Parking Manager

Assignment of Parking Spots

Frederik Bugge Jørgensen, Frederik Peter Bräuner, Jeppe Duedahl Jensen, Lyra Winther Tybo, Mikkel Buus Rasmussen & Noah Horslev Petersen

Software, CS-25-SW-P1-13, 2025

Semester Project

This report is typeset using the Typst system.

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
https://www.cs.aau.dk

Title:
Parking Manager

Theme:
Assignment of Parking Spots

Project Period:
Fall Semester 2025

Project Group:
CS-25-SW-P1-13

Participants:
Frederik Bugge Jørgensen
Frederik Peter Bräuner
Jeppe Duedahl Jensen
Lyra Winther Tybo
Mikkel Buus Rasmussen
Noah Horslev Petersen

Supervisor:
Abiram Mohanaraj

Copies: 1

Number of Pages: 72

Date of Completion:
09/12-2025

Abstract:

The search for parking takes up a somewhat insignificant amount of time each day for the individual motorist, but take that time and multiply it for every day of the week, and multiply that with every car owner of a country, continent or even the whole world and the time starts adding up. Time that could have been spent better, and time that instead contributes to creating traffic congestion and increased pollution. This project aims to produce a software solution that handles the allocation of parking spaces in parking structures. This reduces the time it would take for motorists to look for parking, thus alleviating traffic congestion and pollution.

Chapter 1

Preface

The P1-project ”Parking lot manager” is developed by Jeppe Duedahl Jensen, Frederik Peter Bräuner, Noah Horslev Petersen, Mikkel Buus Rasmussen, Lyra Winther Tybo, 1. semester, Aalborg University. The project is developed from the period of the 9. of october 2025 to the 5. of december 2025.

The project consists of the following:
- Main report
- Product in GitHub repository

The goal of this project was to develop a parking lot manager that minimized the time it takes drivers to find parking spots in parking structures. This is done by allocating a designated parking spot to each driver before entering.

It is recommended to read the entire report in its context to fully understand the intention of the project.

The report and product was developed by the following:

Frederik Peter Bräuner
fpbr@student.aau.dk

Jeppe Duedahl Jensen
jdje@student.aau.dk

Noah Horselv Petersen
?@student.aau.dk

Mikkel Buus
??@student.aau.dk

Lyra Winther Tybo
???@student.aau.dk

Frederik Bugge Jørgensen
????@student.aau.dk

Chapter 1

Contents

Introduction

Searching for parking spaces in crowded urban areas is often slow, frustrating, and inefficient, especially during the peak periods of traffic, like morning rush hour or traffic in the evening. Drivers spend several minutes circling through multi-level parking structures only to discover that the few remaining spaces are reserved or poorly distributed. This wasted time not only delays drivers but also contributes to congestion inside and outside parking structures.

This report addresses the core problem behind this experience and aims to solve it by developing a Parking Structure Manager that models real parking structures and uses algorithms to assign drivers the most optimal parking space based on distance, availability, and vehicle type.

Furthermore, the development of a Parking Space Allocation system will also be detailed throughout this report

The following sections will describe the context of the problem, theoretical foundation, program design and implementation. Through this, the report aims to demonstrate how much faster drivers will be able to find parking spaces in parking structures, by allocating a designated parking space to them when entering.

Chapter 1

Problem Analysis

1.1 Problem with finding parking spots in parking structures

Finding an available parking spot in a crowded parking structure, during peak periods is a problem many drivers have experienced. As traffic builds up, drivers often have to circulate through multiple floors and check most of the spaces to be able find a single available spot. Having to find a parking spot through trail an error is not a very effective way that wastes time, gas, increases congestion and the driver's stress. There is no worse feeling than finally having found an available parking spot, only for it to be a designated handicap spot or a space reserved for electric vehicles. Drivers can spend a lot of time creating unnecessary congestion, while trying to find a parking spot and ultimately having to give up and leave the parking structure.

This problem highlights a bigger issue in p-structures. Drivers lack clear, real-time guidance that directs them to an available parking space.

1.1.1 Economical consequences

Currently, a large amount of time is spent looking for a parking space. In the car-centric USA, motorists spend 17 hours on average every year looking for parking [1]. This hunt for parking leads to an annual loss of 70 billion dollars, and an estimated waste of 1.7 billion gallons of fuel[2]. A lot of this wasted time comes down to factors like congestion from motorists looking for parking, and motorists parking in parking spaces that aren't designated to them[2]. While places like underground parking garages and car parks are made to mitigate this problem, they also bring their own challenges if proper precautions are not taken to avoid confusing or annoying the motorist looking for parking. These precautions can be things like making sure that there is proper and readable signage and incorporating technologies to facilitate easier parking [3]. As such, there can exist places where large parking garages are nowhere near capacity, while motorists are still cruising around the city looking for more "accessible" parking in order to avoid the hassle of having to use large car parks and underground parking garages.

1.1.2 Emissions.

However, the cost of looking for parking is not only monetary, as traffic congestion also causes unnecessary emissions. In Shenzhen, China, different types of emissions have been observed to increase in the range of 14 to 30 percent during rush hour periods with large amounts of traffic congestion[4]. While looking for parking does not account for the increase in its entirety, it is certainly one of the main factors for the congestion. These factors are only predicted to increase in severity, as the number of cars on the road are steadily increasing, with annual global car sales reaching 74.6 million units in 2024[5]. The growing trend of urbanisation also leads to more people being concentrated in urban city centres for a large part of the day, leading to the same type of emission increases as observed in Shenzhen happening more frequently.

1.2 Parking Structures and Basements

A parking structure can be described as an area or building specifically designed to accommodate parked vehicles. These structures come in several forms, like open surface parking lots, multi-level parking garages above ground, and underground parking basements. Several key-factors influence how effective a p-structure is, things like the number of floors, the internal driving aisles, and the placements of the entrance and exit all have an effect on reducing queues and making sure there is efficient traffic flow.

Several layout principles, like logical navigation, smooth traffic flow, and optimal space efficiency, influence how P-structures are designed. Effective design of the p-aisles reduces congestion, shortens the time spent searching for a p-spot, and improves the overall user experience. The core objective of any p-facility is to make the process of entering, parking, and leaving the area as simple as possible. However, surface p-lots have a harder time achieving these goals. While they are easy to construct, they fill a larger space and offer very limited opportunities for vertical expansion, which hinders their overall capacity and efficiency. Their open layout also makes it more difficult to manage traffic flow and guide drivers toward free spaces.

1.2.1 The Importance of Good Parking Layout

An optimized p-structure tries to minimize the time and distance it takes for drivers to enter and exit the p-structure, while also making sure that vehicles' congestion doesn't occur while moving through the facility. A p-structure layout should be designed so cars easily can access the p-spots without excessive back-and-forth movement. For example, the entry and exit lanes are typically placed at opposite ends of the structure to minimize traffic build up at ither point [6]. Another strategy some p-structures implement is to have multiple entrances and exits. This is usually done in high-capacity p-structures. One way to do this is to have specific lanes for entry and exit, which helps congestion at peak periods. Some parking facilities also have signs that try to guide drivers to the nearest available p-spot [7].

The interior driving aisles, meaning the area where vehicles drive between parked cars, also plays a role in the effectiveness of the traffic flow. The P-aisles are most commonly designed in a one-way system to reduce confusion about which way drivers can go but can also be two-way in smaller p-structures [6]. The width of p-spots is also important to have in mind, because too narrow spaces won't allow driver to get out of their cars, while too big spaces can reduce the amount of total p-spots. The standard p-spot on Denmark has a width of around 2.5 metres [8], but differs between structures as there are only guidelines, not laws, about a specific size. To get a truly efficient and well optimized p-structure, the aisles and spots have to be thoroughly thought out, to allow vehicles to easily turn and park without problems.

1.2.2 Automated Parking Systems

Some modern underground p-structures in high-density areas have been implemented as a fully automatic parking structure. Automatic p-structures work by implementing smart technology, like elevators to effectively remove human drivers from most of the p-process, resulting in a drastically improved traffic flow and essentially nullifying the congestion caused by drivers trying to find available p-spots. They also maximize the space-usage, as vehicles can be parked closer together than if human drivers had to park for themselves. One of the more notable examples of a fully automatic p-structure is at DOKK1 in Aarhus in Denmark, which can fit almost 1000 cars and it the biggest automatic p-structure in Europe [9]. Even though automatic p-structures are incredibly smart and seem to solve almost all the issues in p-structures, they come at a big increase in price compared to regular p-structures. DOKK1 was built over multiple years, and the total budget was around 500 million DKK, which really emphasizes the prize of a “perfect” p-structure.

Another smart technology that is used more and more commonly in p-structures is Automatic Number Plate Recognition (ANPR). ANPR-systems consist of cameras and character recognition software to automatically scan vehicles’ number plates. When a car enters or exits the p-structure, a camera takes a picture of the car’s license plate and then the software processes and extracts the license plate number, where it is then being matched against a database to see if the car is entering or exiting. This automatic operation simplifies parking by allowing drivers to park without having to spend time registering at a parking meter, speeding up the user experience and reducing queues. It also helps to enforce the rules and can be integrated with payment systems, allowing drivers to easily pay for their parking.

1.2.3 Parking Space Allocation and Types

In addition to the standard p-spots, modern p-structures have specialized p-spots for different cars or drivers, particularly electric vehicles (EV) and handicap drivers. Having designated p-spots like these makes it so the layout of the p-structure has to accommodate the needs of multiple types of drives. Handicap spaces should be places as close as possible to the building entrances, elevators or other primary access points to reduce the travel time and distance for people with limited mobility. There are also laws stating a specific number of parking spots must be handicap-spots, so this must be taken into great consideration when designing the layout for p-structures. Handicap spaces are most commonly marked with a wheel-chair, so non-handicapped drivers don’t take up spaces that are reserved for handicapped drivers.

EV-spaces should ideally be located close to a power supply to reduce the installation costs for the charging equipment. Because of this, EV-spots of placed on the ground floor or near entrances, so drivers easily can enter, charge and exit without creating congestion. EV-spots are typically marked with a plug symbol and are reserved for electric vehicles or plug-in hybrid vehicles only [10]. Both the handicapped- and EV-vehicle spots should be easily visible and placed in areas that improve the overall user experience for the drivers of these designated spots and regular drivers.

1.2.4 Pros and Cons of Conventional and Automated P-Structures During Peak Periods

Parking demand is exceptionally demanded during peak periods, where congestion is likely to happen, due to drivers panicking, while trying to find an available space. During these times the efficiency of the p-structure primarily depends on its operational design. The conventional p-structures must have great layout and available space to ensure that only a minimal amount of congestion occurs as drivers circulate the p-structure floors in search of any available p-space.

Automatic p-structures, however, remove the need for drivers to manually park their cars, so the panic-related-circulation congestion is eliminated, while also allowing a significant increase in p-space efficiency, allowing more cars to be stored inside [11]. The automated p-systems handle parking through the use of elevators and conveyors that take the cars underground. This also increases safety, while reducing the noise and the emissions.

1.2.5 Most Important Key-point For Optimizing Conventional Parking Structures

Making an efficient p-structure requires optimizing both the functionality of the p-structure and balancing the user experience. The most important thing is making sure that there is a smooth and logical traffic flow, that minimizes congestion, especially during peak periods. This requires having well-designed internal driving aisles with clear entry and exit points, so drivers easily can find a spot, park and exit.

The p-spaces themselves are another consideration. Balancing p-space utilization and capacity between multiple floors, while having easy to park spots that include designated EV- and handicap-spots is a big deal-breaker for the overall layout and user experience of the p-structure. Integrating intelligent parking technologies, like guidance lights or ANPR can further increase the functionality of the p-structure. However, regardless of whether a facility has smart parking technologies or not, a successful p-structure layout ultimately depends on how easy and fast a driver can park their car without causing issues for other drivers.

1.3 How long does parking take in a p-structure?

A study was conducted on two closely situated car parks in the center of Leeds to examine how search times for an empty parking space varied throughout the day. [12] Figure A.1 shows the data collected in the study.
The surface carpark observed had a capacity of 275 cars, and the multi-storey carpark observed had a capacity of 900 cars. The search time of each carpark was observed on a typical working day, starting at 7am.
Both car parks had similar accumulation profiles, and occupancy steadily increases from 7am to near capacity around mid day. The study revealed that the time required to park a vehicle in the surface car park increased to 214 seconds throughout the day, whereas the corresponding time in the multi-story car park was 176 seconds.[12]

Based on the data presented in the study, parking in the surface car park was found to be faster than in the multi-storey car park when occupancy levels were below capacity. Observations conducted in the early morning indicated that it took approximately 49 seconds less to park in the surface car park compared to the multi-storey car park. However, as both car parks approached full occupancy, this trend reversed. Parking in the multi-storey car park became approximately 38 seconds faster, requiring 176 seconds compared to 214 seconds in the surface car park.[12]

Looking across the UK for average parking search times, it can be seen that they have the highest annual search time, compared to Germany and the US, where the search time is annually 44 hours for British drivers. In Germany it is still high with an annual search time of 41 hours and in the US an annual search time of 17 hours per driver.[13]

1.3.1 In peak periods

A case study from Xi'an, China, shows that unbalanced land and limited p-facilities create long congestions throughout the different city districts in the time frames of morning period spanning 7.00-9.00 and the evening period 17.00-19.00.[14] Where there is a search for a limited number of p-spaces, in peak periods such as weekdays and in the evening, it will create long congestions, these long congestions directly affect the parking time, especially with the search for a p-space. Mixed with expensive parking and illegal parking in these areas, the problem grows, and makes the search for a p-space take longer and it becomes more annoying for drivers.[14]

This is also true for p-garages both surfaces and underground ones, when they are limited in number and are more expensive, the use and search time for p-spaces goes up.[14] A direct correlation between search time and how full a p-garage is can be seen. An article from the university of Leeds show that, in an almost empty p-space, search time is at an all time low by taking 19 seconds in total, later in the day when the p-garage is almost full search time has an almost exponential increase by now taking 175 seconds to find a p-space and park your car [12]. This is an increase of 156 seconds, so over 2.5 minutes from finding a p-space in an empty one versus a almost full one.[12] See Figure A.3 for how search time increases over full the p-garage is. So it can be observed that an almost full p-garage and a limited number of p-garages increases the drivers' search time for a p-space, especially under the periods mentioned earlier. This shows that even with an amount of 20 drivers wanting to park in a half full multi level p-garage, with the search time on average being 80 seconds, the 20 drivers are in total using a little over 26.5 minutes of searching time before all are parked.[12]

As mentioned in point 1.1.2, all this search time for parking and the congestion created from the search times, also increases the amount of emissions the drivers produce. [14] The economical consequences is costing not just the drivers, but also the cities.[13] where the average parking search time cost across the US for drivers is 325 USD a year, this includes wasted time, fuel, and emissions.[13] For the US there is an annual cost of 72.7 billion USD, in the UK it is 23.3 billion GBP, and in Germany it is 40.4 billion Euro yearly. [13] There is a large amount of money being wasted, mixed with the increase in emissions and the high search time for cars mixed together creates a big problem for cities, local drivers and tourists visiting the cities. When all this is looked at together, a current and extensive problem can be seen, where the high search time for a limited number of spaces and inefficient p-garages[12] have huge costs for both drivers and the city.

1.4 Stakeholders

To understand the problem in greater detail, a greater understanding of who it affects, and how it affects them is needed. This segment will analyze the needs of the individuals, municipalities and companies. For starters, a diagram over possible stakeholders was made, see Figure 1.1.

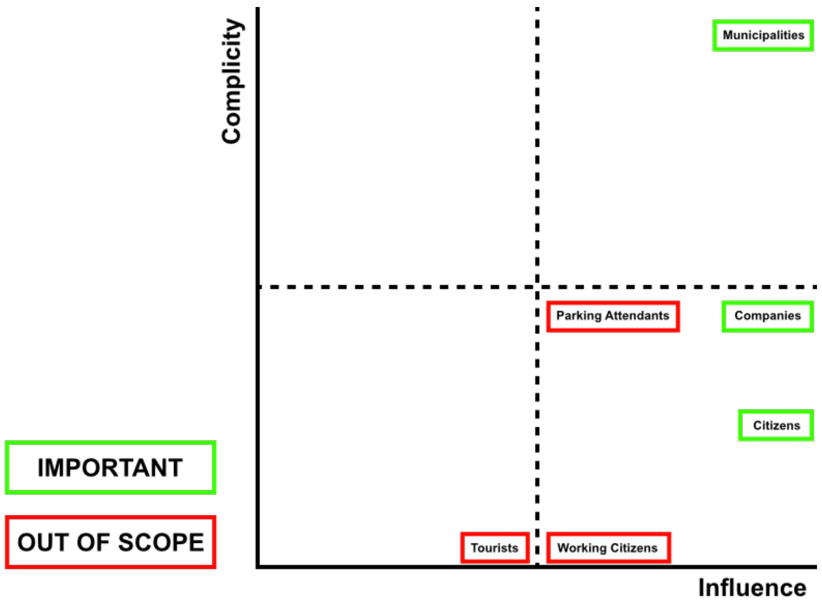


Figure 1.1: Stakeholders Diagram

1.4.1 Citizens

The individuals are the users of the parking spots; therefore, they will be using the front-end of this report's system. It is therefore crucial to understand the primary demographic within these individuals, and at least fulfill their digital needs in the front-end.
Studies conducted by *The Institute of Transport* at *DTU* has tried to segment people on the road. [15]
In the age-segmentation the most prominent age-group are aged 30 to 59 years of age. These individuals, on average, drive 39.7 kilometers per person a day, 34.9 of which are in the driver's seat. However, around 10-15 pct. in this age-group has answered to being digitally challenged in a study conducted by *Danmarks Statistik*. [16]
This sets some limitations in regards to the design of the solution to include next to everybody. A simpler design philosophy will also help us reach other age-groups, hopefully leading to wider adaptation.
The study by *DTU* also had some findings in regards to gender and the household. In general, men drive more than women, and a household consists of a couple with children drive the most. [15]

1.4.2 Municipalities

The municipalities own a large amount of the parking lots in Denmark; therefore, they are one of the target groups that could adopt new parking solutions. For municipalities their requirements in a project to be interesting to them should be examined.
The municipalities have an interest for wide adaptation, which makes one of their needs digital inclusion. [17]

Digital inclusion in itself is the term which describes that all citizens should have the right to participate digitally. To fulfill this requirement, need multiple factors should be kept in mind, such as inclusive design, easy to understand language, in-app guides and alternatives for non-digital citizens.
Also for wide adaptation, the system needs to be able to fit roughly all parking lots, so the system needs to be versatile.

