterating the growth of the network

This shows the implementation of the relaxation of the rules using the variable "chosen". This variable refers to whether a vertex has successfully been chosen from the process of selecting a potential vertex from the set of potential vertices. This process will be considered in more detail in the following sections. This part of the program shows that if

it is unsuccessful in selecting said vertex, it will attempt the same procedure, but with the "relaxed" variable now true. This is similarly repeated with the "veryRelaxed" variable if the result of the relaxed rule set is also empty.

3.7.2 Ordering of filters in the filtering process

Our process to filtering the set of potential vertices was defined prior and listed three separate ways of filtering based on three separate attributes. An important consideration that was not discussed in previous sections is the ordering of these filters. Which attribute should we filter by first? This question becomes more important when we consider that any step of our filtering process does not take into consideration the previous filtering steps. Because of this, we would ideally want to filter in a way that maximises the number of vertices that may proceed at each step, as this reduces the likelihood of potential vertices not being considered when they could. This leads to the following function:

```
def getNextVertex(self,relaxed=False,probV=1/2,threshRoad=1,threshDist=1,veryRelaxed=
    False):

    candidateVertices = self.generateCandidates(relaxed, veryRelaxed)

    candidateVertices = self.splitCandidates(candidateVertices)

    if candidateVertices == [[],[]]:
        return False
    if candidateVertices[0] == [[]]:
        candidateVertices = candidateVertices[1]
    elif candidateVertices[1] == [[]]:
        candidateVertices = candidateVertices[0]
    else:
        if np.random.uniform(0,1) > probV: #Horziontal or vertical choice
            candidateVertices = candidateVertices[0]
        else:
            candidateVertices = candidateVertices[1]

    candidateVertices = self.findLeastRoadCandidates(candidateVertices, threshRoad)

    candidateVertices = self.findClosestCandidates(candidateVertices, threshDist)

    chosen = np.arange(0, len(candidateVertices))

    chosen = candidateVertices[int(np.random.choice(chosen))]

    return chosen
```

Listing 6: Function that performs the filtering steps on our set of potential vertices, returning the final, randomly chosen value

The above shows the ordering we have chosen for our filtering steps. Our first step chosen is "splitCandidates" which implements the separation of the potential vertices into two subsets based on their branching direction. This was chosen as the first filtering step as it reduces the number of potential vertices less than both previous steps. The above part of the program also shows our consideration for choosing one of the subsets depending on if one is empty.

After this filtering step, the choice between which filtering step goes next is not as simple of a decision. "findLeastRoadCandidates" and "findClosestCandidates" implement the filtering by the amount of filled space around a potential vertex and the distance that potential vertex is from the entrance respectively. Both of these filtering steps will significantly reduce the number of potential vertices and will depend more heavily on the current state of the road network. Because of this, both orderings were considered, with

little to no difference found in results. This means that our choice of which filtering step to proceed with after "splitCandidates" can be made rather arbitrarily.

After all the filtering steps, a random value is chosen from the remaining set by choosing a random number that represents the index of the potential vertex to be chosen.

3.7.3 Incorporating uncertainty into the filtering process

We discussed previously the parameters s, d, h that represent the uncertainty we have in our filtering process. s, d, h are provided to our algorithm to encode how much uncertainty we have in any one step of the filtering process. The way these parameters have been implemented will be shown in the following parts of the program. The first part that will be addressed is the parameter h representing our preference of branching direction. This is implemented in the following part of the previous function shown:

```
if np.random.uniform(0,1) > probV: #Horziontal or vertical choice
        candidateVertices = candidateVertices[0]
    else:
        candidateVertices = candidateVertices[1]
```

Listing 7: Function that performs the filtering step that splits the potential vertices into two subsets based on their branching direction.

This part of the previous function is what illustrates our preference for a specific branching direction. The variable "probV" represents the negative of the parameter h. This variable represents the probability we associate with how likely it is for the algorithm to prefer the vertical branching direction. This means that the parameter we provide to our algorithm is effectively $1 - h$. The way we execute this choice in branching direction is to randomly draw a value from a uniform distribution, in the interval $[0, 1]$. If this value is greater than the value of "probV", it will choose the subset of potential vertices with horizontal branching direction. With this implementation, the following scenario will happen with probability $1 - probV = h$; which is our aim. The other scenario represents choosing the subset of potential vertices with vertical branching direction, which should clearly happen with probability $probV$.

Similarly, the same technique for implementing our random choice based on our parameters s, d is used in the filtering steps for the other two considered attributes. This is shown below, where s, d are represented by $1 - threshRoad$ and $1 - threshDist$ respectively:

```
def findClosestCandidates(self, vertices, threshDist=1):
    minimum = math.inf
    minimumSet = [[]]
    randomSet = [[]]
    for v in vertices:
        if v == []:
            continue
        dist = self.getDistance(v[0], v[1])

        if dist == minimum:
            if minimumSet == [[]]:
                minimumSet[0] = v
            else:
                minimumSet.append(v)
        elif dist < minimum:
            minimumSet =[v]
            minimum = dist
    for v in vertices:
        if not v in minimumSet:
            if np.random.uniform(0,1) > threshDist:
```

```
                if randomSet == [[]]:
                    randomSet[0] = v
                else:
                    randomSet.append(v)

    if minimumSet == [[]]:
        return [[]]
    elif randomSet == [[]]:
        return minimumSet

    minimumSet = minimumSet + randomSet

    return minimumSet


def findLeastRoadCandidates(self, vertices, threshRoad=1):
    minimum = math.inf
    minimumSet = [[]]
    randomSet = [[]]

    for v in vertices:
        if v == []:
            continue
        road = self.getFilledSpaces(v[0], v[1])

        if road == minimum:
            if minimumSet == [[]]:
                minimumSet[0] = v
            else:
                minimumSet.append(v)
        elif road < minimum:
            minimumSet = [v]
            minimum = road

    for v in vertices:
        if not v in minimumSet:
            if np.random.uniform(0,1) > threshRoad:
                if randomSet == [[]]:
                    randomSet[0] = v
                else:
                    randomSet.append(v)


    if minimumSet == [[]]:
        return [[]]
    elif randomSet == [[]]:
        return minimumSet

    minimumSet = minimumSet + randomSet
    return minimumSet
```

Listing 8: Functions that perform the filtering steps over the distance the potential vertex is from the entrance, and the amount of filled space surrounding a potential vertex respectively.

3.7.4 Implementation of the cleaning process

The cleaning process that will be implemented after the car park grid has been filled will aim to reduce wasted road vertices in the ways described prior. The rules a vertex must follow to be considered unnecessary, and can therefore be removed, are not necessarily simple to implement. Rule (1) is slightly more simple, involving a calculation over the neighbourhood of every vertex in the neighbourhood of a potential vertex. This process will be described below. The far less obvious rule to implement is rule (2). Determining when a vertex may cause another to become inaccessible at face value requires calculating many paths along the road network to determine whether just one vertex may be removed. A simpler approach that can achieve the same objective may come in using the distance layer we have taken from the deterministic approach. The idea is that if a road network vertex r_{ij} has the largest distance value in the set of $N(i,j) \cup (i,j)$ then it will not cause

any other vertex to become inaccessible from the road network entrance if removed.

