The Self-aware City

by Andres Sevtsuk

Bachelor of Architecture (2003)
Ecole d'Architecture de la Ville et des Territoires à Marne-la-Vallée

Submitted to the Department of Architecture in Partial Fulfillment of the
Requirements for the Degree of Master of Science in Architecture Studies
at the
Massachusetts Institute of Technology
June 2006

© 2006 Andres Sevtsuk. All Rights Reserved.
The author hereby grants MIT permission to reproduce and to distribute publicly
paper and electronic copies of this thesis document in whole or in part in any
medium now known or hereafter created.

Signature of Author……………………………………………………………
Department of Architecture
May 17, 2006

Certified by………..………………………………………………………………
William J. Mitchell
Professor of Architecture and Media Arts & Sciences
Thesis Supervisor

Approved by……..…………………………………………………………………
Julian Beinart
Professor of Architecture
Chair Department Committee on Graduate Students

Thesis Reader
William Lyman Porter
Professor Emeritus of Architecture

Thesis Reader
Carlo Ratti
Director of SENSEable City Laboratory at MIT

The Self-aware City

by Andres Sevtsuk

Submitted to the Department of Architecture in Partial Fulfillment of the
Requirements for the Degree of Master of Science in Architecture Studies
at the Massachusetts Institute of Technology, June 2006

Abstract

This thesis explores the idea of real-time urban space management. While increasing amounts of real-time information about the city, specifically the location of people and resources, appear, it becomes necessary to explore how different strategies of distributing real-time location information can be used as urban design tools for a more sustainable resource allocation.

I focus on the study of street-parking, a system that clearly has a market situation with demand and supply, but due to lack of information is poorly managed today. I argue that an equilibrium state of the parking market in popular areas, similar to many other urban space markets, is a frequent over demand. The important challenges are therefore allocation optimization and queuing management. I propose five different strategies of using real-time location information to reduce search times and analyze the system through computer simulations and logic. Borrowing ideas from Game Theory, I try to illustrate how collaborative behavior between drivers could yield most efficient results from both the individual and the group point of view. Lastly, I outline some challenges that the use of real-time information systems introduce to the realm of urban design in general.

Thesis supervisor: William J. Mitchell
Title: Professor of Architecture and Media Arts & Sciences

Acknowledgements

I am grateful to all the faculty and friends at MIT, in France and in Estonia who have generously contributed to the development of this thesis. In particular I would like to thank my committee: my thesis advisor William J. Mitchell whose support has been invaluable and whose brilliance has led me to many of the ideas below, William Porter who has been the most intellectually supportive and honest reader and Carlo Ratti whose excellent rationality has helped me develop the models and writing presented hereafter.

Many people have enlightened me with interesting discussions and constructive criticism, deepening my interest towards architecture, cities, technology, artificial systems and people. I would like to thank Jean Pascal Ollivry in Estonia for his advice in mathematics, Donald Shoup at UCLA for his insights on parking, Marvin Minsky at MIT for his brilliance in analyzing people and artificial systems, Eric Klopfer and Mitchell Resnick at MIT for their precious advice on StarLogo, Michael Batty at UCS in London for his comments and discussion on urban simulation, Dennis Frenchman at MIT for continuous support and Julian Beinart, my advisor in the department, for challenging me to be critical.

I am also indebted to my dear friends Leonardo Shieh and Talia Dorsey as well as the students and researchers at the SENSEable City Laboratory and the Smart Cities group at MIT who have contributed to the development of this work.

I would like to contribute this to my parents and my brother.

Contents

Acknowledgements……………………………………..…………...…....……7

Introduction………………………………………………………….……..…..11

Chapter One. The Effects of the Street-parking System Today…..…....…....17

How Goal Switching Enhances the Search…………………………...……..…..24

Some Things We Do When Searching for Parking………...……….…………..28

Improving the Current Street-parking System ………...………………………..31

Benefits of a Real-time Guidance System………………….…………………..…40

Chapter Two. The Simulation Approach………………………………….......47

The Rules of the Simulation……..………………………………………………..53

General Rules……………………………………………………………………57

Traditional Parking Search Model……………………………………………....59

Intel Parking Search Models………………………………………………….60

Intel_ 1 Parking Search Model…………………………………………………62

Intel_3 Parking Search Model…………………………………………………..63

Intel_5 Parking Search Model…………………………………………………..66

Intel_7 Parking Search Model……………………………………………..…68

Critical Variables ………………………………………………………….…71

Optimum Strategy Versus *Satisficing* Strategy……………………………..…..73

Hazards of the Simulation ……………………………………………………75

Chapter Three. Results..…..79

Simulation Conclusions…………………………………………………….....96

Intel_9: The Collaborative Equilibrium and Game Theory…………………..…99

Importance of Efficient Queuing in Real-time Systems………………………105

Dynamic Queuing…………………………………………………………...109

Static Queuing……………………………………………………………….110

Chapter Four. Conclusions……………………..………………....…..…...…117

References …………………………………………………………………..123

List of Illustrations………………………………………………………………..126

Appendix ……………………………………………………………………129

Introduction

"The behavior of an artificial system may be strongly influenced by the limits of its adaptive capacities- its knowledge and computational powers". [Herbert Simon, p.29, *Sciences of the Artificial*]

The evolution of human society is built upon the interaction of people. More than any other species', peoples' interactions have created economies and societies, cities and countries. City form is an arena for this interaction. One of the primary tasks of city design is therefore to maximize the use of urban space and resources in order to foster interactions between people and places. Optimization of urban resources has always been a fundamental design challenge for urban designers. Five decades after the dawn of the digital computer, communication and computation offer new opportunities for optimizing the use of urban space.

This thesis is an exploration of using real-time urban information to affect existing relationships between citizens and urban resources. The purpose is to achieve a more intense and sustainable resource allocation. A well-planned distribution of urban resources could lead to significantly smaller zoning requirements of urban infrastructures. I use the term *resources* relatively loosely in this context to signify the functional infrastructure elements of a city that are accessible to the public and for which there is generally a great demand or competition. Such elements are public transportation, curb-side parking spaces, assembly spaces, etc. These elements can be fixed in space (parking spaces, meeting spaces) or moving (public transportation, taxis), but they are all publicly used by a relatively large number of people in daily urban life. I am interested in analyzing how some of these infrastructural systems could acquire different patterns of use if people had real-time information of their availability through portable communication devices.

Specifically I try to illustrate how augmented computational power enables individuals to navigate more efficiently in a complex external world. Already existent, ubiquitously dispersed personal communication devices can be exploited as a network of computational infrastructure for real-time urban resource allocation. I argue that modifying this dispersed communication infrastructure at a personal level can drastically change the interaction between people and places on an aggregated level. I regard such system optimization as fundamentally urban design, which explores alternative futures of how things *could be*. However, this design activity does not explore state descriptions that are proposals for physical states of a city, but rather process descriptions, which similar to differential equations, propose various ways of using information, depending on the goals and the feedback from the environment. Related studies in science[1] have long proven that complex dynamic systems are highly dependent on their initial conditions, and that by altering these conditions, very different dynamic patterns emerge. In the analysis and design of alternative space allocation systems, I shall focus in detail on the universe of internal and external variables that affect the system of street-parking. Specifically I shall argue that in addition to adding a new layer of information to enhance searching for parking, feeding the performance information of group efficiency selectively back to the participants of the system in real-time, can create incentives for collaborative action and can significantly impact people's decision making and distribution in a city. Collaborative behavior at the group level can be further encouraged through dynamic pricing, by offering lower fees to people who are willing to cooperate. By accurately matching demand and supply, I shall propose different strategies of using real-time information for the distribution of public resources and explore how these strategies could help establish their more sustainable allocation.

---

[1] Determenistic Nonperiodic Flow [Edward Lorenz, New York Academy of Sciences 1963]

To test my assumptions and strategies, I use an agent based simulation model[2] and analyze the resulting effects from the model. I study which variables in the simulation model are most critical for good performance and which circumstances jeopardize efficient functioning. My goal is to explore which approaches of real-time information use yield the most efficient distribution of the studied resource.