1.4.3 Companies

The companies is another target group this reports solution should aim to appeal to. When companies adopt technologies they keep multiple factors in mind. [18]

Risks

There are risks involved with incorporating new technologies into a company's environment, especially when being a frontrunner. These new technologies might be unstable and introduce external up-time factors. Down-time results in a loss of profit and public image. [19]
It is therefore important to minimize external factors and enable the system to operate offline.

There are also legal risks in regards to user consent, which may vary depending on the country. This usually has to do with storing user data, which needs to minimized, to appease the most laws around the world. [20]
However, it might be difficult to not store any user data at all, as it is in the interest of the company to keep a log of at least number plates.
The last risk is security, which is essential when storing user data, to ensure malicious actors can not gain access. However, security in regards to not being able to physically bypass the system, e.g. avoid payment also needs to be covered.

Impact on Society

Adopting new technologies can lead to an impact on society. These impacts vary, but they often come down to automation.
Automation can lead to more productive workers. This is because part or all of the work is not done manually anymore; however, the tasks that get automated are usually tedious, which could result in more motivated workers.
There is also a negative side to automation, as it could lead to people losing their jobs; however, it should be noted that automation could also lead to more new jobs than what has been lost, which has been observed in the past, e.g. computers used to be a job.

Inclusion

When adopting a technology, companies prefer *fit-all* solutions, to not have to mash multiple technologies together, or decide to just create an in-house solution.
It is therefore important that the solution is general, which is easy to understand- and use by all drivers.

1.5 Competitors and solutions in the parking lot management sector

When creating a possible solution to the current p-issues in large cities, it is important to take a broad look at the current solutions different companies have developed and are offering. It is especially important to take a look at what solutions are being have been developed for larger p-structures. Although the needs of, for example, danish citizens and the different danish cities are different from people and cities around the globe, possible solutions everywhere could be iterated upon or dismissed entirely.

1.5.1 Privatized public parking management and payment

Currently, much of the privatized public p-management is handled differently worldwide by mobile applications or *apps*. These apps can vary in their exact handling of parking management, but they all typically provide the same service; allowing drivers to pay their parking fee remotely from their mobile phone.

Take for example Denmark; Many of the p-spaces in large danish cities are owned by large companies like *Q-Park* and *APCOA PARKING* which each have their own apps. These proprietary apps offer the user maps of nearby p-facilities, pricing for these areas and in-app payment in order to allow users to pay from their phone, rather than the on-site payment-terminals. [21], [22] Other apps from companies like *EasyPark* aim to collect multiple companies under app in order to ease user experience from having multiple apps to manage. [23]

None of these applications aim to optimize the current flow of p-structures as none of them encourage users to park smarter. These apps do provide a larger value to everyone, being that payment for parking is made faster and more efficient, allowing users to each have their own payment-terminal, avoiding potential lines at the on-site terminals.

1.5.2 Corporate parking and/or allocation solutions

Looking outside of Denmark, a broader range of solutions are currently being offered by large multinational companies. Companies like *Parkable* and *Tidaro* are current market leaders inside the field of company p-management. Their apps offer higher-ups at companies using their app, to delegate p-spaces for different employees or allowing employees to ‘claim’ a parking spot. Their apps have a baseline of features, with a possible expansion of features depending on their customer’s needs. While these apps are great for specific companies and it’s employees using this solution, it does not scale well into larger public, privatized parking. A user could face less or no consequences for claiming a spot they won’t use when parking in a *Q-Park* or *APCOA* p-lot as opposed to a company p-lot, where a company could have stricter enforcement. An app that would allow claimed, yet empty p-spaces to exist would not only be a bad solution, it would entirely worsen the existing problem, as p-lots could not be filled entirely.

Wayleadr offers its costumers a new solution for their company's p-needs, with their AI-driven automatic allocation algorithm. This algorithm allows companies using their app, to allocate different 'zones' of their p-lot, be they for the handicapped, carpooling or their employees, and Wayleadr's *Newton* algorithm automatically allocates a specific p-spot for each user each day they indicate they are driving to work. [24] A key component of their business is selling their p-solutions to other large companies who have/manage their own p-lot. As of current, this app cannot be directly deployed to a larger publicly accessible p-area, as its core functionality depends on company-level enforcement of its rules. As a result, issues similar to those seen in Parkable's and Tidaro's apps would likely still occur, such as pre-allocated spaces appearing vacant. Although this solution can not directly translate to the wider problem of unused p-spaces in the cities, a further specialized allocation algorithm could a promising part of the solution, and should therefore explored.

1.5.3 Merging of- and iteration upon these solutions

In order to optimize the p-structures of today, a comprehensive system encompassing the best qualities of current market leaders' solutions should be the aim for a solution. A system which allocates p-spaces according to drivers' needs, such as handicap-parking, shorter walking distance for the elderly and sick and other relevant factors. This system should allow for system administrators to divide p-structures up into 'zones' which allow a centralized system to automatically assign drivers a p-space according to their needed zone. This p-space should be tied to a payment system, wherein time of parking starts at the checkpoint users would go through to get their p-space, and time of departure could be controlled remotely by the driver themselves on their mobile device.

1.6 Factors That are Currently Prioritised in Parking Lots

When managing a parking lot, there are some factors that need to be prioritised more than others. Which type of vehicle is going to be the most prevalent occupant of your parking space, and for what purpose do they need to use parking?

One such factor can even be mandated by law, which is the requirement for accessible parking spaces, as showcased in Danish building regulations[25]. These spaces are required to be closely situated towards entrances, and some require more space than regular parking spaces to accommodate minibuses. The exact number of required accessible spaces is not explicitly stated in the official building regulations, only that the number of accessible spaces should be appropriately proportional to the number of regular spaces [25].

The increasing popularity of electric vehicles is also a factor to consider. Large municipalities in Denmark, like Aarhus and Copenhagen, have had initiatives such as free parking for the first 4 hours to foster higher EV adoption rates[26]. However, these initiatives have largely been done away with. This is not because they were unsuccessful, but rather because they have accomplished their goal. There are now over 400.000 EVs in use in Denmark[27], and the number of publicly available charging stations in Denmark reached 30,500 by the end of 2024, and 60 percent of all newly registered cars for that year were also EVs[28]. This increase in EVs is felt around the world, causing a large part of recent academic articles on the topic of parking lot management to concern themselves with how to better integrate EVs into parking lot spaces, and how to best utilise the energy grid connected to such spaces[29], [30], [31].

Chapter 2

Problem Statement

With increasing urbanization and an increasing amount of vehicles, modern cities face significant challenges in utilizing parking capacity efficiently. Limited parking leads to prolonged search times, higher traffic congestion and considerable economic and environmental consequences.

Although various digital parking solutions exist today, most focus primarily on payment rather than addressing the core issue of inefficient space utilization and poor traffic flow management within parking facilities, especially during peak periods.

"How can an intelligent and inclusive parking management system for parking facilities be designed to reduce search times and congestion during peak periods, while meeting the needs of citizens, municipalities, and companies through the effective use of existing and emerging technologies?"

Chapter 3

Solution; structure and requirements

With a solid problem statement, further analysis regarding requirements for a potential solution is warranted.

3.1 A MoSCoW analysis

An efficient tool for analyzing and evaluating solutions to a given problem by characterizing the important must-haves for a given solution, is the *MoSCoW* methodology. Examining the *Must-haves*, *should-haves*, *could-haves*, and *would-haves* is an important step, as to fully realize the different needs of the solution.

Must Have	***Should Have***	***Could Have***	***Would Have***
Allocate drivers p-spots for faster parking	Differentiate between p-spot types	Capacity for optimizing p-structures with multiple entrances and exits	P-space size taken into account for prioritization
A p-structure layout designer	Minimize emissions	One-way- and two-way road support	Support for modes of transport other than passenger cars
Optimize and prioritize p-spots	Verify p-spot validity and permission	P-spot checker for p-attendants	
A failsafe for erroneous parking			

Key cornerstones of this solution include the ability for it to allocate drivers a p-spot based on them and their car's attributes. The solution must in an efficient manner be able to deduce which p-spots are currently empty and which of those are the most time-efficient for the current driver. Furthermore, another key attribute of this solution must be a flexible p-structure layout designer, to allow for multiple types of p-structures to be integrated into a p-spot allocation system. An important part of optimizing a given p-structure would be to know the approximate layout and size of said structure. Such a difference between different p-structures would be impossible to account for, unless the workload of designing each individual p-structure is placed upon each individual p-structure's owner/organizer. All of this p-structure data will most importantly be fed into an optimization algorithm, which would take on the workload of allocating drivers their p-spot. Lastly, a must-have of this solution is a way for drivers to allocate themselves a new p-spot, should their first allocated p-spot be erroneously taken by another driver. If such a system is not in place, a cascading effect could occur, where drivers are backed up through the p-structure, waiting for their spot to be freed.

Positive attributes for a solution to the given problem would be for the solution to overall lower emissions of cars idling or make the currently slow search process through p-structures more efficient. A solution which encourages drivers to spend the shortest possible time from the entrance to their p-spot. Another possible user-friendly feature of this solution would be to allow users themselves to doublecheck whether or not they are parked at their assigned p-spot.

Potential attributes for this solution could be adding support for two-way roads and multiple entrances and exits. These additions will, however, introduce a larger degree of complexity to a sorting algorithm, as cars will be moving in a much less predictable manner, and multiple terminals for p-allocation would exist, which would have to interface with each other and share p-spot information. This would further support atypical p-structure design at the cost of increased development complexity.

Some potential aspects of a solution to this problem is considered out of scope, such as adding support for other modes of transport than cars, such as motorcycles and bicycles, as their p-spots are much more atypical compared to passenger cars. The actual size of each p-space and optimizing these p-spaces according to the dimensions of each unique car-model will not be considered as a relevant part of the solution, as such data unnecessarily complicates the development of a practical solution, and users could potentially modify their cars which would render this entire feature invalid.

3.2 Two systems, interfacing with each other

A complete solution which aims to solve this need for optimization within p-structures would ideally be split into two parts; a program for p-structure organizers to interface with and plan out their p-structure's layout, and another from which parkees are assigned an optimal p-spot based on the p-structure they are p-in. This structure of two systems interfacing with each other, allows the production of a general-purpose allocation algorithm which assigns p-spots, tailored to each different p-structure there could exist, by letting the organizer of said p-structure design an export parameters for this allocation algorithm.

These two systems therefore have to interface with each other, as one could not exist without the other. A p-structure organizer needs to provide a standardized prioritized list of p-spaces that an allocation system could assign p-spaces from.

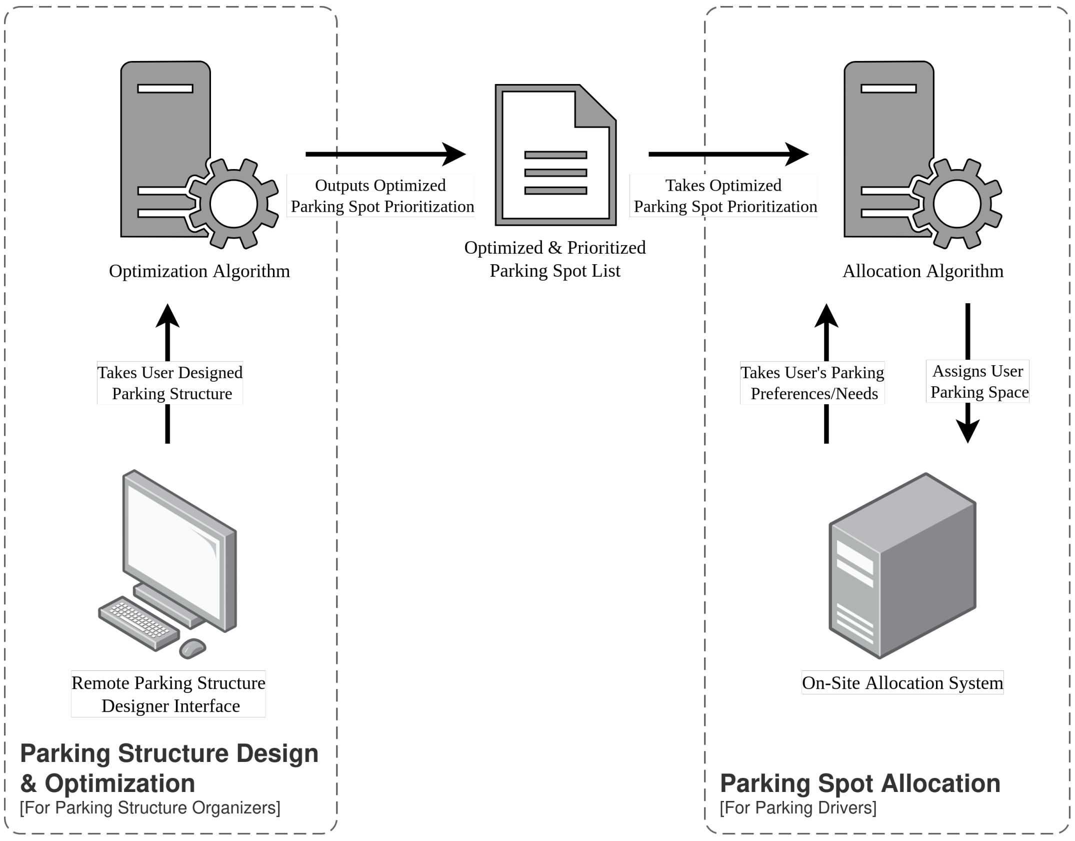

Optimization Algorithm
Outputs Optimized Parking Spot Prioritization
Optimized & Prioritized Parking Spot List
Takes Optimized Parking Spot Prioritization
Allocation Algorithm
Takes User Designed Parking Structure
Takes User's Parking Preferences/Needs
Assigns User Parking Space
Remote Parking Structure Designer Interface
On-Site Allocation System
Parking Structure Design & Optimization
[For Parking Structure Organizers]
Parking Spot Allocation
[For Parking Drivers]

Figure 3.1: Diagram of solution; two systems interfacing

The binding link between these two systems, and therefore a crucial part of this solution, is the *optimized and prioritized p-spot list* and how it is formatted to a certain standard from which an allocation algorithm can extract necessary data in order to allocate drivers their p-spot.

In order to specify which inputs and outputs these two systems should have exactly, each of their respective user-bases, use-cases and applications should be examined. As shown in Figure 3.1, these systems are respectively; *p-structure design & optimization* and *p-spot allocation*. Their general structure and requirements will each be examined.

3.2.1 Parking structure design & optimization system

A system which optimizes any given p-structure first and foremost needs to allow the end user to design any type of p-structure. Therefore, a generalized grid-based p-structure designer application is prudent for the successful implementation across a wide user-base of p-structure owners and organizers. The owners and organizers of these p-structures that would want to have their p-structure optimized need to have this optimization tailored to each of their individually unique p-structures. Offloading this part of the work onto the end user requires a modular approach to implementation of a optimization algorithm and a graphical user interface, or *GUI*, for a p-structure planner.

