eCarPark
The Parking Spot Finder

By
G. P. Kegel

Supervisors:

Johan Lukkien (TU/e)
Remi Bosman (TU/e)
John Byrne (IBM)
Pradhyot Chaudhary (IBM)

Tilburg, March 2009
Abstract

Parking in big car parks can be very challenging. Even when an indication is provided that a small percentage of spots is still available in the enormous car parks, most drivers do not seem to be able to locate those spots.

eCarPark is a system that automates this searching. Visitors of car parks no longer have to spend up to 15 minutes to find one of the last spots because the eCarPark system exactly tells them where to find one of these spots.

How does it work? The system measures the number of cars in the different predefined areas in the car park. When a car is waiting at the entrance, the system can give a recommendation to park in a certain area. One of the spots in this area gets allocated, which means the spot is not yet occupied but also not vacant anymore. The visitor receives directions how to get to the allocated spot. An allocated spot can not be allocated again until the system determines, after some time, that a car did not use a recommendation.

How does the system know which area to recommend and allocate? A number of allocation strategies are proposed to take care of this. The various algorithms are inspired by algorithms used for process scheduling, memory management and caching algorithms. These allocation strategies are, as well as the problem itself, formalized in a model.

When a recommended spot turns out to be occupied when a car arrives, various actuators in the car park will indicate where the last vacant spots could be found.

The document also describes the architecture and design of a system that can simulate a car park so the different allocation strategies can be tested and visualized. Another small program can calculate the cost of the various strategies so they can be compared under different circumstances.

Keywords: parking, RFID, shortest path, location determination, scheduling, caching algorithms, memory management, client-server, AJAX, allocation strategies
Preface

This document describes the process as well as the results of my master thesis, which is the final part of my study at the System Architecture and Networking Department at the Eindhoven University of Technology (TU/e). The work described in this thesis was carried out at IBM Nederland B.V. established in Amsterdam.

I would like to thank a few people for their contribution to my thesis. I would like to thank John Byrne, Pradhyot Kumar Chaudhary, Michiel Koehorst and all the other members of Global Business Services who helped me, for their guidance inside IBM. I would like to thank Peter de Jong, working at Nedap, for some insight in the parking world, as well as Johan Lukkien and Remi Bosman from the TU/e for their refined comments and tips. I would also like to thank Angelo van der Sijpt, David Rokven and Eveline Lindenaar for minor translations and motivation.

Last but not least, I would like to thank my family and friends for their support during my studies.

Amsterdam, May 2007
Gerben Kegel
Table of Contents

1. Introduction ............................................................................................................................... 9
   1.1. Background .......................................................................................................................... 9
   1.2. Previous Work ...................................................................................................................... 9
   1.3. Scope .................................................................................................................................... 9
   1.4. Tooling .................................................................................................................................. 10
2. Problem Description .................................................................................................................... 11
   2.1. Problem Statement ................................................................................................................ 11
   2.2. Approach .............................................................................................................................. 11
   2.3. Questions .............................................................................................................................. 11
       2.3.1 Main Question ................................................................................................................ 11
       2.3.2 Underlying Research Statement .................................................................................... 11
       2.3.3 Subquestions .................................................................................................................. 11
   2.4. Assumptions ........................................................................................................................ 12
   2.5. Use cases ............................................................................................................................. 13
3. Earlier Research .......................................................................................................................... 16
   3.1. History .................................................................................................................................. 16
   3.2. iSpot ..................................................................................................................................... 16
   3.3. IrisNET .................................................................................................................................. 17
   3.4. EzPARK .................................................................................................................................. 17
   3.5. Other Systems ......................................................................................................................... 18
   3.6. Comparison ............................................................................................................................ 19
   3.7. eCarPark ............................................................................................................................... 19
4. Possible Systems .......................................................................................................................... 20
   4.1. Input ...................................................................................................................................... 20
       4.1.1 By Spot ............................................................................................................................ 20
       4.1.2 By Group of Spots .......................................................................................................... 20
       4.1.3 By Area ............................................................................................................................ 21
       4.1.4 By Car Park ...................................................................................................................... 23
       4.1.5 Summary ........................................................................................................................ 23
   4.2. Output .................................................................................................................................... 23
       4.2.1 Recommendation/hint .................................................................................................... 23
       4.2.2 Navigation System ......................................................................................................... 24
       4.2.3 Cellphone ......................................................................................................................... 24
       4.2.4 In Car Park Signs .......................................................................................................... 24
   4.3. Decentralized Solution ........................................................................................................... 25
   4.4. The directed graph ................................................................................................................ 26
   4.5. Storage .................................................................................................................................. 26
5. Allocation strategies ....................................................................................................................... 29
   5.1. Formalization ........................................................................................................................ 29
   5.2. Inspiration ............................................................................................................................. 32
   5.3. Algorithms ............................................................................................................................. 33
       5.3.1 The Common Path .......................................................................................................... 33
       5.3.2 Close to the Passenger Exit ............................................................................................ 34
       5.3.3 Close to the Car Entrance ............................................................................................... 35
       5.3.4 Random ............................................................................................................................ 35
       5.3.5 Fairly Distributed ............................................................................................................. 35
       5.3.6 Last Freed Spot .............................................................................................................. 36


<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.7 Most Recently Allocated</td>
<td>36</td>
</tr>
<tr>
<td>5.3.8 Most Frequently Allocated</td>
<td>37</td>
</tr>
<tr>
<td>5.4. Area-ranking</td>
<td>37</td>
</tr>
<tr>
<td>5.5. Wrap up</td>
<td>37</td>
</tr>
<tr>
<td>6. Architecture</td>
<td>39</td>
</tr>
<tr>
<td>6.1. System Requirements</td>
<td>39</td>
</tr>
<tr>
<td>6.2. Spots, Areas and States</td>
<td>40</td>
</tr>
<tr>
<td>6.3. System Overview</td>
<td>41</td>
</tr>
<tr>
<td>6.4. Components</td>
<td>42</td>
</tr>
<tr>
<td>6.5. Communication</td>
<td>44</td>
</tr>
<tr>
<td>7. Detailed Simulator Design</td>
<td>48</td>
</tr>
<tr>
<td>7.1. Iteration 1: the Basic System</td>
<td>48</td>
</tr>
<tr>
<td>7.1.1 Car Park</td>
<td>48</td>
</tr>
<tr>
<td>7.1.2 User / Car</td>
<td>49</td>
</tr>
<tr>
<td>7.1.3 Input and Output</td>
<td>49</td>
</tr>
<tr>
<td>7.1.4 Detect</td>
<td>49</td>
</tr>
<tr>
<td>7.1.5 Spot Selector</td>
<td>49</td>
</tr>
<tr>
<td>7.2. Iteration 2: Picking the Best Spot</td>
<td>49</td>
</tr>
<tr>
<td>7.2.1 Storage</td>
<td>50</td>
</tr>
<tr>
<td>7.2.2 Timer</td>
<td>51</td>
</tr>
<tr>
<td>7.2.3 Allocate</td>
<td>51</td>
</tr>
<tr>
<td>7.2.4 Check</td>
<td>51</td>
</tr>
<tr>
<td>7.2.5 Full</td>
<td>51</td>
</tr>
<tr>
<td>8. Simulation</td>
<td>52</td>
</tr>
<tr>
<td>9. Allocation Strategy Comparison</td>
<td>54</td>
</tr>
<tr>
<td>9.1. Visitor sequences</td>
<td>54</td>
</tr>
<tr>
<td>9.2. Car Parks</td>
<td>54</td>
</tr>
<tr>
<td>9.3. Goal</td>
<td>56</td>
</tr>
<tr>
<td>9.4. Test Results</td>
<td>56</td>
</tr>
<tr>
<td>9.5. Explanation of Results</td>
<td>61</td>
</tr>
<tr>
<td>9.6. Choose an algorithm</td>
<td>61</td>
</tr>
<tr>
<td>10. Conclusion</td>
<td>62</td>
</tr>
<tr>
<td>11. Summary</td>
<td>64</td>
</tr>
<tr>
<td>12. Samenvatting</td>
<td>65</td>
</tr>
<tr>
<td>13. Glossary</td>
<td>66</td>
</tr>
<tr>
<td>14. Appendices</td>
<td>68</td>
</tr>
<tr>
<td>14.1. Parking Behavior</td>
<td>68</td>
</tr>
<tr>
<td>14.2. Inspiration</td>
<td>69</td>
</tr>
<tr>
<td>14.2.1 Scheduling Processes</td>
<td>69</td>
</tr>
<tr>
<td>14.2.2 Memory Allocation</td>
<td>70</td>
</tr>
<tr>
<td>14.2.3 Cache Versus RAM</td>
<td>71</td>
</tr>
<tr>
<td>14.3. Reply SpotScout, Inc.</td>
<td>73</td>
</tr>
<tr>
<td>14.4. Report ParkeerVak 2007</td>
<td>74</td>
</tr>
<tr>
<td>14.5. CPML-file</td>
<td>75</td>
</tr>
<tr>
<td>14.6. Allocation Algorithms</td>
<td>81</td>
</tr>
<tr>
<td>14.6.1 ClosestVertex Algorithm</td>
<td>81</td>
</tr>
<tr>
<td>14.6.2 Pure Random</td>
<td>82</td>
</tr>
<tr>
<td>14.6.3 Fairly Distributed</td>
<td>82</td>
</tr>
<tr>
<td>14.6.4 Last Freed Spot</td>
<td>82</td>
</tr>
</tbody>
</table>
14.6.5 Most Recent Allocated...........................................................................................................83
14.6.6 Most Frequent Allocated.........................................................................................................83
## Index of Illustrations

<table>
<thead>
<tr>
<th>Illustration</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Use case diagram</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>A car park which is divided in areas</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Architecture solution for a 2 spot car park</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Mapping to directed graph</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>Example of CPML output</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>Parking density where exit is on the bottom right</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>Common path</td>
<td>34</td>
</tr>
<tr>
<td>8</td>
<td>Spot states</td>
<td>40</td>
</tr>
<tr>
<td>9</td>
<td>One of the spots is located</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>Basic architecture</td>
<td>41</td>
</tr>
<tr>
<td>11</td>
<td>Detailed architecture</td>
<td>42</td>
</tr>
<tr>
<td>12</td>
<td>A representation of the sensors and actuators in the car park</td>
<td>43</td>
</tr>
<tr>
<td>13</td>
<td>Interfaces</td>
<td>44</td>
</tr>
<tr>
<td>14</td>
<td>A car drives to another area</td>
<td>46</td>
</tr>
<tr>
<td>15</td>
<td>A user asks for a recommendation at the entrance</td>
<td>46</td>
</tr>
<tr>
<td>16</td>
<td>The timer expires</td>
<td>47</td>
</tr>
<tr>
<td>17</td>
<td>Hardware simulation GUI</td>
<td>48</td>
</tr>
<tr>
<td>18</td>
<td>Graphical representation of internal data structure</td>
<td>50</td>
</tr>
<tr>
<td>19</td>
<td>Simulation system overview</td>
<td>52</td>
</tr>
<tr>
<td>20</td>
<td>Asking for a recommendation</td>
<td>53</td>
</tr>
<tr>
<td>21</td>
<td>Screen shot of the simulation user interface</td>
<td>53</td>
</tr>
<tr>
<td>22</td>
<td>Car park 2 and 3</td>
<td>55</td>
</tr>
<tr>
<td>23</td>
<td>Example of malloc(96)</td>
<td>70</td>
</tr>
<tr>
<td>24</td>
<td>Example of free()</td>
<td>71</td>
</tr>
</tbody>
</table>
Index of Tables

Table 1: Product Overview.................................................................................................................. 10
Table 2: Used software........................................................................................................................ 10
Table 3: Use case 1: Guidance............................................................................................................. 14
Table 4: Use case 2: Leaving the car park............................................................................................ 14
Table 5: Use case 3: Calibrating the car park.................................................................................... 15
Table 6: Use case 4: Showing statistics............................................................................................... 15
Table 7: Earlier research comparison.................................................................................................. 19
Table 8: Input comparison.................................................................................................................... 23
Table 9: Communication data types.................................................................................................... 44
Table 10: Communication between components.................................................................................. 45
Table 11: Visitor sequences................................................................................................................ 54
Table 12: Cost of allocation for Voffice1 in the example car park (* average of 3 tests)................. 56
Table 13: Cost of allocation for Voffice2 in the example car park (* average of 3 tests)............... 57
Table 14: Cost of allocation for Voffice1 in car park 2 (* average of 3 tests)................................. 57
Table 15: Cost of allocation for Voffice1 in car park 3 (* average of 3 tests)................................. 57
Table 16: Cost of allocation for Vshop in example car park (* average of 3 tests)........................... 58
Table 17: Cost of allocation for Vshop in car park 2 (* average of 3 tests)....................................... 58
Table 18: Cost of allocation for Vexit in example car park (* average of 3 tests)............................ 59
Table 19: Crowdedness estimation.................................................................................................... 68
1. **Introduction**

1.1. **Background**

In crowded countries like the Netherlands, more and more people travel by car and it gets harder and harder to find a vacant parking spot. To solve this problem, the government and several companies are building big car parks near crowded places. This only partly solves the problem, because even within these car parks, it can be a real challenge to find a vacant spot in a partly filled car park. Drivers can drive around for many minutes before finding one. This problem was already recognized in Japan in the seventies, both in car parks and urban parking areas.

Nowadays, the problem also exists in many car parks in the Netherlands, including the one located next to the IBM Nederland B.V. headquarters. When people arrive at peak usage hours, they often seem to spend up to ten minutes to find one of the last vacant spots.

This almost pointless driving has many disadvantages. When people drive around, they can not work. In this situation, the longer it takes to find a vacant spot, the more annoyed someone could get. Since gas prices seem to sky rocket lately, it also has a financial disadvantage for the driver. Last but not least, this driving around is an unnecessary impact to the environment, exhaust can get smelly and the cars produce unnecessary noise.

1.2. **Previous Work**

This is not the first project about finding a vacant parking spot. Some of the earlier projects are listed in chapter 3. None of these projects seem to give a complete solution though, especially not to the question which spot or area to assign to a car. In contrast to the corresponding websites which tell that these projects evolved in actual products, these products seem to be hard to find. One company though, Siemens, is building a car park with over nine thousand parking spots near the airport of Munich which provides information to find vacant spots.

1.3. **Scope**

This document describes a possible solution to reduce the time for finding a vacant parking spot and the process of getting there. The main focus of the document is the process of allocating a certain spot or area and assign this spot to a car. A formal model of the problem will be constructed and different allocation strategies will be proposed. The document also describes an architecture and design of a system where these different allocation strategies can be simulated and compared. Because the project ended with a simulation of the system, a proof of concept, the document also ends there. A glossary can be found on page 66.

The document will not describe problems regarding to urban parking or human decision making whether to follow recommendations or not. The consequences of those decisions, will be in the scope of the document though. This document is also not about creating new parking spots. Once a car park is fully loaded, the parking problem remains. There are other solutions to solve that problem, but they are outside the scope of this document.
The following table gives a brief overview of the system that will be described in this document.

<table>
<thead>
<tr>
<th>For</th>
<th>People who need to park their cars in crowded car parks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Who</td>
<td>Don't want to spend much time looking for a vacant spot</td>
</tr>
<tr>
<td>The</td>
<td><strong>eCarPark</strong> is a car park navigation infrastructure</td>
</tr>
<tr>
<td>That</td>
<td>Guides a visiting car driver to a vacant parking spot</td>
</tr>
<tr>
<td>Unlike</td>
<td>GPS navigation systems or conventional car parks</td>
</tr>
<tr>
<td>Our product</td>
<td>Helps you find a vacant parking spot (if it exists).</td>
</tr>
</tbody>
</table>

*Table 1: Product Overview*

The features of this product can be summarized as: minimize the total time a visitor spends in the car park. The following aspects influence this:

- the time (and so the distance) needed to get from the car entrance to a spot;
- the time needed to get out of the car;
- the time needed to walk from the car to the passenger entrance/exit;
- the time needed to walk from the passenger entrance/exit back to the car;
- the time needed to get in the car and
- the time needed to drive from a spot back to the car exit.

All these variables can be determined as a function of the distances, which is again a function of the location of the spot and the car park. The information above will be used to compare different spot allocation algorithms.

### 1.4. Tooling

The following table describes the tools used for this project.

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenOffice.org</td>
<td>2.2.0</td>
<td><a href="http://www.openoffice.org/">http://www.openoffice.org/</a></td>
</tr>
<tr>
<td>Ipe</td>
<td>6.0pre27</td>
<td><a href="http://tclab.kaist.ac.kr/ipe/">http://tclab.kaist.ac.kr/ipe/</a></td>
</tr>
<tr>
<td>Google Spreadsheets</td>
<td>online</td>
<td><a href="http://docs.google.com/">http://docs.google.com/</a></td>
</tr>
<tr>
<td>Python</td>
<td>2.4.4</td>
<td><a href="http://www.python.org/">http://www.python.org/</a></td>
</tr>
<tr>
<td>UltraEdit-32</td>
<td>11.20b</td>
<td><a href="http://www.ultraedit.com/">http://www.ultraedit.com/</a></td>
</tr>
</tbody>
</table>

*Table 2: Used software*
2. Problem Description

This chapter will state the problem which was the first input for this project. From this problem, a question and a number of subquestions can be derived. Later in this chapter, assumptions and use cases will be defined, which may be referred to later in the document.