However, in order to narrow down a vast field of possibilities, I shall mainly focus on an example of a real-time guidance system for street-parking. Many of the more general issues of real-time information allocation will hopefully emerge through this example. In the conclusion I shall eventually come back to a more general discussion and illustrate the implications of this work to other areas of urban planning.

To begin with, Chapter One will discuss how the search for parking works at an individual and aggregate level today. Why is the search so troublesome? There are many valuable strategies that drivers exploit when searching for a parking space, which supposedly increase our capacity to find parking in almost hopeless situations. These strategies generally reduce the time spent cruising, and offer significant insight for the design of a real-time guidance system. Nevertheless, there is room for improvement in the search methods today. First, the current search strategy is based on locally perceptive information, which can lead to results that are only as good as the information perceived from the environment. Broader information could enhance the effectiveness of the search. Secondly, even with the present information, I shall suggest that the current cognitive search method[3] might not yield the best possible results. Decision making, based on immediate perception of information, and not statistical calculations, can lead

[2] An individual based computer modeling technique that allows multiple agents to interact with each other and their surrounding environment.
[3] Read: intuitive searching for curb parking that we regularly use today.

to non-optimal results and jeopardize the performance of a search. I shall propose that computers might help to overcome these shortcomings, and discuss what particular aspect of the search process a real-time guidance system could enhance.

Chapter Two will discuss the general simulation approach by introducing some of the most common cellular automata and agent based modeling concepts for urban simulation. It will explain the particular technique used in this thesis for modeling distant telecommunication, that is, communication through electromagnetic waves over long distances. It shall then propose four different search algorithms that might complement the parking search processes used today. Detailed descriptions of the assumptions and rules of each search model are subsequently presented.

In Chapter Three I shall analyze the results and findings of simulation models. I shall compare the performance of the proposed search strategies and outline the effects of different environmental stimuli on the efficiency of each strategy. Besides the rational allocation strategies studied in the models, I shall propose an additional search strategy (*Intel_9*), which uses Game Theory to provide incentives for collaboration between parkers. I shall try to argue that a collaborative behavior between well informed drivers can be the most efficient way of reducing searching times. Towards the end of Chapter Three, I shall also turn to the question of how to cope with cars that simply can not be immediately allocated a parking spot due to a lack of available spaces. Efficient queuing in situations of over demand becomes a critical issue. Currently the excess cars circulate in traffic with all other vehicles, forming *dynamic queues*. I shall propose an alternative approach where cars could use temporary *static queuing* spaces while waiting for vacating parking spots.

Chapter Four is the conclusion. It will discuss the implications of the simulation results, offering suggestions for future work in the field of real-time urban resource management.

Chapter One

The Effects of the Street-parking System Today

An intoxication comes over the man who walks long and aimlessly through the streets. With each step, the walk takes on greater momentum; ever weaker grow the temptation of shops, of bistros, of smiling women, ever more irresistible the magnetism of the next street corner, of a distant mass of foliage, of a street name. [p. 417, Walter Benjamin 1999]

If the *flaneur* were to stroll around a contemporary American downtown, where would he find those crowds of bustling people and that spectacle of contemporary manners and urban scenes that are as essential to him as water for a fish? Where could the *flaneur* find the hustle bustle of city streets, the very heart of the crowd in a center of a metropolis, dense enough to hide himself and observe the modern urban scene with the eye of an artist? Perhaps in a car, searching for curb-parking?

Eighty seven per cent of all trips in this country are made in personal cars [Shoup, 2005]. Ninety five per cent of each car's lifetime is spent parked, and ninety nine per cent of all parking is free of charge in America [ibid.]. So it is no wonder that nobody wants to pay for parking in a downtown area either. Instead of choosing an available garage that might charge some $10 an hour, most people choose to *cruise*, hoping that they can find a cheaper alternative at the curb if they search long enough. As a result, up to 30% of all traffic in central business districts has been cruising for a cheap spot for decades [ibid.].

Year	City	Share of traffic cruising (percent)	Average search time (minutes)
1927	Detroit (1)	19%	
1927	Detroit (2)	34%	
1933	Washington		8.0
1960	New Haven	17%	
1965	London (1)		6.1
1965	London (2)		3.5
1965	London (3)		3.6
1977	Freiburg	74%	6.0
1984	Jerusalem		9.0
1985	Cambridge	30%	11.5
1993	Cape Town		12.2
1993	New York (1)	8%	7.9
1993	New York (2)		10.2
1993	New York (3)		13.9
1997	San Francisco		6.5
2001	Sydney		6.5
Average		**30%**	**8.1**

The numbers after Detroit, London, and New York refer to different locations within the same city.

Sources: Simpson (1927), Hogentogler, Willis, and Kelley (1934), Huber (1962), Inwood (1966), Bus + Bahn (1977), Salomon (1984), O'Malley (1985), Clark (1993), Falcocchio, Darsin, and Prassas (1995), Saltzman (1997), and Hensher (2001).

Figure 1 Cruising in the 20th century. Source: The High Cost of Free Parking [Shoup 2005]

To put this in perspective, let's listen to Donald Shoup:

"*Even a small search time can create a surprising amount of traffic. Consider a congested downtown area where it takes three minutes to find a curb space. If the parking turnover is 10 cars per space per day, each curb space generates 30 minutes cruising time per day, and if the average cruising speed is 10 miles an hour, each curb space generates five VMT (vehicle miles traveled) per day. As estimated…, the average block is surrounded with 33 curb parking spaces, so cruising for parking creates 165 VMT a day per block. Over a year, this amounts to 60, 000 VMT per block (equivalent to more than two trips around the earth). Because this cruising adds to already congested traffic, it makes a bad situation even worse*."

While cruising for a cheap parking space can bring great financial savings to a driver, the accumulating environmental cost of cruisers is much greater. Studies by Axhausen, Polak and Shoup prove that even a slight reduction of parking search time could significantly reduce environmental impacts of the current parking system [Axhausen, Polak 1991 and Shoup 2005].

The conventional planning response to congested traffic and time-consuming parking search is a provision of more off-street parking space. Most contemporary parking design guidelines demand property developers to host all the potential demand created by their property within off-street parking on the property. Instead of requiring the minimum, parking guidelines are usually set for extreme traffic situations that rarely occur. As a result, most off-street private parking lots are over dimensioned and remain underused a great deal of the time (comprehensive statistics are given on pp. 75-111 in "The High Cost of Free Parking", Shoup 2005).

As off-street parking lots in downtown areas usually charge relatively high fees, then their filling rates are diminished because of people preferring to search for ubiquitously under priced curb-side alternatives. This results in a vicious cycle where demand for street-parking causes congestion and keeps parking requirements from being reduced.

Off-street lots that rarely fill, already consume a surprising amount of urban land. In downtown Albuquerque, for instance, approximately 80% of land is taken up by off-street parking [ibid.]. As a result, such excessive requirements create sparse land-use and restrain building densities, degrading the pedestrian environment even further, which forms another vicious cycle by increasing the demand for driving. Large parking lots result in spread out developments, where even the social *flaneurish* aspect of today's cruising for a cheap curb-side space

loses its flavor- too much parking eventually eliminates the destinations that we drive to in the first place. Such strategy for solving traffic congestion can be successful from a personal savings point of view, but the hidden costs of ubiquitous free parking are unjust, and seen at an aggregated level, the resulting environmental impact is unacceptable. This description, that Shoup has outlined in much more detail than presented here, might sound like an exaggerated dooms-day scenario, but if Benjamin had been able to compare 19th century Paris to 21st century Los Angeles, it might seem disastrous indeed. Figure 2 below, illustrates some common parking coverage in world cities today. Figure 3, further down, shows the amount of land consumed by off-street parking around MIT. Compared to how dense cities like Boston were only a hundred years ago, these are no small indicators.