Proof. Suppose that r_{ij} has the maximum distance value in the set $N(i,j) \cup (i,j)$. If this is true then that means that for all vertices in $N(i,j)$, the distance value is less than that of the distance value at (i,j). This means that vertex r_{ij} must not belong to the path that connects any element in its neighbourhood to the entrance, because vertex r_{ij} would only belong to any of these paths if it had a distance value less than one of its neighbours, continuing the shortest path from itself to that of another vertex. This means that removing the vertex r_{ij} from the road network would not break any paths and would subsequently not cause any other vertex to become inaccessible from the entrance. □

This approach should therefore suffice in determining vertices we can remove from the road network. Both of the above rules are implemented in the following part of the program:

```
1  def clean(self):
2      vertices =[[]]
3      deleted = False
4      for i in range(0, self.w):
5          for j in range(0, self.h):
6              if self.road[int(i+1),int(j+1)]:
7                  flag = True
8                  for m in neighbourhood:
9                      if (i+m[0]>=0) and (j+m[1]>=0) and
10                     (i+m[0]<self.w) and (j+m[1]<self.h):
11                      if not self.distance[i,j] >= self.distance[int(i+m[0]),
12                                                                 int(j+m[1])]:
13                             flag = False
14                             break
15                 if flag:
16                     count = 0
17                     count2 = 0
18                     for m in neighbourhood:
19                         if (i+m[0]>=0) and (j+m[1]>=0) and
20                         (i+m[0]<self.w) and (j+m[1]<self.h):
21                             flag2 = False
22                             if self.road[int(i + m[0] + 1), int(j + m[1] + 1)]:
23                                 continue
24                             count = count+1
25                             for n in neighbourhood:
26                                 if (i+m[0]+n[0]>= 0) and (j+m[1]+n[1]>=0) and
27                                 (i+m[0]+n[0]<self.w) and (j+m[1]+n[1]<self.h):
28                                  if [int(i+m[0]+n[0]), int(j+m[1]+n[1])] == [i, j]:
29                                      continue
30                                  else:
31                                      if self.road[int(i+m[0]+n[0]+1),
32                                                   int(j+m[1]+n[1]+1)]>0:
33                                          flag2 = True
34                             if flag2:
35                                 count2 = count2+1
36
37                     if count2 == count:
38                         self.deleteVertex(i, j)
39                         deleted = True
40                 else:
41                     continue
42     return deleted
```

Listing 9: Function that performs the cleaning process described above.

The above illustrates what we have described. Something that has not been discussed however is how this cleaning process is applied. The idea behind this cleaning process is to remove all the vertices that are unnecessary. Under this scheme however, some vertices that are unnecessary may not be immediately determined as such. This means that this cleaning process must be repeated until there are no more changes to the road network. This is implemented through the variable *deleted* which refers to whether the

road network removed any vertices during this execution of the function. If it returns false, then the cleaning process is terminated and the final car park grid is output.

3.7.5 Brute force approach to optimising over our parameters

We have now illustrated the entire approach of generating a car park grid with a given parameter set. With this, we can simulate the process of generating the car park grid multiple times with any given set of parameters to determine the most optimal result on average. But as described prior, we want to iterate over our parameter space to determine which specific parameter sets may create the best possible results on average. This will be done in a similar way as to the previous brute force minimisation used in the deterministic approach. We will iterate over the parameter space for each parameter in the interval $[0, 1]$, as they are probabilities. At every set of parameters, the algorithm will be simulated a fixed integer number of times. The minimum value of our objective function will be kept along with the parameters used to find this value; these will be returned after the execution has finished for any specific set of simulations. The number of samples to take within each parameters interval range will be provided as an integer to our brute force function. The more samples taken, the greater the likelihood that the specific parameter found is most optimal. However, this also increases execution time. The part of the program listed below shows our function implementing this approach:

```python
def simulate(k, p1, p2, p3, car_park_size, entrance, minimum=math.inf):
    min_flag = False
    for n in range(0,k):
        CarPark = Grid(car_park_size[0], car_park_size[1], entrance[0], entrance[1])
        CarPark.iterate(p1, p2, p3)
        clean_flag = True
        while clean_flag:
            clean_flag = CarPark.clean()

        valu = CarPark.roadCount()
        if valu < minimum:
            minimum = valu
            min_flag = True
            MinCarPark = copy.deepcopy(CarPark)
    if min_flag:
        print( minimum, "Road network vertices\n with parameters", p1,p2,p3 )
        MinCarPark.show()
    return minimum

def bruteForceApproach(simulation_n, car_park_size, entrance, sample_n):
    minimum = math.inf
    for i in np.linspace(0, 1, sample_n):
        for j in np.linspace(0, 1, sample_n):
            for k in np.linspace(0, 1, sample_n):
                minimum = simulate(simulation_n, i, j, k, car_park_size, entrance,
    minimum)
```

Listing 10: Function that simulates the car park grid generation process and the function that performs the brute force approach to determining our optimal parameters as described above.

3.8 Results

The below figures and results will show how this stochastic approach performed under multiple different situations. The program outputs an image which represents the car park grid. Every vertex that is coloured blue represents the boundary of the car park. Any cell coloured red represents a vertex belonging to the road network, where the square red vertex denotes the entrance of the car park. Edges between vertices belonging to the road network are drawn as black lines. All green vertices represent a car parking space. The program used to generate these results can be found in the appendix.

3.8.1 Square car park boundaries

Initially the car park grids that were considered were those with a square boundary shape. For these car park shapes, the entrance was initially placed as central to one axis of the car park as possible, and was required to be adjacent to one of the boundary vertices. The brute force approach was run with parameters where the value of each parameter fell in the interval $[0, 1]$. The number of samples taken within this interval varied. The more samples chosen, the longer the time the program took to return a successful result. This is deliberately chosen to be the same test as performed in the deterministic approach so the two can be compared.

An example of one of these results is the following figure. This result was run with a set of parameters found through the brute force approach. The car park grid size was 15×15 and and the entrance to the car park was placed centrally against the bottom edge of the boundary.

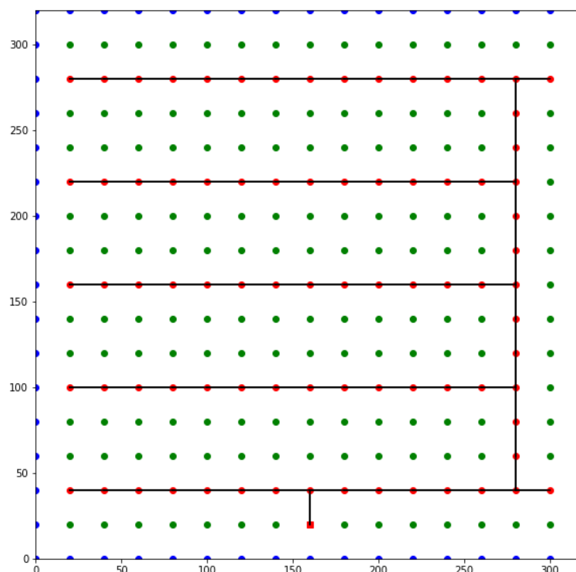


Figure 9: This is the result output from the program listed in the appendix. The car park grid size was 15×15 and and the entrance to the car park was placed at centrally against the bottom edge of the boundary, marked by a square. The parameters used were $(0.0, 1.0, 0.9)$ This result has 81 road network vertices and 144 car parking spaces.

This results merits are visible immediately based on our own intuition of how a car park may be designed. The car park meets the definition of a tree as a requirement, unlike our previous approach. The number of road vertices is 81 against 144 car parking spaces leading to a car parking space to road vertex ratio of $0.1\overline{7}$. This result is an improvement over the result garnered in the deterministic approach with the same dimensions. It has many characteristics we would associate with an ideal car park with many straight lines of road vertices and few intersections. Importantly, the spacing between rows of car parking spaces is consistently two car parking spaces; a previously highlighted, ideal quality. When compared to our optimal solution 2, we can see that *it is in fact the optimal solution*, which is both significant and promising.