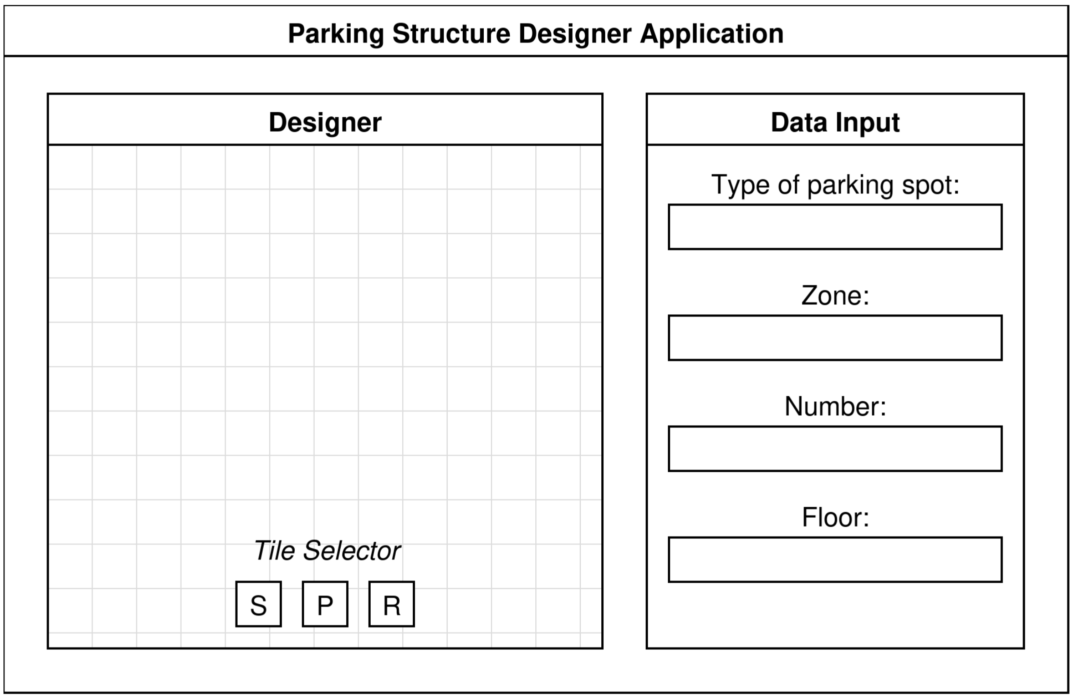


Figure 3.2: Example of GUI designer application

An example of a GUI layout of a p-structure designer application can be seen in Figure 3.2. It is important to note that, as mentioned previously in the stakeholder analysis, the majority of drivers are in the age range of 30-59 years old and 10-15% are self-admittedly digitally challenged. In can be inferred that the owners and organizer's of these p-structures are in the same age range, potentially even older, so catering to their digital challenges is important for the usability of this solution. The application should have two sub-windows, the *designer window* and the *data input window.*

The designer window would allow the user of the application to place *start/entrance-tiles*, *p-space-tiles*, and *road-tiles* onto the grid in the viewer. These tiles together would then represent the layout of their given p-structure. The data input window would then allow the user to enter data relevant to the selected p-space tile, such as *zone, number, floor,* and *type of p-spot.* The type of p-spot is especially relevant, as it would determine whether the p-space is for handicapped, priority, or other.

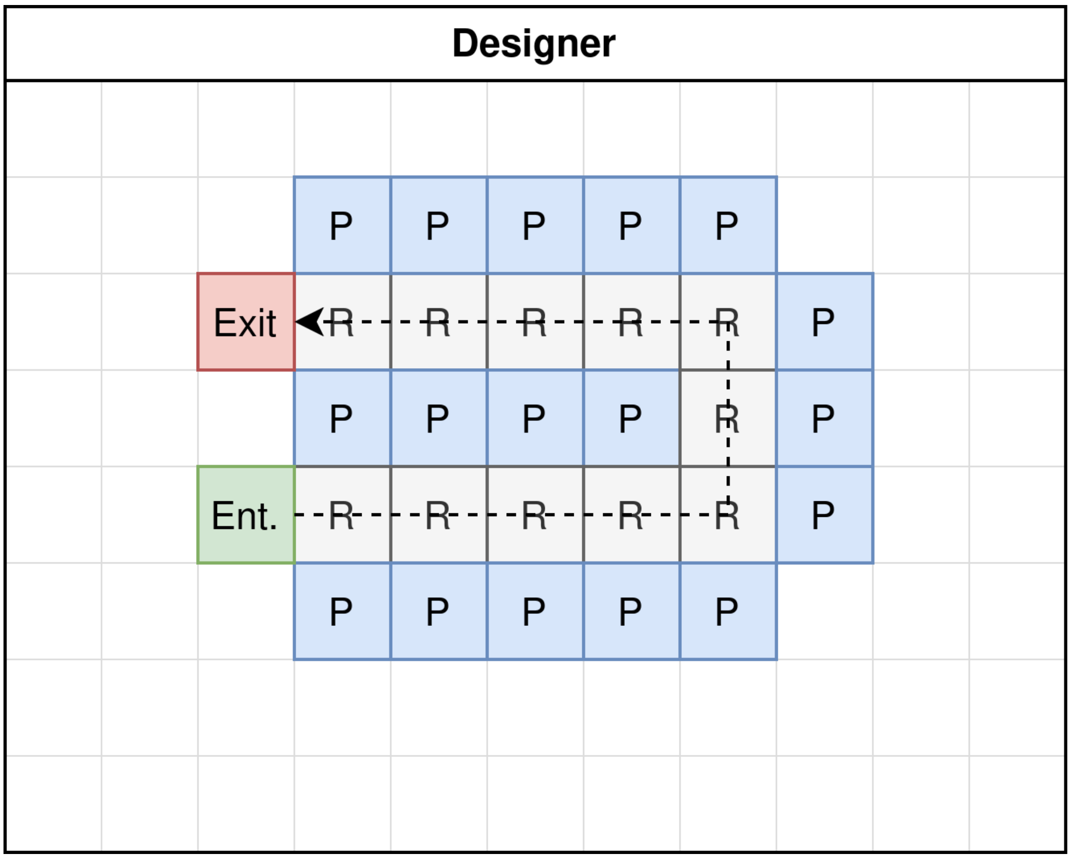


Figure 3.3: Example p-structure designed in designer application

In Figure 3.3, an example of a p-structure designed in the p-structure designer application can be seen. A series of connecting road tiles span from entrance to exit, surrounded by p-space tiles the whole way. After this p-structure is designed, the p-spaces in it, and which ones should be allocated for drivers, can be optimized. This allocation algorithm should measure the distance from a given p-spot and a given exit, and assign each p-space a prioritization number, which would indicate to the p-spot allocation algorithm which p-space to allocate a given driver.

3.2.2 Parking spot allocation system

A system which allocates p-spaces to drivers entering a p-structure should need several different components. Firstly, an *automatic number plate reader* or *ANPR*, which would read the number plate of cars waiting to be allocated a p-space, scanning their vehicle's properties in public registries, such as whether or not the vehicle is registered for handicapped parking or other specific attributes. Then, an *allocation terminal* for drivers to interface with. Drivers would get information about their allocated p-space from this terminal. This terminal would also let drivers through the *toll gate* and into the rest of the p-structure.

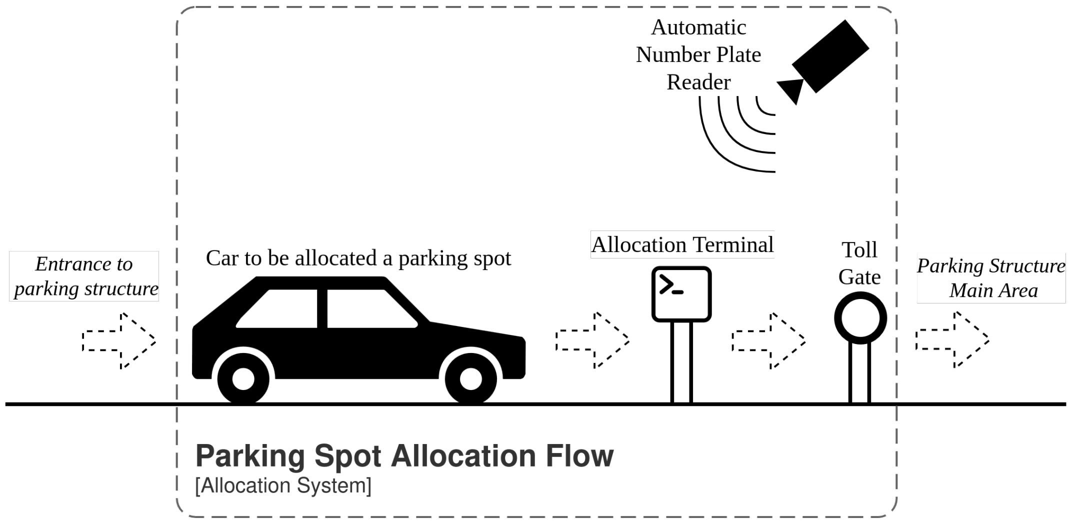

Automatic Number Plate Reader
Entrance to parking structure
Car to be allocated a parking spot
Allocation Terminal
Toll Gate
Parking Structure Main Area
Parking Spot Allocation Flow
[Allocation System]

Figure 3.4: The general flow of p-spot allocation visualized

A driver should be able to enter their own data for parking preferences, if the ANPR does not recognize their number plate. Additionally, another allocation terminal should be posted inside the p-structure to allow for reallocation of p-spaces should the previously allocated space be erroneously occupied.

Chapter 4

Methods

4.1 Analysis strategy

This chapter will go through the different analysis strategies that has been utilized throughout this report. This includes a MoSCoW analysis and a stakeholder analysis. The tools used in the creation of the product will also be listed and explain in this section.

4.1.1 Coordination and management

For coordination and managing the project as a whole, multiple Trello boards were created to see when different parts for the report or the product had been completed. This was a valuable asset and something that made the group's teamwork better. Other things like a logbook and time-plan were created using Microsoft programs, specifically Word and Excel. These have also had a great effect for structuring and collaboration.

4.1.2 Visual elements

When developing the parking lot manager, a video game programming library called raylib was included to visualize the program [32]. It was agreed upon by every member of the group that even though graphics were optional, it was important to include graphics that create a cohesive and flexible user interface.

4.1.3 The programming language

In this project, the programming language used to create the product is C. C is a general-purpose programming language created by Dennis Ritchie in 1972 [33]. It is a very popular language, due to it being fundamental or an inspiration for a lot of other programming languages. C was the required programming language for this project.

4.1.4 Environment for development

During this project, both Visual Studio Code and CLion were utilized for programming. These Internal Development Environments (IDEs) helped to create efficient code, debug code, and test out code. CLion is an obvious choice for coding in C, given that it is made specifically for C and C++, however Visual Studio Code can handle multiple languages, including C, and multiple group members prefer it to CLion.

4.1.5 GUI Clients

The code was shared using GitHub Desktop and GitKraken. This made it easier to work together while developing on the codebase, resulting in efficient and coordinated teamwork.

4.2 MoSCoW

The group chose to use the MoSCoW model to set explicit requirements and expectations for the program. These give the group a better overview over the entire product design, but also what the most important tasks for the program that needed to be done was. As the model is split into four sections, which is essentially a ranking of the different functions. These sections are ordered in priority as 'must have', 'should have', 'could have', and 'would have'.

4.2.1 Must have

The must haves of the model are what is needed to meet the requirements set for the product.[34] In other words, these are the functions that the program must have to achieve what the group wants. This is the foundation and frame of the program. It can even be described as the pillars of the program, because without these functions, the program will not achieve what is required.

4.2.2 Should have

After the foundation has been built and is working, the next point of priority takes place, which can be, for example, the program being able to take more parameters. The program is now gaining more features that will allow for a wider aspect of use. If the 'must haves' are the foundation and frame then 'should have' are the walls of the program.[34]

4.2.3 Could have

The next prioritized sections are where you go even wider with what the program needs to be able to do. These 'haves' are usually out of scope, and would only be done if there is time left when the other 'haves' are done. The two prior sections cover the functionalities that make the program usable, functional and they make make it so that these functions can be implemented without any real problems. But the 'could haves' would add more functionality and improve the quality of life, e.g. making it so that parking attendees will have an easier way to check if every car is parked in the right spot. [34]

4.2.4 Would have

Lastly the 'would haves' are the ideas that are completely out of scope for the program. These are functionalities that would be implemented if there was an endless amount of time to work on the program. They are not essential requirements for the program, as the program functions as expected, and if the 'could haves' are implemented, it even has expanded functionality. The 'would haves' can be thought of as something that would be nice to have, but not prioritized and essential. [34]

4.3 Stakeholder

The stakeholder analysis is a good tool to use for understanding how different stakeholders will have an effect on the project and the product that will be created. [35] It is also great for when you have to work with different stakeholders. In this project, as seen in section 1.4, the specific stakeholder analysis used is the power-interest grid, which divides a two axis grid into 4 equal sized squares. [35] Where the y-axis holds the complicity/power of the stakeholders, and the x-axis is influence/ interest of the different stakeholders.

4.3.1 High complicity, High interest

The first square pertains to the stakeholders who have to be well informed about the product, and who will also take part in almost all the decisions in the making of the product. It is very important to keep these stakeholders happy with the product, as this will ensure a much smoother product development.[36]

In this project, municipalities are the high complicity, high interest stakeholders of the product. They are designated as such, because they have control over their municipalities' parking policies. This means that they can implement the product directly into all p-structures that are municipal, and they also hold sway over private p-structures, as the municipalities are the ones that set the municipal policies that private companies have to adhere to. On the other hand, with private companies the situation is much more diffuse, as every individual company needs to be convinced to implement the product.

4.3.2 High complicity, Low interest

The second square covers the stakeholders that have a lot of power over the product, but are not interested in the details of the product and all of the decisions. These stakeholders have to be informed and take part in the important decision making, but are otherwise more hands-off.[36]

4.3.3 Low complicity, High interest

These stakeholders do not have a lot, if any, power over the product, but they still have a high interest in the product. The stakeholders located here are usually the users of the product, and they need to be informed, but will not take part in any of the decisions. [35]

The stakeholder for the product, who is low complicity and high interest is private companies that own p-lots and citizens who use the p-structures. As the product still needs to take their interests at heart but they don't have any direct control in the decision making.

4.3.4 Low complicity, Low interest

The last square includes the stakeholders that have a limited interest and complicity in the product. They need to be informed about the product and how it is evolving but, they don't need to be informed very often, because of their low interest [35]

Chapter 5

Program Design

This chapter will discuss how the program was designed, with focus on the parking structure design & optimization, and parking spot allocation. Where the development towards the project goals for what the program should be able to do, will discussed, The requirements and structure of the program design from chapter 3 will be used for this. The first part will go into depth with the design, and optimization of the p-structure. This will include the most important functions for the design, how they work and why their are important for the requirements of the program. After that, a walk- will go into how the allocation system works where-in that will go through the same as the design and optimization of p-structure.

The next two sections will discuss how the separate systems flow, and lastly the the whole system and how the two systems flow together to create one program, will be discussed.

5.1 Parking structure design & optimization

This section will detail the original design structure for the visual element of the program, the flow of the design system and psudocode for different functions; *parking structure design & optimization.*

5.1.1 Illustration of the parking spot prioritization

The figure below shows the original layout used to plan for the Parking Lot Manager. The idea was that the first p-spots in the driving direction would be assigned to cars first to shorten the distance and time it took to park. If the first one wasn't available, it would then check the second p-spot and so on, until it found an available spot. Afterward the other p-spot-types were added, specifically handicapped and EV. They were then assigned their own spot that were prioritized and if they were full, would then use the normal p-spots. After this the ability to add multiple floors and prioritize the p-spots between floors was added aswell.

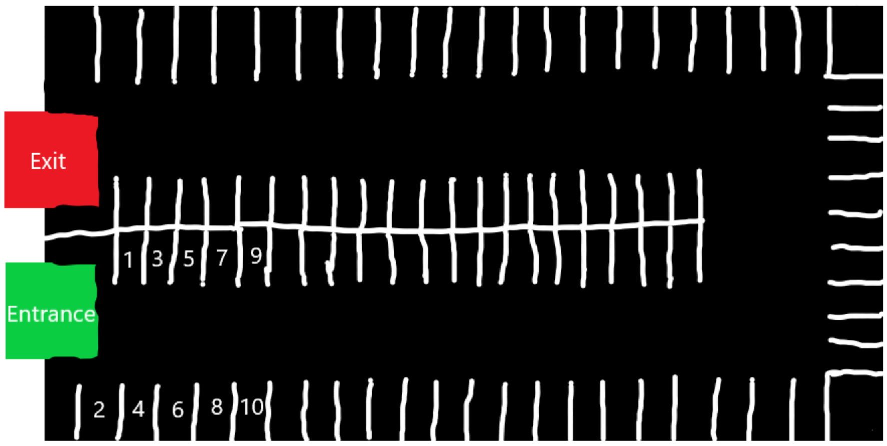


Figure 5.1: Original illustration of how the prioritization was designed.

5.1.2 Pseudocode for the different functions

Algorithm 5.1: General Priority Assignment Algorithm

```
1:  function AssignPriority(roads, entrances)
2:      ▷ Reset all roads' distances.
3:      for $i$ < |roads| do
4:          roads.distance ← 0
5:      end
6:      ▷ Update all roads' distances, which are connected to an entrance—directly
        and indirectly.
7:      for $i$ < |entrances| do
8:          entrance ← entrances[i]
9:          neighbors ← GetSurroundingRoads(entrance.position)
10:         for $j < 4$ do
11:             neighbor ← neighbors[j]
12:             UpdateRoad(neighbor)
13:         end
14:     end
15: end
```

Algorithm 5.2: Update Road

```
1:  function UpdateRoad(road)
2:      ▷ Updates the road distance and neighboring road distances,; if the new
        distance is different from the roads current distance.
3:      distance ← GetRoadDistance(road.distance)
4:      if road.distance = null then
5:          return
6:      end
7:      road.distance ← distance
8:      roads ← GetSurroundingRoads(road.position)
9:      if roads = null then
10:         return
11:     end
12:     for $i < 4$ do
13:         $r$ ← roads[$i$]
14:         if $r$ = null then
15:             continue
```

```
        end
        UpdateRoad(road)
    end
end
```

Algorithm 5.3: Reset Road

```
▷ Resets road value and neighbor values, if not already reset.
function ResetRoad(road)
    if road.distance = 0 then
        return
    end
    road.distance ← 0
    roads ← GetSurroundingRoads(road.position)
    if roads = null then
        return
    end
    for $i < 4$ do
        $r$ ← roads[$i$]
        if $r$ = null then
            continue
        end
        ResetRoad(road)
    end
end
```

Algorithm 5.4: Get Road Distance

```
▷ Gets the lowest distance from neighboring roads, and skips neighbors
  without a distance.
function GetRoadDistance(position)
    hasEntrance ← HasSurroundingEntrance(position)
    if hasEntrance then
        return 1
    end
    roads ← GetSurroundingRoads(position)
    if roads = null then
        return 0
    end
    minDistance ← MAX
    for $i < 4$ do
        $r$ ← roads[$i$]
        isNull ← $r$ = null
        isLarger ← road.distance $\geq$ minDistance
        isNegative ← road.distance $\leq 0$
        shouldSkip ← isNull $\vee$ isLarger $\vee$ isNegative
        if shouldSkip then
            continue
        end
        minDistance ← road.Distance
    end
    if minDistance = null then
        return 0
    end
    return minDistance + 1
end
```

Algorithm 5.5: Parking Spot Prioritization

```
function PRIORITIZE(Vehicle type)
    ▷ Returns list of spot types in priority order; closest to entrance should be chosen first
    if Vehicle type = Handicap then
        return [Handicapped, Normal]
        else if Vehicle type = Electric Vehicle then
            return [EV, Normal]
            return [Normal]
        end
    end
end
```

This function checks a vehicle type and then return a list of available spaces that the car can park on. It checks for handicapped- and EV-cars and if the car is neither or there are no more, then it return the list of Normal spots.