City	Land area (hectares)	Parking spaces	Parking spaces per hectare	Parking area (hectares)	Parking coverage	Employment (jobs)	Jobs per hectare	Parking spaces per job
(1)	(2)	(3)	(4) = (3)/(2)	(5) = (3)/325	(6) = (5)/(2)	(7)	(8)=(7)/(2)	(9)=(3)/(7)
1. Los Angeles	408	107,441	263	331	81%	206,474	506	0.52
2. Melbourne	172	42,601	248	131	76%	126,286	734	0.34
3. Adelaide	181	42,857	237	132	73%	73,868	408	0.58
4. Houston	392	72,797	186	224	57%	118,889	303	0.61
5. Detroit	362	65,639	181	202	56%	93,012	257	0.71
6. Washington	460	80,100	174	246	54%	316,723	689	0.25
7. Brisbane	117	19,895	170	61	52%	61,844	529	0.32
8. Calgary	298	45,260	152	139	47%	86,700	291	0.52
9. Portland	280	41,861	150	129	46%	103,872	371	0.40
10. Brussels	308	45,512	148	140	45%	144,906	470	0.31
11. Vancouver	337	46,053	137	142	42%	104,000	309	0.44
12. Edmonton	297	37,512	126	115	39%	63,200	213	0.59
13. Frankfurt	240	29,487	123	91	38%	119,735	499	0.25
14. Canberra	329	39,558	120	122	37%	22,521	68	1.76
15. Chicago	395	46,653	118	144	36%	363,794	921	0.13
16. Denver	636	67,757	107	208	33%	93,012	146	0.73
17. San Francisco	391	39,756	102	122	31%	291,036	744	0.14
18. Toronto	188	18,436	98	57	30%	174,267	927	0.11
19. Sydney	416	39,031	94	120	29%	175,620	422	0.22
20. San Diego	570	50,234	88	155	27%	72,964	128	0.69
21. Winnipeg	440	37,419	85	115	26%	68,593	156	0.55
22. Boston	868	73,604	85	226	26%	119,189	137	0.62
23. Ottawa	305	25,565	84	79	26%	111,031	364	0.23
24. Perth	759	63,000	83	194	26%	99,819	132	0.63
25. Phoenix	393	31,937	81	98	25%	35,267	90	0.91
26. Montreal	1,224	94,745	77	292	24%	273,203	223	0.35
27. Paris	2,333	172,000	74	529	23%	862,180	370	0.20
28. Munich	795	58,430	73	180	23%	219,518	276	0.27
29. Vienna	298	21,036	71	65	22%	112,770	378	0.19
30. Singapore	725	45,870	63	141	19%	280,000	386	0.16
31. Copenhagen	455	27,400	60	84	19%	122,770	270	0.22
32. Sacramento	462	27,677	60	85	18%	54,121	117	0.51
33. New York	2,331	138,148	59	425	18%	2,305,545	989	0.06
34. Hamburg	460	27,056	59	83	18%	152,590	332	0.18
35. Zurich	152	8,668	57	27	18%	63,410	417	0.14
36. Hong Kong	113	6,376	56	20	17%	193,520	1,713	0.03
37. Kuala Lumpur	1,625	86,030	53	265	16%	290,000	178	0.30
38. London	2,697	138,843	51	427	16%	1,142,781	424	0.12
39. Amsterdam	824	28,600	35	88	11%	80,722	98	0.35
40. Stockholm	424	13,050	31	40	9%	111,233	262	0.12
41. Seoul	2,117	59,758	28	184	9%	1,226,830	580	0.05
42. Bangkok	2,056	50,848	25	156	8%	271,944	132	0.19
43. Tokyo	4,208	98,755	23	304	7%	2,300,738	547	0.04
44. Manila	3,600	22,000	6	68	2%	815,400	227	0.03
Average	**828**	**53,074**	**100**	**163**	**31%**	**321,043**	**403**	**0.36**

Total parking area in column 5 is the surface parking area (in hectares) that all parking spaces in column 3 would occupy.
Each hectare of surface parking accommodates about 325 parked cars.
Source for CBD area and parking spaces: Kenworthy and Laube (1999, Chapter 3).

Figure 2 Parking in central business districts. Source: The High Cost of Free [Shoup 2005]

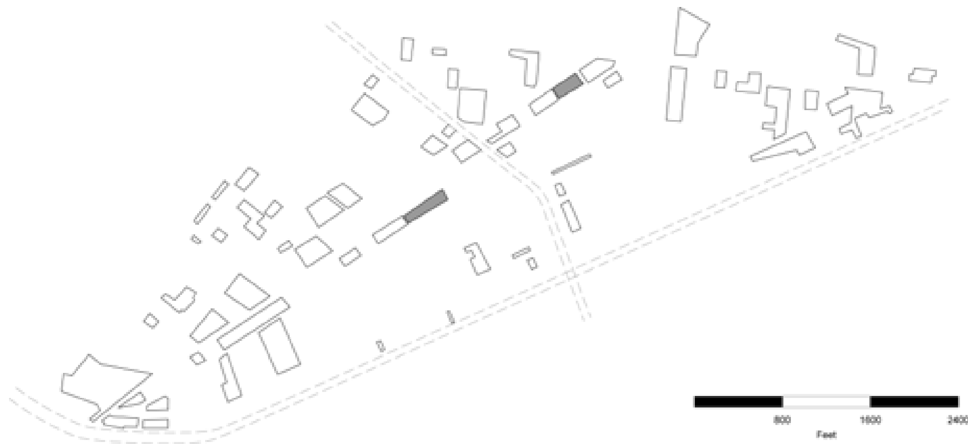


Figure 3 Map of off-street parking lots around MIT. Shaded areas indicate multi-story structures.

An alternative strategy for reducing current cruising seems to be offered by real-time information technology. If drivers knew the exact availability of street-parking in real-time, then they could efficiently find their closest parking spaces, without driving around searching, wasting energy, polluting air and congesting traffic. If the amount of searching cars exceeded the amount of available spots, then drivers could be alerted that their search is probably useless. In order to reach individuals directly, such information seems to be most useful if brought to drivers personally, displayed on their cell-phone screens, personal navigation devices in the car, or as voice directions. If this could be achieved, then could searching times potentially diminish? Could the turnovers of parking spaces increase? Would more cars be accommodated by the same number of parking spots and more people simultaneously occupy a C.B.D.? Off and on-street parking would of course both remain, but by maximizing their efficiency, zoning laws could be revised and their parking requirements could be lowered.

Figure 4 Aerial view of half-parking coverage north of Vassar Street at MIT

One of the potential hazards of this idea is that reduced search time might encourage more people to drive in central business districts. Pricing could be used as a mechanism for controlling demand, turning the rates higher when demand is high and lower if only a few vehicles search for parking. Hence, the real-time guidance system could also function as a free infrastructure for managing dynamic pricing of curb-side parking. Research by [Clinch, Kelly 2003] and [Shoup 2005] has shown how sensitive drivers are even to small fluctuations in pricing. Based on their evidence, and similar precedents in congestion pricing in London, Singapore and L.A., dynamic pricing[4] could offer a powerful tool for managing parking demand.

Whether or not these assumptions would hold in the real world depends on many variables, both technological and human, that might prevent the successful adoption of real-time urban resource management. I shall hypothesize which human requirements a real-time guidance system needs to account for and how

[4] Tolls that vary in real time in response to changing congestion levels.

different environmental conditions affect the performance of the system, using a simulation model to check the validity of my assumptions.

How Goal Switching Enhances the Search

"Occupants of vehicles searching for parking spaces are not doing 'perceiving their environment', they are doing 'searching for a parking space' [Watson 1999]." With this quotation, Laurier has argued that the cognitive mechanisms at work and the attention attributed to parking in the driver's mind are not the same as during normal driving [Laurier 2003]. Minsky talks about this phenomenon as "credit assignment" to different phenomena around us, depending on our current goals [Minsky, 2006]. When the goal of the driver is set to parking, then many senses that would normally be passive or doing other things, get mobilized to help with the search. Similarly, many environmental conditions, which would be overlooked by our senses in different actions, get assigned more "credit" if they are potentially useful for achieving the goal.