An important attribute to note about this result is the parameters it found that more consistently minimised our objective function. The first parameter, associated with our preference of branching direction, is set to 0. This has several implications as to the execution of our algorithm and how we may generate other results. This will be discussed in the result conclusions.

These exact same parameters were used to simulate on the exact same sized car park grid, but with the entrance position changed. This test was also performed in the deterministic approach, and is done here for comparison. The idea behind testing this was to determine whether we had found parameters that were not only optimal for a specific configuration, but could be applied to others. The following shows this test, with the entrance moved several space to the left.

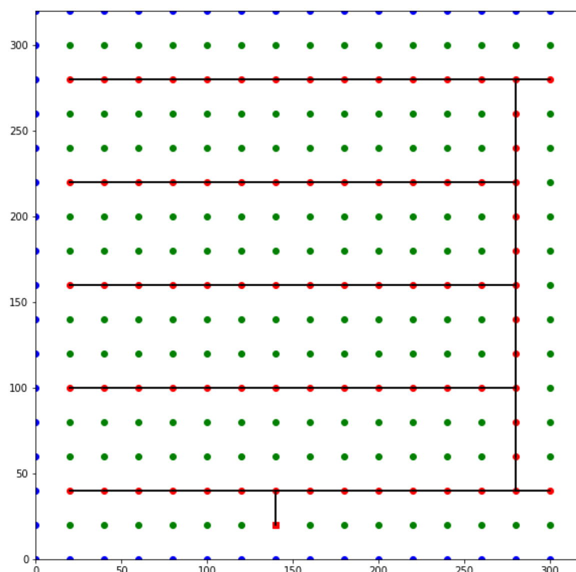


Figure 10: This is the result output from the program listed in the appendix. The car park grid size was 15×15 and and the entrance to the car park was placed at slightly off center against the bottom edge of the boundary, marked by a square. The parameters used were $(0.0, 1.0, 0.9)$ This result has 81 road network vertices and 144 car parking spaces.

This result is virtually the exact same result as the previous and is clearly also *the optimal* when compared to 2. There is only one road different in the two results, the entrance. This is a more promising result that would seem to highlight that the set of parameters found is more applicable to other situations. To test this further, moving the entrance again but more than previous may lead to a greater difference in results. Our choice of entrance will be $(0, 1)$ specifically as this entrance placement, based off of the results gained so far, would seem to give opportunity for an even more efficient road network configuration. Whether this value is found through simulation is a secondary part of this next test.

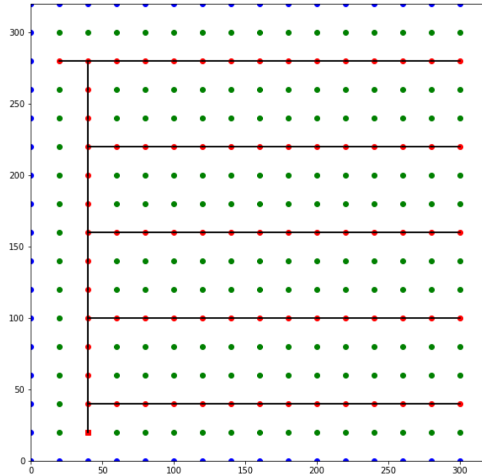


Figure 11: This is the result output from the program listed in the appendix. The car park grid size was 15×15 and the entrance to the car park was placed far off center against the bottom edge of the boundary, marked by a square. The parameters used were $(0.0, 1.0, 0.9)$ This result has 80 road network vertices and 145 car parking spaces.

This result also shows virtually the same result. The placement of the entrance gave the opportunity to find a more optimal solution, which the algorithm found through simulation. This result has 80 road network vertices against 145 car parking spaces. This means that the car park grid has a car parking space to road network vertex ratio of 1.81. This seems to show that the parameters found do have more application across multiple different configurations of car park grids. *This result is the absolute optimal found for a grid of size* 15×15.

3.8.2 Rectangular car park boundaries

After considering square shaped car park grids, the same process was performed providing a rectangular car park. Initially the entrance was placed as central to one axis of the car park as possible and was required to be adjacent to one of the boundary vertices, much the same as the previous example. The brute force approach was run in the same way. The number of samples taken within this interval varied. This test was deliberately made to be similar to the test in the previous approach, that considered rectangular car park boundaries, for comparison.

The following example highlights some of the merits and issues of this approach with the above setup. The parameters were obtained through the brute force approach to maximisation. The car park grid size was chosen as 10×21 and the entrance to the car park was placed centrally against the bottom edge of the boundary.

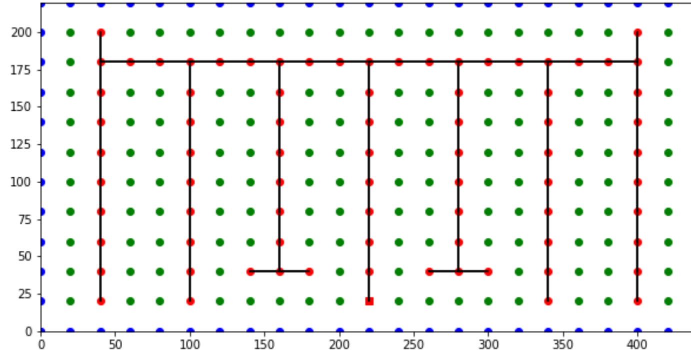


Figure 12: This is the result output from the program listed in the appendix. The car park grid size was 10×21 and the entrance to the car park was placed centrally against the bottom edge of the boundary, marked by a square. The parameters used were $(1.0, 1.0, 0.75)$ This result has 79 road network vertices and 131 car parking spaces.

This result shares many of its merits with the last two examples from the square car park grid. It contains lines of road network vertices, has few intersections, and maintains appropriate space between rows of road network vertices. The number of road network vertices was 79 against 131 car parking spaces. This leads to a car parking space to road network vertex ratio of 1.66. This value is not too dissimilar to those of the previous examples, highlighting that the solution found here is similarly effective.

Importantly, the parameters found to be most effective in this situation are similar to that of the previous examples. The only major difference between them is that the first parameter for preference in branching direction has effectively flipped; it now gives entire preference to vertical branching. This highlights an aspect of the algorithm that was mentioned prior, that will be discussed in section 3.8.5 .

An issue that is shown in this approach is that there is clearly some improvement that could be made. More specifically, there are two points on this result where we, based on our intuition, could remove two vertices. These points can be described as the “T shaped end points”. This is caused by an aspect of how we choose to relax our rules during certain parts of the algorithm depending on circumstance. This issue will be discussed in detail in 3.8.6.

Due to the success from the tests in the square car park grid that related to changing the car park entrance position, a similar test will be performed here. With the same

sized car park grid and the same parameters, the car park can be simulated over to find an optimal configuration. The main difference in this test will be that the entrance position will be significantly different to that of the first; from the bottom boundary edge to the left edge, placed centrally. The result to that test is shown below.

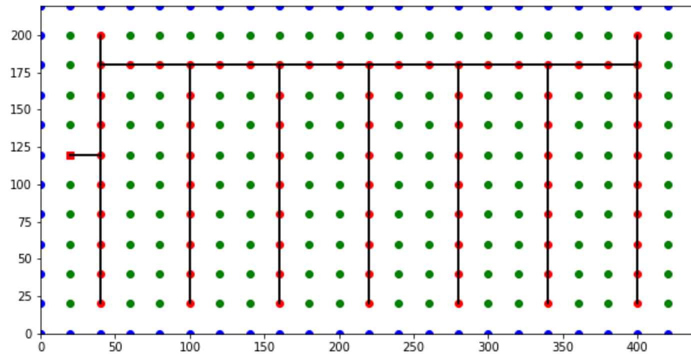


Figure 13: This is the result output from the program listed in the appendix. The car park grid size was 10×21 and the entrance to the car park was placed at centrally against the left edge of the boundary, marked by a square. The parameters used were $(1.0, 1.0, 0.75)$ This result has 78 road network vertices and 132 car parking spaces.