5.1.3 The flow of the system

This part of the program allows the user to either create or open an existing project, where the user then can use different tools to place the entrance, create parking spaces, add floors and set up driving aisles, so the parking structure can be designed to match their requirements or existing layout. Different parking spots, like handicapped- and EV-parking spots, can also be created. Here the user then can select how the parking spots should be prioritized. Once the design is complete, it can then be saved as a list of parking spaces with their type and placement ready to be allocated.

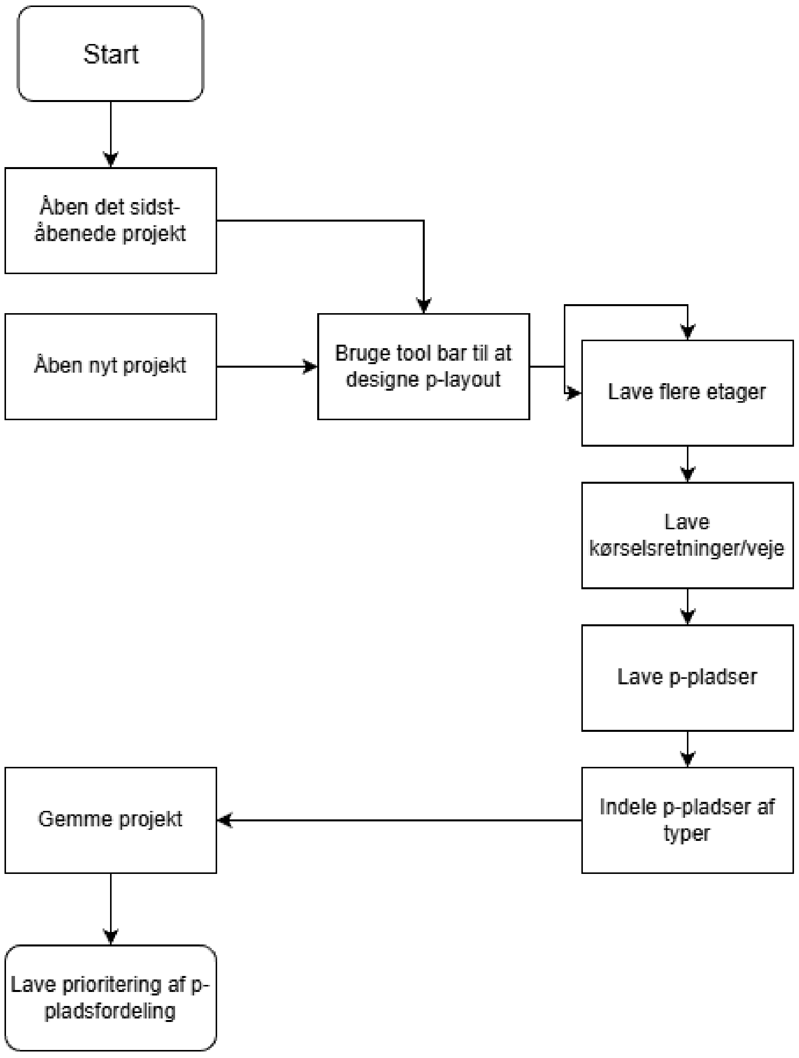

Start
Åben det sidst-åbenede projekt
Åben nyt projekt
Bruge tool bar til at designe p-layout
Lave flere etager
Lave kørselsretninger/veje
Lave p-pladser
Indele p-pladser af typer
Gemme projekt
Lave prioritering af p-pladsfordeling

Figure 5.2: Flowchart of how the parking structure design is supposed to work

5.2 Parking spot allocation

This section will detail the console application; *parking spot allocation.*

5.2.1 Pseudocode for the different functions

Algorithm 5.6: Check occupation

```
1:  procedure
    checkOccupation(▷ Check what spaces are free, and what type it is, )
2:  end
3:  for $i$ < length of total parkingspaces do
4:      if Status = Vacant then
5:          if Type = Standard then
6:              return [Available space [i]]
7:          end
8:          else if Parkinglot type = Car type then
9:              return [Available space [i]]
10:         end
11:     end
12: end
13: return “Error message(All spaces is occupied)”
```

This functions first checks if a spot is vacant or occupied, and if the p-space is vacant it will then check what type the p-space is... if that isn’t true, it will then check for what type the car is and find a suitable p-space. If checkOccupation finds no empty p-spaces it will then return “All spaces occupied”.

Algorithm 5.7: Scan data file

```
1:  procedure                                                    scanDataFile
    (▷ Scan the file for information regarding the parking structure, )
2:  end
3:  while Reading a new line of file is possible do
4:      function Copy line of string(LineCopy, Line)
5:          LineCopy ← Line
6:      end
7:      ParkingID ← Copy of id from the current line of file
8:      if ID = ParkingID  then
9:          ▷ Split the line copy into 3 strings, split for each Comma(,)
10:         return [The information of that parkingspace into respective variables]
11:     end
12:     ▷ Keep going into reaching the end of file.
13: end
```

The purpose of the function scan data file, is making the program able to read the file containing information regarding the parking structure. Furthermore it should be able to store information it reads from the file into variables that can be used to make the program scale to any parking structure.

Algorithm 5.8: Create Parking Lot

```
1:  procedure createParkingLot()
2:      ▷ Creating the parking lot by importing the data from the file into an list
        of structures
3:      for i < size of parking lot do
4:          Struct parking lot [i] ← Result of scanDataFile()
5:      end
6:  end
```

This function is what will create import all the data of the parking lot, into an array of structs. It will run as many times as there are lines in the parking structure file.

So it will be able to scale to any amount of parking lots.

Algorithm 5.9: Get Assigned lot of car

```
1: procedure GETASSIGNEDLOTOFCAR()
2:     ▷ Function to get information regarding an specific car, by searching on
       the numberplate
3:     for i < amount of cars in the parking lot do
4:         if Searched plate = cars[i].plate then
5:             return assigned lot of car
6:         end
7:     end
8: end
9: return -1 if no car was found in the list by for the searched number plate.
```

For the program to be able to check where a car should be located in the structure after allocation, therefore this function will make it possible to search through all the current allocated cars in the parking structure. The search will be done with the numberplate, and the function will return the lot the car is assigned to. Otherwise it will just return with a error message, incase it is not possible to find the car.

5.2.2 The flow of the system

One of the first parts of the program's design phase was to construct a flowchart that illustrates the processes that are part of the software program's allocation program.

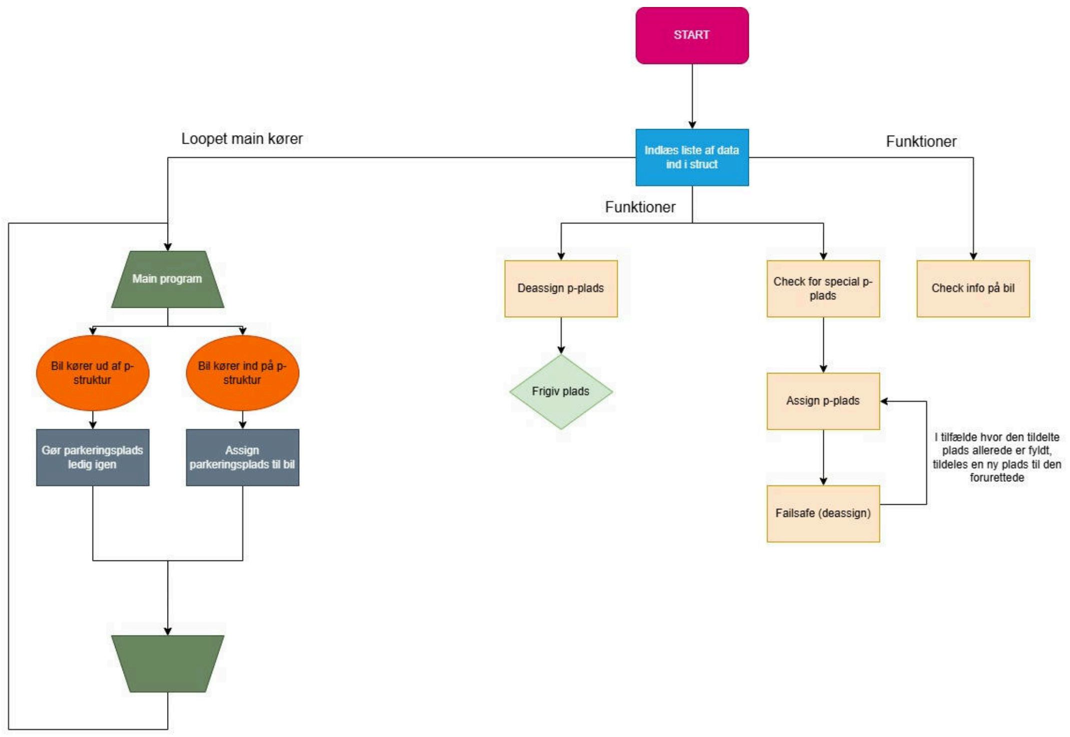

START
Loopet main kører
Indlæs liste af data ind i struct
Funktioner
Funktioner
Main program
Bil kører ud af p-struktur
Bil kører ind på p-struktur
Gør parkeringsplads ledig igen
Assign parkeringsplads til bil
Deassign p-plads
Frigiv plads
Check for special p-plads
Assign p-plads
Failsafe (deassign)
I tilfælde hvor den tildelte plads allerede er fyldt, tildeles en ny plads til den forurettede
Check info på bil

Figure 5.3: Flowchart of the allocation program, as envisioned by the project group

The left part of the flowchart illustrates the main program loop, whereas the right side shows the functions that are part of the program. The idea is for the program to parse data that has been gathered from a file. If this was a real-life scenario, the program would most likely rely on data gathered from scanners in the p-structure itself, but adding a scanner functionality would far exceed the boundaries of the current scope. Therefore, the group has elected to have the data be simulated instead. This data is parsed into two types of structs, one for the p-spaces themselves and one for the cars that are going through the p-lot.

The middle branch of the flowchart describes the functions that are used to assign and unassign a p-space to a given motorist traversing the p-structure. Firstly, a function is used to check for special flags that are attributed to the cars of some motorists, those flags being whether they are an electric vehicle (designated in the code as car_EV), or a car that is cleared for accessible p-spaces (designated in the code as car_HANDICAPPED). If one of those flags are raised, an appropriate p-space will be assigned to the car in question. In the case of EVs specifically, there is not a strict requirement that this type of vehicle can only occupy EV p-spaces, so in the event where no EV p-spaces are available, a standard p-space will be assigned to the given EV instead. In the event that there are no unoccupied p-spaces left, a message will inform the motorist that all spaces are occupied. In the event that a motorist should get assigned a p-space that is already occupied by another vehicle, the motorist is then to return to the, in this case hypothetical, parking machine to be assigned a different p-space, and a hypothetical parking attendant will then be dispatched to rectify the situation in the already occupied p-space. Barring incidents like the aforementioned, the intention is for the p-space to be unassigned once the car that has been assigned to the space leaves the p-structure, and once the p-space has been unassigned from this car, the space is then marked as vacant once more.

The right-most branch describes a function that checks the info of a given vehicle. It is meant to be used through other functions to store the information for later use. One might also imagine a scenario where the information of a car on a specific p-space needs to be read, to check whether the car matches the assigned p-space.

The left-most branch of the flowchart describes the program's main function and what sort of actions the program perform, and it also illustrates the looping nature of the program.

Chapter 6

Implementation

This chapter will focus on how the different libraries and packages have been implemented into the coding of the Parking Lot Manager. Different functions and algorithms from the code will also be shown and explained. The visual elements of the Parking Lot Manager, i.e. the UI, will also be shown and explained.

6.1 Integrating the code

The product has the following file structure.

```
└── src
    └── camera
        ├── camera.c
        ├── camera.h
        ├── CMakeLists.txt
    └── project
        ├── CMakeLists.txt
        ├── project.c
        ├── project.h
    └── utils
        ├── CMakeLists.txt
        ├── utils.c
        ├── utils.h
    ├── CMakeLists.txt
    ├── definitions.h
    └── main.c
```

6.2 Included libraries and packages

6.2.1 CMake

CMake is a standardized software development tool that makes it possible to describe how to build simple and very complicated software systems with a single set of input files. Below is a CMakeLists.txt file located in the “src” folder in the parking structure design & optimization portion of the code.

```cmake
cmake_minimum_required(VERSION 3.15)
project(frontend C)

# Tell CMake to use vcpkg installed libraries
set(CMAKE_TOOLCHAIN_FILE "C:/Users/jdj10/Documents/Programmering/vcpkg/installed/x64-windows/include" CACHE STRING "")

# Find external libraries
find_package(raylib CONFIG REQUIRED)
find_package(nfd CONFIG REQUIRED)

# Main executable
add_executable(main
    main.c
)

# Add subdirectories for your own libraries
add_subdirectory(utils)
add_subdirectory(project)
add_subdirectory(camera)
add_subdirectory(selection)

# Link libraries to main
target_link_libraries(main
    PRIVATE
        raylib
        nfd::nfd
    PUBLIC
        utils
        project
        camera
        selection
)
```

6.2.2 Raylib

Raylib is a programming library designed for making videogame graphics. An extension for raylib, called raygui, is also implemented. Raylib does not include any visual aid or GUI, requiring pure programming to create anything visual using it.

This program utilizes raylib to visually show the parking lot, parking spots, different floors, roads and entrance. The ability to inspect each parking spot to see their type, number and ID is also functional. This can be seen and is explained in more detail in 6.3.2.

6.2.3 NativeFileDialog -extended

Native File Dialog (NFD) is a small library used to natively open files, select folders and and save dialogs. NFD-extended however is an expanded upon version that is overall better than the original version and is still receiving updates.

6.2.4 Mtest

For backend unit-testing, a C unit testing framework called mtest was utilized. Mtest was developed by Morten Schou and uploaded to GitHub. It is used in this project to verify parking spot assignment and deassignment. This can be seen and is explained more in 6.6.

6.2.5 Stdio.h

Stdio stands for Standard Input/Output and is a header file that includes functions to read off of the keyboard (input) and show it on the screen (output). It also handles file management, like printf, scanf and fopen.

6.3 Implementation of the code

6.3.1 Prioritizing the different parking spaces

Prioritizing a parking structure's parking spaces is done by assigning each road-tile a number, given by its distance from the entrance-tile. Thereafter, a parking space is given its prioritization number by the road-tiles number adjacent to it. In order to enable this functionality, several functions are called.

`update_road()` is called whenever an entrance-tile is placed or whenever a road-tile is placed. The function recursively assigns road-tiles each their distance from the entrance-tile. The function firstly needs to get the distance of the road through the `get_road_distance()` function. That function takes the road-tile's *position* value and passes it through several checks. Firstly, an array of pointers of the Road struct type is created by the `get_surrounding_roads()` function. This array of pointers is 4 elements large, and contains the pointers to the surrounding road-tiles. Then, a for loop checks to see if this array is empty, because if it is empty then that should mean that this road tile is unconnected. If it had an entrance-tile next to it, it would have been caught by the `has_surrounding_entrance()` function call and the entire function should return 1. This is so that the first road-tile in a series of road-tiles has a number of 1, and the subsequent road-tiles can have their distance value calculated. The next part of the function checks for each connected surrounding road-tile (skipping the empty array elements) and checks whether or not these tiles' distance value are greater than or equal to the min_distance, which is set to the maximum integer value, and skips them if they are. It also skips a tile if its distance is 0 or less. All of this is error handling. Finally, the min_distance value is overwritten and replaced with the current road's distance and the function can free the roads in memory and return the min_distance plus 1, if the min_distance doesn't exceed the INT_MAX. All in all, the function reads adjacent road-tiles for the one with the smallest distance value which is over 0, and assigns the current road that value plus 1.

Returning this value allows the `update_road()` function to establish a distance value from the given road-tile and return the function if it is the correct value. Otherwise, it assigns the road's distance value to the calculated distance of the tile and moves onto a neighboring tile. It keeps going until a neighboring tile's road's distance is correct and closes this recursive loop. This function just loops through every road tile until they are all checked and verified.