If the higher level goal of a driver is to find a space for the car, then the sub-goals prescribing the particular kind of space that is acceptable, are constantly changing. The relationship between a satisfactory parking space for a driver and the options an environment has to offer is a dynamic one, frequently shifting, depending on many concurrently active variables. Amongst many influences, the time spent on searching is itself a crucial factor that affects our ambitions, usually making us revert to less desirable or more expensive goals if over extended. Laurier, who conducted an ethnographic study of parking-search in London, emphasized the importance of this continuously adaptive aspect of the search, by showing in his study how a driver's goals constantly readjusted according to varying circumstances.

In Laurier's study, Mms. Marge, one of the subjects, was a delivery driver who had heavy boxes to deliver to a hotel lobby. Setting out on her daily route, she was anxiously hoping to have good luck and find a parking spot right in front of the hotel. Approaching the destination, she passed by one spot, but as it looked too small and was located several blocks away from the hotel, she decided not to take it. As she passed the hotel, she learned that parking right in front it was impossible. Hence she had to reconsider her strategy and try the next best option. She readjusted her goal to find a parking space close to the hotel on the same street and set out for a new round. She ended up driving several circles around the block, each time looking more attentively for people leaving, or other cues to help her accomplish her goal. By now her fellow passenger, the ethnographer himself had also engaged in helping her observe the environment by looking at side streets. After a couple of unsuccessful rounds they happened to pass by the small and distant spot they noticed at the very beginning again. This time they decided to consider the option seriously. Marge remarked that she was already late for her appointment at the hotel and estimated that further searching would extend her delay even more. They discussed whether the car would fit in the small space and not block the adjacent car's passenger doors and decided to go ahead and try. This option didn't seem unacceptable to them anymore, because they had learned that it would be hard to find anything better. They drove in, discovered that they didn't block the other car's doors and decided to park the car. The study illustrated how the driver's goals became progressive less selective as time went by and no optimal conditions emerged.

Time also determines the financial resources that a driver agrees to spend. For Marge, who was a delivery driver, and did not have a budget to spend on more expensive parking, a faster way of parking at a paid parking structure was already ruled out before she started her search. However, this is often times not the case. Many people spend a substantial amount of their income on parking fees. Needless to say, the ability to afford paid off-street parking does not imply that a

person will actually choose that costlier option. Instead, the option a person prefers is normally determined by a comparison of alternatives that an environment offers and the time that the person agrees to spend on searching. Hence, we could suggest that the choice of parking is dependent on the long term characteristics of the driver (financial resources, speed of life etc.), the momentary circumstances of a person (in a hurry, unwilling to search, heavy items to carry etc.), and the opportunities an environment offers. Certain drivers might always prefer to park at paid off-street spaces, while others only use paid parking if they are in a great hurry. Others, such as delivery drivers, might never use paid parking, even if they are in a great hurry. The exact behavioral psychology of drivers is yet impossible to predict with certainty, but studies demonstrate [Klinch and Kelly 2003], that on average, the amount a driver is willing to spend is inversely proportional to his available time.

Consider the examples below that illustrate how parking searching time and willingness to spend interact in daily life. For instance, imagine a scenario of a wealthy person with a meeting in a city center. As it happens to be during work hours, she does not have much time to spend looking for parking. She first drives towards her destination. A few blocks away she starts to drive slower to make sure she does not miss any vacant cheap spots. After passing her destination and making a second tour around the block, she slows down the car even more, almost to walking speed, but the cars behind her form a queue and force her to move faster. After having searched for five minutes, she decides to drive to a nearby parking lot that charges 5$ / hour instead of 50cents, but which offers her a spot immediately.

On another occasion, she might go shopping downtown on a Sunday afternoon. She might have a lot of spare time, and can therefore spend some of it searching for a cheap curb-side parking spot. The available spots she finds might be several blocks away from the stores she plans to go to, but even though she can afford a

closer space in a garage, she is not in a rush and will accept the walk, as well as the walk back with her shopping bags.

A week later, she is in a terrible rush, afraid of missing a train. While driving to the train-station she thinks over all the options but decides that she has no time to waste. When arriving at the station, she hastily looks around for vacant parking spaces, but seeing none, she drives directly in front of the station and uses the valet service to park her car. She knows that after a fifteen minute search she might be able to find a cheap space that would cost her less than a dollar, but afraid of missing her train, she decides not to take the chances and agrees to pay 8$ dollars for valet fees plus 10$ / hour at an off-street parking lot at the station.

From the driver's point of view, this capacity to use different strategies, adapting the goals along the way, is natural to anyone searching for parking. The actual choice of parking is not merely a clear outcome of a user's goals, but a multi-faceted product of the user's ambitions, time availability, financial resources and the environment's changing circumstances. If one strategy fails, then instead of wasting any further time applying it, a driver can modify her goals and test a different strategy. When designing a technological addition to the system, it is important to allow for such adaptive flexibility.

The list below outlines some common techniques and strategies we use when searching for street-parking today. The features in the list are mainly based on Laurier's study of street-parking [Laurier 2003] and my own empirical reflections on the process.

Some Things We Do When Searching for Parking:

Senses:

- Use the help of fellow passengers to strategize and observe. As the driver is forced to multitask between driving and searching, the person in the passenger seat or in the back can greatly increase the driver's scope of observation.

- Look for cues from other drivers on the street, who are also searching for parking, in order to understand their goals and learn from it.

- Increase attention paid to surroundings and the sharpness of senses as time moves on and the goal is not achieved. For example, an initially passive passenger might actively engage in the search over time.

- Observe pedestrians and other drivers on the street, who might be potentially leaving parking spaces. People carrying shopping bags, keys in hands, finishing conversations on the street, people not wearing overcoats in cold weather etc. are all signs that sharpen our attention and make us slow down or wait in anticipation for a potential soon to be vacated spot.

- Observe further peripheral environmental signs like the amount of traffic in the general area. For instance the presence of road-blocks or construction works that can increase the amount of traffic around the destination, further inform us of the parking demand in the area and help us choose the appropriate goals.

- In addition to visual cues, sounds provide cues to enhance the search process. For example, an igniting car engine behind, can alarm the driver of a spot about to be vacated.

Strategies:

- Slow down the car, in order to improve the observation of the environment and potential for response.

- If there happens to be a car ahead, also looking for parking, then slow down or pass that car, deliberately increase distance in order to not be the "second in line" and loose the first available spot to the car ahead.

- Estimate the social situation of the road and categorize other drivers as competitors, non-competitors, polite, impolite, cheaters etc. Such categorization can influence the behavior of the driver, by switching to a more aggressive strategy for instance.

- Use previous knowledge about the usual availability and demand of parking in a given area in order to set a strategy. For instance, knowledge about how difficult it might be to find parking in a specific area at a given time can help a driver to revert to a different search behavior or even change his plans to drive all together.

- Different parking distances from the destination are acceptable depending on specific personal parameters e.g. heavy items to carry, children to walk with and other factors influencing the effort to walk.

- Oftentimes a driver notices an available parking space that is either too small or inconveniently far away from the destination and therefore doesn't accept it.

However, this undesirable option is nevertheless recorded in memory and its acceptance probability grows as the search continues fruitlessly. Drivers frequently return to the spot, which they knew about since the beginning of the search. This was also confirmed by Thompson's model [Thompson 1996], where he confirmed that "search does not necessarily lead to better car parks being selected".

Considering the late origin of the parking problem, it is quite amazing how in the course of a 70-year evolution such complex skills, of which these are only a few, have been acquired by almost all drivers. Has it really been worth bothering to learn such skills? Yes, Figure 5 below shows some common financial benefits that cruising gives.