This result is promising as it shows very little change to the car park road structure, except for a slight improvement of one extra car parking space. This shows that even with a large change to the car parks configuration, the algorithms growth process is not significantly changed. This will be discussed further in section 3.8.4 . The resulting ratio of car parking spaces to road network vertices for this example is 1.69.

3.8.3 Result conclusions

3.8.4 Performance in comparison to deterministic approach

We have deliberately chosen similar examples when generating the results from both approaches. This is so that we may compare their performance directly to determine which one is more effective at tackling our problem.

When comparing our two sets of results, we can see noticeable differences at a glance. For example, the first result from the first approach 3 seems to create a pattern that almost represents hooks in a repeating pattern. The chosen parameter set for this result clearly influences the algorithm to make this specific type of arrangement, regardless of

the orientation, entrance location, and car park dimensions. This happens at the possible detriment to overall performance. This can be illustrated by looking similarly at 4. Our current approach however seems to be far less influenced be these small changes, as illustrated by 9 and 10. Both of these figures also illustrate optimal car parks for their configurations. This seems to show that this stochastic approach achieves our ultimate goal of generating optimal car park configurations.

When comparing the two sets of results, we must also consider the details of the performance; the car parking space to road network vertex ratio. In 3 and 5, the ratio found was 1.65 and 1.39 respectively. This in comparison to the results from 9 and 12, $0.1\overline{7}$ and 1.66 respectively, shows a significant improvement in this approach. These are also not the most optimal configuration found for the car park boundaries at hand, where 11 and 13 show an even greater increase in performance with ratios 1.81 and 1.69.

Another important aspect of comparison is the efficiency of the algorithms implemented in both approaches. Through testing, this approach was significantly less computationally demanding than the deterministic approach. Increasing the number of simulations lead to an increased time of execution, but for the number of simulations required to determine the car park grid that minimised the objective function, the execution time was still less on average. Especially considering the brute force approach to minimisation used in both approaches, this approach was significantly quicker. This process however was still not necessarily quick enough to consider running on every configuration as that would still be too demanding, much the same as the deterministic approach.

This increase in performance will be partially related to having to compute less attributes at every iteration of the growth algorithm. This approach used less information about the road network at every iteration of the growth algorithm to achieve this improved result. It is also important to note that as this approach used to determine the car park grid that should be returned probabilistic simulation, the crossover of parameters was greater than in the previous approach; if the value of a parameter was used to produce a result, the exact same parameters were not required to produce the same result.

3.8.5 Parameter choice and general application

After evaluating our results for this approach, a pattern in the parameters preferred emerges. Through the brute force approach to determining optimal parameters, our results would often resemble $(1, 1, 0.75)$ or $(0, 1, 0.9)$. This trend is consistent through almost every run of our algorithm and highlights some important factors of how our algorithm is best used.

Every single result found from our brute force optimisation had the first parameter h set to either 0 or 1. What does this imply? It means that the best use of our approach was to only consider horizontal or vertical branching opportunities at every iteration of our growth algorithm, removing all uncertainty from this step. When there were no branching opportunities left in the preferred direction, our algorithm defaulted to selecting the non-empty subset of branching directions as described prior. This effectively can be described as choosing a direction in which the road network should have long lines of road network vertices, and only when you have reached a point where that isn't possible do you consider other options. This consistency would seemingly provide a large benefit as it would reduce a fair amount of the chaotic nature of an approach that is mostly random.

The other two parameters, on the whole, fell in the interval $[0.7, 1]$ depending on the circumstance. This also makes sense as although greater structure is created by setting the first parameter to 1 or 0, some guidance in how it may choose to branch out after it has reached a point where it can no longer branch in the preferred direction may lead to optimal results. The variance in the choice of these other parameters coming from the fact that this process is random, and certain parameters will make certain results more likely, but not guaranteed. This small uncertainty would potentially lead to results that may be more optimal and not as likely to be considered if no uncertainty had been encoded.

So after this we can see that a result of the form $(0, a, a)$ or $(1, a, a)$, where $a \in [0.7, 1]$, will in almost all cases be found to produce an optimal solution for a given car park grid, given enough simulations. The two different parameter sets will prefer the two different branching directions. This means that the result, depending on the choice of the first parameter, will either make rows or columns of road network vertices with road network vertices connecting them. The fact that these two parameter sets apply to so many cases is promising, as it means that the process of determining optimal parameters for a given car park configuration may not be necessary. This is of specific benefit as this is the most costly aspect of this approach in terms of computation. This in comparison to the deterministic approach shows far broader application and becomes far less computationally demanding overall as you do not need to run the brute force approach to minimisation of the objective function.

3.8.6 Issue of how our rule choices negatively effect road network

One issue that was observed prior in 12 was related to the relaxation of rules. The circumstance that was found in this situation is related to the two visible aspects of the car park grid where it branched in a away from the boundary to access two car parking spaces that were accessible in a simpler way. This is due to the fact that for the algorithm to consider branching so that it is adjacent to the car park boundary, it must relax its rule set. This means that it must have no other option, before relaxing the rule set so that it may be considered as a possibility. This will very rarely happen as in this specific circumstance, there is a way to access vertices in a way that doesn't require relaxing the rule set. This issue is difficult to tackle as it will cause the algorithm in the vast majority of cases to choose a few vertices to add to the road network that do not contribute to the most optimal solution. This may require greater consideration, but an initial thought of improvement may be the idea of adding a possibility for a vertex that requires the relaxed rule set to be considered to pass through the "filter" with some probability. This idea then turns the relaxation of rules to instead be another filter that we associate with a probability.

3.8.7 Overall approach conclusions

This approach, taking a set of parameters and a car park grid with specified dimension, was able to produce a filled car park grid which approximates a car park configuration in some real context. The car park grids produced were found to be the optimal in most cases, with some adjustment required in certain circumstances to reach an optimal solution. These generated car parks can be used as candidates for a potential car park topology that could be optimised through the approach of optimising over road networks designed in [1]. This approach in comparison to the deterministic approach is less computationally demanding and has been shown to have a greater overall performance, assessed by the

ratio of car parking spaces to the amount of road. This approach in comparison to the previous deterministic approach is also far more applicable practically and can be applied in a much broader range of contexts. These contexts could include a more complex car park boundary or the inclusion of obstacles. The issues described above do have an effect on the potential of this approach, but potential solutions have been presented that may help mitigate these factors. Due to the performance provided with only a few set of parameters, an optimal configuration may quickly be determined through this approach of simulating the algorithm that grows our network.

4 Conclusions and possible extensions

We have demonstrated two simple approaches to generating multiple candidate car park networks using a branching algorithm. The deterministic approach attempts this through a cost function that is optimised over, while the stochastic approach uses simulation and random selection. Both algorithms dynamically grow the road network to fill the space inside the car park, reaching an optimal result. These algorithms have been tested on rectangular car park boundaries, and have not been explored in more complicated configurations. These tests however illustrate the efficacy of these approaches and show their potential application. These candidate car parks can then be used in conjunction with [1] to create a fully automated approach to optimising over a road network, as the need for candidate road networks can be provided by our approaches.

The possible extensions to this project that have been considered are listed below.