The `reset_road()` function works the same way recursively, but instead of setting the distance values to a calculated value, it set them all to the value of 0 to reset the entire roadway.

```c
void update_road(Road *road) {
  int distance = get_road_distance(road->position);
  if (road->distance == distance)
    return;

  road->distance = distance;

  Road **roads = get_surrounding_roads(road->position);
  if (roads == NULL)
    return;

  for (int i = 0; i < 4; i++) {
    Road *neighbor = roads[i];
    if (neighbor == NULL)
      continue;

    update_road(neighbor);
  }

  free(roads);
}
```

```c
void reset_road(Road *road) {
  if (road->distance == 0)
    return;

  road->distance = 0;

  Road **roads = get_surrounding_roads(road->position);
  if (roads == NULL)
    return;

  for (int i = 0; i < 4; i++) {
    Road *neighbor = roads[i];
    if (neighbor == NULL)
      continue;

    reset_road(neighbor);
  }

  free(roads);
}
```

```c
int get_road_distance(Vector2 position) {
  if (has_surrounding_entrance(position))
    return 1;

  Road **roads = get_surrounding_roads(position);
  if (roads == NULL)
    return 0;

  int min_distance = INT_MAX;
  for (int i = 0; i < 4; i++) {
    Road *road = roads[i];
    if (road == NULL)
      continue;

    if (road->distance >= min_distance)
      continue;

    if (road->distance <= 0)
      continue;

    min_distance = road->distance;
  }

  free(roads);
  return min_distance == INT_MAX ? 0 : min_distance + 1;
}
```

```c
int checkOccupation(struct parking_lot parking_lot[], int length,
struct car car) {
    for (int i = 0; i <= length; i++) {
        if (parking_lot[i].status == VACANT ) {
            if (parking_lot[i].type == STANDARD) {
                return i;
            }
            else if (parking_lot[i].type == car.type) {
                return i;
            }
        }
    }
    printf("All spaces is Occupied");
    return -1;
}
```

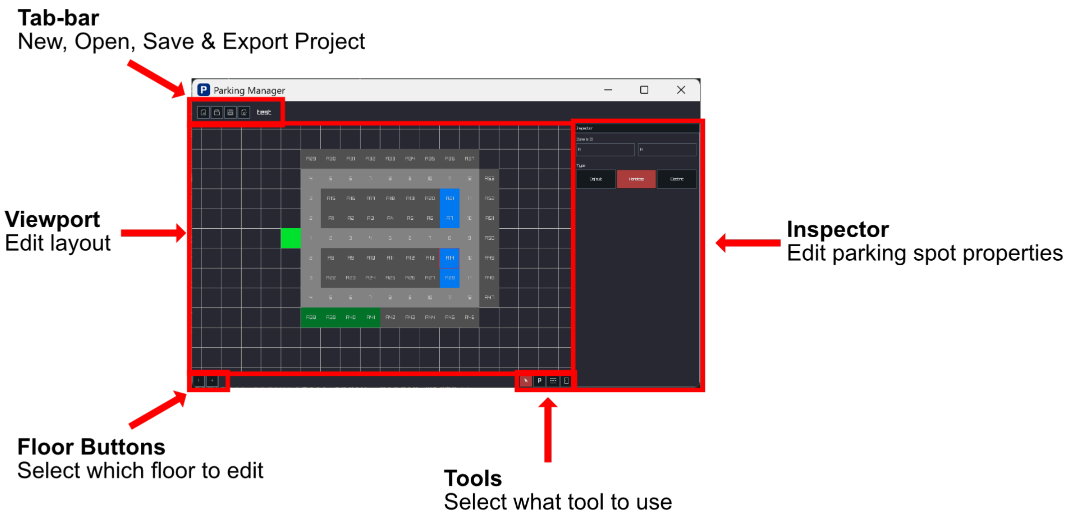
```c
int getAssignedLotOfCar(struct car cars[], int amount) {
    char plate[16];

    printf("Enter plate: ");
    scanf("%15s", plate);
    printf("Lookingd for %s\n", plate);
    printf("%s\n", cars[1].number_plate);

    for (int i = 0; i < amount; i++) {
        if (strcmp(plate, cars[i].number_plate) == 0) {
            printf("Car found on spot %d\n", cars[i].assigned_lot);
            return cars[i].assigned_lot;
        }
    }

    printf("No car with that number plate\n");
    return -1;
}
```

6.3.2 UI



6.3.2.1 Tab-bar

The tab bar in the product consists of a text element and four buttons. The text element only appears while having an active project. The first two buttons are to initialize a project—you can either create a new project or open an existing one.

The function `draw_tab_bar()` uses the library *raygui* within `<raylib.h>` to draw the tab-bar to the screen. The function starts out by drawing the background—consisting of a solid color rectangle and a line to separate the tab-bar from the rest of the window.
When creating the buttons a tool-tip is set to ensure users will understand the functionality, without initially understanding the icon. The four similar with the only difference being the position and icon. The state of the button is then stored for later use.

To capture the name of the project the character '\' and '\0' are taken advantage of. The backslash indicates when changing directory. After the last backslash is the file name. Therefore; by starting the string one character after the last backslash, the filename is gotten. The character '\0', however; indicates the termination of a string, and by changing the last '.' in the acquired file name, the file extension is removed, however; it is important to perform this operation on a copy of the path, as this manipulates the string, and the original string is critical for use outside this function.

```c
void draw_tab_bar() {
  DrawRectangle(0, 0, SCREEN_WIDTH, TAB_BAR_HEIGHT, get_bg_color());
  GuiLine((Rectangle){0, TAB_BAR_HEIGHT, SCREEN_WIDTH, 1}, NULL);

  const char *tooltips[] = {"New Project", "Open Project", "Save
  Project", "Export Project"};
  const char *icons[] = {NEW_ICON, OPEN_ICON, SAVE_ICON, EXPORT_ICON};
  bool pressed[4] = {false};

  GuiEnableTooltip();
  for (int i = 0; i < 4; i++) {
    GuiSetTooltip(tooltips[i]);
    pressed[i] = GuiButton(
      (Rectangle){
        .x = TAB_BAR_PADDING + (BUTTON_SIZE + BUTTON_SPACING) * i,
        .y = TAB_BAR_PADDING,
        .width = BUTTON_SIZE,
        .height = BUTTON_SIZE
      },
      icons[i]
    );
  }
  GuiDisableTooltip();

  if (pressed[0]) new_project();
  if (pressed[1]) open_project();
  if (pressed[2]) save_project();
  if (pressed[3]) export_project();

  if (project == NULL) return;

  char *path_copy = strdup(project->path);
  if (!path_copy) return;

  const char *project_name = path_copy;
  char *last_slash = strrchr(path_copy, '\\');
  char *last_dot = strrchr(path_copy, '.');
  if (last_slash) project_name = last_slash + 1;
  if (last_dot) *last_dot = '\0';
  const int font_size = 18;
  DrawText(
    project_name,
    TAB_BAR_PADDING + (BUTTON_SIZE + BUTTON_SPACING) * 4 +
    BUTTON_SIZE / 2,
    TAB_BAR_PADDING + BUTTON_SIZE / 2 - font_size / 2, font_size,
    WHITE
  );
  free(path_copy);
}
```

6.3.2.2 Viewport

The viewport is based on a camera rendering to a texture, which is then drawn within the viewport's designated area.
What is rendered within the viewport, however; consists of several elements. The elements are the background grid, parking spots, roads, entrances, selection preview and selection.

The functions `draw_spots()`, `draw_roads()` and `draw_entrances()` are very similar with the only difference being what array to loop through, color and text. For the sake of simplicity and redundancy, only `draw_spots()` will be explained.
`draw_spots()` works by looping though the *spots* array within the global *project* struct. It then draws a rectangle at the spot's properties, such as *position* and *type*, and overlays a text element, displaying the spot's *zone* and *id*.

```c
void draw_spots() {
  if (project == NULL)
    return;

  Spot **active_floor = &project->floors[project->active_floor].spots;
  int *spot_count = &project->floors[project->active_floor].spot_count;

  for (int i = 0; i < *spot_count; i++) {
    Spot *spot = &(*active_floor)[i];
    Color color = get_spot_color(spot->type);
    DrawRectangleV(spot->position, (Vector2){50, 50}, color);
    draw_centered_text(TextFormat("%c%d", spot->zone, spot->id),
                       spot->position.x + 25, spot->position.y + 25, 12);
  }
}
```

6.3.2.3 Floor Selection

The floor selection consists of a button list based on the floors and a button to add a new floor to the list. When right-clicking a floor button you initialize a confirmation menu to delete that floor.

The function `draw_floor_buttons()` has a static variable *deleting_floor_index* variable declared inside, meaning it is only accessible inside the function, however; consistent between function calls. If the *deleting_floor_index* is not (-1) the confirmation menu to delete a floor is shown. If confirmed, the floor at index *deleting_floor_index* will be deleted. After the confirmation menu has been drawn or skipped, the floor buttons will be drawn. This in done through a loop, drawing and checking whether the drawn buttons are pressed. If a button is pressed, the active floor is changed to the index *i* of the button. However; if the button is right-clicked, then the *deleting_floor_index* is set to *i* and the confirmation menu will be drawn next call. Lastly the button to add a new floor is drawn, and if pressed it will call the function `add_floor()`, which allocates space for a new floor and increments the *floor_count* integer within the *project*.

```c
void draw_floor_buttons() {
  static int deleting_floor_index = -1;

  // Handle delete confirmation
  if (deleting_floor_index != -1) {
    int result = GuiMessageBox(
      (Rectangle){
        (float)(SCREEN_WIDTH - INSPECTOR_WIDTH) / 2 - 125,
        (float)(SCREEN_HEIGHT - TAB_BAR_HEIGHT - BUTTON_SIZE) / 2 +
        TAB_BAR_HEIGHT - 50,
        250, 100
      },
      "#191#Delete Floor",
      TextFormat("Are you sure you want to delete floor %d?",
      deleting_floor_index + 1),
      "Confirm;Cancel"
    );
    if (result != -1) {
      if (result == 1) remove_floor(deleting_floor_index);
      deleting_floor_index = -1;
    }
  }

  // Draw floor buttons
  for (int i = 0; i < project->floor_count; i++) {
    int x = (BUTTON_SIZE + BUTTON_SPACING) * i;
    Rectangle btn_rect = {x, SCREEN_HEIGHT - BUTTON_SIZE, BUTTON_SIZE,
    BUTTON_SIZE};
    bool pressed = GuiButton(btn_rect, TextFormat("%d", i + 1));
    bool right_clicked = IsMouseButtonPressed(MOUSE_BUTTON_RIGHT) &&
                         CheckCollisionPointRec(GetMousePosition(),
                         btn_rect);

    if (right_clicked && project->floor_count > 1) {
      deleting_floor_index = i;
      break;
    }
    if (pressed) project->active_floor = i;
  }

  // Add floor button
  int x = (BUTTON_SIZE + BUTTON_SPACING) * project->floor_count;
  if (GuiButton((Rectangle){x, SCREEN_HEIGHT - BUTTON_SIZE, BUTTON_SIZE, BUTTON_SIZE}, "+"))
    add_floor();
}
```

6.3.2.4 Tool Selection

The tool selection consists of a toggle-group, and updates the current tool index by the index of the button in the toggle-group. There is also a line drawn at the top of the tool selection to separate the element from the rest of the window.

The function `draw_tool_bar(...)` takes two inputs. A pointer to the *tool_index* variable and the width of the renderer/viewport.
In the function a line and a toggle-group is drawn, and when drawing the toggle-group, the *tool_index* is passed, both setting the default value and updating the value, if a toggle-group button is pressed.

```c
void draw_tool_bar(int *tool_index, int render_width) {
  GuiLine((Rectangle){0, SCREEN_HEIGHT - BUTTON_SIZE,
                      SCREEN_WIDTH - INSPECTOR_WIDTH, 1},
          NULL);

  GuiToggleGroup((Rectangle){.x = (float)render_width - 4 * BUTTON_SIZE -
                                  3 * BUTTON_SPACING,
                             .y = SCREEN_HEIGHT - BUTTON_SIZE,
                             .width = BUTTON_SIZE,
                             .height = BUTTON_SIZE},
                 TextFormat("%s;%s;%s;%s", INSPECT_ICON, PARKING_ICON,
                            ROAD_ICON, ENTRANCE_ICON),
                 tool_index);
}
```

6.3.2.5 Inspector

The inspector—as of now—allows to manipulate properties of just parking spots, and is split into two functions.
One function as a general inspector-drawer and another to draw the specific properties of the selected parking spot.
The manipulation of said properties is handled through two input-fields for the *zone* and *and*, and a toggle-group for the *type*.
The data from the input-fields are then attempted to be parsed back into a char and integer respectively.

The function `draw_inspector(...)` draws a background for the inspector and calls `draw_spot_inspector(...)` if a parking spot is selected.
`draw_spot_inspector(...)` has two static boolean variables to determine whether an input-field is currently selected for editing, which they are when pressed inside its area, and not when pressed outside its area. The spots properties are saved into new variables and converted to strings, which can be used when setting the default value

of the input-field and allowing to retrieve the edited values. After retrieving the values are converted back into their respected types and assigned to their original variables.

The function `draw_inspector(int render_width)`

```c
void draw_inspector(int render_width) {
  GuiPanel((Rectangle){render_width, TAB_BAR_HEIGHT,
                       SCREEN_WIDTH - render_width,
                       SCREEN_HEIGHT - TAB_BAR_HEIGHT},
           "Inspector");

  if (selection.type != SPOT)
    return;

  draw_spot_inspector(render_width);
}
```

The function `draw_spot_inspector(int render_width)`

```c
void draw_spot_inspector(int render_width) {
  static bool zone_edit = false;
  static bool id_edit = false;
  Spot *spot = (Spot *)selection.ptr;

  // Convert spot->id (int) to string
  char new_zone = spot->zone;
  int new_id = spot->id;
  char zone_str[16] = "";
  char id_str[16] = "";

  sprintf(zone_str, "%c", new_zone);
  if (spot->id >= 0)
    sprintf(id_str, "%d", new_id);
  int half_width = (float)(INSPECTOR_WIDTH - 24) / 2;

  // Zone & ID
  GuiLabel((Rectangle){render_width + 8, TAB_BAR_HEIGHT + 24,
                       INSPECTOR_WIDTH - 16, 32},
           "Zone & ID");

  Rectangle zone_rect = (Rectangle){
      render_width + 8, TAB_BAR_HEIGHT + 24 + 8 + 18, half_width, 32};
  Rectangle id_rect = (Rectangle){render_width + half_width + 16,
                                  TAB_BAR_HEIGHT + 24 + 8 + 18,
                                  half_width, 32};

  if (IsMouseButtonPressed(MOUSE_BUTTON_LEFT)) {
    Vector2 mouse_pos = GetMousePosition();
    bool is_inside_zone = CheckCollisionPointRec(mouse_pos, zone_rect);
    bool is_inside_id = CheckCollisionPointRec(mouse_pos, id_rect);

    if (is_inside_zone) {
      zone_edit = true;
      id_edit = false;
    } else if (is_inside_id) {
      zone_edit = false;
      id_edit = true;
    } else {
      zone_edit = false;
      id_edit = false;
    }
  }

  GuiTextBox(zone_rect, zone_str, sizeof(zone_str), zone_edit);
  GuiTextBox(id_rect, id_str, sizeof(id_str), id_edit);

  if (sscanf(zone_str, "%c", &new_zone) == 1) {
    spot->zone = new_zone;
  }

  if (sscanf(id_str, "%d", &new_id) == 1) {
    spot->id = new_id;
  }

  // Type
  GuiLabel((Rectangle){render_width + 8, TAB_BAR_HEIGHT + 24 + 48 + 18,
                       INSPECTOR_WIDTH - 16, 32},
           "Type");
  int type_index = (int)spot->type;
  GuiToggleGroup(
      (Rectangle){render_width + 8, TAB_BAR_HEIGHT + 24 + 48 + 44,
                  (float)(INSPECTOR_WIDTH - 16 - BUTTON_SPACING * 1.5) / 3, 48},
      "Default;Handicap;Electric", &type_index);
  spot->type = (SpotType)type_index;
}
```

6.3.3 Reading the data of the parking structure

The function `scanDataFile()` is used to read a csv (comma separated values) file, which contains all the information regarding the designed parking structure, created by the design tool described earlier. It works by opening the file `FILE *file = fopen("data.csv", "r");` in an read only environment. To use the data stored in the file, the function `fgets()` is used, which is an built-in function in the `<stdio.h>` library. It works by reading an line, and storing that into an array until it reaches an new line or the end of the file. The first line of the data file is an header line, only containing titles of the data stored in the file. Therefore the program needs to skip this line, and this is done by running the `fgets()` function once. Then the function utilize an while loop `while (fgets(line, sizeof(line), file))` which runs the code inside the loop, as long as it is still possible to read an new line in the data file. Then a specific ID in the data is looked for, in order to get the information regarding an specific parking lot. This information is then being converted into the correct types, and then gets stored into variables.

```C
int scanDataFile(int id, int *floor, char *zone, int *number, enum type *type) {
    // Open file
    FILE *file = fopen("export.pexport", "r");
    if (file == NULL) {
        printf("Could not open export.pexport\n");
        return 1;
    }
    char line[MAX_LINE];
    int found = 0;

    // Skip header line (The first line in the file)
    fgets(line, sizeof(line), file);


    while (fgets(line, sizeof(line), file)) {

        char lineCopy[MAX_LINE];
        strcpy(lineCopy, line);
        char *parkingID = strtok(lineCopy, ",");
        int compareID = atoi(parkingID);

        // Ensure right parking spots in list
        if (id == compareID){
            // Separates string at every comma
            char *floorStr  = strtok(NULL, ",");
            char *zoneStr   = strtok(NULL, ",");
            char *numberStr = strtok(NULL, ",");
            char *typeStr   = strtok(NULL, ",");


            if (floorStr && zoneStr && numberStr && typeStr) {
                // Saves data in pointers
                *floor = atoi(floorStr);
                strcpy(zone, zoneStr);
                *number = atoi(numberStr);
```

```c
                *type = atoi(typeStr);

                found = 1;
                break;
            }
        }
    }

    if (!found) {
        printf("No record found for %d\n", id);
    }

    fclose(file);
    return 0;
}
```