		Price of parking for one hour ($ per hour)		
City	State	Curb	Off-street	Savings for finding a curb space ($ per hour)
(1)	(2)	(3)	(4)	(5)=(4)-(3)
Baltimore	MD	$2.00	$6.00	$4.00
Berkeley	CA	$0.75	$1.00	$0.25
Boston	MA	$1.00	$11.00	$10.00
Buffalo	NY	$1.00	$3.00	$2.00
Cambridge	MA	$0.50	$4.00	$3.50
Chicago	IL	$1.00	$13.25	$12.25
Houston	TX	$0.25	$1.50	$1.25
Long Beach	CA	$2.00	$2.50	$0.50
Los Angeles	CA	$1.50	$3.30	$1.80
New Orleans	LA	$1.25	$3.00	$1.75
New York City	NY	$1.50	$14.38	$12.88
Palo Alto	CA	$0.00	$0.00	$0.00
Pasadena	CA	$1.00	$6.00	$5.00
Philadelphia	PA	$1.00	$3.00	$2.00
Portland	OR	$1.00	$1.50	$0.50
San Diego	CA	$1.00	$6.00	$5.00
San Francisco	CA	$2.00	$2.00	$0.00
Santa Barbara	CA	$0.00	$5.00	$5.00
Santa Monica	CA	$0.50	$4.20	$3.70
Seattle	WA	$1.00	$8.00	$7.00
Average		**$1.17**	**$5.88**	**$4.71**

Assumptions: A solo driver parks for one hour at noon on a weekday. The prices refer to the first hour of parking in the spaces nearest the City Hall. The data were collected in 2001–2003.

Figure 5 Financial benefits of cruising (parking one hour at the curb). Source: "The High Cost of Free Parking" [Shoup 2005].

Improving the Current Street-parking System

Most of the strategies in the list above greatly enhance our capacity to efficiently find parking. The list revealed how complex a person's parking search behavior can be, using various senses and strategies to help achieve the goal of finding a vacant space. It is not the current search behavior of drivers that causes congestion, on the contrary, the current search mechanisms significantly help to reduce cruising time. Then why is searching still so long?

On the one hand, we might conclude that a maximum capacity of street parking has been achieved and the reason why we cruise is not because we don't search well enough, but because there is simply nothing to find. Indeed, during experiments in Westwood, California, Donald Shoup's analysis showed that "for every 100 curb spaces, seven cars are hunting for parking; that is, 107 cars want to park in 100 curb spaces, so seven cars must *wait* in the traffic flow." [p. 353 Shoup, 2005.]. In popular areas, especially during rush hours, demand surpasses supply exceedingly more than in this example of Westwood, California. Street parking is an illustration of a space market that is rarely in a condition of an economic demand and supply equilibrium, that is, a condition where the amount of parking spaces are matched with the amount of searchers in a perfect balance. I would like to argue that a moderate over demand in street-parking is in fact positive for the overall efficiency at the group level and that the individual search can still be optimized further.

On a general level, street-parking parking simply exemplifies a fluctuating high demand condition that is also natural to other public uses of urban space. Bernard Landau, the principal architectural surveyor of the city of Paris, has argued [5] that optimal design of urban spaces, and here I mean not only functionally optimal,

[5] Course in history of urbanism, Ecole d'Architecture de la Ville et des Territoires a Marne-la-Vallee, France.

but also optimal in relation to cultural and social requirements, cannot satisfy everyone, but has to satisfy a necessary majority of people. This has been intuitive to city designers throughout history. Similarly to parking, benches in a park or a plaza, outdoor recreation spaces, popular restaurants, bus stops and sidewalks witness constant fluctuations of high demand, and are traditionally designed to function coherently with a changing demand and supply. These spaces are dimensioned deliberately smaller than demand at peak conditions would require- there are often more people at a door of a restaurant than tables can accommodate. In a long run this pays off. A moderate over demand to use urban spaces creates a necessary density where spaces do not only function efficiently during short instances of extreme demand, but rather over time, during any hour of the day. What is this optimal balance? I believe that an urban space is optimally dimensioned if it is as large as necessary, but as small as possible. Public space, and especially parking, should not be dimensioned according to rush hour conditions, but rather according to different demand fluctuations over time.

However, zoning laws for parking in this country seem to have forgotten the necessity to optimize the use of public urban space. Excessively large land use requirements are granted to parking, which degrade the quality of urban space, as we saw in the beginning of this Chapter. The trouble, as Shoup has pointed out, is that adding parking spots one by one passes almost unnoticed in cities until we perceive that parking has become the single largest land use in most American cities [Shoup and Menville 2005]. It is necessary and critical from the point of view of sustainability that planners design fewer parking spaces than satisfying demand during peak hours would require and optimize allocation instead of increasing supply. This is precisely where the current parking system could use an improvement.

Despite the use of skills we saw in the list above, the supply and demand of available parking spaces are not coordinated in the most efficient and sustainable manner. In a situation where the amount of cars searching for parking is low and the amount of available parking spots is equally low- a situation close to equilibrium- the filling rate of the available spots is small, it is difficult for the few searchers to find the few parking spots. Furthermore, in conditions of high demand, drivers do not find available parking spaces in a time-efficient manner. For instance, as will be shown, simulations with 6 parking spaces and 24 cars demonstrate that there are always a few spots unoccupied at all times.

Figure 6 Low cost street-parking in popular areas is often filled to almost full capacity, making it difficult for drivers to find the remaining few hidden parking spaces. Beacon Hill, Boston.

This leads me to suggest that the efficiency of the current street-parking system could be improved in two aspects:

1. By providing broader information to drivers, than is currently available to them in their immediate visual surroundings.

2. By using combinatorial and probabilistic calculations on a computer to enhance decision making with the available information.

Let us expand these two claims. The first argument claims that the current search behavior could be rendered more efficient with a use of broader information. The skills we have seen thus far are useful for achieving certain parking goals. Most strategic parking information that is available to us today is obtained from our immediate visual and aural environment during the process of driving and searching in combination with information we have learned from the past. The list above describes some examples of using such information in order to achieve goals more efficiently. The goals that we set however are modified according to the information that is available to us. In other words, we can only set goals based on the information we are aware of. This leads me to suggest that we are not solving the goals inefficiently, but rather we might be solving the *wrong* goals. If drivers had better information of the overall parking situation, then they could apply similar tools for solving a better informed goal. Seen from a distance, the effectiveness of the current search behavior is only successful within a local context around the driver, generally limited by a person's visual field. Many available options outside of this field remain unnoticed and underutilized. Due to the limited geographical dimensions of visual perception, drivers often fail to find the closest parking spot by virtue of chance. Hence, the long searching does not result from poor searching behavior, but rather the limited awareness of parking availability beyond the scope of sight. Due to the lack of such information, we have no capacity to assess the broader efficiency of our strategies.

The lack of wide-ranging coordinated information as the cause of wasteful searching and queuing becomes apparent at an aggregated city-wide level. As all drivers are limited by a similar local search technique, the higher level view of street parking is not a random sum of unpredictable individualistic behaviors, but shows some clear common patterns. For example Axhausen and Polak found in their experiments in UK and Germany that in the overall process from leaving the home to arriving at the destination, the average ratio between the time spent on driving to the destination area (access time) and the time spent on searching for a parking spot (search time) was roughly 2 to 1 [Axhausen, Polak 1991]. The ratio between access time and the additional time of walking from the parking spot to the destination was roughly 2.15: 1. Hence, of the total process, roughly a half was spent on driving, a quarter on searching and a quarter on walking. In areas of high demand, close to a third of total time was used for searching. These studies were done in Karlsruhe, Germany and Birmingham, Sutton and Coventry in UK. In larger cities like Boston, London or New York, where demand is much higher, the search can be far longer. Yet studies show [Shoup 2005] that even there the average search times are fairly constant. While depending on environmental variables, the balance between demand and supply, as well as cultural characteristics of drivers, many central districts have relatively stable search times for parking [ibid].