2.1. Problem Statement

More and more cars populate our crowded world and it gets harder and harder to find a vacant parking spot in a car park. At the entrance of most modern car parks, as well as on several digital displays around the city, the number of vacant spots in a car park can be read. When a car enters a car park during peak usage hours however, it can take ages before the driver finds one of those vacant spots because the driver has no idea where those vacant spots are located within the car park. If the driver would know the location of such a vacant spot, because it was recommended to him when he entered, it would have taken a lot less time to get to that spot because there is no more searching involved. Second of all, the location of this recommend spot also influences the time that is spent in the car park. So the better the recommended place, the shorter the time the visitor has to be in the car park.

2.2. Approach

The goal of the project is to design a system that guides a visitor to a vacant parking spot. The focus will be on algorithms that determine which vacant spot to recommend. We will first dive into the literature to see whether there are any initiatives like this. People in the parking business, mainly on ParkeerVak 2007 (more info in appendix 14.4), will be interviewed for more information. Different input and output signals which could be used will be investigated. A good combination of in- and outputs should be chosen because it can highly influence the architecture. A model will be created to formalize the problem and clarify the different allocation strategies. An architecture will be designed to use the inputs and calculate proper outputs. Some extra thought will be put into determining which vacant spot to allocate for a new visitor, such that the total time that is spent in the car park can be minimized. Different allocation algorithms will be proposed and compared using a cost function from our model.

2.3. Questions

There are several big and small question to ask ourselves. This section lists those questions.

2.3.1 Main Question

“How to minimize the time visitors spend in a car park?”

2.3.2 Underlying Research Statement

“How could a system and the underlying architecture, that notices a car, allocates a spot, and guides the car to that spot look, and which allocation strategy would have to be used to minimize the total time the visitor spends in the car park?”

2.3.3 Subquestions

[Q1] How can a car be noticed when it tries to enter?
[Q2] How to determine whether a parking spot is empty or not?

[Q3] How to measure the number of empty spots?
[Q4] How to decide which vacant spot to pick for a car?
[Q5] How to react when a driver neglects the guidance that was recommended?
[Q6] When can a spot be said to be empty again?
[Q7] How can people's preferences be taken into account? Think about reservations.
[Q8] How can a car be guided to a spot?

[Q9] How to model the behavior of a car?
[Q10] How to model the entire system?
[Q11] How could a system like this be demonstrated?

More questions might come up during the process of answering the ones above. Some questions are easy to answer, others might lead to big design decisions.

2.4. Assumptions

The next thing to do is setting the assumptions and constraints of the environment the system will live in.

[ASS1] The total number of spots in the car park is clearly defined
[ASS2] Every vehicle can be noticed by the system when it tries to enter the car park.
[ASS3] A driver is able to choose whether to follow the instructions that are provided or not. He or she is still able to choose any spot (s)he likes.
[ASS4] Every spot can only hold one car.
[ASS5] Each car will only occupy one spot.
[ASS6] Cars enter the car park one by one.
[ASS7] No traffic jams occur in the car park so congestion control is not an issue.
[ASS8] If a car is waiting at the entrance of a full car park, it keeps waiting until a car leaves the car park.
[ASS9] Motorcycles will be handled like cars would.
### 2.5. Use cases

Here you can find the use cases which are used for this system. Keep in mind that only a simulation will be created.

There exist only two actors in this system: the Administrator and the Driver. The Administrator will be the guy who installs and maintains the system. The Driver is, obviously, one of the car park visitors who is looking for a vacant spot. The following diagram shows all the use cases.

![Use case diagram](Illustration 1: Use case diagram)

<table>
<thead>
<tr>
<th>Name</th>
<th>Use case 1: Guidance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>A visitor enters the car park and receives a recommended spot. He drives to the spot and parks there.</td>
</tr>
<tr>
<td>Actors</td>
<td>Driver</td>
</tr>
<tr>
<td>Assumptions</td>
<td>The car park is fully calibrated.</td>
</tr>
<tr>
<td>Description</td>
<td>10 The driver tries to enter the car park.</td>
</tr>
<tr>
<td></td>
<td>20 The system determines which spot to assigns to the car, using a chosen allocation algorithm, and recommends this to the driver.</td>
</tr>
<tr>
<td></td>
<td>30 The driver receives information about where to park and how to get there.</td>
</tr>
<tr>
<td></td>
<td>40 The driver drives to the recommended location.</td>
</tr>
<tr>
<td></td>
<td>60 The system determines whether the number of cars in the area in question exceeds the limit and indicates this if necessary.</td>
</tr>
<tr>
<td></td>
<td>70 The driver parks his car on the spot.</td>
</tr>
<tr>
<td>Exceptions</td>
<td>In the meantime, a driver B neglected his recommendation and parked on the spot that was allocated for car A.</td>
</tr>
<tr>
<td></td>
<td>Driver A can now still see where the other vacant spots are located, using indicators in the car park.</td>
</tr>
<tr>
<td>Result</td>
<td>The driver is guided to the vacant spot.</td>
</tr>
</tbody>
</table>
Table 3: Use case 1: Guidance

<table>
<thead>
<tr>
<th>Name</th>
<th>Use case 2: Leaving</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>A car leaves and a spot becomes vacant again.</td>
</tr>
<tr>
<td>Actors</td>
<td>Driver</td>
</tr>
<tr>
<td>Assumptions</td>
<td>The car park is fully calibrated.</td>
</tr>
<tr>
<td></td>
<td>The driver parked his car somewhere in the car park.</td>
</tr>
<tr>
<td>Description</td>
<td>10 A driver parked his car on a spot in the parking car park and wants to leave.</td>
</tr>
<tr>
<td></td>
<td>20 He gets in his car, and leaves his spot.</td>
</tr>
<tr>
<td></td>
<td>30 He leaves the area he was parked in.</td>
</tr>
<tr>
<td></td>
<td>50 The system marks the spot as vacant as soon as it registers the car leaves the area.</td>
</tr>
<tr>
<td></td>
<td>60 The driver drives to the exit and leaves the car park.</td>
</tr>
<tr>
<td>Exceptions</td>
<td>The driver parks somewhere in another area.</td>
</tr>
<tr>
<td></td>
<td>When the car entered another area, where there was still one spot vacant, that spot state became occupied when the car entered, and would turn vacant again when the car leaves the area in question.</td>
</tr>
<tr>
<td></td>
<td>If the car would park in a spot that looked vacant to him, but was in fact allocated for another car, see the exception at the previous use case.</td>
</tr>
<tr>
<td>Result</td>
<td>The driver left and the spot is vacant.</td>
</tr>
</tbody>
</table>

Table 4: Use case 2: Leaving the car park
The following use cases are optional if there will be any time left.

<table>
<thead>
<tr>
<th>Name</th>
<th>Use case 3: Calibrating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>A new car park can be defined.</td>
</tr>
<tr>
<td>Actors</td>
<td>Administrator</td>
</tr>
<tr>
<td>Assumptions</td>
<td>All sensors and other hardware infrastructure are in place.</td>
</tr>
<tr>
<td>Description</td>
<td>Most car parks have different layouts. This is the reason the system should get calibrated to fit the car park.</td>
</tr>
<tr>
<td></td>
<td>10 The system administrator configures the car park.</td>
</tr>
<tr>
<td></td>
<td>30 The system boots and initializes the configuration file.</td>
</tr>
<tr>
<td>Exceptions</td>
<td>An error raises if the configuration could not exist.</td>
</tr>
</tbody>
</table>
The system is now calibrated. The system knows what to conclude out of the information received from the sensors.

**Table 5: Use case 3: Calibrating the car park**

<table>
<thead>
<tr>
<th>Name</th>
<th>Use case 4: Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>Statistical data of the car park can be viewed to the administrator.</td>
</tr>
<tr>
<td>Actors</td>
<td>Administrator</td>
</tr>
<tr>
<td>Assumptions</td>
<td>The car park is fully calibrated.</td>
</tr>
</tbody>
</table>
| Description   | On any moment, the Administrator can see statistical information about the car park. The following information will be available:  
  ● number of cars in the car park;  
  ● filled percentage in car park;  
  ● number of cars per area and  
  ● filled percentage per area. |
| Exceptions    | None                   |
| Result        | Statistical data is shown |

**Table 6: Use case 4: Showing statistics**
3. Earlier Research

Before developing the system any further, let's have a look at the rest of the world for some inspiration and to see what's already been done and what's not. This project is not the first looking at the parking problem from this point of view. Though the problem only arose a couple of years ago in the Netherlands, more crowded areas, like the big cities in Japan, are facing the problem for decades already. In the following chapter we will have a look at several systems and what kind of inputs, outputs and allocation strategies they use.

3.1. History

Apparently, this problem is not very new. Siemens already wrote about it in the early 90's and they were already thinking about solutions back than. Concrete results where not found at that time though.

Toyota (car manufacturer from Japan) started developing Parking Guidance and Information (PGI) systems in 1995. It looks like there isn't much guidance though. The users seem to guide themselves, after receiving information about parking intensity around the city. Initiatives like this also arose in the Netherlands but they only indicate an approximation of the number of vacant spots per car park.

3.2. iSpot

One of the initiatives out there is iSpot, also referred to as iPark [ISPOT]. The creators, four students from Boston University, call it a vision based awareness system. The project identified the exact same problem and tried to find a solution for it.

Though the system uses digital cameras to detect whether spots are vacant or not, it does offer single space monitoring [Q2] as well as parking spot reservation [Q7]. Parking spot reservation can be compared with the user preference for certain spots. The information about which spot is vacant and which one isn't, is communicated to the user at the entrance using an image of the car park [Q8], so the user can choose one of the vacant spots and drive there.

Though their system has quite some overlap with eCarPark, there are some problems which are not addressed. First of all, the system uses digital cameras in combination with license plate extraction to detect whether certain spots are occupied, and to keep track of the cars. Though this might sound promising on paper, an initiative like this could get quite some resistance from organizations which are fighting for privacy.

Another problem which is not addressed is the chaos of the crowd. Everybody sees which spots are vacant, but the system does not assign specific cars to specific spots. This could lead to a situation where car A sees a vacant spot on the screen at the entrance, and decides to go there. Car B sees the same vacant spot, because car A did not yet reach that spot, and also decides to choose that one. When car B arrives at the spot, he sees that car A already parked there. Car B is now lost in the big car park and the only way to find himself a vacant spot it to search for it himself in the old-fashioned way. This could not only happen for two cars, but as long as the first car did not reach the spot. All but the first of those cars will end up unsatisfied. If this keeps happening, the original problem isn't really solved at all. Cars drive to spots which turn out to be occupied when they arrive.

Investors are said to be enthusiastic about iSpot and expect it to hit the market within the next 2 or 3 years. Though iSpot as it exists today, might not yet be the problem solving technology, they sure
are a couple of steps ahead.

### 3.3. IrisNET

Another interesting project is called IrisNET from Intel [IRIS]. The project doesn't have much to do with our problem but defines a platform that connects cheap sensor-nodes, to a large network. To speed up the acceptance of this platform, Intel created a couple of demo implementations. One of those demo implementations is called the Parking spot Finder. Webcam information from a large network of webcams, can be used for a higher goal, if combined. Feeds from webcams all over a car park, or as Intel states it, all over the city, can be combined to gather information about vacancy of parking spots.

What does the demonstration look like? Cheap webcams are pointed to parking spots, or small groups of parking spots. Computers near these webcams can process the information, and determine whether certain parking spots are vacant or not [Q2]. This information can be fed to the network. Central computers gather all this information via IrisNET and combine it to make a list or vacant parking spots. This list can then be offered to other services, like Yahoo maps, for example [Q8]. People could see the vacant spots on their car navigation system using existing technology. It's just a matter of connecting the services via IrisNET.

It seems like a complete system, even for a demo implementation. It is not clear whether the problem of guiding multiple cars to the same spot or how to keep a reservation, was solved. Knowing the fact that this is just a demonstration of IrisNET, could mean that the system does not provide these extras. We can conclude that this Parking spot Finder is a great source of inspiration, but not a solution for the parking problem as stated in the previous chapter.

### 3.4. EzPARK

EzPARK is the name of a company as well as the product they are developing [EZP]. The mission statement of EzPARK is the following: "EzPARK is a low-cost, wireless parking lot infrastructure that enables the customers to see the empty spaces at the entrance, and leads them to their vehicles on their way back." This system seems to satisfy quite some of our requirements. EzPARK does what iSpot does, and on top of that, also leads customers back to their vehicles. Apart from that, the system also gives a hint to the visitor, where to park the car, which is a form of guiding him [Q8]. So if all the cars park where they are told to park, the system would work like the one we want. Unfortunately, that is a significant assumption. Apart from this, it is unclear how the system decides which spot to hint.

How does EzPARK solve the problem? When the user enters the car park, he receives 2 RFID tokens and a hint where to park. One of the RFID tokens would be left in the car, to identify the car, and the other one would identify the user itself. The system can associate the two when the user returns for its car. The RFID tag in his car is also used to register whether a parking spot is occupied or not [Q2]. Combining the availability information of all spots can result in a hint for a new visitor.

The entire car park will have to be filled with RFID readers, which communicate to each other wireless so all the information of the spot states can be centralized. These wireless units, so called MOTES, are prefabbed sensor nodes.

Apart from the fact that there is no actual guidance during the driving through the car park, this seems to be a very good product to solve the original problem. But the problem remains, what if driver D decides to park on the same spot as driver E, driving in front of him. Driver E would have to go all the way back to the entrance to find a new parking spot, or he could solve it the old fashioned way. And why would the system prefer one spot over another when giving a hint?
3.5. Other Systems

The Urban Parking Finder is another project [UPF], done by a small group of students. They tried to find the closest parking spot in an urban setting. Though their report doesn't describe how to measure whether a spot is vacant or not, their simulation works like you would expect. They made a bunch of virtual streets with cars parked all over the place, and some vacant spots. A car could drive around within the streets and whenever the user is interested in a parking spot, he can press a button and the system calculates the closest parking spot, as well as the shortest path to that spot, and gives directions to the user to guide him. This solution only solves a small part of our problem but interestingly enough, a part that was not mentioned in the other initiatives. The students don't describe how exactly they calculate the closest parking spot, but it probably looks very much like the “close to the exit” allocation algorithm proposed in section 5.3.2.

The Parking Space Optimization Service [PSOS] from the university of Zürich was presented a little more than two years ago at the Swiss Transport Research Conference. They describe an e-Parking model where not only the driver and parking lots communicate with the system, but also events and businesses in the neighborhood, payment services etc.

The system also (effectively) takes care of reservations. Cars can identify themselves using Bluetooth when they enter or leave the car park. The payment is made automatically when the car leaves the car park.

Please note that this system might seem brilliant on paper, the future seems to be far from a living implementation of such a system. Every user should have a properly configured Bluetooth device in his car. This is far from the case in the Netherlands. Another identification method could be used to solve this though. Apart from the identification issue, the paper does not mention any active guidance.

The Parking Meter Supervision System [PMSS] describes an urban non-free parking system. This might not be the kind of system we are looking for, but this is a system that was actually tested in a Japanese city and the results are very promising. The system displays vacant parking spot information around the city so drivers can spend less time searching for a free spot. Without the system 14.1 cars parked on a spot each day on average. After introducing the system, this increased to 15.9 cars on a spot on average. These figures show the need for a parking guidance systems, though the problem might be slightly bigger in urban areas.

Evaluation of Parking Search using Sensor Network [EPSSN] tries to solve the problem a little differently. Sensors are placed within the vehicles. These vehicles gather information while they drive. Vacant spots can be located this way. The information (including a location) is wirelessly transmitted to other cars within a certain range. These cars retransmit the information to the cars they "meet", and so on. If one driver is looking for a vacant spot, (s)he will be informed of the existence of that spot by a passing car. All the information expires, so it doesn't leave the surrounding area of the parking spot. Thought this initiative might not directly solve our problem, the approach of the problem is original and worth mentioning. The system was simulated but never built.

IcanPARK is a remote management system for all types of car parks. Each spot is equipped with a little sonar system which determines whether a car is parked on that spot. Signs inform the driver where to find the nearest vacant spot. The entire system can be monitored by an operator, who can check the car park status, make reservations or close certain areas for maintenance. The allocation problem is not addressed in this project.

Close to finishing this project I found out about Sipark [SIPARK]. Sipark is the world's largest Parking Guidance System is in the making at Munich Airport, developed by Siemens. This system
eCarPark: the Parking spot Finder

monitors each individual parking spot using ultrasound sensors and guides cars to vacant spot using this information. The guiding is done by hundreds of LED displays all over the car park. The solution also includes zone and aisle counting. This means that any vehicles still en route to a parking space are also acquired by the system which avoids guiding too many drivers into a sector that only contained a few unoccupied spaces when the car entered the car park.