6.3.4 Getting the size of the parking structure

For our program to be able to work on different parking structures, that varies in size and amount of parking lots. Therefor the program needs to be able to recognize the size of the parking structure, this is implemented through the function `int lenghtOfDataFile()` which returns an integer value. The function works by calculating the amount of new lines.
It does this by looking at each character in the file, by utilizing the function `fgetc()`. Every time it reads a new line character, the count for lines will rise by one. This action will keep going until the function reaches the end of the file.

```c
int lengthOfDataFile() {
    // Open file data.csv and save it
    FILE *file = fopen("data.csv", "r");
    // Print if an error occurs during initialization of file.
    if (file == NULL) {
        printf("Could not open file data.csv");
        return 1;
    }
    int current_line = 1;
    char c;
    do {
        // Read next character from file
        c = fgetc(file);
        // The character '\n' indicates a new line in the file.
        if (c == '\n') current_line++;
        // Continue until we hit the end of the file
    } while (c != EOF);
    // Close the file
    fclose(file);

    // Returns the length of the file
    return current_line;
}
```

6.3.5 Adding a car

For the program to be able to handle new cars entering the parking structure, the function addCar() was implemented. The program stores the amount of cars, that is in the parking structure in an Array. But when the program is initialized its unknown how many cars, there will be demand for. Therefore it was implemented every time a new car gets added, the size of the array was resized. The function will take the original array, and return a new array with an added car inside it.

```c
struct car* addCar(struct car *cars, int *carCount) {
    char plate[16];
    int type;

    printf("Enter number plate:");
    scanf("%15s", plate);

    printf("Car type (0 = STANDARD, 1 = HANDICAPPED, 2 = EV): ");
    scanf("%d", &type);

    // Check at rigtig type er indtastet
    if (type < 0 || type > 2) {
        printf("Invalid type. Car not added.\n");
        return cars;
    }

    // Resize array, +1 bil
    struct car *newCars = realloc(cars, (*carCount + 1) *
    sizeof(struct car));

    if (newCars == NULL) {
        printf("Adding car failed. Car not added.\n");
        return cars;
    }

    cars = newCars;

    // Tilføj bil til listen
    strcpy(cars[*carCount].number_plate, plate);
    cars[*carCount].type = (enum type)type;
    cars[*carCount].assigned_lot = -1;

    (*carCount)++;

    printf("Car %s added successfully!\n", plate);

    return cars;
}
```

6.3.6 Removing a car

Removing an car from the program, works in the same way as adding a new car. The program needs to be able to adjust the size of the list containing the cars. The function `removeCar()` will there for reduce the size of the list with one, every time an car gets removed from the parking structure.

```c
struct car* removeCar(struct car *cars, int *carCount, int indexToRemove) {
    if (*carCount == 0) return cars;

    // Flyt alle elementer 1 plads til venstre
    for (int i = indexToRemove; i < *carCount - 1; i++) {
        cars[i] = cars[i + 1];
    }

    (*carCount)--;

    // Gør array 1 mindre i størrelse
    struct car *newArr = realloc(cars, (*carCount) * sizeof(struct car));

    if (newArr == NULL && *carCount > 0) {
        printf("Removing car failed, returning old list.\n");
        return cars;
    }

    return newArr ? newArr : cars;
}
```

6.3.7 Getting information about a car

For the program to be able to get information regarding a car, the function `getAssignedLotOfCar()` was implemented. It works by letting the user search through the cars assigned to the parking structure, by making a search with the numberplate. It uses the function `strcmp()`which compares if two strings is the same. If that statement is true, the program will print information regarding that car, and also which parking lot it is assigned to.

```c
int getAssignedLotOfCar(struct parking_lot lots[], int lotCount, struct car cars[], int carCount)
    {
        char plate[16];
        while (getchar() != '\n'); // flush leftover input
        printf("Enter number plate: ");
    scanf("%15s", plate);

    // Søg gennem liste af biler udfra nummerplade
    for (int i = 0; i < carCount; i++) {
      // Check op den indtastede nummerplade matcher en nummerplade i listen
      if (strcmp(plate, cars[i].number_plate) == 0) {

        int lotNumber = cars[i].assigned_lot;
        // Hvis nummerpladen ikke er assignet til nogen plads
        if (lotNumber == -1) {
          printf("Car %s is not assigned to any lot.\n", plate);
          return 0;
        }
        // Print information ift bilen
        for (int j = 0; j < lotCount; j++) {
          if (lots[j].number == lotNumber) {
            printf("\n=== Car Found ===\n");
            printf("Number Plate: %s\n", cars[i].number_plate);
            printf("Car Type: %s\n", typeToString(cars[i].type));

            printf("\n--- Assigned Lot ---\n");
            printf("Lot Index:   %d\n", j);
            printf("Lot Number:  %d\n", lots[j].number);
            printf("Floor:       %d\n", lots[j].floor);
            printf("Zone:        %s\n", lots[j].zone);
            printf("Type:        %s\n", typeToString(lots[j].type));
            printf("Status:      %s\n\n", statusToString(lots[j].status));
            return 0;
          }
        }
        // Print information ift bilen
        printf("Car found but assigned to unknown lot number %d\n", lotNumber);
        return 0;
      }
    }
    // Hvis ikke nogen bil med nummerpladen findes.
    printf("No car with that plate exists.\n");
    return -1;
  }
```

6.3.8 Assigning car to a lot

The function `autoAssignNewCar()` assigns a car to an empty parking lot, it uses the implemented function `checkOccupation()` to get which parking lot the car should be assigned to, and the assigns the car to that lot with the function `assignSpace()`

```c
void autoAssignNewCar(struct car *newCar,
                      struct parking_lot parkinglots[],
                      int lotCount)
{
    int freeIndex = checkOccupation(parkinglots, lotCount, *newCar);

    if (freeIndex == -1) {
        printf("No free compatible parking lot available for this car.
\n");
        newCar->assigned_lot = -1;
        return;
    }

    assignSpace(&parkinglots[freeIndex], newCar);

    printf("Car %s assigned to lot number %d (index %d)\n",
           newCar->number_plate,
           parkinglots[freeIndex].number,
           freeIndex);
}
```

6.4 Complexity

In the first semester of Software a mandatory course, called The Theoretical Foundations of Computer Science (DTG), has a subject about functions and complexity (block 2). This block taught how different algorithms have different complexities, categorized by the *big O notation*. This allows someone to calculate the theoretical amount of time an algorithm takes when it is called in a program.

The two main components of this software solution are firstly prioritizing a parking list and secondly allocating cars to a prioritized parking structure list.

Looking at the code for prioritizing the parking list, several different coding structures appear. Firstly, functions like the `update_road()` function are called recursively, with a simple check for a base case which would close the loop. Functions like these are *linear* and therefore fall under *O(n)*. Secondly ...

6.5 Simulation

This section will simulate how long it takes to fully allocate all the cars in a given parking structure. The first test will check whether or not front to back allocating or back to front allocating is fastest for a given list of cars. The test here is to see what saves the most about of time, if there is any difference at all. The second is to test how long it takes for a single floor p-structure with a given amount of p-spaces compared to a multiple story p-structure with the same amount of p-spaces.

6.5.1 General structure of simulation

This simulation should simulate the time required for a line of cars of different types to park in a given parking structure. The simulation should be flexible to account for multiple different parking structures, so that the flexibility of the parking structure planner is not invalidated. The general structure of this simulation should then account for this line of cars, each of which can be given a index according to their placement in the line.

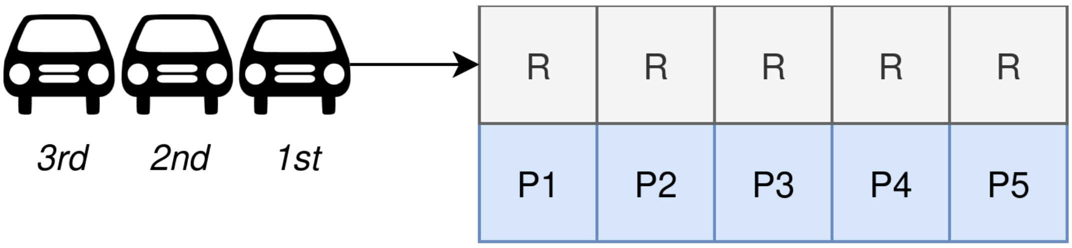


Figure 6.1: Parking simulation outline

A parking structure which has had its parking spaces prioritized by the solutions prioritization algorithm, would then have a prioritization number for each parking space. A visualization of this general outline can be seen in Figure 6.1.

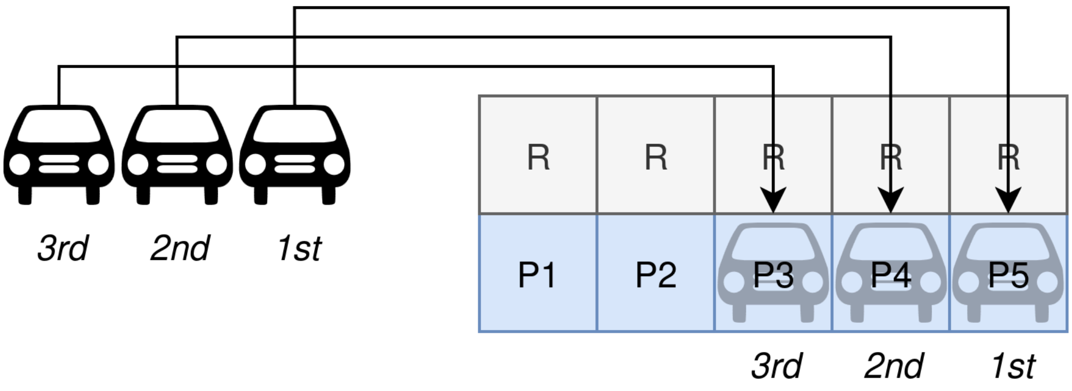


Figure 6.2: Parking simulation, back to front assignment

Now, for the sake of explanation, assume the cars are to be parked *back to front*, so each car is assigned the highest prioritization number in the parking space list. If each car is of the same type and the entire parking structure is empty, each car should easily fill the parking structure, as no car would ever be stuck behind another car slowly entering their assigned parking space. This speed of actual parking is important to take note of, as the time taken for actually maneuvering a given car into its assigned parking space would be slower than the cruising time for a car to drive from the entrance to said allocated parking space. A visualization of this can be seen in Figure 6.2.

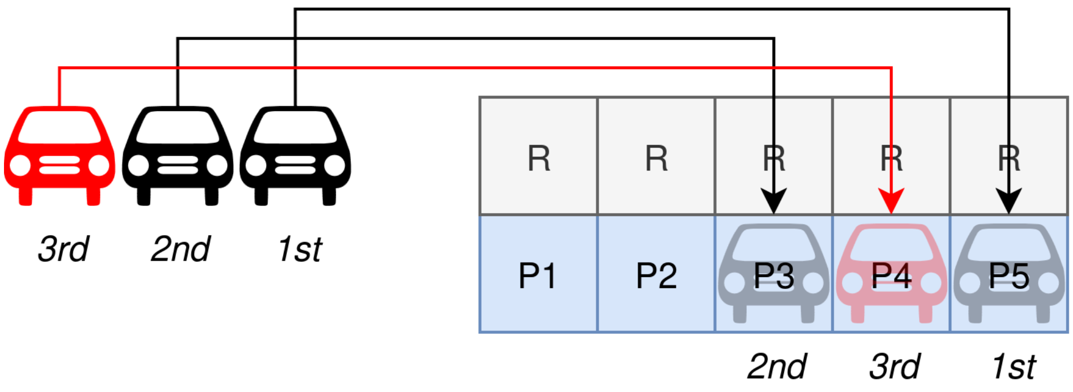


Figure 6.3: Wait time in back to front assignment

These circumstances are ideal and only true in quite a niche scenario of cases, cases where this allocation system would also be obsolete, as an empty parking structure wouldn't exactly need an allocation system for drivers to quickly find an empty parking space. It is therefore important to account for errors and a line of cars backing up. In Figure 6.3, a visualization of a common occurrence in parking structures can be seen; a driver slowly parking their car while the next car in line needs to go further into the parking structure.

In order to accurately model these characteristics of time taken filling a parking structure, these key points need to be taken account for. Cars who don't block each other should run right after another, and as soon as a car is making the entire line wait, the simulation should take that into account.

6.5.2 Simulation over front to back vs. back to front allocation

...

6.5.3 Simulation of allocation in single story vs. multiple story parking structure

...

6.5.4 Simulation of parking structure at almost full capacity.

To test how much our program optimize the average time for an driver to park, in an parking structure that is almost at full capacity. The program simulated a case where a parking structure containing 91 parking lots, during an simulated peak period. The parking structure was simulated to only have 22 empty spaces located randomly throughout the parking space, and then was being loaded with 20 extra cars.
Figure A.4 in the appendix visualize the simulated parking structure.
The way the simulation was executed, was by having every car start on an base time of 20 seconds, this time was set as a base for taking the time, it tajes for the driver to park in the lot after driving to it into account. The parking structure designed for this test, was divided into different zones. These zones were used to calculate how long time it would take an driver, to drive from the entrance of the structure

to the allocated parking space.
In the simulation it took in average for the cars 46.25 seconds to park the car. With the slowest time for an driver being 65 seconds, and the fastest time for an driver being 45 seconds.
In the study done over a simular parking structure without an parking manager from Leeds, it took up to 214 seconds for a driver to find a parking lot [12]. So from simulating the program it can be argued that it improves and optimize the search time for drivers when an parking structure is almost full.

6.6 Testing

It was stated as a requirement that the product should have some kind of unit testing of the code. This type of testing is very important for making sure the code functions as it should. So it is used not just because it is a requirement but also because using testing is very useful, especially for bigger projects where there are a lot of functions with various sizes that need to function together. This chapter will go through the use of mtest, the testing framework chosen for unit testing, and then how the different test cases work for the product.

6.6.1 The use of mtest

To be able to do unit testing on the product, a testing framework was needed, the easiest to understand and use was mtest, which was made by the professor of our imperative programming course, Morten K. Schou. This unit testing framework is inspired by doctest and CuTest, and it is a simple framework made only for C. It was chosen, because the group has worked with it before, and knows how to use it.

In the framework "*we*" use macros like "TEST_CASE" to test the different functions in the program. There is a lot of different macros that can be used, some are dedicated macros for example "CHECK_GT_INT" which lets us compare integers, with the GT meaning greater than.[37] Another example of a check made using mtest is: "REQUIRE_EQ_DOUBLE" which is a require test, which purpose it is to check if the expected action or output has been done correctly, but if it has not done this, then it has failed the require test. The test will then stop the code there, and not let it keep running through the rest of the code.

6.6.2 Unit testing the Allocating of cars

There have been implemented multiple unit tests for the assigning of p-spaces, deassigning of p-spaces, reassigning of p-spaces, and checking if a p-space is occupied. These 4 functions is the whole loop of the allocation and deallocation of p-spaces of vehicles in the p-structure.

```
TEST_CASE(assignLot_test, {
    struct parking_lot lotArray[10];
    struct car car = {"ABC123", STANDARD, -1 };
// Intializing the two structs for the parking lot and the car itself for
the test case

    for (int i = 0; i < 10; i++) {
        lotArray[i].floor = 1;
        strcpy(lotArray[i].area, "A");
        lotArray[i].type = STANDARD;
        lotArray[i].status = VACANT;
    }

    assignSpace(&lotArray[0], &car);

    CHECK(car.assigned_lot == 0, "Car should be assigned lot 0");
    CHECK(lotArray[0].status == OCCUPIED, "lot number 0 should be
occupied");
})
```

Above is the first test case for the four functions, this test checks if the function "assignLot" assigns the lot to the car, and changes the status of the p-space to occupied instead of VACANT. The test case starts by initializing a parking lot and a car, by using the structs created for each. A for loop is now run to check through the p-lot for an empty spot, and if an empty spot that fits the type of the vehicle is found, it will then assign that space. Then the code check if the car's assigned lot is the one that was assigned to it, and furthermore if the p-space status has changed from VACANT to OCCUPIED.

```
TEST_CASE(deAssignSpace_test, {
    struct parking_lot lotArray[10];
    struct car car = {"ABC123", STANDARD, -1 };

    for (int i = 0; i < 10; i++) {
      lotArray[i].floor = 1;
      strcpy(lotArray[i].area, "A");
      lotArray[i].type = STANDARD;
      lotArray[i].status = OCCUPIED;
    }

    deAssignSpace(&lotArray[0], &car);

    CHECK(lotArray[0].status == VACANT, "Lot status changed to vacant");
    CHECK(car.assigned_lot == -1, "Car is deassigned from the space and
returns -1");

})
```

The second test run is for the function "deAssignSpace" where as explained above it deassigns a space from a car and changes the lot status to VACANT. The test wants to test exactly if it will return these two things. It is essentially the opposite of the test for assigning a p-space. Its structure is almost the same as the test above, as the code initializes the two structs and has a for loop that checks for occupied spaces, and then deassigns that space. The code then check that the lot status has changed to VACANT and that the car's assigned lot is changed to -1.

```
TEST_CASE(reAssignSpace_test, {
    struct parking_lot lot = {1, "A", 0, STANDARD, VACANT};
    struct car car = {"ABC123", STANDARD, -1 };

    reAssignSpace(&lot, &car, 0);

    CHECK(lot.status == ERROR, "Lot status needs to be set to Error when
reassigned");
    CHECK(car.assigned_lot == 0, "Car assigned lots is set to 0");
})
```

The next test seen above is for the function to reassign a p-space, which should be able to allocate a new p-space for a car, and then set that lot's status to "ERROR". As the other tests, first the test case initializes a parking lot and a car with the assigned lot of -1. So the test runs the function and then checks to see if the lot status is changed and returns the correct status and does the same with the assigned lot to the car.

```
TEST_CASE(checkOccupation_test, {
    struct car car = {"ABC123", STANDARD, -1};
    struct parking_lot lotArray[10];

    for (int i = 0; i < 10; i++) {
      lotArray[0].floor = 1;
      strcpy(lotArray[0].area, "A");
      lotArray[0].type = STANDARD;
        if (i < 3) {
            lotArray[i].status = OCCUPIED;
        } else {
            lotArray[i].status = VACANT;
        }
        lotArray[i].number = i;
    }
    int index = checkOccupation(lotArray, 9, car);

    CHECK(index == 3, "checkOccupation will skip over the first 3 lots,
since they are occupied and return index 3");
})
```

The last test for the allocation of cars is the function for running through a p-lot. This function is very important for the whole allocation process of cars parking. Therefore it is important that the function works, and the test is therefore very important. The test, same as all the others, starts with initializing a car and a p-lot. A for loop then starts going through the p-lot where it checks if index i for the p-lot number is lower than 3, then the status is occupied, otherwise the status is vacant, so the function exits the if-else loop and sets the lot array number equal to “i”. Then the function “checkOccupation” runs for the lotarray, the length of the index, and then the cars’ information. The test now checks for index 3, as the function should return this index because C is an index-0 based language and so the first three p-lots are occupied.