4.1 Testing both approaches with more complex car park configurations

Both of the approaches were tested with a variety of car park grid configurations. Different entrance locations and different dimensions were specifically considered. more complex boundaries and obstacles, although included in the scope of this problem, were not directly considered. This was mainly due to the focus of our results being on the comparison between the two approaches, and the deterministic approach did not seem effective enough to be considered in the more complex context. The stochastic approach did achieve more promising results however, meaning that testing more complex configurations of the car park with this approach would be a logical next step. For example, the algorithm could be tested in a car park grid with a rectilinear boundary and with obstacles present.

4.2 Using a generative model and the Fourier transform to construct the road network

When judging what we would consider to be an optimal car park, some noticeable consistent traits can be found. Almost all of the optimal car park road network configurations, in a non-simplified context, can be described as a set of repeated rows or columns of road network vertices that are at some angle in relation to the car park boundary. This could lead to a natural description of a car park being said rows or columns, repeating at some fixed interval. This can then be naturally described by some frequency, for which rows are placed.

This then leads into the idea that the road network can be decomposed into the relevant frequencies through a Fourier transform. This then gives a description of a road network in the frequency domain. This leads to a possible learning context where you can

construct a generative model around the boundary of the car park and the associated frequency domain representation of the car parks road network. This model could be trained on car parks that exist in the real world through images, with a function associated that represents the boundary; perhaps an approximated boundary so that the function is less complex. With this, the idea would then be to be able to provide the model a boundary function, for which it would then be able to return a result in the frequency domain. This would then be converted back through the Fourier transform to construct the road network.

4.3 Using genetic algorithms to improve the road network structure

We briefly mentioned genetic algorithms earlier in this report. In that context, genetic algorithms would have been used on the parameters provided to our algorithm to optimise over the road network. This idea however employs a different use of genetic algorithms. Instead, you would use genetic algorithms with some base candidate road network. The genetic algorithm would add or remove vertices depending on the "genes" passed forward from other generations of the road network. This would be executed over the same objective function used in our deterministic and stochastic approach. This idea was thought of as a natural extension to the issue found in our stochastic approach, where some uncertainty in our relaxation of the rule set was considered as a solution to rules discluding potential optimal configurations. The use of genetic algorithms would encode this uncertainty in each generation of the road network.

4.4 Reinforcement learning as an approach to optimising the road network

A reinforcement learning approach [11] could also be considered for this type of problem. The idea behind a reinforcement learning approach is to take a state of the current car park, and using a policy, determine what action the algorithm should perform, where doing any action on any state has an associated cost. This action will change the state of the car park to a new state, which is then used to continue repeating the process. The idea behind this approach would be that we do not know the policy that would optimally create our car park. We would instead try to learn our policy based off of the total cost that the car park would have garnered after executing our algorithm that grows the network. This is not too dissimilar to our deterministic approach, as you could describe our deterministic approach as a brute force reinforcement learning task. In this approach however, instead of using our intuition of what would create an ideal car park to make a cost function, we may make this variable latent. We would still have a cost function, but it would now be used to assess our performance, instead of determining the most optimal choice at any given iteration.

5 Appendices

5.1 Python code for the Deterministic approach

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import copy
4 import sys
5 GraphFigure=None
6 axs=None
7 from scipy.optimize import fmin
8 import math
9
```

```
10 neighbourhood = [[0,1], [0,-1], [1,0], [-1,0]]
11
12 def CreateRoadLayer(width, height, startx, starty, obstacleConstant=1):
13     grid = np.zeros((width+2, height+2))
14     grid[:,0] = obstacleConstant
15     grid[:,height+1] = obstacleConstant
16     grid[0,:] = obstacleConstant
17     grid[width+1,:] = obstacleConstant
18     grid[startx+1,starty+1] = 1
19     return grid
20 #These two layers don't need to consider the boundary so they are smaller. this needs to
   be
21 #reflected in the indices used to traverse elements
22 def CreateDistanceLayer(width, height, startx, starty):
23     grid = np.zeros((width, height))
24     grid[startx,starty] = 1
25     return grid
26
27 def CreateFilledSpaceLayer(width, height, startx, starty):
28     grid = np.zeros((width, height))
29     for m in neighbourhood:
30         #Will probably need to add protection for bounding on other side too
31         if (startx + m[0] >= 0) and (starty + m[1] >= 0):
32             grid[startx + m[0], starty + m[1]] = 1
33     return grid
34
35 class Grid:
36     def __init__(self, width, height, startx, starty):
37         self.w = width
38         self.h = height
39         self.start = [startx, starty]
40         self.road = CreateRoadLayer(width, height, startx, starty,1)
41         self.distance = CreateDistanceLayer(width, height, startx, starty)
42         self.filledSpace = CreateFilledSpaceLayer(width, height, startx, starty)
43
44     def show(self):
45         fig = plt.figure(figsize=(10,10))
46         ax = fig.add_subplot(111)
47         ax.set_xlim([0,(self.h+1)*20])
48         ax.set_ylim([0,(self.w+1)*20])
49         for x in range(0, self.w+2):
50             ax.plot(0, x*20, 'bo')
51             ax.plot((self.h+1)*20, x*20, 'bo')
52         for y in range(0, self.h+2):
53             ax.plot(y*20, 0, 'bo')
54             ax.plot(y*20, (self.w+1)*20, 'bo')
55         for i in range(1, self.w+1):
56             for j in range(1, self.h+1):
57                 if self.road[i,j] > 1:
58                     if i == self.start[0] + 1 and j == self.start[1] + 1:
59                         ax.plot(j*20, i*20, 'rs')
60                     else:
61                         ax.plot(j*20, i*20, 'ro')
62                     for m in neighbourhood:
63                         if (i + m[0] >= 1) and (j + m[1] >= 1) and (i + m[0] <= self.w)
   and (j + m[1] <= self.h):
64                             if self.road[i + m[0], j + m[1]] > 0:
65                                 ax.plot([(j + m[1])*20, j*20],
66                                         [(i + m[0])*20, i*20], 'k-')
67                 else:
68                     ax.plot(j*20, i*20, 'go')
69         ax.set_aspect(1)
70         plt.savefig('result' + str(self.w) + str(self.h) + str(self.start[0]) + str(self.
   start[1]) + ".png")
71         plt.show()
72
73     def updateRoad(self, i, j):
74         #make sure to pass i , j as 1 extra for border
75         adjacentCount = 0
76         for m in neighbourhood:
77             if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w+2) and (j + m
   [1] < self.h+2):
78                 if self.road[i + m[0], j + m[1]] > 0:
```