Figure 1 in the introduction indicated that these search times in CBDs have been roughly the same over an almost 80 year period (average 30%) since 1927. This is very interesting, because over this time the amount of cars per people in America has almost quadrupled (from 220 per 1000 people in 1935 to 800 per 1000 people in 2005) [Shoup 2005]. How can searching times remain comparable while the potential demand for parking grows remarkably? One of the explanations to this phenomenon might be that the policy of street parking has remained virtually unchanged since its creation. Shoup pointed out that that taking inflation into account, the cost of metered street-parking, which was

initially started in Oklahoma City in 1935 with the charge of a nickel per hour, has not changed at all till 2004. “The main change in 70 years is that few meters now take nickels. In real terms however, the price of most curb parking hasn’t increased; adjusted for inflation, 5 cents in 1935 was worth 65 cents in 2004, less than the price of parking for an hour at many meters in 2004” [p. 381 ibid.]. The same price and policy directing the flux of parkers over 70 years might explain why a dynamic system like parking has reacted linearly over decades.

Analogous to most complex systems, the behavior of drivers searching for street-parking is influenced by the parameters and variables of their environment. Herbert Simon provided a comprehensive theory of the relationship between an individual and the environment, arguing that the majority of the constraints guiding a system to a given outcome are imposed by the external environment rather than isolated individual thinking [Simon 1996].

His example of this idea is an ant walking on a beach. As ants follow relatively simple rules of how they should react to environmental stimuli, then more complex environments will make their behavior look more sophisticated. “Viewed as a geometric figure, the ant’s path is irregular, complex, hard to describe. But its complexity is really a complexity in the surface of the beach, not a complexity in the ant.” [p. 51, Simon 1996] Following this argument, the relative stability of searching times in street-parking can be caused by the stable characteristics of the pricing policy as well as the stable perceptive feedback that drivers have at their disposal for evaluating the success or failure of their strategies.

Simon continues to suggest that “a human being can store away in memory a great furniture of information that can be evoked by appropriate stimuli. Hence I would like to view this information packed memory less as part of the organism than as part of the environment to witch it adapts” [p. 53 Ibid.]. Minsky has

elaborated Simon's idea of environmental feedback as the basis of successful functionality of any artificial system, by explaining the importance of cognitive feedback that takes place inside the individual's mind[6]. The variations in search behavior are not only determined by the external environmental stimuli, but also the internal states of a person's mind (mood, ambition, habits, self-critics etc). This idea supports the evidence seen in Laurier's ethnographic study of parking, where the driver gradually changed her goal along the way and finally chose a parking spot that she initially discarded. The reason why a specific parking spot was unacceptable at the beginning of the search, but did become acceptable after searching for a while, was caused by the external feedback from the environment as well as internal feedback of the driver, specifically her self critique of being late. As time passed, and the external environmental conditions remained unchanged (no new parking spaces occurred), the driver changed her goals from ideal to less ideal.

Hence, the deviations in the behavior of cruising for street-parking are a combination of internal stimuli that can cause goal switching, and external stimuli offered by the surrounding environment. Like in most systems, it is natural for anyone in the parking system to be attracted to options of least effort and greatest self-interest. We choose the more comfortable or cheaper alternative offered by the environment. Though cruising behavior appears complex at an individual level, at an aggregate level cruising patterns appear fairly repetitive. To search a larger geographic area, a driver must constantly move around and *scan* the environment by applying his limited sight radius on successive streets until a vacant spot comes to view. Hence, cruising is the result of applying the currently possible search behavior within a given built environment according to the current pricing policy of street-parking. In the long run, the statistical constancy of cruising over the past decades might imply that a certain level of complexity of searching has stabilized and adapted to the unchanged price of

[6] Society of Mind course at M.I.T. Spring, 2005.

street parking and feedback of the search. “An intelligent system’s adjustment to its outer environment (its substantive rationality) is limited by its ability, through knowledge and computation, to discover appropriate adaptive behavior (its procedural rationality).” [p. 25, Simon]. Unfortunately, the point at which the current parking system has stabilized is unacceptable because of its grave environmental impact.

The second problem suggested above is that our current method of search in a condition of uncertainty might not necessarily lead us to the best possible solution with the given local information. Let us also expand this question of rational decision making in a situation of uncertainty in order to explore how digital computation might complement this process.

Research on bounded rationality has unveiled interesting results in the recent years. I would specifically like to refer to the work by Daniel Kahneman on the topic [Kahneman, Slovic, Tversky, 1982]. In his 2002 economics Nobel Prize talk, Kahneman claimed that under conditions of uncertainty human decision making often arrives at solutions, which are not the best that could be computed with the given information. He argued that this is mainly caused by the fact that the value function of alternative decisions is defined by gains and losses, and not by steady states. This means that people base their decisions largely upon the immediately perceivable gains and losses that a given decision offers, and not on a slightly more computationally optimal calculation, that could also be derived from the same evidence. Here is an example he gave. A person **A** is shown two sets of silverware and asked to evaluate the price of each set. Set **A** is composed of 10 pieces of perfect quality. Set **B** contains the same 10 pieces of perfect quality, and in addition, three damaged pieces. How does a person assess the value of **A** and **B**?

Experiments have shown, that if both sets are shown to the subject together, then the vast majority of people assess that set **B** is more valuable than **A**. This is logically the optimally correct decision, as **B** contains all the elements of **A**, and just three additional damaged elements, which do not diminish the value of the 10 remaining perfect pieces. However, if only one of the sets is shown to half of the people and the other set to the other half, then people who see **A** evaluate its value to be higher than people who see **B**. It is easy to see the erroneous nature of this decision if we have already seen two sets, but why does it happen? Kahneman explains that people base their decisions on simplistically perceived averages derived from the information they are exposed to, instead of slightly more computational statistical averages, which requires the help of additions. If people see both sets of silverware, then they visually perceive that **B** is on the average more valuable than **A**, because it contains all elements of **A** and more. If people only see one set, however, then again they average the value, but this time, the pros and cons are averaged from within the given sample set itself. It follows that as **B** contains damaged elements, then the average value of each element in **B** is less than perfect. In case of set **A**, though the set is slightly smaller, each element has a perfect average value. People normally fail to see that instead of averaging the whole group in a single piece, the set **B** could be broken down into two groups, each containing 10 and 3 elements, which added together compose a higher value than the 10 elements alone.

Kahneman has provided several other similar experiments [Kahneman, Slovic, Tversky, 1982], which demonstrate that in situations of limited information, people tend to use perceptive averaging, instead of additive averages. Kahneman's work demonstrated that in many everyday situations of limited information we act irrationally and do not calculate in the most optimal way with the information given. Instead, we often arrive at non-optimal decisions due to a simpler perceptive calculation. In the search for street-parking, this might suggest that we search more based on what we perceive immediately, than based on more

general statistical calculations. For instance, we normally tend to pay most of our attention to vacant spots. Even though we pass numerous occupied spots, we rarely include them in our equations for finding a parking space. In many areas, parking periods are limited, for instance to one or two hours, hence the limited spaces have a frequent turnover. Using common sense knowledge of the types of parkers that might be using these spaces, we could compute the probability of one of these spaces vacating in a few minutes. A high probability result could advise us to wait for a spot to vacate and save us a longer trip to a more distant area. (I shall explain the idea of waiting in designated queuing spaces at the end of Chapter Three.) Though this is certainly done by some drivers, on an aggregate level we do not seem to account for such information that is not immediately apparent, but requires some statistical computation. That is, we do not assign much credit to such information, even though it could potentially lead us to a better search outcome. This is a very loose hypothesis and a great effort of additional experimental research is required to be able to claim any factual evidence. However, there seems to be a great potential to use computers for overcoming such decision errors that Kahneman outlines. Unlike people, computers are extremely efficient and fast for calculating optimal statistical decisions. Thinking along these lines aided me in the construction of the simulation models where the use of local information could be compared with the use of more general statistical information to determine which method yields more efficient search results.

Benefits of a Real-Time Guidance System

So far in this chapter I have discussed how the current parking system works at a personal level and what the consequent group impact of this system is like. From

here on I shall introduce the addition of real-time information to the current system and analyze the effects of this proposal.

In light of a stabilized parking system, the use of real-time information sets a new paradigm for managing street-parking. Real-time information about the demand and supply of street-parking adds a new variable to the search behavior that enables drivers to achieve goals more efficiently.