6.6.2.1 Summarization of unit testing

>

Chapter 7

Discussion

7.1 Limitations and restrictions

Due to several limiting factors, such as budget, time and technical experience, several limitations and restrictions were either directly or indirectly imposed upon the product solution. A key distinction to make, is between *limitations* and *constraints*. Several constraints have been willfully put onto the product solution, during the several stages of development, such as the ones outlined in the MoSCoW analysis. Constraints are boundaries set for the product solution, as to not veer out of scope for the given problem statement, or to not add unnecessary workload during the product development stage. Limitations, however, are not boundaries that have been willfully set and are instead set by outside factors, such as the ones described earlier. These limitations and the reasons for them will be outlined here.

7.1.1 Limitations of C

C is fundamentally a good programming language, however there are many limitations that have had an effect on the development of the product throughout this project. One of these things is the lack of a feature like inheritance in Object Oriented Programming (OOP). Inheritance in OOP works by letting subclasses or child classes acquire the methods and properties of classes. In this project there has been spent a lot of time writing the same variables and parameters that otherwise could have been saved, while also making the code clearer.

Another is the need to implement libraries in C. Base C lacks many features required to create a project such as this one. These can be implemented using libraries like Native File Dialog Extended and Stdio etc.

7.1.2 Grid-based parking structure designer

The p-structure designer developed as part of this project's solution is entirely *grid-based*, meaning that p-spaces and roads are all represented by the same perfectly square tiles with the exact same dimensions. This is not wholly representative of real world p-structures. P-structure roads and p-spaces vary greatly in size and shape, and a program which is rigidly grid-based lacks the capability to totally conform to different p-structures and their design. Another point against this grid-based design, is that many p-structures have diagonal roads and p-spaces, which the current p-structure designer application can not fully represent. This further limits the usability of this solution.

A possible mitigation to these grid-based problems, could be to assign roads and p-spaces varying dimensions, made up of smaller grids. Meaning, instead of having a p-space be a perfect square, taking up 1x1 grid space, a p-space could instead be represented by 6 grids, forming a 2x3 space.

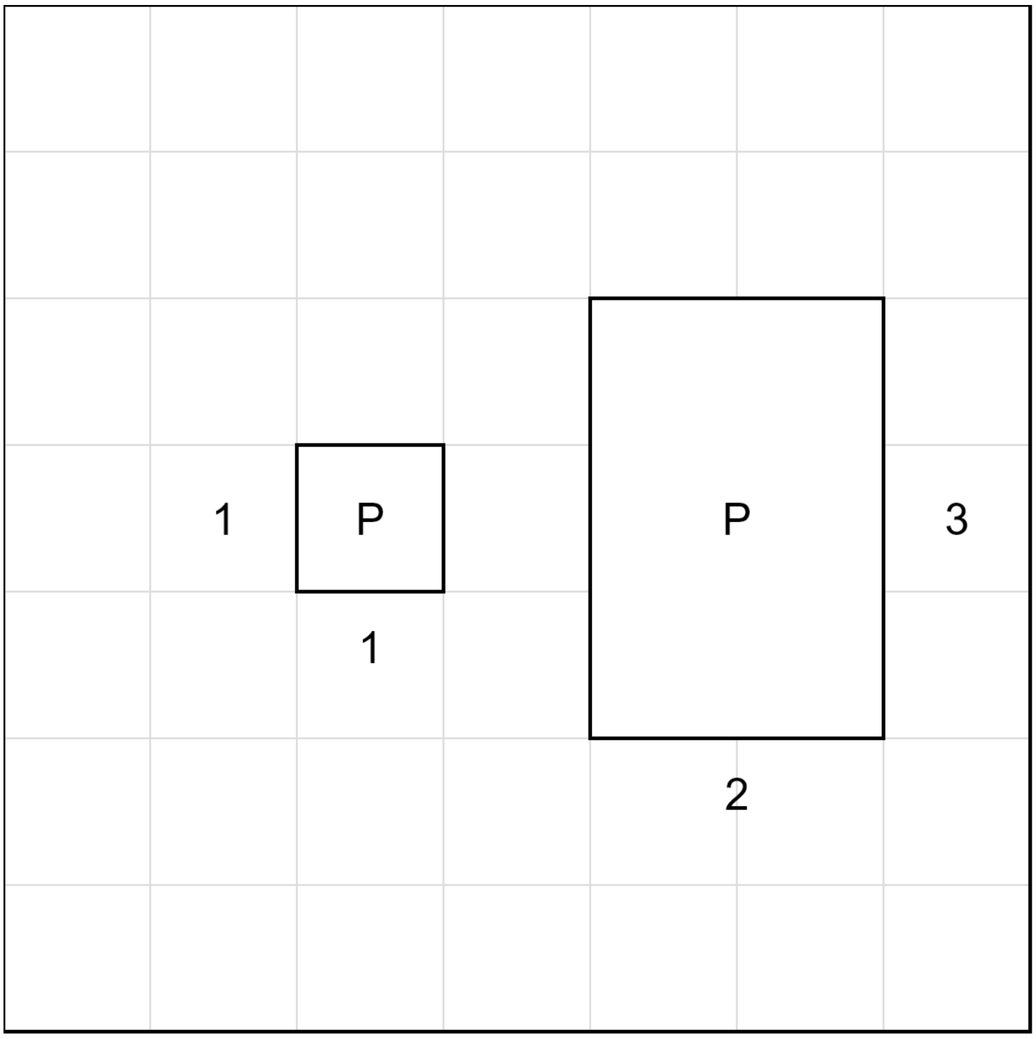


Figure 7.1: Visualization of variation in p-space size

In Figure 7.1, a visualization of this way of representing a p-space with different widths and heights can be seen. This alternate way of visualizing and designing p-structures in the p-structure designer would be able to reflect real world p-structures more accurately.

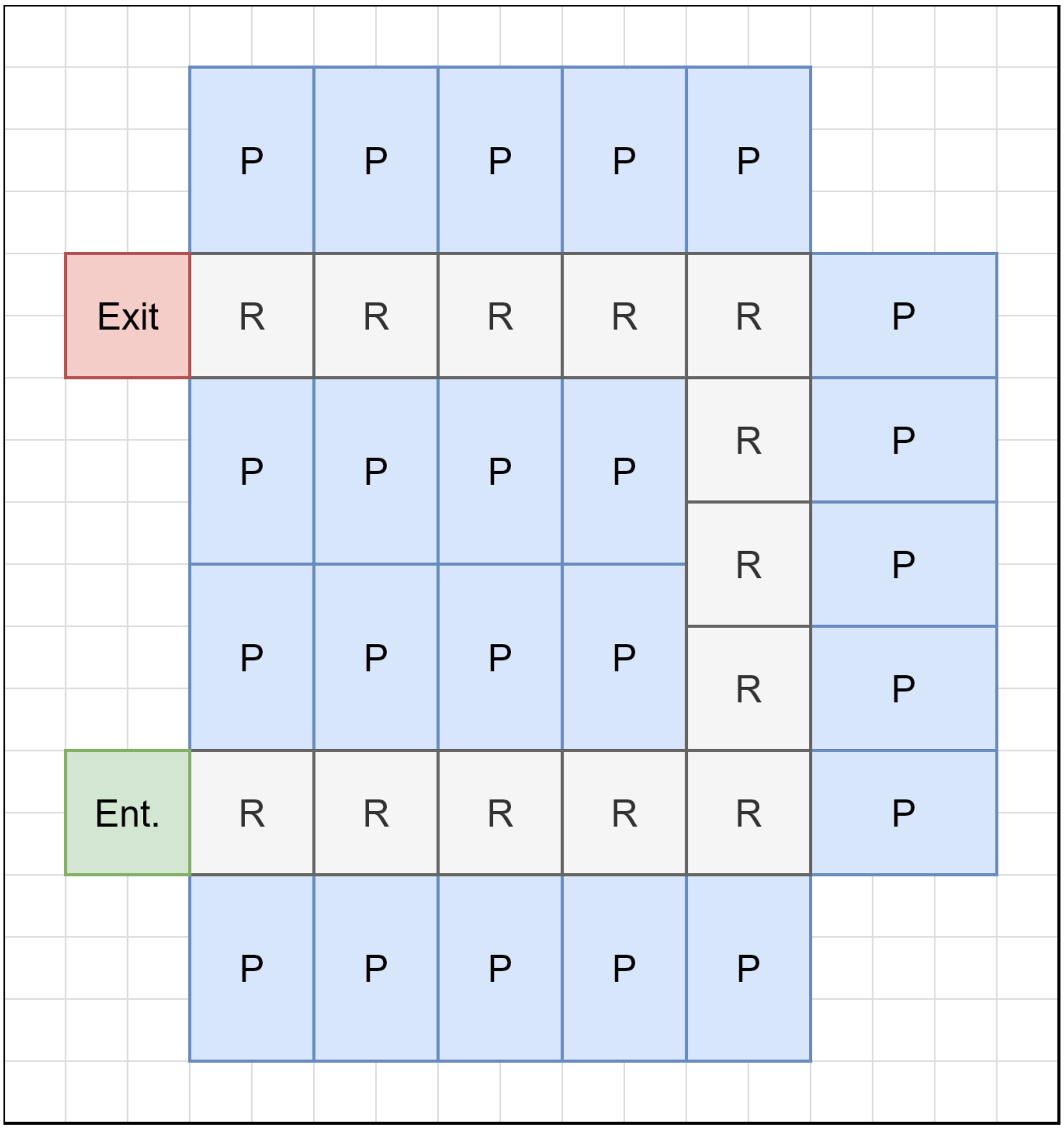


Figure 7.2: p-structure example with varying widths and heights

An example of a complete p-structure designed according to these different p-space sizes can be seen in Figure 7.2. While the exact dimensions in this example are still not wholly representative of an actual real-world p-structure and its layout, the rectangular p-spaces are much more realistic than the current project's solution of perfectly square p-spaces.

These specific limitations and the reason for it not being part of the current solution, is mainly due to time constraints. Allowing tiles to be of varying sizes and placing them on a grid should be a rather straight-forward matter, yet a time consuming one. A large part of this is the redesign of the algorithm which assigns prioritization to each different p-space in the p-structure designer. The current algorithm counts the tiles from a given p-space to the entrance according to the grid; varying tile sizes would warrant further abstraction of how these tiles are represented in code and would add significant development time and effort, which was not available.

7.1.3 Different prioritization algorithm

While not a limitation, the current prioritization algorithm which assigns p-lot spaces works by counting the roads tiles from a given p-space to the entrance of the p-lot and ordering the p-spaces according to this. This is functional, but during development a different approach was proposed; A sort of pathfinding algorithm, which would travel along the road tiles of a given p-structure and assign adjacent p-spaces increasing prioritization values.

Such an algorithm would take it's current position, and check adjacent tiles starting north and going clockwise until it has checked all four adjacent tiles.

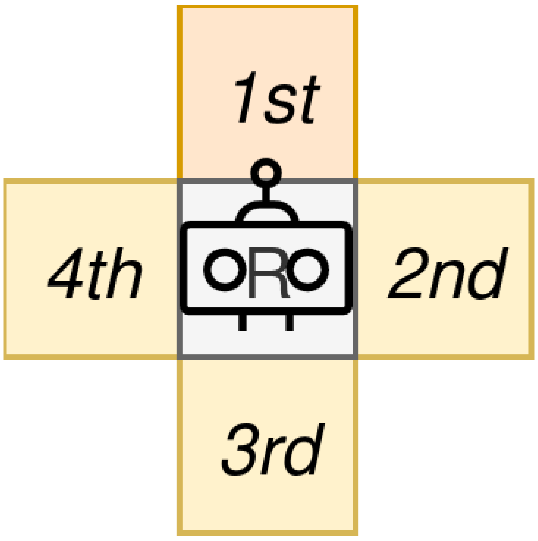


Figure 7.3: Visualization of alternative prioritization algorithm

It would then start at the entrance and travel along the road tiles, assigning a prioritization value for tiles which has not yet been assigned in the order as seen in Figure 7.3.

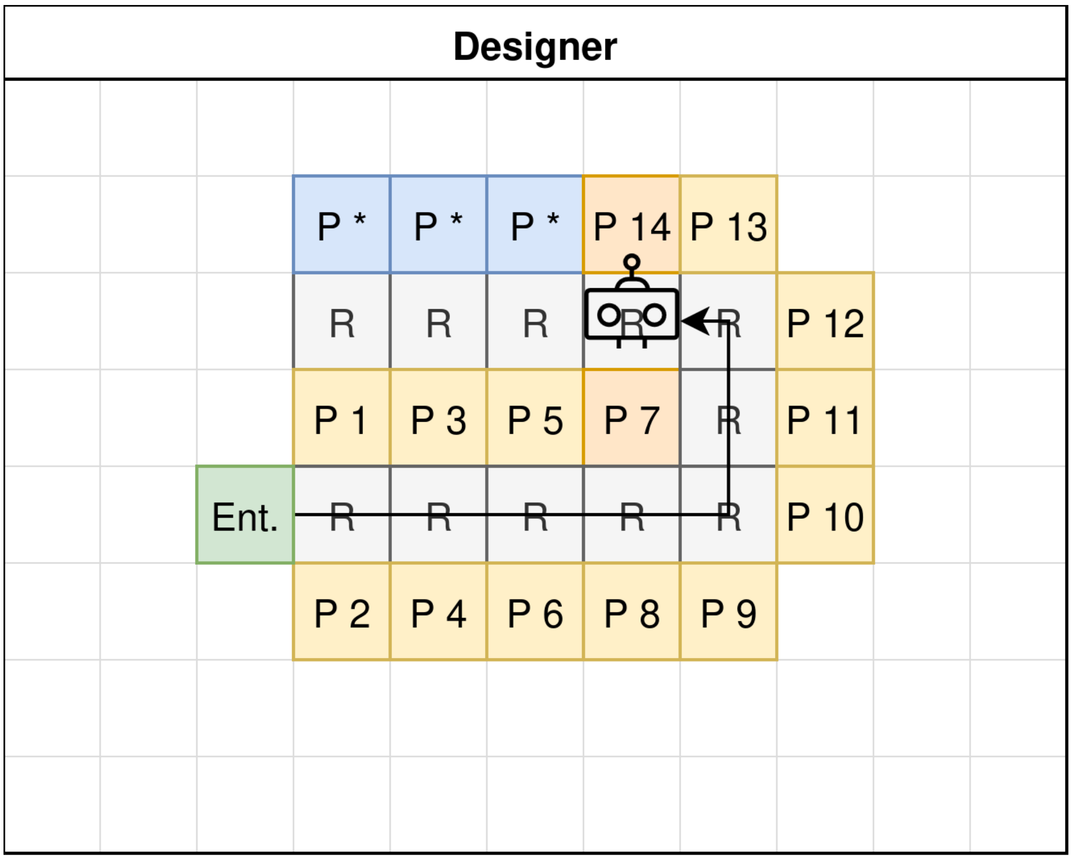


Figure 7.4: Example of p-structure's p-spaces prioritized by a pathfinding algorithm

This approach to prioritization could work, but was ultimately turned down in favor of the simpler approach of counting the road tiles from a given p-space. This choice was made not necessarily due to technical difficulty, but more so a prioritization of time. Counting the tiles would achieve the same result, while being easier to implement. A notable case for the simpler approach is when a road *diverges*. Several p-structures have diverging roads within, which serve to create multiple rows of p-spaces. Simply counting the tiles from a p-space to the entrance would work just fine and would not require much alteration of the base code in order to account for this. However, achieving this with a pathfinding algorithm would require splitting this algorithm into multiple instances when it reaches a diverging road, and have those multiple instances communicate to each other. This would greatly increase development complexity with no significant benefit. Therefore this was a conscious choice to not pursue this proposal.