```
79                      adjacentCount = adjacentCount + 1
80                      self.road[i + m[0], j + m[1]] = self.road[i + m[0], j + m[1]] + 1
81
82          self.road[i,j] = adjacentCount + 1
83
84      def updateDistance(self, i, j):
85          adjacentCount = 0
86          minDistance = math.inf
87          for m in neighbourhood:
88              if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m[1]
    < self.h):
89                  if self.road[i + 1 + m[0], j + 1 + m[1]] > 0:
90                      if self.distance[i + m[0], j + m[1]] < minDistance:
91                          minDistance = self.distance[i + m[0], j + m[1]]
92          self.distance[i, j] = minDistance + 1
93
94          for m in neighbourhood:
95              if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m[1]
    < self.h):
96                  if self.road[i + 1 + m[0], j + 1 + m[1]]:
97                      if not self.distance[i + m[0], j + m[1]] == minDistance:
98                          self.updateDistance(int(i + m[0]) , int(j + m[1]) )
99
100     def updateFilledSpace(self, i, j):
101         self.filledSpace[i, j] = 1
102         for m in neighbourhood:
103             if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m[1]
    < self.h):
104                 #the idea with this being that we want to view the road as also a filled
    space
105                 self.filledSpace[i + m[0], j + m[1]] = 1
106
107     def update(self, i ,j):
108         self.updateRoad(int(i+1), int(j+1))
109         self.updateDistance(i, j)
110         self.updateFilledSpace(i, j)
111
112     def gridSum(self, i, j):
113         #make sure to give this i + 1, j+1
114         total = 0
115         for m in [i-1, i , i+1]:
116             for n in [j-1, j , j+1]:
117                 if self.road[m, n] > 0:
118                     total = total + 1
119         return total
120
121     def roadSum(self, i, j):
122         #make sure to give this i + 1, j+1
123         total = 0
124         for m in [i-1, i , i+1]:
125             for n in [j-1, j , j+1]:
126                 if (m >= 0) and (n >= 0) and (m < self.w+2) and (n < self.h+2):
127                     total = total + self.road[m, n]
128         return total
129
130     def getAmountEmpty(self):
131         t = 0
132         for i in range(0, self.w):
133             for j in range(0, self.h):
134                 if self.filledSpace[i, j] == 0:
135                     if not self.road[i+1, j+1] > 0:
136                         t = t + 1
137         return t
138
139     def roadCount(self):
140         t = 0
141         for i in range(1, self.w+1):
142             for j in range(1, self.h+1):
143                 if self.road[i, j] > 0:
144                     t = t + 1
145         return t
146
147     def getDistance(self, i, j):
```

```
148         if self.distance[i, j] < 1:
149             minDistance = 10000
150             for m in neighbourhood:
151                 if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m
    [1] < self.h):
152                     if self.distance[i + m[0], j + m[1]] < minDistance and self.distance[
    i + m[0], j + m[1]] >= 1:
153                         minDistance = self.distance[i + m[0], j + m[1]]
154             return minDistance + 1
155         else:
156             return self.distance[i, j]
157 
158     def getFilledSpaces(self, i, j):
159         total = 0
160         for m in neighbourhood:
161             if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m[1]
    < self.h):
162                 total = total + self.filledSpace[i + m[0] , j + m[1]]
163         return total
164 
165 
166     def cost(self, i, j, k):
167         return (self.gridSum(i+1, j+1) * k[0] +
168                 self.roadSum(i+1, j+1) * k[1] +
169                 self.getDistance(i, j) * k[3] +
170                 self.getFilledSpaces(i,j) * k[2])
171 
172     def minimumCost(self, k, excluded=0):
173         minimum = math.inf
174         index = []
175         for i in range(0, self.w):
176             for j in range(0, self.h):
177                 if self.road[i+1, j+1] > 0:
178                     if (i >= 0) and (j >= 0) and (i < self.w) and (j < self.h):
179                         count = 0
180                         for m in neighbourhood:
181                             if self.road[int(i + 1 + m[0]), int(j + 1 + m[1])] > 0:
182                                 count = count + 1
183                         if count == 4:
184                             continue
185                         else:
186                             for m in neighbourhood:
187                                 if not self.road[int(i + 1 + m[0]), int(j + 1 + m[1])] >
    0:
188                                     val = self.cost(i + m[0], j + m[1], k)
189                                     if val < minimum:
190                                         minimum = val
191                                         index = [int(i + m[0]), int(j + m[1])]
192 
193         return index
194 
195     def iterate(self, k):
196         while True:
197             if self.getAmountEmpty() == 0:
198                 break
199 
200             chosen = self.minimumCost(k)
201             if chosen:
202                 nextRoad = chosen
203             else:
204                 print('exit - help')
205                 self.show()
206                 break
207             self.update(nextRoad[0], nextRoad[1])
208 
209         ret = self.roadCount()
210         return(ret)
211 
212 
213 
214 
215 def bruteForceMaximisation(beginning, end, sample_n, car_park_size, entrance):
216     minimum_road = math.inf
```

```
217     results = [[]]
218     for q in np.linspace(beginning,end,sample_n):
219         print("progress: ", (q/end)*100, "%\n")
220         for w in np.linspace(beginning,end,sample_n):
221              for e in np.linspace(beginning,end,sample_n):
222                 for r in np.linspace(beginning,end,sample_n):
223                     CarPark = Grid(car_park_size[0], car_park_size[1], entrance[0],
    entrance[1])
224                     k = [q,w,e,r]
225                     valu = CarPark.iterate(k)
226                     if valu == minimum_road:
227                         results.append(k)
228                     if valu < minimum_road:
229                         results =[[]]
230                         results.append(k)
231                         minimum_road = valu
232     return results
```

5.2 Python code for the Stochastic approach

```python
import numpy as np
import matplotlib.pyplot as plt
import copy
import sys
GraphFigure=None
axs=None
from scipy.optimize import fmin
import math

neighbourhood = [[0,1], [0,-1], [1,0], [-1,0]]

bigNeighbourhood = [[0,1], [0,-1], [1,0], [-1,0]]

#bigNeighbourhood = [[0,1], [0,-1], [1,0], [-1,0], [1,1], [-1,-1], [1,-1], [-1,1],
#                     [0,2], [0,-2], [2,0], [-2,0]]

#bigNeighbourhood = [[0,1], [0,-1], [1,0], [-1,0], [1,1], [-1,-1], [1,-1], [-1,1],
#                     [0,2], [0,-2], [2,0], [-2,0], [2,2], [-2,-2], [2,-2], [-2,2],
#                     [1,2], [-1,-2], [1,-2], [-1,2], [2,1], [-2,-1], [2,-1], [-2,1],
#                     [0,2], [0,-2], [2,0], [-2,0]]

def CreateRoadLayer(width, height, startx, starty, obstacleConstant=1):
    grid = np.zeros((width+2, height+2))
    grid[:,0] = obstacleConstant #-1
    grid[:,height+1] = obstacleConstant #-1
    grid[0,:] = obstacleConstant #-1
    grid[width+1,:] = obstacleConstant #-1
    grid[startx+1,starty+1] = 1
    return grid
#These two layers don't need to consider the boundary so they are smaller. this needs to
    be
#reflected in the indices used to traverse elements
def CreateDistanceLayer(width, height, startx, starty):
    grid = np.zeros((width, height))
    grid[startx,starty] = 1
    return grid

def CreateFilledSpaceLayer(width, height, startx, starty):
    grid = np.zeros((width, height))
    for m in neighbourhood:
        #Will probably need to add protection for bounding on other side too
        if (startx + m[0] >= 0) and (starty + m[1] >= 0):
            grid[startx + m[0], starty + m[1]] = 1
    return grid

class Grid:
    def __init__(self, width, height, startx, starty):
        self.w = width
        self.h = height
        self.start = [startx, starty]
        self.road = CreateRoadLayer(width, height, startx, starty,1)
        self.distance = CreateDistanceLayer(width, height, startx, starty)
        self.filledSpace = CreateFilledSpaceLayer(width, height, startx, starty)

    def show(self):
        fig = plt.figure(figsize=(10,10))
        ax = fig.add_subplot(111)
        ax.set_xlim([0,(self.h+1)*20])
        ax.set_ylim([0,(self.w+1)*20])
        for x in range(0, self.w+2):
            ax.plot(0, x*20, 'bo')
            ax.plot((self.h+1)*20, x*20, 'bo')
        for y in range(0, self.h+2):
            ax.plot(y*20, 0, 'bo')
            ax.plot(y*20, (self.w+1)*20, 'bo')
        for i in range(1, self.w+1):
            for j in range(1, self.h+1):
                if self.road[i,j] > 1:
                    if i == self.start[0] + 1 and j == self.start[1] + 1:
                        ax.plot(j*20, i*20, 'rs')
                    else:
```