3.6. Comparison

Let's make a brief overview of the inputs, outputs and allocation strategies in these systems.

<table>
<thead>
<tr>
<th>System</th>
<th>Input</th>
<th>Output to user</th>
<th>Allocation strategy</th>
<th>Guidance</th>
</tr>
</thead>
<tbody>
<tr>
<td>iSpot</td>
<td>Multispot camera</td>
<td>Visualization of location of vacant spots at entrance</td>
<td>By user</td>
<td>None</td>
</tr>
<tr>
<td>IrisNET</td>
<td>Single/multispot webcam</td>
<td>Not implemented</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>EzPARK</td>
<td>Single spot RFID</td>
<td>Hint at entrance</td>
<td>Unknown</td>
<td>Hint at entrance</td>
</tr>
<tr>
<td>Icanpark</td>
<td>Single spot sonar</td>
<td>Signs within car park</td>
<td>Unknown</td>
<td>Unclear</td>
</tr>
<tr>
<td>Sipark</td>
<td>Single spot ultrasound</td>
<td>LED-displays within the car park</td>
<td>By user</td>
<td>LED-displays</td>
</tr>
</tbody>
</table>

Table 7: Earlier research comparison

3.7. eCarPark

So what makes this project any different from the ones above. The main goal of this project will be to find out which spot to assign to a certain car, in a certain parking lot structure. None of the projects above seem to worry about that detail. There are different possible algorithms to do this allocation and we are interested in the one that minimizes the time a visitor spends in the car park.

We will also try to find a way to guide a car to a specific spot, and not just mention the spot to the visitor.

Why did we not continue from one of the projects above? There are two main reasons.

- None of the project responsibles ever replied to information requests.
- I wanted this graduation project to cover as many different development phases as possible within the available time.
4. Possible Systems

During the beginning of the project, when the assumptions and use cases where formed, lots of ideas and possible solutions came up. This chapter will describe the most relevant ones. Before we go into any total solutions, let's analyze the physical system we need by having a look at the various input and output signals we could use, which can be interpreted and generated by a centralized system. We will also have a look into a decentralized solution. A model will be proposed including a couple of allocation strategies. After that, we will take a look at the car park definition and a couple of total solutions.

4.1. Input

One of the input signals this system needs, is the knowledge about the vacancy of the spots. We look at four possible ways to do this:

- by spot;
- by group of spots;
- by area or
- by car park

Let's have a deeper look into these options.

4.1.1 By Spot

The first and most obvious manner to determine whether a spot is vacant or not, is to measure that for every individual spot. There are several products on the market which can measure whether there is a car very nearby (on top of it). These sensors can measure metal (or object) presence, pressure or light for example. It is also possible to not measure the car, but something that is in the car, like an RFID tag. This way, you don't just know whether there's a car parked on the spot in question, you also know which car it is. If all the spot state information is gathered, the system could determine the number and location of vacant spots. A disadvantage of this measuring method is that a hardware sensor needs to be placed near every spot. This can be quite cost inefficient. On the other hand, the input generated is the most accurate. If the system would, for any reason, have to be restarted, the state of each spot can easily be rechecked.

4.1.2 By Group of Spots

A second way to determine whether a spot is empty or not, is to group the spots in small groups and determine the number of cars in these groups. This can be done by vision for example, like in the iSpot project. A camera is pointed at a group of three spots. A program analyzes the video stream and determines for each spot individually, whether there is a car parked on that spot or not. When solved visually, individual spot state information could still be provided.

Another way to determine the number of cars in this small group, is again, using RFID. A passive RFID tag can easily be read from 5 meters, so one RFID scanner could read about three or four tags. The number of cars in the immediate area of the scanner can be determined.

Note that you need less sensors using this method, but the sensors might be a little more expensive, primarily because of the bigger range that is needed.

According to Peter de Jong from Nedap, this mode of measuring already brings in a disadvantage,
which only grows in the next proposals. Imagine that a car doesn't park exactly within the borders of a spot, but crosses one of the white lines on the floor a little, what makes it impossible, or at least very unattractive, to park in the remaining spot. This remaining spot could get allocated over and over again. People are sent to the spot but none of them will actually park there. Luckily, assumptions 4 and 5 take care of these situations for now. In the real world, we'd like this situation to be measurable, using sensors or cameras, so we can take counter measures. We could also prevent this situation by placing small (flexible) poles between the spots, so the cars can not cross these lines.

If the system would, for any reason, have to be restarted, the spot states can easily be recalculated because the system can always measure the state of each spot.

4.1.3 By Area

A third way to keep track of vacant spots is to divide the car park in areas such that each area, has a very small number of entrances and exits. Such an area could be seen as a smaller embedded or nested car park. Illustration 2 gives an example of this. If you can configure the car park like this, you only have to measure when a car enters an area, and when a car leaves an area. Because the number of spots within the area is clearly defined, the system can easily determine whether there are any spots left in a specific area. When one of these areas does not contain any more vacant spots the system could indicate this by visually closing the area. This is why each area should be bound by the car parks intersections.

There are two important considerations. First of all, what happens if a car drives through area A to go to area B? The system might think that the car is parked in area A until it leaves area A again. But this does not have to be a problem, because there is, at this point, a possibility that the car is actually going to park in area A. It could be wise not to recommend this area as long as there is a car driving around in it, which is exactly what will happen because the system thinks the car actually occupied a spot in area A. If the area already was full, it's also not really a problem when a car would enter because it can't park there anyway.

Another important thing to keep in mind is that cars can park outside the spots or on two spots at once. Fortunately, assumption 4 and 5 take care of this. So, if an area provides 5 spots, 5 cars can be parked in this area.

Again, there are different techniques to measure whether a car enters or leaves an area, like sensors which measure metal presence, sonar, RFID scanners or video cameras. RFID scanners or video cameras might be a little more expensive but they are harder to fool and cars can easier be tracked around the car park, what could produce interesting statistics to improve the guidance. Apart from this, video cameras are obliged in car parks anyway, for safety reasons.
Note that this way of measuring is significantly less precise than the ones above but it still is precise enough to send a car to an area that still offers vacant spots.

When someone parks his car in a car park, he often remembers where he parked his car in an area, but is not able to find the area where the car was parked in (because they all look alike). If we'd like to guide someone back to his car, it is often enough to just guide him to the area of his parked car, assuming the user is able to distinguish his own car from about twenty other parked cars. Note that this functionality is beyond the original project goal, but it's mentioned because the EzPARK project provides this functionality.

A disadvantage of this method is, if the system, for any reason, has to be restarted, there is no way to determine how many cars each area is holding at that moment. If the car park would never be empty again, this would be a problem. Luckily most car parks are empty each night, so the system can be reset with an empty car park. If this would not be the case, the amount of cars per area could be configured manually or some algorithm could approximate the number of cars in the car park, but this is outside the scope of the project. If we assume that no car can enter or exit the car park during the restart, we could tackle this problem by storing the spot states on some persistent medium, which we can read again after a restart.
4.1.4 By Car Park

This is obviously the solution with the lowest significance. Many car parks or parking lots are already equipped with systems that can measure (or approximate) the amount of cars in the car park by counting the cars which drive into the car park as well as the cars that drive out of the car park. The difference between these two equals the number of cars in the car park. Though this might be useful for very small car parks (less than 20 spots), it is of no use within the scope of this project. In fact, this way of measuring is the same as for a single area, as described in the previous section.

4.1.5 Summary

<table>
<thead>
<tr>
<th>Measure by</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
</table>
| spot       | ● very accurate  
            ● not influenced by restart | ● cost inefficient  
            ● has been done before |
| group      | ● quite accurate  
            ● not influenced by restart | ● sensitive for mistakes  
            ● has been done before |
| area       | ● cost efficient  
            ● Interesting for research | ● influenced by restart  
            ● vacancy is calculated, not measured |
| car park   | ● cost efficient | ● highly influenced by restart  
            ● lack of accuracy |

Table 8: Input comparison

Note that measurement by area is the most interesting from a scientific point of view because none of the earlier projects seem to use this method. It is also more cost efficient because we need less sensors in the car park, but the information we could get is still accurate enough for making a recommendation. This is why we will choose to measure by area.

4.2. Output

There are also several ways to actually guide the car to a location. Some of the ways will be listed below. Note that the output can not be more precise than the input. If we do not have information about the vacancy of one individual spot, a car can not be guided to an individual spot. The next methods might be useful:

- Recommendation/hint at entrance (EzPARK);
- Navigation System (EPSSN);
- Cellphone;
- in car park (digital) signs (SiPark).

Let's go into these items a little further.

4.2.1 Recommendation/hint

Influenced by EzPARK, giving a recommendation or hint does not sound like a bad idea at all. A visitor would know from the beginning where he can park and simple signs can guide him to the recommended area. We do have to put some thought in how to determine which location to recommend.
4.2.2 Navigation System

In the ideal situation, updates, including the map and availability of spots, are sent wirelessly to a car navigation system. This navigation system can interpret this data and use its existing guidance instructions to guide the user to a vacant spot, which was picked by the eCarPark. Though this might sound like the perfect solution, there are a couple of reasons why this output would not work in practice:

- the GPS signal to determine a car's position within the car park might be blocked in case the car park is covered by a roof;
- all visitors should own a navigation system or there should be an alternative output for those who don't;
- all navigation systems will have to have an API, supported by eCarPark, to communicate with the eCarPark. If the navigation system does not provide such an API, it is useless and the bullet above applies. TomTom, a popular navigation system in the Netherlands, does provide such an API, where points of interest can be added to a map.

4.2.3 Cellphone

Another device which could be used to output guidance information is the cellphone. This could be more useful than the navigation system because a lot more people own a cellphone. Directions could be SMS-ed to the user or the user could run a small Java application or WAP site and be able to see graphical directions. Though this output sounds more usable than the navigation system, it still might not be the way to go:

- when people forget their (or do not own a) cellphone, they cannot use the service anymore;
- SMS delays could lead to confusion;
- SMS costs money per message (about 15 – 20 cents), a graphical (Java) solution would consume bandwidth which also costs money;

An advantage of using a cellphone as an output device is that third party navigation tools can easily be used. Look at Google Maps for example. Google provides an API so third party programmers can relatively easy add things to Google Maps. The car park could be inserted into Google Maps for example, so people don't have to switch applications. The cost of bandwidth will even become higher in this case though.

Notice that these in car solutions can be hard and expensive to deploy, and, in the case of cellphones, can be distracting in use. Let's have a look at a solution on the outside of the car.

4.2.4 In Car Park Signs

The car park could be equipped with:

- signs that indicate where a specific area could be found;
- LED-bars which tell where the vacant spots can be found;
- digital LED-arrows which point in the direction of the vacant spots;
- A sign could indicate the number of vacant spots at the entrance of each area.

An advantage of this is that you do not depend on the users equipment. A disadvantage is that multiple users could act on the same output. An output system like this could excellently be combined with the area oriented input from the previous section.
4.3. **Decentralized Solution**

The sections above both describe input and output signals which could be used in centralized systems. A central computer reads all the inputs, makes certain decisions, and creates outputs. This is not the only solution. The problem could also be solved decentralized.

Think about a situation where each parking spot is equipped with a sensor device to sense whether there is a car parked on that specific spot, as well as an actuator, which could indicate whether the spot is vacant or not. When a car drives through the lane, the driver can easily see whether there are any vacant spots, and if there are, where to look for them. The only thing that should be done is equip every spot with a small device that provides these both tasks.

This solution is simple but there are a couple of disadvantages for such a solution. First of all, information about the vacancy of a spot does not leave the spot unless the nodes communicate in a p2p-like way (like the MOTE in EzPARK), which drastically complicates the system. Every node would have to get server-like features so a cheap embedded system might no longer suffice. The cost of each node would increase from just a couple of euros to forty or fifty euros. If the server-like features would not be present in every node, the information in the network can not be used for any other purpose, like giving recommendations at the entrance; determine whether the car park is full; actually finding the one vacant spot from the other side of the car park or making reservations.

One big advantage should also be mentioned. This system is without any effort more scalable than any centralized solution because you do not depend on the amount of connections a server can handle, or the capacity of the infrastructure.

An even simpler solution would be one where each spot is only equipped with an actuator, which is blocked if and only if a car is parked on the spot. You could think about a laser pointer, located in the floor, that beams to a ceiling. When a driver sees a red dot on the ceiling, the spot is vacant. This would even make the sensor and computer obsolete. The same disadvantages hold as above. Apart from this, visual contact is required with the spot indicators.

Again, we could also think in areas instead of in spots. Each area could be a separate sub system where the number of vacant spots is constantly administered and some output device communicates this information to the cars which want to enter the area. The number of vacant spots could be displayed at the entrance of an area for example. These different areas would have to communicate with each other. In order to be able to give useful recommendations, there would still have to be some server which could compute these. Note that this approach very much looks like the one we were planning to take. It only adds more complexity to the communication, which is not where we want our focus to be.

![Illustration 3: Architecture solution for a 2 spot car park](image_url)
4.4. The directed graph

Because we would like to do some calculations on our car park later on, we map the car park to a directed graph. Each vertex represents an area that contains a certain number of vacant spots. Each edge will be labeled by the distance between these different vertices, which is a function of the number of spots of the connecting areas $s$ and $t$:

$$d = \max \left( \frac{s + t}{N}, M \right)$$

where $s$ and $t$ are the number of spots in the areas between which the distance is measured. $N$ is a constant that gives the number of spots within one distance unit. $M$ is the minimum distance between two areas. When we set $N$ to 6 and $M$ to 5, the distances between the different areas in the example car park in appendix , turn out as expected. These values might need some tuning for other car parks. Notice that the distance on the edges gets bigger when the number of spots in the connecting areas gets higher.

The picture below visualizes this mapping.

4.5. Storage

Another problem that should get some attention in this chapter is the storage of the car park data. Some administrator should be able to easily define the car park in some way so the system can understand the lay-out. There are several properties which come into scope here:

- easy to create and maintain;
- mappable to a directed graph and
- easy to visualize.

Lets go into these aspects a little deeper. First of all, the digital representation of the car park should be easy to create and to modify. Unless we make an additional tool to manage this data, this digital representation should be human readable. Because there is no plan to write such an extra tool at this moment we will choose an ASCII representation.

The second requirement is that the digital representation should be able to get mapped to a directed graph [DIE97] [BON76], a natural model of a car park which is needed for a couple of the allocation algorithms which are represented in the next chapter. Because each area will be presented
as a single vertex, these areas should somehow be defined in the digital representation. We need to be able to calculate the size of each area, and the distance between two areas.

The last requirement is that the digital representation has to get mapped to a visual representation. To do this, it would be useful to add coordinates to each spot so it can be drawn. In our examples, we only use x- and y-coordinates to simplify the visualization. Note that adding more dimensions won't influence the way our algorithms work, it only complicates the visualization.

To meet these requirements (and because it's the driving force behind interoperation), we decided to use XML as an underlaying structure. We created an XML-based format: Car Park Markup Language (CPML). CPML is designed to define car parks. If we want to use this XML based method, there are two things to make: an XML schema and an example. The first one can be generated out of the second one, so let's concentrate on making an example that holds the information we need.

There are different items in a car park to identify. First of all, there are a couple of areas we would like to identify, using the “<area>”-tags. Each area should have a unique identifier (or name). This is added as an attribute. In order to visualize the car park, the location of the spots is of importance, so we decided to give each spot an x and y coordinate. For this reason, the “<spot>”-tag is introduced, which lives inside an “<area>”-tag. To identify the spots in the future, we also add unique identifiers.

So a number of areas containing spots can be defined now. The next thing to worry about is how these areas are connected to each other. To provide this information, the “<connect>”-tag is introduced. This tag also lives in an area and tells to which area there is a one way connection. If the tag lives outside the scope of an area, there is a connection from outside the car park to this area. This means that the car entrance and exit connects to this area. If the <connect> tag is empty, this means that the passenger exit connects to this area.
This information can later on be used to generate a directed graph needed for some of the allocation algorithms.

```
<!-- CPML file of example car park //-->
<carpark>
  <connect>A</connect> <!-- car entrance //-->  
  <area id="A">  
    <spot id="1" x="1" y="1"></spot>  
    <spot id="2" x="1" y="2"></spot>  
    <connect>B</connect>  
  </area>
  <area id="B">  
    <connect>A</connect>  
    <spot id="3" x="5" y="1"></spot>  
    <spot id="4" x="5" y="2"></spot>  
    <connect></connect> <!-- passenger exit //-->  
  </area>
</carpark>
```

*Illustration 5: Example of CPML output*

The full example can be found in appendix 14.5.
5. Allocation strategies

One of the questions that none of the existing systems seems to address, is which spot or area to allocate in which situation? There are various areas we could choose for different reasons. This chapter will propose a number of allocation strategies.

5.1. Formalization

Before we start talking about solutions, let’s define a formal model and a cost function so our proposals can be compared later on. Our goal here is to identify the allocation algorithms that allocate a spot such that the time spent in the car park is minimal. To do this we define a cost function which computes the cost of a certain allocation in terms of time.