The idea that I am exploring is based on a proposal for the Zaragoza Digital Mile urban design studio (MIT, fall 2005) and the subsequent Smart-Park project [Lee, Sevtsuk, Ratti 2006] that we are developing at the SENSEable City Lab at MIT.

The system uses both environmental sensors and the user's mobile communication device (e.g. cell phone, PDA, GPS) to help drivers conveniently locate parking spots relative to their position in real-time. One of the differences that I am proposing to the initial Smart Park idea, is that it does not necessarily require help from a telecommunications company to position users. Instead, real-time positioning could be done inside a personal communication device itself, hence protecting a person's location information from second parties. What is necessary for computing one's location, is a map of the communication network and the current signal strength of available antennae. However, I shall try to demonstrate later how an aggregate sharing of personal location information can improve the efficiency of the search even further by providing drivers with information about other drivers competing for the same parking spaces. The system also requires an online database to keep track of the environmental sensors and respond to user requests.

Curb-side parking spaces in a downtown area would be interspersed with tiny sensors that could detect whether a car is parked in the space in front of it or

behind it. The sensors would use a microcontroller with wireless capabilities to communicate with each other and communicate with the server. They would be powered by a battery that recharges from solar energy cells that cover most of the sensor's surface area. A light on top of the sensor on the road side would indicate the status of the parking spot, such as "available," "restricted," "paid," or "unpaid" to passing drivers.

The user interacts with these devices with his cell phone, PDA or an in-car communication device, such as a GPS receiver. While driving around the city, he can query an on-line database for vacant curb-side parking spaces. Having determined its own geographic position in the network, the device can download the parking availability information from the server and offer the closest unoccupied parking space to the user. Users can either reserve that space or simply approach it, hoping it will remain vacant while they drive to it. The communication device would then direct the user to parking by a displayed map or voice directions. Once the user is parked in a chosen spot, the sensor detects the vehicle's presence and informs the server to take the spot off the availability list. In case of a reservation, the sensor also checks if the parked car corresponds to the reservation and then initiates the electronic payment count, at which point the e-ink on the ground indicates this status as a parking meter would. The idea is summarized in the following storyboard.

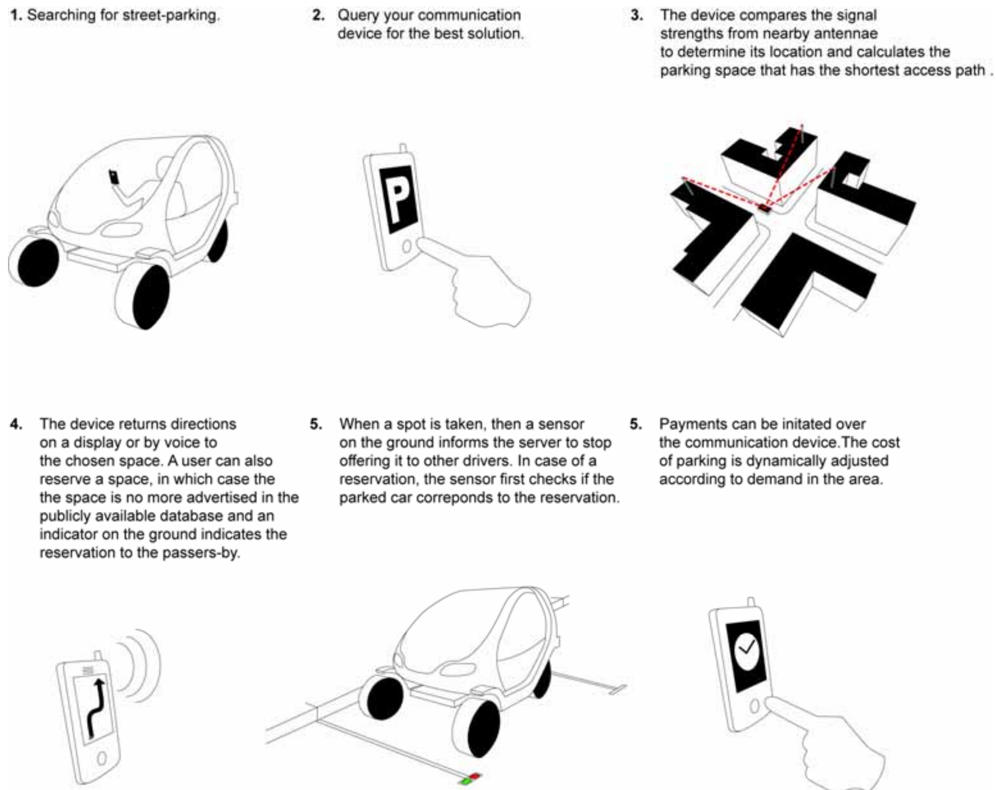


Figure 7 Illustration of the guidance system for street-parking.

Compared to the current searching of street parking the potential advantages of this real-time guidance system are summarized in the points below.

- Currently we rarely find the closest parking space, but rather pick the space we happen to stumble upon first. The real-time system would have an accurate overview of all vacant spaces of a neighborhood and is therefore able to indicate spaces that are within the shortest access distance from the driver or closest to the desired destination.

- We usually miss many available spots which are close to us, but not within our field of sight. The real-time system could extend our knowledge of available spaces to areas we cannot see or guess, such as spaces behind corners, further down the street, on an adjacent block etc. It could also warn drivers if no parking is currently available in the area in order to avoid unnecessary trips.

- Currently drivers often snap away a parking spot in front of someone else, who has been searching for a longer period of time. For good or for bad, it is like cutting into a line. Accurate information about the competition for a given spot would allow the system to assess the chances of obtaining a spot and only offer a particular spot to the driver if he is surely capable of obtaining it. A reservation policy in the system could further help guarantee that searchers will not loose a spot to a newcomer. Reservations naturally help satisfy specific individual demands of drivers.

- It is well known that long term parkers[7] today consume most of the capacity of all on-street parking. Presently there is no efficient way of discouraging that from happening. Current parking meters, which impose a maximum time limit, have no way of prohibiting drivers from paying the cycle multiple times. This has in fact become so common, that "re-feeding" the parking meter has become a popular verb in standard American English. In addition, current parking meters charge at a constant rate, regardless of how long people stay parked. If the system could alter parking fees dynamically and keep track of the period of occupancy of a parking space in real-time, then the price per time could increase exponentially instead of linearly. Also, a priority of using the

[7] People parking for several hours, often a whole day.

guidance information could be given to short-term parkers, with high penalties in case of violations.

- Under-priced street-parking today is a major cause for cruising and excessively large parking requirements in zoning laws [Shoup 2005]. An intelligent management system could respond to demand by fair market prices in real-time, therefore balancing demand and reducing parking requirements. Analogous to economic markets, subsidized supply generally creates higher demand. For example, how much bigger would the city of London have to be to satisfy housing demands if housing were uniformly under priced, equivalent to prices in a small rural village? This is the case with under priced curb-parking today.

These ideas and assumptions were tested in agent-based simulations, which gave approximate estimates of the benefits of the proposed real-time allocation strategies. The topic of the next chapter is to give a detailed overview of these simulation models.

Chapter Two

The Simulation Approach

In her book *The Death and Life of Great American Cities*, Jane Jacobs emphasized the importance of thinking about cities as organized complexity where discrete interrelated variables influence each other simultaneously, where players are many and solutions more complicated than simple formulas.

Jane Jacobs paraphrasing W. Weaver states: "Cities happen to be problems in organized complexity, like the life sciences. They present "situations, in which a half-dozen or even several dozen quantities are all varying simultaneously and in subtly interconnected ways." Cities again, like the life sciences, do not exhibit one problem in organized complexity, which if understood explains all. They can be analyzed into many such problems or segments which, as in the case of the life sciences, are also related to one another. The variables are many, but they are not helter-skelter; they are interrelated into an organic whole." [p.433 The Death and Life of Great American Cities, Jacobs 1961]

During the peak of modernist planning, she warned that cities do not embody disorganized complexity, where order is only to be found by reducing everything to averages and probabilities. This is an important aspect of this thesis: rather than using pure mathematical probability to prove certain benefits or failures of one real-time system over another, I deliberately use multi-agent simulation models, which not only tell us about the broader behavioral patterns of a dynamic system, but arrive there by visually modeling the behavior of each specific member of the system, the sum of which creates the whole. Hence, simulation

modeling is a way to quantitatively test some of the assumptions made at the end of the last chapter. I believe that this technique provides more information than a clean mathematical proof, precisely because it doesn't rule out individual differences and because it works as well with very few agents as it does with hundreds or thousands of agents. This is the opposite of a probabilistic approach, where statistics only get better when the group of study is larger and periodic changes over time are not accounted for.