7.2 Future work/implementations

If more time had been allocated to this P1 project, such as the time given to a full semester project, there are things that would have been implemented into the project or improved upon. General refinements to both the visual elements and the code are obvious things that could be improved but it is more important to look at functions that could be implemented into the product to give it applications to adapt more different p-structures.

7.2.1 P-aisle design and direction

One of these things is the ability to decide whether the p-aisles, i.e. the roads inside the p-structure, are one-way or two-way. In the current finalized version of the project, only one-way roads are fully implemented. Two-way roads technically exist, but are non-functional and don't have an effect on the allocation of parking spots.

7.2.2 Improving UX in designing parking lots

Improvements to the User Interface and the general user experience when designing the parking lots in the program would be a priority. Some quality of life things, like being able to drag over grids to create p-spots, instead of having to click for every individual one, would be one of the many things that would have been implemented, if there had been more time in the project.

7.2.3 Emptying the parking lot

The current Parking Lot Manager can fill an empty parking lot and prioritize the different spots, but it cannot empty it. It can check if cars are parked and can set the spots to vacant, but it cannot empty the spots and simulate a full parking experience. Future work on the project would implement this and show how long individual cars have parked.

7.2.4 Other vehicles

The most common vehicle is the standard person vehicle, but there are other vehicle types that exist, which could be implemented into the parking lot manager. Vehicle types like motorcycles, bikes, vans and, small compact trucks were already considered, but decided less important to the overall problem that the parking lot manager was going to fix. With more work and fine-tuning, however, this could have been implemented.

7.2.5 P-spot measuring and prioritizing according to car sizes

Another feature that was discussed was the ability to assign a set size for p-spots and the ability to prioritize the parking spots according to this. It would have worked in the sense that there would be different sizes for different parking spots. This would have worked well with the implementation of different vehicle sizes and the idea that handicapped and EV p-spots should be wider than normal p-spots.

7.2.6 P-structures choosing how the allocation algorithm prioritizes p-spaces

A thing the group has talked about is how the allocation algorithm should fill up p-structures, and which way is more efficient than the other. Is it front to back, back to front, or other ways? But with more time, the group wanted to make it so that the owners of the p-structures could choose themselves where in the p-structure the prioritization should take place. This would make the product more customisable for the users, and would help with how different p-structures efficiently fill up their p-spaces. Given that a p-basement under a shopping mall you might want to fill up the parking lot, around where the elevator or stairs entrance is into the mall. This would entail the shoppers are closer to where they will spend their money. There could also be a desire for prioritizations when there are multiple exits and entrances or multiple of one of them.

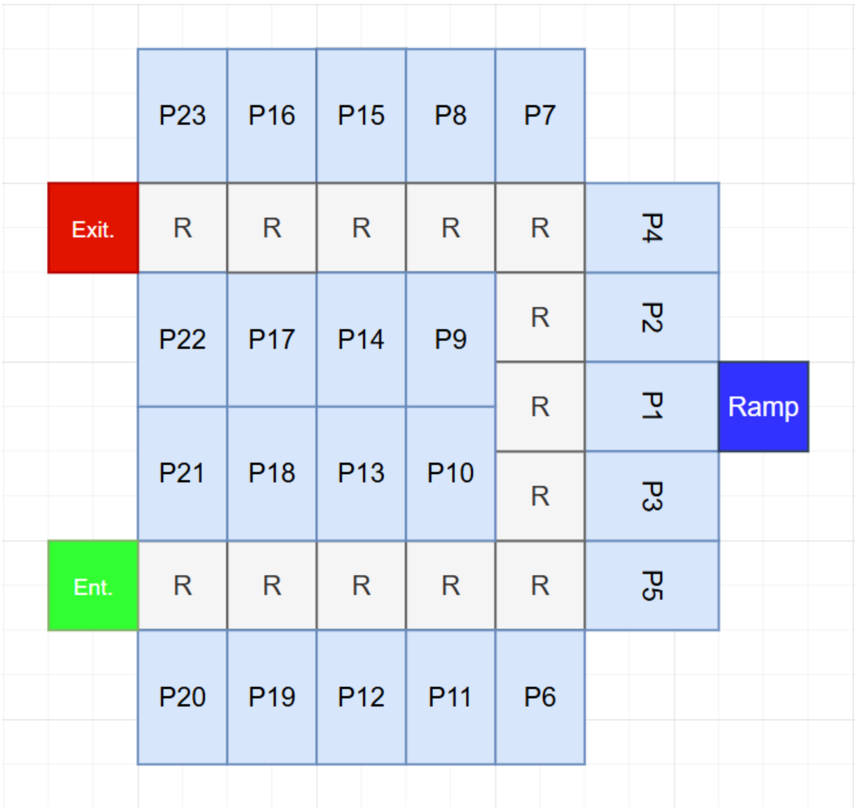


Figure 7.5: Parking lot prioritized for a entrance to a mall

The figure shows the example where the allocation is prioritized around an entrance into a mall instead of being prioritized around the entrance to the p-structure. This is just one of the many examples of what is possible with an allocation algorithm that can be changed to what the user wants.

Conclusion

List of Acronyms

- "P-" means parking. It is used as a prefix in cases such as: p-structure, p-space, p-options and more.
- "Apps", shorthand for mobile applications installed through either Android's *Play Store* or Apple's *App Store*
- "GUI", shorthand for graphical user interface, a style of application with graphical elements.
- "ANPR" is short for Automatic Number Plate Registration.
- "IDE" is short for Internal Development Environment.
- "NFD" is short for Native File Dialog.

References

[1] K. Mccoy, “Drivers spend an average of 17 hours a year searching for parking spots.”

[2] A. B. S. A. T. A. Abdelrahman Osman Elfaki Wassim Messoudi, “A Smart Real-Time Parking Control and Monitoring System.”

[3] B. Khan, “Challenges Associated with Big Car Parks & Their Solutions.”

[4] J. Z. S. B. X. W. D. Y. Y. W. Yifan Wen Shaojun Zhang, “Mapping dynamic road emissions for a megacity by using open-access traffic congestion index data.”

[5] acea, “Economic and Market Report: Global and EU auto industry – Full year 2024.”

[6] C. Walker, “Parking Structure - Design Guidelines.” [Online]. Available: https://ccdcboise.com/wp-content/uploads/2014/11/CCDC-Boise-Parking-Structure-Design-Guidelines_2016-Final-Draft-08-04-2016.pdf

[7] Vejdirektoratet, “Anlæg for parkering og standsning i byer.”

[8] M. Bregenov-Pedersen, “Smalle P-pladser presser bilisterne.”

[9] DOKK1, “Parkering.”

[10] NRGi, “Parkeringsregler for elbiler.” 2024.

[11] L. Industries, “Europe's Largest Automated Parking System Turns 10 – A Decade of Innovation at DOKK1, Aarhus.”

[12] C. Balijepalli, P. Kant, and S. Shepherd, “Calibration and validation of parking search-time function.” 2015.

[13] I. Research, “The Impact of Parking Pain in the US, UK and Germany.”

[14] T. S. A. Y. H. A. M. M. T. A. J. L. A. X. H. A. L. Wu, “How does parking availability interplay with the land use and affect traffic congestion in urban areas? The case study of Xi’an, China.” 2020.

[15] DTU Institut for Transport, “Transportundersøgelsen Faktaark om biltrafik i Danmark.” 2013.

[16] Danmarks Statistik, “It-anvendelse i befolkningen 2022.” 2022.

[17] Kommunernes Landsforening, “Digital Inklusion.” 2025.

[18] centurion, “The Ethics of Technology: What Businesses Need to Consider When Adopting New Technologies.” 2023.

[19] J. C. Villanueva, “The Impact of System Downtime - 5 Consequences | JSCAPE — jscape.com.”

[20] Datatilsynet, “Grundlæggende om GDPR — datatilsynet.dk.”

[21] Q-Park, “Q-Park App.”

[22] APCOA, “APCOA App.”

[23] EasyPark, “EasyPark App.”

[24] Wayleadr, “Customize your parking allocation based on your office’s needs..”

[25] bygningsreglementet, “Vejledning til kommunerne om krav til parkering i forbindelse med byggeri.”

[26] FDM, “Slut med gratis parkering for elbiler i Aarhus.”

[27] DST, “Der er nu over 400.000 elbiler på de danske veje.”

[28] Transportministeriet, “2024 sluttede med stor vækst i ladepunkter.”

[29] L. I. D. K. G. M. J. P. V. Mehmetkr Şükrü Kuran Aline Carneiro Viana, “A Smart Parking Lot Management System for Scheduling the Recharging of Electric Vehicles.”

[30] T. G. Y. L. S. G. Zexin Yang Xueliang Huang, “Real-Time Energy Management Strategy for Parking Lot Considering Maximum Penetration of Electric Vehicles.”

[31] T. T. L. Asaad Mohammad Ramon Zamora, “Transactive Energy Management of PV-Based EV Integrated Parking Lots.”

[32] “Raylib.”

[33] “C (programming language).”

[34] “MoSCoW Prioritization Technique in Product Management.”

[35] “Stakeholder Analysis.”

[36] “What is the Power/Interest Grid?.”

[37] MortenSchou, “mtest.”

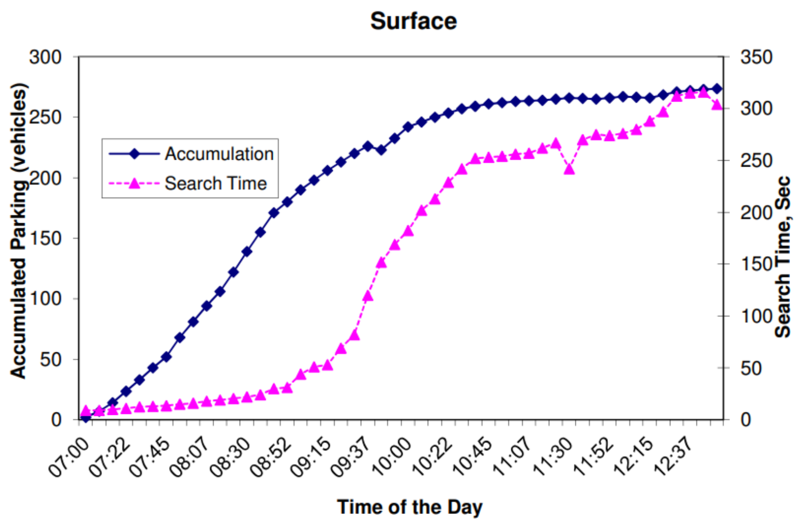


Figure A.1: Surface Car Park – Observed Parking Accumulation and Search Time

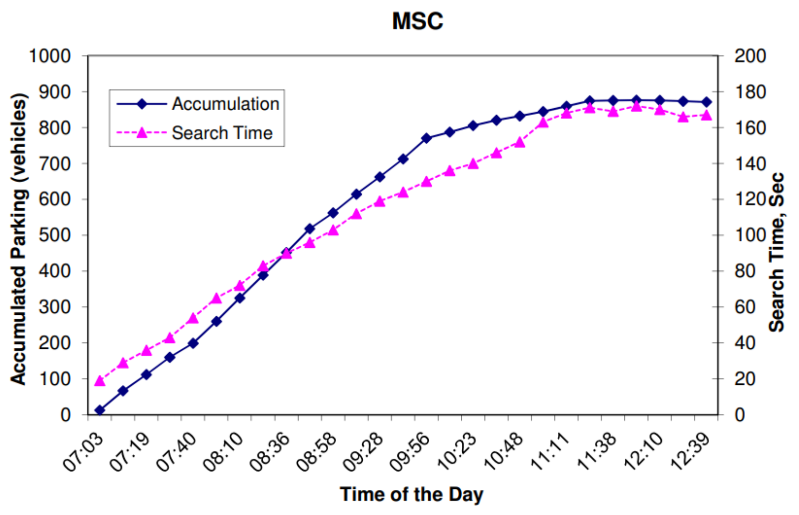


Figure A.2: MSC – Observed Parking Accumulation and Search Time

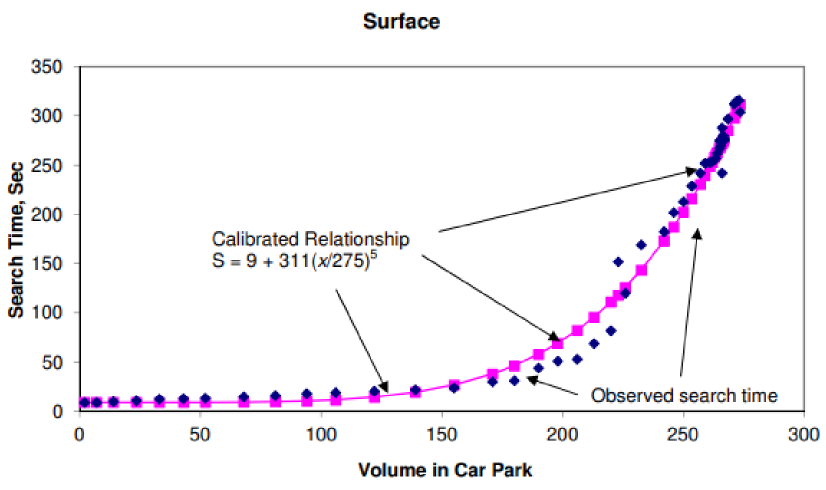


Figure A.3: Surface - Calibration of Search Time at Surface Car Park

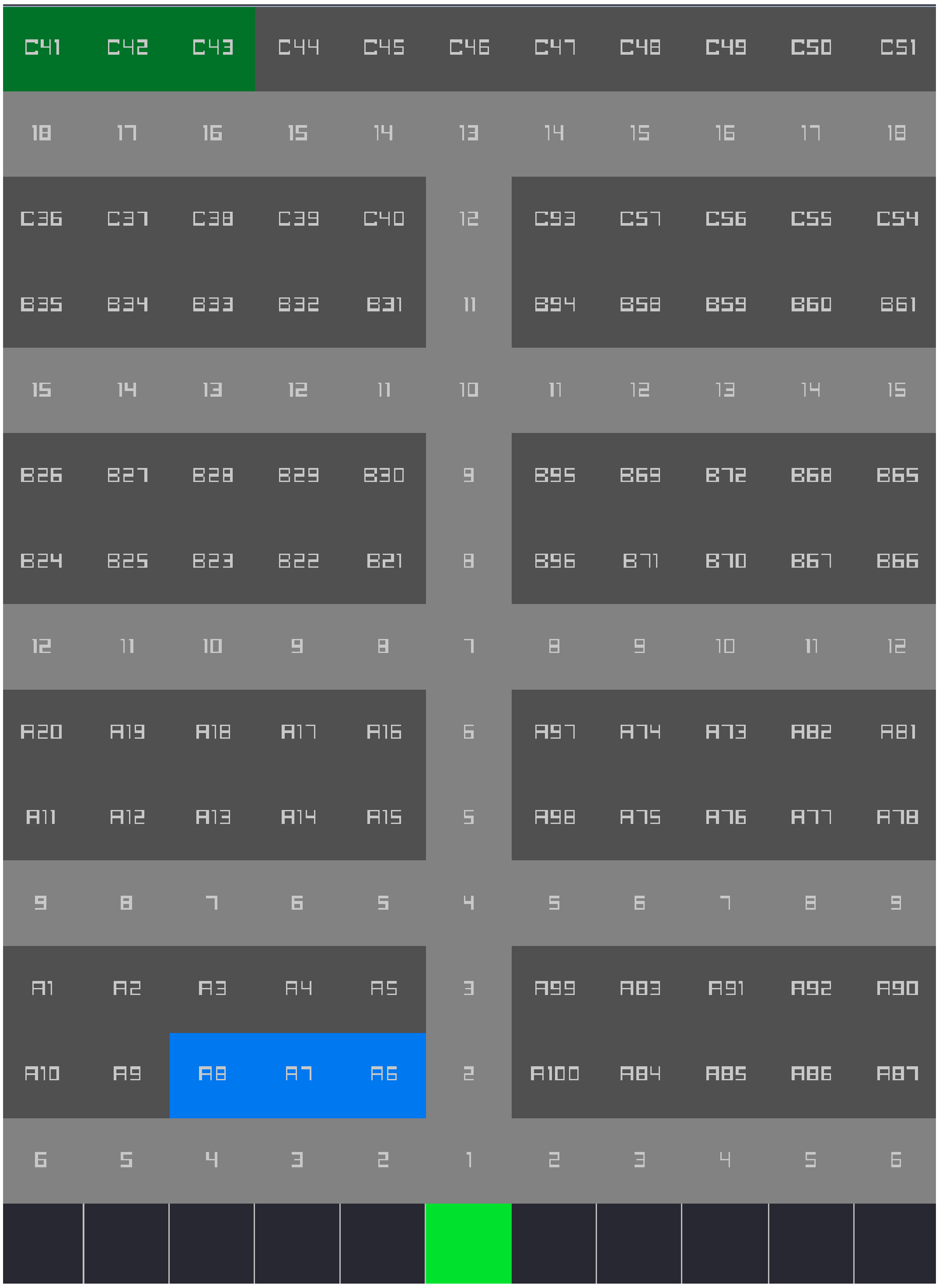


Figure A.4: Parking lot created for the simulation