- $A$ : a set of areas $a_0$ to $a_k$ ;
- $d(a)$ : the distance from the area that contains the car entrance $a_{ce}$ to the allocated spot $a$, in accordance with the traffic rules and using the shortest path. (driving);
- $e(a)$ : the distance from the allocated spot $a$ and the area that holds the car exit $a_{cx}$, in accordance with the traffic rules and using the shortest path. (driving);
- $p(a)$ : the distance from the allocated spot $a$ and the area that holds the pedestrian exit $a_{px}$. (walking);
- $c$ : the constant time it takes to leave or enter the car;
- $y_a$ : the number of cars in area $a$;
- $s$ : a state of the car park, which holds the current number of cars for each area;
- $S$ : a sequence of states $s_0$ to $s_n$ where each element describes a car park filling such that $s_i = \{y_0...y_k\}$ ;
- $\text{cap}(a)$ : the capacity measured in spots of area $a$ ;
- $\text{cars}(a,s)$ : the current number of cars in area $a$ in state $s$ ;
- $V$ : a sequence of visitor tuples $v_0$ to $v_r$ where $v_i = (\text{timeArrival}, \text{timeSpent})$ .

Each tuple gives the time a car arrives, and the time that car will spend. Please note the algorithm does not know $\text{timeSpent}$, but the cost function does;
- $E$ : a sequence of events $e_0$ to $e_m$ where $e_i \in \{\text{enter}(a), \text{exit}(a)\}$ which can be constructed by our sequence $V$ and our preferred allocation strategy;
- $\text{alg}$ : a mapping from a state $S$ to a recommended area $a \in A$ ;
- $g_c$ : the constant speed of a car within the car park in distance units per time tick.
- $g_p$ : the constant speed of a pedestrian within the car park in distance units per time tick.
Knowing these variables and functions, we can compute the cost of a certain allocation in a certain state. The formula that will be used to compute the cost of the different algorithms is the following:

\[ C(\text{enter}(a)) = \frac{1}{g_c} \times d(a) + \frac{1}{g_p} \times p(a) + c \]

\[ C(\text{exit}(a)) = \frac{1}{g_c} \times e(a) + \frac{1}{g_p} \times p(a) + c \]

Note that \( d(a), e(a) \) and \( p(a) \) represent distances.

We also define a transition function starting in state \( s \) for both types of events. To compute the current state after an \( \text{enter} \) action, we take the previous state and add one car to the number of cars for the allocated area \( a \). To compute the current state after an \( \text{exit} \) action, we take the previous state and subtract one car from the number of cars for area \( a \).

\[ F(s, \text{enter}(a)) = s[y_a := y_a + 1] \]

\[ F(s, \text{exit}(a)) = s[y_a := y_a - 1] \]

The precondition for any event \( \text{enter}(a) \) in a certain state \( s \) is the following:

\[ \exists a : a \in A : \text{cars}(a, s) < \text{cap}(a) \]

And we have an invariant:

\[ \forall a, s : a \in A, s \in S : \text{cars}(a, s) \geq 0 \]

To evaluate an algorithm, we calculate a sequence of states \( S \) for a certain sequence of visitors \( V \) and a start state \( s_0 \). The cost of the algorithm is the sum of all the costs of the state transitions.

Why can't we just generate a sequence of events \( E \) and use it to evaluate all allocation strategies? Well, imagine a situation where a new event \( \text{exit}(a) \) is generated, but area \( a \) does not (yet) hold any cars when we used algorithm \( X \), but is does hold any cars when we used algorithm \( Y \). How can these two situations ever be compared? Is the event cost-free for one of the algorithms? This would not be very fair. So, one universal event sequence for each algorithm is not always possible. What we do want, is to generate a sequence using our algorithm, such that an area \( a \) will always hold at least one car if event \( \text{exit}(a) \) occurs. To do this, we generate algorithm specific event sequences out of a universal visitor sequence. This visitor sequence \( V \) lists a number of visitors, the time they enter the car park, and the amount of time they will spend in the car park. The allocation algorithm determines for every visitor which recommendation it would give. Two events are added to the event sequence on given times. The state is also changed each time an event is added because this influences the next area that gets computed by the algorithm. A visitor sequence can result in different event sequences for different algorithms. The following algorithm is used to generate the event sequences:
def \textit{minT}(X): // gets the earliest leaving car
  \textit{result} := (\infty, \infty);
  \textit{min} := \infty;
  \text{for } (t, z) \in X:
    \text{if } t < \textit{min} then
      \textit{min} := t;
      \textit{result} := (t, z);
    \text{fi}
  \text{end}
  \text{return } \textit{result};
\text{end}

def \textit{createE}(V, s):
  \textit{E} := <>; // create an empty event sequence
  \textit{P} := <>; // create a sequence to store end times of allocated areas
  \textit{S} := <s>; // create a state sequence width a start state
  \text{for } (t_v, p_v) \in V:
    \textit{t_v, a} := \textit{minT}(P); // get the earliest exit event
    \text{do } (t_p < t_v) : // for all exit event that occur before the current enter event
      \textit{s} := \text{\textit{F}}(s, \text{\textit{exit}}(a));
      \textit{S} := \textit{S} + + s;
      \textit{E} := \textit{E} + + \text{\textit{exit}}(a);
      \textit{P} := \textit{P} − − (t_p, a);
    \text{od}
    \textit{t_p, a} := \text{\textit{minT}}(P);
    \text{do}
      \textit{a} := \text{\textit{alg}}(S);
      \textit{s} := \text{\textit{F}}(s, \text{\textit{enter}}(a));
      \textit{S} := \textit{S} + + s;
      \textit{E} := \text{\textit{E}} + + \text{\textit{enter}}(a);
      \textit{P} := \textit{P} + + (t_v + p_v, a);
    \text{end}
    \text{return } \textit{E};
  \text{end}

We can now calculate $\sum_{i=0}^{m} C(e_i)$ for each allocation algorithm. Our goal is to minimize this value because this value represents the total time all the visitors spend in our car park. Let's give one small example using our empty example car park.

We have a sequence $V = [(0,850), (200,300)]$ This means that one car will enter at $t_0 = 0$ and spend 850 time units in our car park. A second car will enter at $t_1 = 200$ and will spend 300 time units in our car park. We can now map this sequence to an event sequence using the algorithm.
above:  \( E=[\text{enter}(a_0), \text{enter}(a_1), \text{exit}(a_1), \text{exit}(a_0)] \)

Note that each event will result in a new state which will influence our allocation algorithm.

Knowing this event sequence, we can now calculate the total cost for this sequence. In this step, we also define the constants, \( c=2 \), \( g_c=1.5 \) and \( g_p=1 \). This leads to:

\[
\left( \frac{2}{3} \times d(a_0) + p(a_0) + 2 \right) + \left( \frac{2}{3} \times d(a_1) + p(a_1) + 2 \right) + \left( \frac{2}{3} \times e(a_1) + p(a_1) + 2 \right) + \left( \frac{2}{3} \times e(a_0) + p(a_0) + 2 \right)
\]

where \( a_0 \) and \( a_1 \) still depend on our preferred allocation algorithm. The values can easily be calculated using the directed graph representation of our car park. If we would have used a random allocation algorithm, this could lead to \( a_0=A \) and \( a_1=D \). If we now apply the functions \( d() \), \( e() \), and \( p() \), the result will look like this:

\[
\left( \frac{2}{3} \times 10 + 20 + 2 \right) + \left( \frac{2}{3} \times 10 + 5 + 2 \right) + \left( \frac{2}{3} \times 10 + 5 + 2 \right) + \left( \frac{2}{3} \times 15 + 20 + 2 \right) = 88
\]

We also calculate the average cost per event (22) as well as the standard deviation from that average, which is in this case:

\[
\sigma = \sqrt{\frac{(22-28)^2 + (22-13)^2 + (22-13)^2 + (22-32)^2}{4}} = \sqrt{\frac{44 + 69 + 69 + 100}{9}} \approx 16.83
\]

5.2. Inspiration

Where would we get inspiration for allocation algorithms? Our first approach is to compare our problem to a classic scheduling problem. After some investigation we concluded that there are two major differences with our problem and a dynamic on line multi-resource scheduling problem. Scheduling algorithms often describe how to distribute a set of jobs over a set of resources. One important characteristic of a job is that we can somehow determine how much time it will take to finish the job. This might be possible for a job, but is not possible for a parked car. (See 14.2.1)

So a second source of inspiration is memory management, which is more about space and not time. We dove into the working of the malloc() function and found out that there are quite some similarities between managing memory and managing parking spots. Nonetheless, the real problems which are assigned by malloc() are not the ones we are looking for. Malloc() allocates pieces of memory while minimizing memory fragmentation and this problem does not occur in our car park because we assumed that each spot could hold exactly one car and each car would only occupy one spot. (See 14.2.2)

A third source of inspiration were caching algorithms. Cache is a fast but small piece of memory, located close to a CPU for example, that is used to cache data and instructions which are needed in the near past and future. Because this piece of memory is so small, the available space should be managed wisely. There are different algorithms to do this and they can all be applied to the car parks. These algorithms delete pieces of data from the cache which are least recently allocated and least frequently allocated. (See 14.2.3)
Our last source of inspiration was observation. Where do visitors want to park? Simply by looking at people's parking behavior, we can already determine that parking close to a passenger exit is very popular, as you can see in the photos below. We could write an algorithm that allocates exactly such an area if there are any vacant spots left.

This is actually part of the problem. Because people want to park near the exit, they first drive to the exit to find out that all the spots there are already occupied. Measurement results to illustrate this can be found in appendix 14.1.

5.3. Algorithms

This section will propose several different spot allocation algorithms which will be evaluated later on in this document using the simulation which implements the cost function.

5.3.1 The Common Path

Before we start summing up possible allocation strategies, we'd like to make clear what happens if such a strategy is not used. In our example car park, cars seem to be willing to park close to the exit, as mentioned before. How will they try to manage this without any guidance and without any knowledge about the vacancy of the spots near the exit? Well, they go take a look themselves. They drive all the way to the exit and look around whether there are any spots vacant over there. It is not really a problem if they find a vacant spot. But if they don't they have to drive back from the exit in the direction where they came from. During peak usage time, they spend more and more time in the car park looking for their preferred spot. It seems that a lot of cars drive a typical path, or a common path as we will call it, that will be illustrated in the picture below.
If we are going to calculate the costs of our different algorithms, it could also be interesting to know the cost if no allocation strategy was used. That's why we introduce a first strategy, that investigates whether there are any vacant spot next to the blue arrow in the illustration above. Now we have to find another allocation strategy that can beat this cost substantially.

5.3.2 Close to the Passenger Exit

\[
\text{alg}_{\text{exit}}(S) = a \text{ such that } (p(a) = \min_{a \in A} : \text{cars}(a, s) < \text{cap}(a) : p(a)) \text{ where } s = \text{last}(S)
\]

The first real algorithm we take a look at is the one that determines which area is providing vacant spots and is closest to the passenger exit. To do this, we would like to introduce the closestVertex algorithm, which basically is a modified version of Dijkstra's shortestPath algorithm. We can use the directed graph as introduced above. The car entrance is located at the top right of illustration 4. The fat node in the bottom right is the one where the passenger exit of the car park is located. As long as this vertex can offer any vacant spots, this is the area which will be recommended. When the area is completely filled, the closestVertex algorithm will look for the closest vertex that satisfies the vacant property \(\text{cars}(a, s) < \text{cap}(a)\). This algorithm was written in Python and can be found in appendix 14.6.1. Note that we first have to convert the directed graph into an undirected graph because the visitors do not have to follow traffic rules as long as they do not drive their cars.

Some car parks have several exits for the visitors where they can leave by foot, or several entrances for the cars. Several entrances is not really a problem. Because the system is centralized, it doesn't really matter where a certain recommendation is made, with the close to the exit parking idea in our mind. The fact that a car park can provide multiple exits can be a problem though. How would we know near which exit a visitor would like to park. There are a couple of possible solution we would like to propose, but these will not be implemented in our simulation. First of all, we could give multiple recommendations. If the visitor receives a different recommendation for each exit, he can decide for himself which one is more interesting for him. Both areas could be allocated but at least one of the allocated areas is very likely to expire within a certain time.

Another solution would be to let a visitor choose at the entrance near which exit he would like to
park. The system could then use the visitors preference to give one recommendation. The system could also store the visitors preference and try to recognize him the next time he arrives (using the license plate for example).

### 5.3.3 Close to the Car Entrance

\[ \text{alg}_{\text{entrance}}(S) = a \text{ such that } (d(a) = \min_{a \in A} : \text{cars}(a, s) < \text{cap}(a), d(a)) \text{ where } s = \text{last}(S) \]

If we would like to minimize the distance cars drive within the car park, we should recommend them to park close to the entrance. It should be obvious that this algorithm does not differ from the near the exit algorithm. The only difference is that we’d have to choose another starting area: the one next to the entrance instead of next to the exit. Apart from this, instead of using the undirected version of the graph, the directed version would have to be used because cars may only move according traffic rules, what may influence distances.

Note that this algorithm might sound nice considering the environmental impact, but users might not choose to park within the recommended area when it is not the one they would have chosen themselves. And if users start neglecting recommendations, the value of the system decreases until it becomes worthless, because visitors lose trust.

On the other hand, when the car park would be close to filled, and there are only a couple of spots left, the users will have less problems with unpopular recommendations, because they might think it could be the last vacant spot.

### 5.3.4 Random

\[ \text{alg}_{\text{random}}(S) = \text{select } a \in A \text{ with } \Pr[\frac{\text{cap}(a) - \text{cars}(a, s)}{\sum_{i=0}^{k} (\text{cap}(a) - \text{cars}(a, s))}] \text{ where } k = \text{size}(A) - 1 \land s = \text{last}(S) \]

Another algorithm we would like to try is the random one. What would happen if we would give random recommendations (where areas containing more vacant spots have more chance to be chosen then the smaller ones, so every vacant spot would get an equal chance of getting chosen and every area has an equal chance of getting filled first. The algorithm is listed in appendix 14.6.2.

### 5.3.5 Fairly Distributed

\[ \text{alg}_{\text{distributed}}(S) = a \text{ such that } \text{cars}(a, s) < \text{cap}(a) \land \left( \forall b \in A : a \neq b : \frac{\text{cars}(a, s)}{\text{cap}(a)} \leq \frac{\text{cars}(b, s)}{\text{cap}(b)} \right) \text{ where } s = \text{last}(S) \]

Another approach, somehow inspired by process scheduling, is called fairly distributed. When a series of processes have to be assigned to different processors, it is done in a way such that each processor has to spend about the same time on its job list. Faster processors will get bigger or more jobs and slower processors will get smaller or less jobs.

In our car park, there are differently sized areas with different numbers of spots. Like distributing jobs over processors, we could distribute cars over areas. Bigger areas, providing more spots, will receive more cars, and smaller areas will receive less cars. This way, the cars will be evenly distributed over the car park and each area will contain about the same percentage of vacant spots.

The way to do this is to calculate occupation percentages for each area and recommend the one with the lowest percentage. The car park will seem to be almost filled earlier so users are more likely to follow up recommendations, even when there are less cars parked in the car park. An advantage is that not only the cars are evenly distributed, but also traffic is. This means that the car density on
In a lot of daily situations, a system like last freed spot is used. Take for example a gas station, that has two gas pumps and one waiting line. When a driver is waiting at the head of the line, he does not yet know which pump he is going to use. His choice depends on which of the gas pumps will be available first, or as you could say, freed last. A car park might not be the same as a gas station, mainly because the lack of a queue, but this manner of allocating is worth looking at.

The algorithm will not be very sophisticated. It just selects the \( a \) of the last \( \text{exit}(a) \) event. This algorithm will always try to restore the previous configuration. The algorithm is listed in appendix 14.6.4.

Note that this algorithm does not work when there are no \( \text{exit}(a) \) events. That's why we initially fill sequence \( E \) with one \( \text{enter}() \) and \( \text{exit}() \) event. This first area can be computed by any other allocation algorithm.

### Most Recently Allocated

\[
\text{alg}_{\text{mostrecent}}(S) = a \text{ such that } (\max_i : a \in A \land e_i \in S2E(S) \land e_i = \text{enter}(a) : i)
\]

Borrowed from the world of caching algorithms, we can also allocate the most recent allocated spot (if vacant). Again, the code is not very complex, but it might be interesting to see how this system turns out in the simulation. The code is listed in appendix 14.6.5.

Like before, in the initial situation, there is no most recent allocated area. In that case we also pick
the first area out of the bag of areas

### 5.3.8 Most Frequently Allocated

\[ \text{alg\_mostfrequent}(S) = a \text{ such that } (\max_{a \in A} : (\#_{e \in S2E(S)} : e = \text{enter}(a))) \]

Last but not least, also borrowed from the caching algorithm world, the one that allocated the most frequently allocated spot or area. Again this algorithm might not be very hard to create, but let's have a look at how it handles the test during the simulation.

Again, initially we just have to pick one of the areas.