```
71                          ax.plot(j*20, i*20, 'ro')
72                      for m in neighbourhood:
73                          if (i + m[0] >= 1) and (j + m[1] >= 1) and (i + m[0] <= self.w)
    and (j + m[1] <= self.h):
74                              if self.road[i + m[0], j + m[1]] > 0:
75                                  ax.plot([(j + m[1])*20, j*20],
76                                          [(i + m[0])*20, i*20], 'k-')
77                  else:
78                      ax.plot(j*20, i*20, 'go')
79          ax.set_aspect(1)
80          plt.savefig('result' + str(self.w) + str(self.h) + str(self.start[0]) + str(self.
    start[1]) + ".png")
81          plt.show()
82
83      def updateRoad(self, i, j):
84          #make sure to pass i , j as 1 extra for border
85          adjacentCount = 0
86          for m in neighbourhood:
87              if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w+2) and (j + m
    [1] < self.h+2):
88                  if self.road[i + m[0], j + m[1]] > 0:
89                      adjacentCount = adjacentCount + 1
90                      self.road[i + m[0], j + m[1]] = self.road[i + m[0], j + m[1]] + 1
91
92          self.road[i,j] = adjacentCount + 1
93
94      def updateDistance(self, i, j):
95          adjacentCount = 0
96          minDistance = 999999999
97          for m in neighbourhood:
98              if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m[1]
    < self.h):
99                  if self.road[i + 1 + m[0], j + 1 + m[1]] > 0:
100                     if self.distance[i + m[0], j + m[1]] < minDistance:
101                         minDistance = self.distance[i + m[0], j + m[1]]
102         self.distance[i, j] = minDistance + 1
103
104         for m in neighbourhood:
105             if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m[1]
    < self.h):
106                 if self.road[i + 1 + m[0], j + 1 + m[1]]:
107                     if not self.distance[i + m[0], j + m[1]] == minDistance:
108                         self.updateDistance(int(i + m[0]) , int(j + m[1]) )
109
110     def updateFilledSpace(self, i, j):
111         self.filledSpace[i, j] = 1
112         for m in neighbourhood:
113             if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m[1]
    < self.h):
114                 #the idea with this being that we want to view the road as also a filled
    space
115                 self.filledSpace[i + m[0], j + m[1]] = 1
116
117     def deleteVertex(self, i, j):
118         self.distance[i, j] = 0
119         self.road[i+1, j+1] = 0
120         self.filledSpace[i, j] = 1
121
122
123     def update(self, i ,j):
124         self.updateRoad(int(i+1), int(j+1))
125         self.updateDistance(i, j)
126         self.updateFilledSpace(i, j)
127
128
129     def getAmountEmpty(self):
130         t = 0
131         for i in range(0, self.w):
132             for j in range(0, self.h):
133                 if self.filledSpace[i, j] == 0:
134                     if not self.road[i+1, j+1] > 0:
135                         t = t + 1
136         return t
```

```
137
138     def roadCount(self):
139         t = 0
140         for i in range(1, self.w+1):
141             for j in range(1, self.h+1):
142                 if self.road[i, j] > 0:
143                     t = t + 1
144         return t
145
146     def neighbourRoadCount(self, i, j):
147         roadCount = 0
148         for m in neighbourhood:
149             if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m[1]
    < self.h):
150                 if self.road[int(i + m[0] + 1), int(j + m[1] + 1)]:
151                     roadCount = roadCount + 1
152         return roadCount
153
154     def getDistance(self, i, j):
155         if self.distance[i, j] < 1:
156             minDistance = math.inf
157             for m in neighbourhood:
158                 if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m
    [1] < self.h):
159                     if self.distance[i + m[0], j + m[1]] < minDistance and self.distance[
    i + m[0], j + m[1]] >= 1:
160                         minDistance = self.distance[i + m[0], j + m[1]]
161             return minDistance + 1
162         else:
163             return self.distance[i, j]
164
165     def getFilledSpaces(self, i, j):
166         total = 0
167         for m in bigNeighbourhood:#
    ##############################################################################
168             if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m[1]
    < self.h):
169                 total = total + self.filledSpace[i + m[0] , j + m[1]]
170         return total
171
172     def neighbourEmptyCount(self, i, j):
173         total = 0
174         count = 0
175         for m in neighbourhood:#
    #############################################################################
176             if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w) and (j + m[1]
    < self.h):
177                 count = count + 1
178                 total = total + self.filledSpace[i + m[0] , j + m[1]]
179         return count - total
180
181
182     def generateCandidates(self, relaxed=False, veryRelaxed=False):
183         minimum = math.inf
184         candidates = [[]]
185         for i in range(0, self.w):
186             for j in range(0, self.h):
187
188                 #############################
189                 # Initial vertex selection #
190                 #############################
191                 if not self.road[i+1, j+1] > 0: #Has to be a road
192                     continue
193
194                 if not (i >= 0) and (j >= 0) and (i < self.w) and (j < self.h): # Has to
    be within boundary
195                     continue
196
197                 count = 0
198                 for m in neighbourhood:
199                     if self.road[int(i + 1 + m[0]), int(j + 1 + m[1])] > 0:
200                         count = count + 1
201                 if count == 4: # Has to not be surrounded
```

```
202                     continue
203
204                 ###########################################################
205                 # Now individual branching oppurtunity consideration #
206                 ###########################################################
207                 for m in neighbourhood:
208                     flag = True
209
210                     if self.road[int(i + 1 + m[0]), int(j + 1 + m[1])] > 0: # Has to not
    already be a road
211                         continue
212
213                     if self.neighbourEmptyCount(int(i + m[0]), int(j + m[1])) < 1:
214                         continue
215
216                     for n in neighbourhood:
217                         if self.road[int(i + 1 + m[0] + n[0]), int(j + 1 + m[1] + n[1])]
    > 0: # Don't check roads
218                             if [int(i + m[0] + n[0]), int(j + m[1] + n[1])] == [int(i),
    int(j)]:
219                                 continue
220                             else:
221                                 if not veryRelaxed:
222                                     if relaxed:
223                                         if self.road[int(i + m[0] + n[0]+1), int(j + m[1]
    + n[1]+1)] == 1:
224                                             continue
225                                         else:
226                                             flag = False
227                                     else:
228                                         flag = False
229
230
231                         if self.neighbourRoadCount(i + m[0] + n[0], j + m[1] + n[1]) > 1
    and not relaxed: # Check if the road is already accessible in 2 ways
232                             flag = False
233
234                     if flag:
235                         candidates.append([int(i + m[0]), int(j + m[1])])
236
237         return candidates
238
239     def findClosestCandidates(self, vertices, threshDist=1):
240         minimum = math.inf
241         minimumSet = [[]]
242         randomSet = [[]]
243         for v in vertices:
244             if v == []:
245                 continue
246             dist = self.getDistance(v[0], v[1])
247
248             if dist == minimum:
249                 if minimumSet == [[]]:
250                     minimumSet[0] = v
251                 else:
252                     minimumSet.append(v)
253             elif dist < minimum:
254                 minimumSet =[v]
255                 minimum = dist
256         for v in vertices:
257             if not v in minimumSet:
258                 if np.random.uniform(0,1) > threshDist:
259                     if randomSet == [[]]:
260                         randomSet[0] = v
261                     else:
262                         randomSet.append(v)
263
264         if minimumSet == [[]]:
265             return [[]]
266         elif randomSet == [[]]:
267             return minimumSet
268
269         minimumSet = minimumSet + randomSet
```