### 5.4 Area-ranking

There is another allocation strategy that we would like to propose, but this one differs from the previous proposals because it depends on the neglection of recommendations by users, such that \( s_i \neq F(s_{i-1}, \text{enter}(a)) \). Because this project does not focus on the fact whether a visitor will follow a recommendation, we can not model such a scenario.

The algorithm is inspired by Google's PageRank [GOO06]. What exactly is page-ranking? Google is a popular search engine that indexes the world wide web and allows users to search that web. Search results will have to be presented in some order. Different search engines use different ordering techniques. Google uses page-ranking to tackle this problem. When new sites are indexed, Google uses the links on the sites to determine the popularity of the sites where the links are pointed to. When there are more links to a certain place on the internet, Google thinks it's more popular, so it deserves a higher page ranking. This means the site will end up higher in the ordering.

The eCarPark could do something similar to find out which spots or areas are more popular than others. If some spot is recommended to a car and the system is able to successfully determine that the car did not follow the recommendation, the popularity of that area can be decreased. Meanwhile the system might in some situations also be able to determine that a non recommended area is chosen over a recommended area by one of the drivers. In this case, the popularity of that area is increased.

How can be determined which area to recommend? This will simply be the one with the highest area-ranking, which still contains any vacant spots.

\[ \text{recommendation} = \text{popularity} \times \min(\#\text{ vacant}, 1) \]

Popularity is a function of recommendation neglection, which can have both positive and negative effects on the popularity of an area, depending on whether the area in question was the recommended area or not.

\[ \text{popularity} = \text{neglectInFavor} - \text{neglectNotInFavor} \]

NeglectInFavor is an integer which will be increased when the program can determine a car enters this area without the area being recommended.

NeglectNotInFavor is an integer which will be increased when the program can determine that a car did not enter an area after the area was recommended.

### 5.5 Wrap up

The next thing to do would be implementing these algorithms as well as the cost function to compute the cost for a number of scenarios. But we do not have an actual car park so we would
have to simulate one. The next chapter will describe how this simulation was created. Once we have such a simulated car park, we can run our algorithms to see which one minimizes the total time a visitor spends within the car park. This time depends on the location of the allocated spot. Which spot to allocate is a function of the selected allocation algorithm and the current situation in the car park. We can calculate the cost of each allocation in each scenario. The results of this test will be listed in chapter 9.
6. Architecture

In order to test the allocation algorithms as proposed in the previous chapter, there is a need for some system to test them on. This chapter will describe that system.

6.1. System Requirements

In this section, the requirements of the system will be listed. Each requirement will be explained if necessary. Each requirement is numbered so it can be referred to later on. The numbering of the requirements is related to the order in which they will be implemented. In the first iteration of the system, only the first couple of requirements will be met. In the second iteration, a couple of new requirements will be added, and so on. The use cases described in the next section, each combine a number of these requirements, so the system can be designed in an iterative manner inspired by the spiral model [BOE88].

[REQ1] If a car enters the car park, the system notices.
   *It should be obvious that the system should know when a car wants to enter the system because this is where a recommendation should be made.*

[REQ2] The system always knows how many parking spots are vacant in each area.
   *If the system wants to point a car to a specific spot, it should know from which spots it can choose.*

[REQ3] If a car enters the car park, the system determines which vacant area to recommend to the car and allocates this spot for that car.
   *The system picks one of the vacant spots, which is the result of the current state and the preferred allocation algorithm, and marks it as allocated.*

[REQ4] The system is capable of detecting when a driver neglects recommendations.
   *When drivers neglect recommendations, the allocated spot becomes vacant again, because the driver probably parked on another spot. The spot can be allocated again for another car as long as it's vacant.*

[REQ5] If each spot in a certain area is occupied or allocated, the area does not get recommended until one of the spots becomes vacant again.
   *If a spot is allocated for car A, it can not be allocated for any other car until car A leaves or doesn't reach the recommended spot within a certain time frame.*

[REQ6] The system will communicate to the car where a allocated spot can be found.
   *After the system picked one of the vacant spots, the user should get informed about how to get there.*

[REQ7] Driving directions will at all times be in accordance with the rules in the car park.
   *A car will never be guided to the wrong direction in a one way street.*

[REQ8] If a vacant spot which was allocated for car A, is unforeseen picked by car B, car A will still be able to find a vacant spot.
   *Indicators in the car park will show where the remaining vacant spots are located.*

[REQ9] When the number of occupied spots plus the number of allocated and reserved spots equal the total number of spots in the car park, the system will not give any more recommendations, but tell the visitor the car park is full instead.
   *Does not need an explanation. Full is full.*
[REQ10] If a car leaves the car park, the system notices and changes the spot state. *The system should know this so the parking spot where this car was parked, can be freed again.*

[REQ11] If a certain area in the car park is full, the system will indicate this.

[REQ12] A system administrator can define a car park layout.

These are all the requirements of the system. It should be clearer what the system does now, and more importantly, what the system does not.

In case there is any time left, the following requirements might be interesting to take a look at.

[REQ13] The system learns which spots are more popular. *The system can see when cars don't park on recommended spots and can often see where they park instead. These spots or areas will be marked as more popular (they get a higher priority) so the system can recommend them earlier in the future.*

[REQ14] Visitors can reserve a certain spot for a certain time over the internet. The system will not allocate reserved spots for a new visitor. *Though the system can not guarantee that the spot will be available, it can guarantee that there will be a spot available, and do what it can to make that the preferred spot.*

[REQ15] The system sends a message to a user if a reserved spot gets taken. This message contains a new reserved spot, close to the original reserved spot. *In case some driver neglects its recommendation and parks his car on a reserved spot, the requester of this reserved spot will receive a notification.*

[REQ16] A system administrator can take a look at the current car park statistics. *The system can output statistics about occupation and area usage over time.*

### 6.2. Spots, Areas and States

Let's have a closer look at the all the properties of a single parking spot. A spot can have four possible states: vacant, occupied, allocated and reserved. The states are related to each other as shown in the illustration below.

**Illustration 8: Spot states**

Because we are not planning to manage the state of every individual spot, we use a graph to represent all the states of the spots in one area. Each vertex in the graph holds the number of spots which are in that state. The transitions are the same. The following illustration shows that in a certain area, we have 56 vacant spots, 0 occupied spots and 0 allocated spots.
6.3. System Overview

The system consists of several logical building blocks. These building blocks are defined in the illustration below.

The big blocks on the top and bottom of the architecture are the hardware including sensors and actuators. The hardware communicates with the input and output blocks. These blocks can be altered when different input devices would be used. The core of the system obviously does all the work.

The system gets inputs and gives outputs from and about two different parts. One of those parts is the car park itself. The system receives information about the number of cars in each area by measuring the number of cars entering and exiting each area. Using this information, the system will return information for the indicators in the car park, so they can show which areas are already filled. In our system, we will use red and green lights to indicate whether there are any free spots in an area.

The other part, the system notices that a car wants to enter the car park. When it does, it returns a
recommendation to the user.

6.4. Components

The blocks can again be divided into smaller modules with dedicated functions. The illustration below will define these modules where each component will take care of one or more requirements.

![Illustration 11: Detailed architecture](image)

The input and output of the system both are divided in two modules. The upper module of both blocks is responsible for the communication with the car park. The input will process information received from the car parks sensors, which register when cars enter or leave areas, and the output is responsible for sending information to the actuators in the car park, represented by the traffic lights.
The bottom modules in the input and output blocks are responsible for communicating with the user at the entrance. On the input side, the system registers the user wants to enter the car park, on the output side, the user can read the recommendation, if there is any.

The core block also might need a little explanation. This block consists of several modules. The upper five modules are all responsible for the administration of the vacant spots in areas. The first one (detect) writes received sensor information about the availability of spots to the storage component. The third one (allocate) allocates a spot. An allocated spot is vacant nor occupied, but can become either. When a car parks on the spot within a certain amount of time, the spot becomes occupied, else vacant. The original allocated spot can be allocated for someone else again. The second module is the timer needed for this behavior.

The fourth module (check) is responsible for selecting the best vacant spot out of all the vacant spots. Which spot is the best spot, is described in the chapter 5. If the car proceeds, this spot can be allocated for the car and what follows is described above. The last module of the upper five (full) is responsible for deciding whether a certain area or row is full or not. The module gets triggered when the number of vacant spots in an area changes. If the module concludes that a certain area is full, it sends an output to the output block.
6.5. Communication

The next thing to do is define the interfaces between these different components. The illustration below shows the numbering of the different interfaces. These numbers can be associated with the descriptions below.

The table below describes the different data types which are used in these messages.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area</td>
<td>Integer</td>
<td>The identifier of the area in question</td>
</tr>
<tr>
<td>Direction</td>
<td>Boolean</td>
<td>The direction of the car, true for entering, false for leaving</td>
</tr>
<tr>
<td>Knock</td>
<td>Boolean</td>
<td>True if there's a car waiting at the entrance of the car park</td>
</tr>
<tr>
<td>State</td>
<td>Boolean</td>
<td>True if there are vacant spots in the area. The indicator will be green. False if the area is completely filled. The indicator will be red.</td>
</tr>
</tbody>
</table>

Table 9: Communication data types
Table 10: Communication between components

The following three figures explain these action in sequence diagrams.
Illustration 14: A car drives to another area

Illustration 15: A user asks for a recommendation at the entrance
With this information, we should be able to implement our simulation.
7. Detailed Simulator Design

After making an architecture, a detailed design was made so the demonstration can actually be implemented. This chapter describes how the system was implemented. Several iterations are described, each divided in a number of components and processes. The first iteration describes a skeleton system. More features and use cases will be implemented in the second iteration.

7.1. Iteration 1: the Basic System

The first iteration consists of several processes that fulfill the basic requirements. Some simplified input will be received, some simplified output will be generated, the storage component will be realized, and a recommendation will be made.

7.1.1 Car Park

The car park represents the input sensors and output actuators of the car park. Because we use sensors at the entrance and exit of each area, these sensors could be simulated by placing buttons in an image map. For the output, traffic lights were chosen, which can also be presented in the image map by colored arrows.

The input signal is basically just a message that contains two areas: the one where the car comes from and the one where the car goes to.

To implement this component, we create a website where one can see a visual representation of the car park, together with some arrows where the area borders collide. If one clicks on one of these arrows, an event is generated. The image is illustrated in the picture below. If the component receives a traffic light change, it updates one of the corresponding layers in the image. This means that one or more of the green arrows can turn red in case one of the areas will be filled. This component will be implemented using PHP and its GD library, so it can easily be showed on a website. AJAX will be used for communication purposes.
7.1.2 User / Car

The user component simulates the hardware at the entrance of the car park. This hardware registers when a user attempts to enter the garage. A simple button on the website will suffice to mimic this behavior. As a reply, this component will receive an identifier for a certain area, as well as a path how to get there, represented in the identifiers of the areas to follow.

For convenience, this component will be simulated together with the car park. This makes the system easier to test. Again, AJAX will be used for communication.

7.1.3 Input and Output

Because no hardware will be used for this simulation, the input and output components of the system don't really have to do anything. They just forward messages as they are. All input and output components are joined in one communication component which receives something from a PHP script over a TCP connection, and forwards this to one of the Python components, or the other way around.

7.1.4 Detect

Detect is the component that detects any movement in the car park and updates this in the system, for future recommendations. The component receives a movement from area A to area B, so has to modify the two finite state machines linked to those areas. Area A will get an extra vacant spot, and will lose an occupied spot, where area B gets an extra occupied spot, and loses a vacant spot. Note that the number of vacant spots can be vacant if a car enters an already filled area.

7.1.5 Spot Selector

This component is called when a new visitor wants to enter the garage. The recommend button is clicked and the system receives a notification that a new recommendation has to be calculated. The check component will determine which area would be the best one to park in, using the information in the storage component and a selected allocation algorithm. The area, which is a result of this, will be returned to the spot selector. Allocation is a problem for the next iteration.

7.2. Iteration 2: Picking the Best Spot

The algorithm for picking the best spot out of all the vacant spots will be made in this iteration.
7.2.1 Storage

This component realizes the storage of the actual car park. The component should be able to read the CPML file and generate an internal structure to work with, being a directed graph as well as a finite state machine for each area. This structure should hold the information directed graph as well as a finite state machine for every area in the car park. The illustration above shows a graphical representation of this structure.

The main graph will be represented as a 2 dimensional dictionary where the first index represents an area, the second index represents a second area and the value of this element represents the physical distance between the areas in the car park, from the first to the second area. If the value refers to a two-way road, it should also occur somewhere else in the dictionary. The element below represents this directed graph.

```
G = {'A': {'D': 10},
     'B': {'F': 10},
     'C': {'A': 10, 'D': 5},
     'D': {'C': 5, 'E': 5},
     'E': {'D': 5, 'B': 10, 'F': 5},
     'F': {'E': 5}}
```

Illustration 18: Graphical representation of internal data structure
A second two-dimensional dictionary represents the finite state machines. The first index of this array is the area identifier and the second index represent a state \{VACANT, OCCUPIED, ALLOCATED, RESERVED\} in a finite state machine. The values represent the number of spots which share that state. The structure looks as follows:

\[
P = \{\text{'A': \{`VACANT': 56; `OCCUPIED': 0; `ALLOCATED': 0; `RESERVED': 0\},}
\text{'B': \{`VACANT': 56; `OCCUPIED': 0; `ALLOCATED': 0; `RESERVED': 0\},}
\text{'C': \{`VACANT':  4; `OCCUPIED': 0; `ALLOCATED': 0; `RESERVED': 0\},}
\text{'D': \{`VACANT':  3; `OCCUPIED': 0; `ALLOCATED': 0; `RESERVED': 0\},}
\text{'E': \{`VACANT':  4; `OCCUPIED': 0; `ALLOCATED': 0; `RESERVED': 0\},}
\text{'F': \{`VACANT':  5; `OCCUPIED': 0; `ALLOCATED': 0; `RESERVED': 0\}}\}
\]

Note that

\[
\sum_{a \in A, u \in U} P[a][u] = \text{TOTAL SPOTS CAR PARK}
\]

where \(U = \{\text{VACANT, OCCUPIED, ALLOCATED, RESERVED}\}\) and \(A = \{a, b, c, d, e, f\}\)

### 7.2.2 Timer

The timer is a very simple component. It starts a separate thread when someone gets a recommendation. For some time, a certain area has an allocated spot. This means that the spot can not get allocated for someone else, until the allocation is expired again. When this allocation expires, two things could have happened. A car entered the area or not. In the first case, the area temporarily counts one vacant spots less, and one occupied spot extra. After the allocation expires, there will be an extra vacant spot again, that can be allocated for new arriving cars.

If no car entered the area, nothing happened and after expiration, the allocated spot will simply become vacant again.

### 7.2.3 Allocate

This component will only do one thing which is modifying one of the finite state machines. If the system gives a recommendation, one of the spots will be allocated, so the number of allocated spots in a certain area increase by one, and the number of vacant spots decrease by one.

### 7.2.4 Check

This component determines which area will be recommended. It implements the closestVertex algorithm, the shortestPath algorithm and keeps track of the popularity of the states.

### 7.2.5 Full

The full component gets triggered whenever one of the finite state machines changes. The component computes whether any of the affected areas does or does not have any vacant spots, since the last change. This simple function simply iterates over \(P\). If there are any changes, lets say the last spot in area A just got occupied, the component sends this to the cap park, so the traffic light indicator can turn red.
8. **Simulation**

It is time to test the system and optimize and tweak the parameters.

To do this, we start our web server, start our Python application and our web browser and the following system will be running.

Let's describe the components above. The CPML component does not need much explanation. It is an XML based file that describes any car park. The GD component, which is a PHP extension, opens this file and draws a graphical representation of the car park, as well as the connections between the different areas. The PHP component renders a simple website that embeds this image and generates links so cars can be moved around using user interaction. This result is sent to the users browser as a HTML file that includes Javascript.

Meanwhile, the CPML2DG component also opened the CPML file and generated several other data structures out of this, like a directed graph and finite state machines. This directed graph can then be used by the Python server to calculate shortest paths.

Now the user can take some action in his browser, like move a car around, enter the garage, ask for a recommendation, etc. Javascript passes this request to a PHP script using Ajax. This PHP script will forward the request again to the python server. The python server, updates its finite state machine and sends back results. When the user asked for a recommendation, the allocation components starts computing the best area to park in and returns this to the Python server using the preferred algorithm. The allocation component will be variable, meaning, we will use different kinds of allocation algorithms to determine which one would be the better choice. These algorithms are described in the chapter 5 and listed in appendix 14.6.
We can now play around with this system. We load a car park, set the allocation strategy and fill the car park with a random filling. After this, we can ask for a recommendation using the browser. One of the spots in the recommended area will get allocated. The number of allocated spots for that area, increases by one, and the number of vacant spots decreases by one. After receiving a recommendation, we can move a car from the entrance to the recommended spot. Each time we enter an area, the number of occupied spots increases by one, and the number of vacant spots decreases by one. So we end up in a situation where the number of vacant spots in the recommended area decreased by two, the number of occupied spots increased by one, and the number of allocated spots increased by one. After some predefined number of seconds, the recommendation expires and the number of allocated spots decreases by one, and the number of vacant spots increases by one.

Meanwhile, we can ask for another recommendation. This time, we decide not to follow the recommendation because we think we can find a better spot. Again, the number of allocated spots in the recommended area is increased by one. But in the end, the car did not park in the recommended area, so after the recommendation gets expired, the situation, for this area, is the same as before the car asked for a recommendation. In some other area, the number of vacant spots decreased by one, and the number of occupied spots increased by one. This is the area where the car, that neglected the recommendation, did park. The following screen shot shows the user interface for this simulation.
9. Allocation Strategy Comparison

In order to determine which allocation algorithm would be the better choice, there are four things we need:

- a cost-function;
- some car park definitions;
- initial fillings and
- some visitor sequences.

The cost function is something we already described in the formal definition of the problem. We also have one car park definition to start with and we will define some others in this chapter. An initial filling $s_0$ can also easily be created (by random), but we prefer to start with an empty car park, and let the algorithm do all the filling.

9.1. Visitor sequences

Generating a visitor sequence can also be a challenge. We expect that different algorithms give better results in different situations. That's why we construct a number of visitor sequences to find these differences. The following table lists these sequences.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Goal</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{office1}$</td>
<td>A typical office day where people can park the entire day, half a day or just a short time for a quick meeting</td>
<td>See what would happen in a car park near an office.</td>
</tr>
<tr>
<td>$V_{office2}$</td>
<td>As above, but another random sequence</td>
<td>As above</td>
</tr>
<tr>
<td>$V_{shop}$</td>
<td>The car park near a shopping center where people come in to park between 15 and 120 minutes</td>
<td>See if the results differ if people only park for a short time.</td>
</tr>
<tr>
<td>$V_{exit}$</td>
<td>A fictional sequence that's optimized for alg\textsubscript{exit}(). First the cheap areas will be occupied by short parkers. Than the expensive spots will be occupied by cars that park the entire day. After that, more short parkers arrive.</td>
<td>Will the costs of the different allocation strategies be even closer to each other.</td>
</tr>
</tbody>
</table>

*Table 11: Visitor sequences*

9.2. Car Parks

Maybe one single car park doesn't really do it if we want to compare the different allocation strategies. That's why we introduce three extra car parks with different properties. Car park 2 has an extremely inefficient common path. Car park 3 does not have any one-way streets which cause loops. The following illustration shows the new car parks.
Illustration 22: Car park 2 and 3
9.3. Goal

We are interested in two values: the average cost of an event (enter, exit) given a car park lay-out and an allocation strategy, as well as the standard deviation which is computed by the following formula:

\[ \sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}} \]

where \( \bar{x} \) represents the average which is computed by:

\[ \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \]

and \( n \) is the number of samples.

9.4. Test Results

The following tables list the costs for the visitor sequences each of the proposed algorithms including the one that uses the common path. For each algorithm, the absolute cost, the average cost per event and the standard deviation are printed.

<table>
<thead>
<tr>
<th>V_office1, example car park</th>
<th>Recommendations</th>
<th>Absolute</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td>237</td>
<td>12365</td>
<td>26,09</td>
<td>4,94</td>
</tr>
<tr>
<td>Entrance</td>
<td>237</td>
<td>12458</td>
<td>26,28</td>
<td>4,82</td>
</tr>
<tr>
<td>Random</td>
<td>237</td>
<td>12321*</td>
<td>25,99*</td>
<td>5,07*</td>
</tr>
<tr>
<td>Fairly distributed</td>
<td>237</td>
<td>12363</td>
<td>26,08</td>
<td>5,05</td>
</tr>
<tr>
<td>Last freed</td>
<td>237</td>
<td>12363</td>
<td>26,08</td>
<td>5,14</td>
</tr>
<tr>
<td>Most recently</td>
<td>237</td>
<td>12417</td>
<td>26,20</td>
<td>5,05</td>
</tr>
<tr>
<td>Most frequently</td>
<td>237</td>
<td>12327</td>
<td>26,01</td>
<td>5,03</td>
</tr>
<tr>
<td>Common</td>
<td>237</td>
<td>15338</td>
<td>32,36</td>
<td>13,52</td>
</tr>
</tbody>
</table>

*Table 12: Cost of allocation for V_office1 in the example car park  (*average of 3 tests)*
### Table 13: Cost of allocation for $V_{office2}$ in the example car park (* average of 3 tests)

<table>
<thead>
<tr>
<th>Recommendations</th>
<th>Absolute</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td>10559</td>
<td>26,14</td>
<td>4,85</td>
</tr>
<tr>
<td>Entrance</td>
<td>10571</td>
<td>26,16</td>
<td>4,98</td>
</tr>
<tr>
<td>Random</td>
<td>10590*</td>
<td>26,22*</td>
<td>4,78*</td>
</tr>
<tr>
<td>Fairly distributed</td>
<td>10575</td>
<td>26,18</td>
<td>4,87</td>
</tr>
<tr>
<td>Last freed</td>
<td>10681</td>
<td>26,44</td>
<td>4,65</td>
</tr>
<tr>
<td>Most recently</td>
<td>10708</td>
<td>26,50</td>
<td>4,59</td>
</tr>
<tr>
<td>Most frequently</td>
<td>10545</td>
<td>26,10</td>
<td>4,91</td>
</tr>
<tr>
<td>Common</td>
<td>13064</td>
<td>32,34</td>
<td>13,47</td>
</tr>
</tbody>
</table>

### Table 14: Cost of allocation for $V_{office1}$ in car park 2 (* average of 3 tests)

<table>
<thead>
<tr>
<th>Recommendations</th>
<th>Absolute</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td>16976</td>
<td>46,38</td>
<td>8,73</td>
</tr>
<tr>
<td>Entrance</td>
<td>16914</td>
<td>46,21</td>
<td>8,33</td>
</tr>
<tr>
<td>Random</td>
<td>16899*</td>
<td>46,17*</td>
<td>8,66*</td>
</tr>
<tr>
<td>Fairly distributed</td>
<td>16964</td>
<td>46,30</td>
<td>8,80</td>
</tr>
<tr>
<td>Last freed</td>
<td>17078</td>
<td>46,66</td>
<td>8,80</td>
</tr>
<tr>
<td>Most recently</td>
<td>17078</td>
<td>46,66</td>
<td>8,80</td>
</tr>
<tr>
<td>Most frequently</td>
<td>16735</td>
<td>45,73</td>
<td>8,43</td>
</tr>
<tr>
<td>Common</td>
<td>19613</td>
<td>53,59</td>
<td>30,24</td>
</tr>
</tbody>
</table>

### Table 15: Cost of allocation for $V_{office1}$ in car park 3 (* average of 3 tests)

<table>
<thead>
<tr>
<th>Recommendations</th>
<th>Absolute</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td>10155</td>
<td>28,53</td>
<td>4,67</td>
</tr>
<tr>
<td>Entrance</td>
<td>10329</td>
<td>29,01</td>
<td>4,66</td>
</tr>
<tr>
<td>Random</td>
<td>10223*</td>
<td>28,72*</td>
<td>4,64*</td>
</tr>
<tr>
<td>Fairly distributed</td>
<td>10330</td>
<td>29,02</td>
<td>4,72</td>
</tr>
<tr>
<td>Last freed</td>
<td>10260</td>
<td>28,82</td>
<td>4,64</td>
</tr>
<tr>
<td>Most recently</td>
<td>10260</td>
<td>28,82</td>
<td>4,76</td>
</tr>
<tr>
<td>Most frequently</td>
<td>10321</td>
<td>28,99</td>
<td>4,39</td>
</tr>
<tr>
<td>Common</td>
<td>15874</td>
<td>44,59</td>
<td>30,80</td>
</tr>
</tbody>
</table>
Before we continue, doing more tests, with different visitor sequences, let's analyze the results we have got so far. The first thing we clearly see is that the number of recommendation does not differ for different allocation strategies for both office visitor sequences. The reason for this is, no matter how smart we allocate spots, we will never get more spots. We already talked about this earlier, but the results prove this once more.

A second observation is that the average cost of a certain strategy does not differ so much from any other strategy, except for the common path. Why would this be the case? The reason that it differs quite much from the common path is that the user now knows where to park up front and doesn't have to look for a vacant spot himself. He can just drive the shortest path to the spot that is allocated for him. The reason why the differences between the strategies are so small, is that the car park is filled most of the time. When there is only one vacant spot, it doesn't really matter which strategy we use.

It starts to get interesting when we also look at table 16 and 17. We see that the number of recommendations for this sequence equals the number of visitors. This means that both car parks never got completely filled. This also results in a bigger difference between the average costs. In table 16, we clearly see that the $\text{alg}_{\text{exit}}$ has a lower average cost than the others. We also see that the difference with the common path approach is way smaller than before and that this approach is not
the most expensive anymore. The last thing we notice is that the standard deviation in table 17 is way lower for the common path. Two strategies even have an extremely low standard deviation.

Can we explain these results? Because the car park never gets completely filled, the efficiency of the algorithm we use is more important in this scenario. Because the absolute cost of one allocation with the alg\text{exit} is lower, and the chance that there is a vacant spot near the exit is bigger, this strategy scores better. The fact that the average cost of the common path is lower, has the same reason. Because the car park never gets completely filled, the chance that a car can park in area B is bigger, so we don't pay for the extra loop in the car park. The reason why it even scores better than some other strategies, is that the common path, despite the inefficient way to get to a certain spot, does allocate a cheap spot, when it exists. This also explains the lower standard deviation in this scenario.

Table 17 looks more like we had expected again. Most strategies score about the same and the common path is again quite a bit more expensive. The only real interesting thing here is the low average cost of the “most frequently” strategy. Looking back at table 14, we also see that this strategy scores slightly better for car park 2, but the difference might be too small to draw conclusions.

The reason why all the other average costs are so alike is the following. We randomly generated a visitor sequence for a car park. The randomness in this queue possibly makes the differences in cost between the different strategies small. If we would generate such a visitor sequence with a certain allocation strategy in mind, we could probably improve the behavior of that one strategy, and possibly make the other behave worse.

\text{V}_{\text{exit}} describes such a sequence for our “close to the exit” strategy in the example car park. The way to do this, is to allocate the cheap areas for cars that only park short, and the relatively more expensive areas, for people that park long. This result are shown in the following table.

<table>
<thead>
<tr>
<th>\text{V}_{\text{exit}}, example park</th>
<th>Recommendations</th>
<th>Absolute</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td>510</td>
<td>21417</td>
<td>21,00</td>
<td>6,85</td>
</tr>
<tr>
<td>Entrance</td>
<td>510</td>
<td>26151</td>
<td>25,64</td>
<td>5,55</td>
</tr>
<tr>
<td>Random</td>
<td>510</td>
<td>26626*</td>
<td>26,44*</td>
<td>4,94*</td>
</tr>
<tr>
<td>Fairly distributed</td>
<td>510</td>
<td>25577</td>
<td>25,07</td>
<td>5,87</td>
</tr>
<tr>
<td>Last freed</td>
<td>510</td>
<td>28797</td>
<td>28,23</td>
<td>2,95</td>
</tr>
<tr>
<td>Most recently</td>
<td>510</td>
<td>28797</td>
<td>28,23</td>
<td>2,95</td>
</tr>
<tr>
<td>Most frequently</td>
<td>510</td>
<td>26861</td>
<td>26,33</td>
<td>3,55</td>
</tr>
<tr>
<td>Common</td>
<td>510</td>
<td>25857</td>
<td>25,35</td>
<td>9,68</td>
</tr>
</tbody>
</table>

\text{Table 18: Cost of allocation for } V_{\text{exit}} \text{ in example car park} (* average of 3 tests)

How did we get these results? We first occupied all the cheap spots with short parkers. Then we occupied all the expensive spots with long parkers. The rest of the day, we only added short parkers so the cheap spots would get allocated over and over again.

What can we learn from table 18. The first thing we see is that the number of recommendations does not equal the number of visitors. This means that the car park was completely filled at some
times. The second thing we see, is that indeed, \textit{alg_{exit}} has the best results for this sequence. The difference is over twice as big as the differences we've seen in table 16.

Another observation we make is that the common allocation strategy doesn't perform as bad as it used to. Why could this be the case? The reason is that this strategy will too, allocate the spots close to the exit first, but doesn't provide a shortest path. Because the spots near the exit are vacant most of the time, this doesn't matter because the common path doesn't differ that much from the shortest path for spots close to the exit. Note that the standard deviation for common allocation strategy is quite high, because the spots far from the exit are still extremely expensive, and they get allocated once for this visitor sequence.

The last thing we would like to comment on is the high costs of the most recent allocation strategy. The reason for this is that, using this visitor sequence, the car park gets filled first. Every time the system allocated a spot, it searches in the same area as for the previous allocation. But this is not the case for the first one, which is actually determined quite randomly. In this case the algorithm unluckily, chose the area with the highest cost (A). Every time a spot in this area gets freed, it can be allocated again. This adds up, causing this algorithm to be the most expensive for this visitor sequence.

An optimal visitor sequence can also be generated for various other allocation strategies. If, for example, we use, once more our example car park, and the first 64 visitors intent so spend the entire day in the car park, and the rest of the visitors only spend an hour or so, the close to the entrance approach will give us very good results.
9.5. **Explanation of Results**

The first thing we notice is that the results are all relatively close to each other, except for the approach where no algorithm is used. Why could this be the case? The answer to this question is simple. Differences can be large for a single allocation, but at some point in the morning, if there is any need for such an algorithm, the car park will be filled. At this point, each spot is allocated at least once, so it doesn't matter if we first allocate the expensive spot, and later the cheap one, or the other way around. If we allocate all the spots in any order, the sum of the total costs will be the same.

But there are small differences, how do we explain them? If a visitor spends less time in the car park, a spot can be allocated again. If this is a cheap spot, we could win some time here, if it's not, we won't. Can we use this knowledge to optimize our allocation algorithms? The answer is no, because our car park does not provide any information about how much time someone is going to spend in the car park. If we would have that information, we could send the short time visitors, to the less expensive spots.

Another thing that needs some attention is the relatively higher cost of the common path approach. Why is this cost so much higher? Again, the answer is simple. Visitors are not directly sent to a vacant spot. They have to look for a vacant spot themselves. This means that if there are no vacant spots in the cheap areas, the cost of such an allocation increases quickly. In our car park, this only happens in roughly half the spots, and the chance it happens is smaller because the more expensive spots will be allocated later. This means that the cheaper spots will be chosen more frequently. My intuition tells that the difference should be about 25% (0.5*0.5) and this seems to be correct if we look at the results.

9.6. **Choose an algorithm**

Some allocation algorithms have a preferred visitor sequence that minimizes the total cost of the sequence. If one would expect a certain sequence of visitors, an optimal allocation algorithm can be chosen, such that the short parkers get the cheap spots and the long parkers get the expensive spot. If such a sequence can not be identified up front, the “close to the exit” algorithm seems to give the best result in most cases.
10. Conclusion

So after designing this system and evaluating different allocation strategies, there are several conclusions to draw. To do so, let's answer the subquestions from section 2.3.

First of all, cars can easily be noticed at the entrance using some hardware sensors. This could trigger a recommendation request. The system could allocate a spot in an area, using the allocation algorithm preferred by the car park owner. Information about spot states can relatively easy be managed when the system administrates the number vacant, occupied and allocated spots in each predefined area in the car park. This can be done my measuring the number of cars that enter and exit each area.

We did not find a method to prevent neglecting of recommendations. We did find a way to deal with these neglections. Whenever a driver might decide to neglect a recommendation, this has only little influence on the system. The allocated spot will expire within a certain amount of time, ready to get allocated again. The spot that is chosen by the driver will get occupied as soon as he enters that area. And just in case that spot was allocated for a later visitor, that visitor can use the digital indicators in the car park to see where other vacant spots are located.

A spot is said to be vacant again as soon as a car leaves the area or as soon as an allocation expires. When the spot is vacant, it can be allocated again by the system. The system does not take visitors preferences into account yet, but in a future version of the system, people could be able to make a reservation over the internet or using their mobile phones. This reservation could be linked to their license plate our cellphone.

There are three methods described how to guide a car to a vacant spot. First of all, after calculating a recommended area, the system will also calculate the shortest path to that area. The visitor gets a notification of this path. Second of all, old-fashioned signs in the car park can indicate where to find a certain area. And third method is to use digital indicators in the car park to indicate where to find the vacant spots, just in case the visitor forgets which area was recommended to him, or in case someone else neglected a recommendation and the recommended area turns out to be completely filled.

Because there was no actual car park to run tests on, a car park was simulated. This was done using a small website showing a graphical representation of the car park, that communicates with the system using AJAX. The simulation can model the most basic movements in a car park and the recommendation system can “real-time” adept to new situations. The simulation is a tool to demonstrate the working of the different allocation algorithms.