```
270
271             return minimumSet
272
273     def findLeastRoadCandidates(self, vertices, threshRoad=1):
274         minimum = math.inf
275         minimumSet = [[]]
276         randomSet = [[]]
277
278         for v in vertices:
279             if v == []:
280                 continue
281             road = self.getFilledSpaces(v[0], v[1])
282
283             if road == minimum:
284                 if minimumSet == [[]]:
285                     minimumSet[0] = v
286                 else:
287                     minimumSet.append(v)
288             elif road < minimum:
289                 minimumSet = [v]
290                 minimum = road
291
292         for v in vertices:
293             if not v in minimumSet:
294                 if np.random.uniform(0,1) > threshRoad:
295                     if randomSet == [[]]:
296                         randomSet[0] = v
297                     else:
298                         randomSet.append(v)
299
300
301         if minimumSet == [[]]:
302             return [[]]
303         elif randomSet == [[]]:
304             return minimumSet
305
306         minimumSet = minimumSet + randomSet
307         return minimumSet
308
309     def splitCandidates(self, vertices):
310         horizontal =[[]]
311         vertical =[[]]
312
313         if vertices ==[[]]:
314             return [[],[]]
315
316         for v in vertices:
317             if v ==[]:
318                 continue
319             for m in neighbourhood:
320                 if (v[0] + m[0] >= 0) and (v[1] + m[1] >= 0) and (v[0] + m[0] < self.w)
    and (v[1] + m[1] < self.h):
321                     if self.road[int(v[0] + m[0] + 1), int(v[1] + m[1] + 1)] > 0:
322                         if v not in horizontal and v not in vertical:
323                             if ( not m[0] == 0 ) and m[1] == 0:
324                                 if vertical == [[]]:
325                                     vertical[0] = v
326                                 else:
327                                     vertical.append(v)
328
329                             if ( not m[1] == 0 ) and m[0] == 0:
330                                 if horizontal == [[]]:
331                                     horizontal[0] = v
332                                 else:
333                                     horizontal.append(v)
334         return[horizontal, vertical]
335
336     def clean(self):
337         vertices =[[]]
338         deleted = False
339         for i in range(0, self.w):
340             for j in range(0, self.h):
341                 if self.road[int(i+1),int(j+1)]:
```

```
342                     flag = True
343                     for m in neighbourhood:
344                         if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w)
    and (j + m[1] < self.h):
345                             if not self.distance[i,j] >= self.distance[int(i+m[0]), int(j
    +m[1])]:
346                                 flag = False
347                                 break
348                     if flag:
349                         count = 0
350                         count2 = 0
351                         for m in neighbourhood:
352                             if (i + m[0] >= 0) and (j + m[1] >= 0) and (i + m[0] < self.w
    ) and (j + m[1] < self.h):
353                                 flag2 = False
354                                 if self.road[int(i + m[0] + 1), int(j + m[1] + 1)]:
355                                     continue
356                                 count = count+1
357                                 for n in neighbourhood:
358                                     if (i + m[0] + n[0] >= 0) and (j + m[1]+ n[1] >= 0)
    and (i + m[0] + n[0] < self.w) and (j + m[1] + n[1] < self.h):
359                                         if [int(i + m[0] + n[0]), int(j + m[1] + n[1])]
    == [i, j]:
360                                             continue
361                                         else:
362                                             if self.road[int(i + m[0] + n[0] + 1), int(j
    + m[1] + n[1] + 1)] > 0:
363                                                 flag2 = True
364                                 if flag2:
365                                     count2 = count2+1
366
367                         if count2 == count:
368                             self.deleteVertex(i, j)
369                             deleted = True
370                     else:
371                         continue
372         return deleted
373
374
375
376
377     def getNextVertex(self, relaxed=False, probV=1/2, threshRoad=1, threshDist=1,
    veryRelaxed=False):
378         candidateVertices = self.generateCandidates(relaxed, veryRelaxed)
379
380         candidateVertices = self.splitCandidates(candidateVertices)
381
382         if candidateVertices == [[],[]]:
383             return False
384         if candidateVertices[0] == [[]]:
385             candidateVertices = candidateVertices[1]
386         elif candidateVertices[1] == [[]]:
387             candidateVertices = candidateVertices[0]
388         else:
389             if np.random.uniform(0,1) > probV: #Horziontal or vertical choice
390                 candidateVertices = candidateVertices[0]
391             else:
392                 candidateVertices = candidateVertices[1]
393
394         candidateVertices = self.findLeastRoadCandidates(candidateVertices, threshRoad)
395
396         candidateVertices = self.findClosestCandidates(candidateVertices, threshDist)
397
398         chosen = np.arange(0, len(candidateVertices))
399
400         chosen = candidateVertices[int(np.random.choice(chosen))]
401
402         return chosen
403
404
405     def iterate(self, probV=1/2, threshRoad=1, threshDist=1):
406         while True:
407             if self.getAmountEmpty() == 0:
```

```
                break

            chosen = self.getNextVertex(False, probV, threshRoad, threshDist)

            if chosen:
                nextRoad = chosen
            else:
                chosen = self.getNextVertex(True, probV, threshRoad, threshDist)

                if chosen:
                    nextRoad = chosen
                else:
                    chosen = self.getNextVertex(True, probV, threshRoad, threshDist, True)

                    if chosen:
                        nextRoad = chosen
                    else:
                        print('Error - could not iterate as no potential vertices')
                        self.show()
                        break

            self.update(nextRoad[0], nextRoad[1])

        ret = self.roadCount()

        return(ret)


def simulate(k, p1, p2, p3, car_park_size, entrance, minimum=math.inf):
    min_flag = False
    for n in range(0,k):
        CarPark = Grid(car_park_size[0], car_park_size[1], entrance[0], entrance[1])
        # 1-0.4 = 60% chance of going through
        CarPark.iterate(p1, p2, p3)
        clean_flag = True
        while clean_flag:
            clean_flag = CarPark.clean()

        valu = CarPark.roadCount()
        if valu <= minimum:
            minimum = valu
            min_flag = True
            MinCarPark = copy.deepcopy(CarPark)
    if min_flag:
        print( minimum, "Road network vertices\n with parameters", p1,p2,p3 )
        MinCarPark.show()
    return minimum

def bruteForceApproach(simulation_n, car_park_size, entrance, sample_n):
    minimum = math.inf
    for i in np.linspace(0, 1, sample_n):
        for j in np.linspace(0, 1, sample_n):
            for k in np.linspace(0, 1, sample_n):
                minimum = simulate(simulation_n, i, j, k, car_park_size, entrance, minimum)
```

References

[1] Ian Wise, Roland Trim, *Optimisation of car park design*, ARUP (Bristol, UK), 2013.

[2] *http://publications.lib.chalmers.se/records/fulltext/238498/238498.pdf*

[3] *https://en.wikipedia.org/wiki/Tree_(graph_theory)*

[4] *https://en.wikipedia.org/wiki/Planar_graph*

[5] *https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree*

[6] Nelder, J.A. and Mead, R. (1965), *A simplex method for function minimization*, The Computer Journal, 7, pp. 308-313

[7] *https://en.wikipedia.org/wiki/Integer_programming*

[8] *https://www.python.org/downloads/release/python-363/*

[9] *https://www.numpy.org/*

[10] *https://www.scipy.org/*

[11] *https://en.wikipedia.org/wiki/Reinforcement_learning*

[12] *http://www.bristol.ac.uk/engineering/media/engineering-mathematics/esgi91/esgi-arup-parking.pdf*