The main focus of the project turned out to be spot allocation. A number of different spot allocation algorithms were proposed, implemented and tested. These algorithms were inspired by process scheduling, memory allocation and caching algorithms, as well as common sense (for the random solution). The first proposed algorithm, that allocates the spot closest to the exit, seems to mimic visitors preferences the most, but over the long run, it didn't pay off as we hoped because eventually, each spot will be occupied anyway.

So what can we conclude about allocation strategies? There is not just one allocation algorithm which produces the best allocation in all cases. The cost will depend on the behavior of our visitors and the following example will show why. Imagine a car park with 2 areas which will result in different cost functions. The area with the lower cost will be allocated first, but the cars in this area all spend the entire day parked. New visitors arrive all day and they are sent to the expensive area for their short visits. In this case it would have been wiser, to allocated the more expensive area first. The problem is that we would have to know how long a car will be parked in our car park and
our infrastructure does not provide this information. An optimal solution could not be given up front.
11. Summary

Problem
More and more cars populate our crowded world and it gets harder and harder to find a vacant parking spot in a car park. When a car enters a car park during peak usage hours, it can take ages before the driver finds one of the last vacant spots. This almost pointless driving around can be extremely annoying for the driver and has an unnecessary impact on the environment.

What the world needs is some guidance system to guide a car or car driver to one of these vacant spots. Some thought was put into determining which vacant spot to allocate for a new visitor. Different allocation algorithms are proposed and compared.

Earlier researchers
This project is not the first one to address this problem. Some earlier projects, like iSpot and EzPARK already came up with solutions how to guide a car to a parking spot. Which question they did not address is the one which spot to allocate and why.

Possible solutions
Based on the earlier research and common sense, some solutions where proposed for a system that would solve this problem. The focus here lays on the combination of input and output signals used.

Spot allocation
The focus of this project lays on the question which spot would have to be allocated. To answer this question, different spot allocation algorithms were proposed. A cost function is created to measure which of these algorithms would give better results under which circumstances. The differences between these strategies are very small and depend on the pattern of visitors.

Architecture
After having some thought about input signals, output signals and spot allocation algorithms, an architecture was made. Different functions were addressed and different components were defined, as well as the communication between those components.

Design and implementation
Each component from the architecture got a description so it could be implemented. The components were implemented over two iterations. First the basic skeleton of the system was made, with a standard hard coded car park definition and predefined allocation areas. In the second iteration, these lacks were added to the system so different car parks could be used in the Car Park Markup Language (CPML) file.

Simulation
The system can mimic most legal car movements. The car park can be filled in any configuration and all the allocation algorithms can be tested in these configurations. The cost function can calculate the efficiency of an allocation, so the different algorithms can be compared under different circumstances.

Conclusions
To conclude, cars can be directed to recommended areas by calculating the shortest path and notify them. A recommended area can be calculated using one of the proposed algorithms. The simulation that was created can test these algorithms and they can be compared using the cost function. Differences are small but there is quite something to win over the old situation where no recommendations were given.
12. Samenvatting

Probleem
Er komen steeds meer auto's op de weg waardoor het steeds moeilijker word om een vrije parkeer plek te vinden in een parkeer garage. Wanneer een auto een parkeer garage binnenrijdt tijdens de piek uren, kan het erg lang duren voordat er een vrije plek gevonden wordt. Dit zinloze rondrijden kan erg frustrerend zijn voor een bestuurder en ook is het een onnodige aanslag op het milieu.

Wat de wereld nodig heeft, is een systeem dat een auto naar een beschikbare parkeer plaats leidt. Belangrijk is naar welke plek een auto geleid moet worden. Verschillende algoritmes zijn mogelijk.

Eerder onderzoek
Dit project is niet de eerste welke naar dit probleem kijkt. Eerdere projecten, zoals iSpot en EzPARK kwamen al met oplossingen om een auto naar een parkeer plek te leiden. Er werd echter niet gekeken, naar waarom een bepaalde plek gekozen wordt boven anderen.

Mogelijke oplossingen
Gebaseerd op eerder onderzoek zijn er enkele mogelijk oplossingen voorgesteld voor een systeem dat het hierboven beschreven probleem op kan lossen. Het accent ligt hier nog op de combinatie van invoer en uitvoer signalen.

Spot Allocatie
Het accent van dit project ligt op de vraag welke beschikbare parkeer plek nu toegewezen moet worden. Om deze vraag te beantwoorden zijn er verschillende algoritmes voorgesteld. Een evaluatie functie en enkele test sets zijn gemaakt om te meten welk algoritme onder welke omstandigheden beter functioneert. Hieruit blijkt dat het close-to-the-exit algoritme vaak iets betere resultaten geeft.

Architectuur
Nadat gekeken is naar invoer en uitvoer signalen, en enkele toewijs algoritmes, is er een architectuur gemaakt. Verschillende functies zijn geadresseerd en verschillende componenten zijn gedefinieerd, evenals de communicatie tussen deze componenten.

Ontwerp en implementatie
Elk component uit de architectuur heeft een beschrijving gekregen zodat ze geïmplementeerd konden worden. De componenten zijn geïmplementeerd in twee iteraties. Eerst is het skelet van het systeem gemaakt, waarin onder andere een voor gedefinieerde parkeer garage zat. In de tweede iteratie zijn de tekortkomingen aangepakt zodat verschillende parkeer garages gedefinieerd konden worden met behulp van een Car Park Markup Language (CPML) file.

Simulatie
Het systeem kan de meeste legale auto bewegingen nabootsen. De parkeer garage kan worden gevuld in elke wenselijke configuratie en verschillende toewijzing algoritmen kunnen worden getest. De kosten functie kan de efficiëntie van een bepaalde toewijzing uitlezen zodat de verschillende algoritmes kunnen worden vergeleken onder verschillende omstandigheden.

Conclusie
Om te concluderen, auto's kunnen worden verwezen naar toegewezen area's door het kortste pad naar een area te berekenen. Een aanbevolen area kan worden berekend door de voorgestelde algoritmes. De simulatie welke is gemaakt kan deze algoritmes testen en de kosten functie kan ze vergelijken. Verschillen zijn klein maar er is zeker wat te winnen ten opzichte van de oude situatie.
13. **Glossary**

*allocated spot*: a spot is called allocated when there is no car parked at the spot, but the system is guiding a car to that specific spot.

*area*: a predefined group of spots.

*available spot*: see vacant spot.

*car park*: a building or area that contains spots where cars can be parked. Spots can be grouped in areas. A car park has at least one entrance where cars can enter and exit, and one exit, where passengers can exit and enter.

*entrance*: a location in the car park where cars can enter or exit the car park

*exit*: a location in the car park where passengers can exit or enter the car park

*full*: a car park is called full or filled, if all the spots are occupied, allocated or reserved. This means that no spot is vacant.

*graph*: Mathematical structures used to model pairwise relations between objects from a certain collection. A "graph" in this context refers to a collection of vertices and a collection of edges that connect pairs of vertices. In a directed graph, the edges are directed. [DIE97] [BON76]

*input*: The input is everything that can be measured, like user actions, sensors for vacant spots etc. Possible inputs are pressure or vision sensors, video camera streams, cellphone position, a car counting tile or some authentication item that a user can carry.

*spot*: a certain amount of space in a car park, often surrounded by white stripes, that can hold exactly one car.

*spot state*: A parking spot knows the following states: vacant -> allocated -> occupied -> vacant

*occupied spot*: a spot is called occupied if and only if a car is parked on the spot.

*output*: every manner used to communicate to the user, for guiding him to a specific spot. Possible output is a LED-path, laser guidance or displays.

*reserved spot*: a spot is called reserved when someone indicated that he preferred the spot, over the internet for example. The spot will not get allocated for someone else during the reserved time frame.

*unavailable spot*: see occupied spot.

*vacant spot*: a spot is called vacant when there is no car parked on the spot, and the system is not guiding any car to this spot at the moment.
References

14. **Appendices**

14.1. **Parking Behavior**

In order to choose or design a working scheduling algorithm, a nearby car park was monitored for several days. Results are grouped in the table below. We divide the car park in three equal sized sectors: S1, near the exit; S2, near the middle and S3, near the entrance. We record the crowdedness on different times.

<table>
<thead>
<tr>
<th>Day</th>
<th>Time</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monday</td>
<td>9:00</td>
<td>90%</td>
<td>20%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>10:00</td>
<td>100%</td>
<td>40%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>11:00</td>
<td>100%</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>12:00</td>
<td>90%</td>
<td>80%</td>
<td>30%</td>
</tr>
<tr>
<td>Tuesday</td>
<td>9:00</td>
<td>100%</td>
<td>60%</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>10:00</td>
<td>100%</td>
<td>90%</td>
<td>80%</td>
</tr>
<tr>
<td></td>
<td>11:00</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>12:00</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Thursday</td>
<td>9:00</td>
<td>80%</td>
<td>90%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>10:00</td>
<td>100%</td>
<td>90%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>11:00</td>
<td>100%</td>
<td>90%</td>
<td>90%</td>
</tr>
<tr>
<td></td>
<td>12:00</td>
<td>90%</td>
<td>90%</td>
<td>80%</td>
</tr>
</tbody>
</table>

*Table 19: Crowdedness estimation*

The percentages in the table above are no hard figures, but only estimations. It's pretty clear where the more popular spots are located on any given day: S1, near the exit. This is the reason why these spots or areas closer to the exits have higher priorities for the scheduling algorithm.
14.2. Inspiration

14.2.1 Scheduling Processes

There are various concepts in computer science which can be compared with picking the right parking spot. One of them is process scheduling [OS]. If we would compare the cars to processes and spots to resources, the problem of picking a parking spot out of a set of all vacant parking spots, suddenly sounds like a classic scheduling problem. There are three major differences though: (1) usually, there are more processes than resources, (2) a process queue is not desirable in a car park situation and (3) there is no way a car can be moved from a spot back into a queue (or another spot). If the car park is full, there is a very low chance that a driver is willing to wait in a queue for a spot to become vacant.

There are three types of scheduling available in modern operating systems [OS]. These are long-, medium- and short-term scheduling. Long-term scheduling is used when a task is about to get started. When the resource utilization is low, that is, resources are still available some part of the time, a task can get started. If the resources are already occupied by earlier tasks, the new task can not be started. Long-term scheduling is all about available resources over time.

The second form is called medium-term scheduling. This kind of scheduling hits the scope when a task is already started, but there is no space left in memory to write the results to, or some other task needs more space in memory. A medium-term scheduler can now decide to swap the tasks data to another storage level, the harddisk for example. When the task needs the data again, it can be swapped back to the system memory. Note that medium scheduling is not about resources over time, but resources over space.

The last kind of scheduling, short-term scheduling assigns a task to a resource. It decides what to execute next. There are several reasons to switch to another task like, when a task must wait for another resource, when an event occurs or when a task terminates. The short-term scheduler is obviously the busiest of the three. Short-term scheduling is also about resources over time. So, if we want to compare picking a parking spot with scheduling a task, we might want to have a closer look at medium-term scheduling. In other words, memory allocation might be an interesting topic to have a look at. We will do that in the following section.

A typical scheduling algorithm takes a number of preferences into account, which often conflict with each other. In the process context, one could think about I/O throughput, CPU utilization, response time, urgency of fast response, priority, maximum time allowed, total time required and so on. In the car park context, we could think about minimizing the driving or walking distance, (and so the time spent in there), maximize distributivity or maximize convenience. Note that these preferences can not easily be mapped from one context to another.

The reason for this is that scheduling is more about when, not so much about where. Scheduling picks something out of a queue of options and assigns it to a certain resource for a certain time. Only medium-term scheduling makes some sort of distinction between different kinds of memories but this is probably not enough to useful map it to our problem.

Generally, when we talk about scheduling in computer science, there are a number of variables which influence in which group of scheduling algorithms we would have to look for a fitting scheduling algorithm. Let’s have a look at these variables.

1. preemption. Our system is non-preemptive. A car is assigned to a certain spot, it can not be removed from that spot until the its parking time is finished. This variable does not occur in our solution.
2. precedence relations. eCarPark does not have any precedence relations. This variable does not occur in our solution.

3. release times. Because there are no known release times in our system, this variable does not occur in our solution.

4. restrictions on processing times. Because those kind of restriction do not apply to our problem, this variable does not occur in our solution.

5. deadlines. Again, our problem is about assigning space, not time. This variable does not occur in our solution.

As you can see above, non of these variables occur in our solution, which means that non of the known scheduling algorithms, existing in the predefined groups, apply to our problem. Maybe we should look to assigning spots not from a process scheduling point of view, but form a memory allocation point of view.

14.2.2 Memory Allocation

What does memory allocation have in common with parking spot allocation more than process scheduling? Well, memory allocation is more about space, where process scheduling is more about time. When a program needs a certain amount of space, it asks the operating system for this amount of free space. The operating system will than tell the program where to start using this space, unless it is not available. Doesn't this sound like finding a vacant parking spot?

There are various ways this free space can be managed, but that is not really the problem here. The problem is why to choose a certain place in memory over another place that also meets the requirements. So let's have a look into malloc, the C(++) memory allocation function [MEM].

When malloc is called, it scans through a list of pointers, each pointing to available space of a certain amount. When a point in the list is reached, where a pointers points to a equal or larger space then needed, this pointer is stored. The amount of requested space is extracted from the available chunk, and a new pointer will occur in the list, right after the requested chunk. Now, the original pointer can be released. The illustrated below shows this for malloc(96).

Illustration 23: Example of malloc(96)
The space can be freed again using the free() function, as showed in the following illustration.

Illustration 24: Example of free()

Note that there are a couple of problems with this approach. First of all, malloc will use a chunk of memory which is big enough, and split the requested space of. The result of this is that the memory gets fragmented in smaller and smaller pieces. This can result in situations where there is more than enough memory available, but it comes in chunks too small to use. Proposed solutions for this are to allow the OS to grow the amount of memory that can be dynamically addressed, or choose the chunks more wisely, the smallest one that fits. This last solution brings more problems. The algorithm has to search the entire list, or the list should be kept sorted.

So how can this approach be mapped to allocating parking spots. Well, first of all, we can conclude that the size problems do not really matter because every car has roughly the same size. So, instead of counting in bytes, we count in spots and those will be allocated not more than one at a time. So, if we keep a list of available spots, it is easy to find one that can offer enough space to park a car in.

Finding space that is at least as large as the requested space, is the only problem malloc, and the entire memory management problem solves. Except for the size of available chunks there are no reasons to choose one available chunk over another. So taking a look at memory allocation did also not really help finding a solution for our problem. Let's take a look at cache versus RAM whether we can find any inspiration there.

### 14.2.3 Cache Versus RAM

There are different levels of memory in a computer. In many cases we will see the following three levels: cache, RAM (Random Access Memory) and a swap file. Cache is the memory close to the CPU. There can be several levels of cache but we will not go into that. Cache memory is usually small and extremely fast. RAM is not as fast as cache, and not as close to the CPU. It comes in larger amounts. When information can no longer be stored in cache memory, it can be stored in RAM. When the RAM is completely filled, the swap file on the harddisk can be used. How does a computer decide on which level certain data is stored. There are various so called cache algorithms that can take care of it, but let's talk about cache a little more first.
Cache can be used in three cases: to get the next instruction, to read needed data or to write a result to. Not every instruction can be stored in the cache, but it is not really a problem to guess which instruction to put in the cache. In case the cache runs out of new instructions, it takes quite some time to load them from the RAM, and the CPU can only wait. The same story holds for data to read. Not all data might be present in the cache, so an algorithm somehow has to decide which data will be present in the cache. Writing is not much of a problem. As long there is space in the writing queue, some result can be put into that queue.

So the important thing a cache algorithm has to take care of is that the data to read is present. There are basically four groups of algorithms that try to solve this problem:

A group called Least Recently Used assumes that the datum referred to is least likely to be read next and removes it from the cache. A group called Least Frequently Used assumes that the least frequently used datum is least likely to be read next so deletes that one from the cache. Belady Min deletes the datum that will never be read again. Of course it is impossible to implement this in hardware. The last group of algorithms, Adaptive Replacement Cache, constantly balances between recency and frequency. There are of course other things to consider like cost, size and time, but this is of less importance in the car park environment.

Could those algorithms be of any use considering finding a parking spot? Maybe not when we see a register as a parking spot and a datum as a car. Cache algorithms don't really manage the data, they manage the registers. And we want to manage the cars. If we would see the spots as data, the algorithms above suddenly make a lot more sense. For example, we could choose to guide a car to the spot that was least recently recommended. On the other hand, we could also recommend the spot that was most recently recommended (and also vacant). We could do the same trick for the lowest and highest frequencies of recommending. Especially the most recent and most frequent recommended spot will overlap highly with recommending on grounds of a popularity index.
14.3. **Reply SpotScout, Inc.**

- How do you "measure" whether a certain spot is vacant or occupied?
  A: It will ultimately be up to space providers (SpotCasters) to keep track of their space(s)—what is available and when. That being said, SpotScout will be privy to arrival and departure times. It will be up to the space provider to make sure that the space is in fact available for an incoming driver (SpotScout) at the designated arrival time.

- Can you guarantee that a reserved spot won't be used by someone else than the reserver?
  A: No. Like eBay, you will be able to rate the quality of service you experience with a particular SpotCaster. Those SpotCasters who do not in fact make their spaces available will have a hard time doing business in the long run.

- Can you guarantee that someone will actually park on the spot where you want him to park?
  A: Again, the answer is no. But just as SpotCasters are rated, so are SpotScouts. Poor SpotScouts will have a hard time finding people willing to rent them a space.

ParkeerVak is een organisatie die bedrijven in de parkeersector bij elkaar brengt. Ik ben een van hun jaarlijkse bijeenkomsten bijgezeten en dit is wat ik vond (in het Nederlands).

Keyprocessor Scanton is gespecialiseerd in geoïntegreerde elektronische beveiligingsystemen en ook actief in de parkeermarkt, waar zij een speler van vormaat beweren te zijn. Na het probleem van het lang zoeken naar een parkeerplek te hebben uitgelegd, legden ze mij uit dat dit niet als probleem werd gezien. Een exploitant zou volgens hen geen probleem vinden wanneer klanten langs de parkeergarage rijden, op zoek naar een vrije plek. Meerdere bedrijven kwamen met ditzelfde verhaal.

Wanneer ik aangaf dat het niet om een commerciële parkeergarage zou gaan, maar om een faciliteit welke aan werknemers aan zou worden geboden, veranderde hun mening niet. Als werknemers er 10 minuten over doen om een parkeerplaats te vinden, moeten ze maar 10 minuten eerder vertrekken thuis. Bovendien zijn dit naar hun zeggen ook altijd dezelfde mensen die later zijn, en daardoor moeite hebben om de parkeerplek te vinden.

Een systeem dat werknemers zou helpen bij het vinden van een vrije plek wordt vaak gezien als extra secundaire arbeidsvoorwaarde. Vanwege de hoge prijs van een dergelijke voorwaarde, zou dit systeem minder interessant kunnen zijn. In het kort, een dergelijk systeem zou volgens veel spelers in de markt als onzinnig worden gezien.

Wel wisten meerdere bedrijven te melden dat er verschillende systemen zijn waar per parkeerplaats met een simpel lampje wordt aangegeven of deze nog vrij is. Ook de parkeerplaats bij het vliegveld van Antwerpen zou al een dergelijk systeem hebben.

Na enkele vragen en discussies over het feit hoe auto's geteld zouden kunnen worden, kwamen de volgende antwoorden naar boven. Veel bedrijven stelden dat dit visueel te doen. De parkeergarage hangt wegens beveiligingsdoeleinden toch al vol met video camera's, dus deze kunnen net zo goed gebruikt worden om auto's te tellen. Het spreekt voor zich dat deze camera's dan wel met dit doel op moeten worden gehangen.

Jade B.V. is een bedrijf dat gespecialiseerd is in detectie systemen. Op de vraag wat voor producten zij hadden om auto's te detecteren die een area in of uit rijden lieten zij enkele producten zien waarvan er een wel erg interessant leek. Het product kon in twee richtingen meten of er auto's voorbij kwamen, en kon geoïntegreerd worden in een groter systeem.

Tenslotte sprak ik Peter de Jong van Nedap. Hij zag wel degelijk iets in het verwijzen van auto's naar specifieke plaatsen of delen van de parkeergarage. Volgens hem was het echter niet de allocatie waar het probleem in zat, maar juist het tellen van de auto's. Volgens Peter is het tellen per area geen goed idee omdat mensen buiten de vakken kunnen parkeren. Daardoor zou het systeem kunnen denk en dat er nog een plek vrij is, en die keer op keer al meer, terwijl er geen auto meer bij kan staan. Wanneer auto's geregeld moeten worden, moet dat dus per plek, wat een hoop extra hardware en dus kosten met zich meebrengen. Vervolgens zei Peter dat je er niet naar moet streven elke area vol te krijgen. Ondieke plekken zijn nu eenmaal minder gewild dan anderen. Als voorbeeld nam hij een plek die vlak bij de ingang was. Om hierin een auto te parkeren is een manoeuvre nodig die enige tijd in beslag zou kunnen nemen. Het overige verkeer zal hierdoor worden gehouden. Ten eerste wil je om die reden de plek niet aanraden, omdat er dan files ontstaan. Mocht de plek toch aanraden, laat zeggen aan iemand met een bepaalde vorm van parkeer vrees, zal hij of zij de auto daar toch niet willen parkeren.
14.5. CPML-file

This appendix lists the example CPML-file of the example car park, and the source code of the program that visualizes this CPML-file.

```
<carpark>
  <connect>C</connect>
  <area id="C">
    <spot id="301" x="18" y="1"></spot>
    <spot id="303" x="17" y="1"></spot>
    <spot id="305" x="16" y="1"></spot>
    <spot id="307" x="15" y="1"></spot>
  </area>
  <connect>A</connect>
  <connect>D</connect>
</area>

<area id="A">
  <spot id="201" y="1" x="14"></spot>
  <spot id="202" y="3" x="14"></spot>
  <spot id="203" y="1" x="13"></spot>
  <spot id="204" y="3" x="13"></spot>
  <spot id="205" y="1" x="12"></spot>
  <spot id="206" y="3" x="12"></spot>
  <spot id="207" y="1" x="11"></spot>
  <spot id="208" y="3" x="11"></spot>
  <spot id="209" y="1" x="10"></spot>
  <spot id="210" y="3" x="10"></spot>
  <spot id="211" y="1" x="9"></spot>
  <spot id="212" y="3" x="9"></spot>
  <spot id="213" y="1" x="8"></spot>
  <spot id="214" y="3" x="8"></spot>
  <spot id="215" y="1" x="7"></spot>
  <spot id="216" y="3" x="7"></spot>
  <spot id="217" y="1" x="6"></spot>
  <spot id="218" y="3" x="6"></spot>
  <spot id="219" y="1" x="5"></spot>
  <spot id="220" y="3" x="5"></spot>
  <spot id="221" y="1" x="4"></spot>
  <spot id="222" y="3" x="4"></spot>
  <spot id="223" y="1" x="3"></spot>
  <spot id="224" y="3" x="3"></spot>
  <spot id="225" y="1" x="2"></spot>
  <spot id="226" y="1" x="1"></spot>
  <spot id="227" y="1" x="1"></spot>
  <spot id="228" y="4" x="4"></spot>
  <spot id="229" y="2" x="1"></spot>
  <spot id="230" y="4" x="4"></spot>
  <spot id="231" y="3" x="1"></spot>
  <spot id="232" y="4" x="1"></spot>
  <spot id="233" y="4" x="1"></spot>
  <spot id="234" y="5" x="1"></spot>
  <spot id="235" y="5" x="1"></spot>
  <spot id="236" y="7" x="1"></spot>
  <spot id="237" y="7" x="2"></spot>
  <spot id="238" y="7" x="2"></spot>
  <spot id="239" y="7" x="2"></spot>
  <spot id="240" y="7" x="2"></spot>
  <spot id="241" y="7" x="3"></spot>
  <spot id="242" y="4" x="3"></spot>
  <spot id="243" y="7" x="4"></spot>
  <spot id="244" y="7" x="4"></spot>
  <spot id="245" y="7" x="5"></spot>
```
```
<spot id="232" y="4" x="6"></spot>
<spot id="247" y="7" x="6"></spot>
<spot id="234" y="4" x="7"></spot>
<spot id="249" y="7" x="7"></spot>
<spot id="236" y="4" x="8"></spot>
<spot id="251" y="7" x="8"></spot>
<spot id="238" y="4" x="9"></spot>
<spot id="253" y="7" x="9"></spot>
<spot id="240" y="4" x="10"></spot>
<spot id="255" y="7" x="10"></spot>
<spot id="242" y="4" x="11"></spot>
<spot id="257" y="7" x="11"></spot>
<spot id="244" y="4" x="12"></spot>
<spot id="259" y="7" x="12"></spot>
<spot id="246" y="4" x="13"></spot>
<spot id="261" y="7" x="13"></spot>
<spot id="248" y="4" x="14"></spot>
<spot id="263" y="7" x="14"></spot>
<connect>D</connect>
</area>
</area id="D">
<connect>C</connect>
<area id="E">
<spot id="302" y="4" x="17"></spot>
<spot id="304" y="5" x="17"></spot>
<spot id="306" y="6" x="17"></spot>
<connect>E</connect>
</area>
</area id="E">
<connect>D</connect>
<area id="B">
<spot id="101" y="8" x="14"></spot>
<spot id="102" y="11" x="14"></spot>
<spot id="103" y="8" x="13"></spot>
<spot id="104" y="11" x="13"></spot>
<spot id="105" y="8" x="12"></spot>
<spot id="106" y="11" x="12"></spot>
<spot id="107" y="8" x="11"></spot>
<spot id="108" y="11" x="11"></spot>
<spot id="109" y="8" x="10"></spot>
<spot id="110" y="11" x="10"></spot>
<spot id="111" y="8" x="9"></spot>
<spot id="112" y="11" x="9"></spot>
<spot id="113" y="8" x="8"></spot>
<spot id="114" y="11" x="8"></spot>
<spot id="115" y="8" x="7"></spot>
<spot id="116" y="11" x="7"></spot>
<spot id="117" y="8" x="6"></spot>
<?php
    define("ARROW", 10);

    function startElement($parser, $name, $attrib) {
        global $area;
        global $spot;
        global $stack;
        global $thisarea;

        $stack[count($stack)] = $name;
        switch ($name) {
            case $name == "AREA" :
                $thisarea = $attrib["ID"];
                break;
            case $name == "SPOT" :
                $spot[$attrib["ID"]]["x"] = $attrib["X"];  
                $spot[$attrib["ID"]]["y"] = $attrib["Y"];  
                $area[$thisarea]["minx"] = isset($area[$thisarea]["minx"] ? min($area[$thisarea]["minx"], $attrib["X"]): $attrib["X"]);  
                break;
        }
    }

    function endElement($parser, $name) {
        global $stack;
        unset($stack[count($stack)-1]);
    }

    function characterData ($parser, $data) {
        global $stack;
        global $thisarea;
        global $connect;

        if (count($stack)==0) {return;}
        if ($stack[count($stack)-1]=="CONNECT") {
            $connect[] = array($thisarea, $data);
        }
    }

    function maxX($s) {

$x = 0;
foreach ($s as $spot) $x = max($x, $spot["x"]);
return $x;
}

function maxY($s) {
$y = 0;
foreach ($s as $spot) $y = max($y, $spot["y"]);
return $y;
}

function imagetriangle($im, $x1,$y1, $x2,$y2, $x3,$y3, $colour) {
$coords = array($x1,$y1, $x2,$y2, $x3,$y3);
imagefilledpolygon($im, $coords, 3, $colour);
}

function imagelinearrow($gd, $x1, $y1, $x2, $y2, $colour) {
$k = 20;
$dx = abs($x2 - $x1);
$dy = abs($y2 - $y1);
$len = sqrt(pow($dx, 2) + pow($dy, 2));
$alfa = $dx != 0 ? atan($dy / $dx) : 0;
$kx = cos($alfa) * $k;
$ky = sin($alfa) * $k;

// Make the arrows a little shorter
$x1 = $x1 > $x2 ? $x1 : $x1 + $kx;
$x2 = $x2 > $x1 ? $x2 - $kx : $x2;
$y1 = $y1 > $y2 ? $y1 : $y1 + $ky;
$y2 = $y2 > $y1 ? $y2 - $ky : $y2;
$gamma = atan((ARROW/2) / ARROW);
$beta = $alfa - $gamma;
$theta = $alfa + $gamma;
$n = sqrt(pow(ARROW, 2) + pow(ARROW/2, 2));

$dxl = $x1 == $x2 ? $x2 + cos($beta) * $n : ($x2 > $x1 ? $x2 : $x2 - cos($beta) * $n);
$dyl = $y2 > $y1 ? $y2 - sin($alfa) * $n : ($y1 > $y2 ? $y1 : $y1 + sin($alfa) * $n);

$dx = $x2 - $x1;
$dy = $dx != 0 ? atan($dy / $dx) : 0;
$dx = $dx > $x1 ? (abs($dx - $x1) != 0 ? $dx - $x1 : $x1) : ($x1 > $dx ? $x1 : $x1 + $dx);
$dy = $dy > $y1 ? (abs($dy - $y1) != 0 ? $dy - $y1 : $y1) : ($y1 > $dy ? $y1 : $y1 + $dy);

\}
<? ($x1 == $x2 ? $y2 - ARROW : $y2 - sin($beta) * $n)
  : ($x1 == $x2 ? $y2 + ARROW : $y2 + sin($beta) * $n);

$dxr = $x1 == $x2
  ? $x2 + ARROW/2
  : ($x2 > $x1
      ? $x2 - cos($theta) * $n
      : $x2 + cos($theta) * $n);

$dyr = $y2 > $y1
  ? ($x1 == $x2 ? $y2 - ARROW : $y2 - sin($theta) * $n)
    : ($x1 == $x2 ? $y2 + ARROW : $y2 + sin($theta) * $n);
  imageline ($gd, $x1, $y1, $x2, $y2, $color);
  imagetriangle($gd, $x2, $y2, $dxl, $dyl, $dxr, $dyr, $color);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////

$filename = "carpark2.xml";
$cpml = file_get_contents($filename);

$area = array();
$spot = array();
$thisarea="";

$xml_parser = xml_parser_create();
xm1_set_element_handler($xml_parser, "startElement", "endElement");
xm1_set_character_data_handler($xml_parser, "characterData");
xm1_parse($xml_parser, $cpml);
xm1_parser_free($xml_parser);

$maxX = maxX($spot);
$maxY = maxY($spot);

$gd = imagecreate($maxX * 20+11, $maxY * 20+11);
$grey = imagecolorallocate($gd, 0xEE, 0xEE, 0xEE);
$black = imagecolorallocate($gd, 0x00, 0x00, 0x00);
$green = imagecolorallocate($gd, 0x00, 0xDD, 0x00);
$blue = imagecolorallocate($gd, 0x00, 0x00, 0xFF);
$red = imagecolorallocate($gd, 0xFF, 0x00, 0x00);

foreach($spot as $v => $s) {
  imagerectangle ($gd, $s["x"]*20-20, $s["y"]*20-20, $s["x"]*20,
    $s["y"]*20, $black);
  imagestring($gd, 2, $s["x"]*20-20 + 2, $s["y"]*20-20 + 4, $v,
    $black);
}

include("class.rectangles.php");
$myRectangles = new Rectangles();

foreach($area as $v => $s) {
  $rect[0]["x"] = $s["minx"]*20-17;
def closestVertex(G, P, start):
    """This function finds the vertex where a certain predicate
    vacant() holds and which is closest to a certain starting point
    G is a dictionary that contains the graph that describes the
    parking lot.
    P is a dictionary that describes the number of vacant spots per
    vertex
    start is the name of the vertex where to start, in our case,
    where the exit is located
    """
    if vacant(P, start):
        return start
    D = {}
    X = {}
    X[start] = 0
    while X != {}:
        #print "X:", X
        k = getClosest(X)
        if vacant(P, k):
            return k
        for key in G[k].keys():
            if not X.has_key(key) and not D.has_key(key):
                X[key] = G[k][key] + X[k]
                D[key] = X[k]
        #print "D:", D
        del X[k]
        del D[k]

- 81 -
14.6.2 Pure Random

```python
def random2(G, P):
    V, total = {}, 0
    for p in P:
        total += P[p]["VACANT"]
        V[p] = total
    r = random.randint(0, total-1)
    for p in P:
        if r < V[p]:
            return p
```

14.6.3 Fairly Distributed

```python
def faioredistribute(G, P):
    T = {}
    for p in P:
        total = P[p]["VACANT"] + P[p]["OCCUPIED"] + P[p]["ALLOCATED"] + P[p]["RESERVED"]
        T[p] = P[p]["VACANT"] / float(total)
        maximum = p
    for t in T:
        if T[t] > T[maximum]:
            maximum = t
    if P[maximum]["VACANT"] > 0:
        return maximum
    else:
        return None
```

14.6.4 Last Freed Spot

```python
class MostRecentFreed:
    Freed = []

    def init(self, P):
        for p in P:
            self.Freed.append(p)

    def allocate(self, P):
        if self.Freed == []:
            self.init(P)
        for f in self.Freed:
            if vacant(P, f):
                return f
        return None
```
14.6.5 Most Recent Allocated

class MostRecent:
    R = []
    def init(self, P):
        for p in P:
            self.R.append(p)
    def allocate(self, P):
        if self.R == []:
            self.init(P)
        for r in range(len(self.R)):
            if vacant(P, self.R[r]):
                recommendation = self.R[r]
                del self.R[r]
                break
        return recommendation

14.6.6 Most Frequent Allocated

class MostFrequent:
    A = {}
    def init(self, P):
        for p in P:
            self.A[p] = 0
    def allocate(self, P):
        if self.A == {}:
            self.init(P)
        V = self.A.copy()
        recommendation = maxFreq(V)
        while recommendation and not vacant(P, recommendation):
            recommendation = maxFreq(V)
        if recommendation:
            self.A[recommendation] += 1
        return recommendation