Urban simulation is a growing research area and many thorough studies have been conducted for modeling urban growth [Batty 2005], response to policy change [Flaxman 2002] and social networks [Metcalf 2005]. UrbanSim software (developed at the University of Washington) and ILUTE software (developed at the University of Toronto) are examples of large-scale urban simulators designed for use by urban planners. The Santa Fe Institute has for years been pioneering scientific advancement of agent-based computation in economics and social systems. City simulators are generally agent-based simulations with explicit representations for land use and transportation. The simulation environment primarily used in this thesis, however, is Star Logo[8]. It is not specifically designed for urban applications, but had great advantages due to its open framework of coding possibilities, which allowed modifying agent communication in aspects discussed at the end of this section.

The two principal computational concepts used in most simulation environments are *Cellular Automata* and *Agent-Based modeling*. I shall give a very brief description of the key characteristics of those two techniques in order to show how they can be used for street-parking simulation.

[8] StarLogo is a specialized version of the Logo programming language. It is developed at Media Laboratory and Teacher Education Program, MIT, Cambridge, Massachusetts, with support from the National Science Foundation and the LEGO group.

The origins of both Cellular Automata (CA) and Agent-based models are closely related to the work of John von Neumann and John Conway. Before cellular automata[9] obtained this name, it was a concept that von Neumann proposed as a theoretical machine with self-replicating capacity. The idea initially consisted of *cells* on grid paper, with specific rules and purposes for communicating with each other. The cells mimicked very simple intelligent beings and their behavior was determined by a few internal rules of interaction and by neighboring cells that interact with it. Conway took the idea forward in his Game of Life[10], where he implemented the idea in a virtual context on a computer. As the interaction between cells is largely determined by the rules that a programmer ascribes to the cells, CA can simulate selected real-world situations of localized simplicity, but overall complexity.

CA consists of an infinite field of equally sized cells, that can all have a finite number of states. Each cell is surrounded by a neighborhood of cells, which are commonly described as the Moore or the Van Neumann neighborhoods, depending if diagonal neighbors are considered. Hence, each cell has eight neighbors in a Moore neighborhood and four neighbors in a von Neumann neighborhood. A communication signal that travels across a CA field will be passed from each affected cell to its neighborhood, therefore changing the state of adjacent cells. Though there is potentially an infinite number of states, cells can only communicate with other cells within their immediate neighborhoods.

There are surprisingly many processes in nature that follow the principles of cellular automata. From the diffusion of particles to the societies of insects, communication happens from one member of the system to another, resulting in complex orders of dynamic systems. It is commonly believed that the fairly sophisticated societal structures in ant colonies are determined by very simple

[9] A detailed description can be found in the New Kind of Science [Stephen Wolfram, Wolfram Media 2002] and on the Wikipedia on-line encyclopedia.
[10] First published in the October 1970 issue of Scientific American.

rules that ants follow without having any knowledge of what higher level structures they are actually part of[11]. Similarly, a flock of birds flying in a neat triangular shape or a flock of fish swimming in groups have no higher level knowledge of how to organize themselves in neat shapes. Instead the shape is an emergent result of self-organization, where birds or fish only know what position to take according to each other. Amongst humans, oral communications also frequently follows diffusion patterns, similar to CA. Hence, CA is a suitable method for analyzing systems where communication flows continuously from a locus to its neighboring loci and so on. Cellular automata is not suitable for processes, however where communication does not happen in a physically proximate manner.

Agent based-modeling is directly related to the concepts of cellular automata. Perhaps the main difference in agent-based modeling is that in addition to fixed cells, the concept uses dynamically interacting rule based agents, which can move around on top of the cell-grid. The agents are representations of small programs, which have well-defined rules of interaction. However, agents not only interact with the background cells, but also with each other. Hence the agents are intelligent and purposeful and they reside in space and time. The modeler ascribes the rules of interaction from real-world assumptions he thinks as most relevant to the processes he wants to model, and can then study how a phenomenon emerges from the agent's interaction. Agent- based models are very useful for studying how a certain system reacts to external and internal forces and how/if a new equilibrium is established after a change. Therefore, agent-based modeling can have a wide range of applications in urban planning to analyze the impacts of both internal changes in people's behaviors and external changes in physical building or legal policy interventions.

[11] For a great description, refer to *Turtles, Termites and Traffic Jams* by Mitchel Resnick, MIT Press 1994.

The simulations used in this thesis use an agent based modeling approach. Agent based modeling can fairly precisely characterize the formation and movement of traffic jams that mostly form from two simple principles amongst agents: "decelerate if there is a car in front of you" and "accelerate up until the speed-limit if there is nobody ahead". The conventional search process for street-parking can be analogously modeled through an agent based model. Drivers looking for parking spaces can be symbolized as agents. Static cells in the background can indicate parking spaces. The physical distance between a driver and a parking space that is within the driver's visual reach, can be represented by a neighborhood of CA cells around a driver. Should a space appear in a neighborhood, an agent can approach and seize it akin to the way a driver would seize a parking space that comes in sight in the real-world.

However, for modeling real-time communication networks of a contemporary city, the traditional method of communication between agents and cells, where messaging happens only between immediate neighbors, is insufficient. In the real-world, the use of telecommunication channels, such as telephones, faxes, cellular-phones, radio, television, satellites etc, have the capacity to transmit information over long distances without involving intermediate places and channels. This capacity for distant communication is generally not part of cellular automata or agent based modeling techniques.

For instance, before going out to a retailer, a person can telephone the shop to inquire if the product he is looking for is available. If it is, he might set out for the trip, if not, he might call other stores before making a move. The information that determines the action is acquired distantly. Similarly, on-line databases of public transportation and traffic congestion can tell a person the real-time traffic situation on major highways and streets (e.g. Los Angeles City Traffic Info, MIT

Shuttle Track)[12], hence influencing the choice and mode of the route the person might take. Simple electromagnetic signals can encrypt messages in binary form and almost anyone connected to the telecommunication network can access them by using a device that decrypts the message back into text format. What is most significant in an urban context is that messages can travel through electromagnetic waves in the air, connecting cars, phones, computers and wristwatches to the rest of the network without any apparent physical connections. If the amount of portable communication technology continues to spread at the current rate[13], then an increasing amount of urban decision making could happen with the help of distant sources. All this facilitates distant communication but complicates urban modeling through classical cellular automata.

In the models used in this thesis, I am proposing a way to make real-time distant information available to agents that represent cars. In the following models the ability to acquire and report information distantly also works in parallel with local cellular automata communication. The key additional communication feature to traditional cellular automata communication is a dynamic real-time list in the form of a global variable that reports the availability of a particular resource to all agents regardless of their position. At the same time, the list is also continuously updated by all the agents themselves, based on their interactions in the model. If a cell representing a parking space is occupied, then the agent occupying that cell reports to the system that it has absorbed the particular spot and this spot is no longer announced as a vacant parking space to other agents. This form of distant communication through a list of information that is updated in real-time allows me to symbolize agents the way people in the real-world would acquire distant information through their mobile communication devices.

---

[12] http://trafficinfo.lacity.org/ ; http://shuttletrack.mit.edu

[13] In 2005 there were 194,479,364 cellular phones in the U.S., which is equal to 0.65 cell phones per person. In 2003 there were 0.54 cell phones per person. (CIA Wold Factbook, 2005) If continued at this rate, everyone would have a cell-phone in about seven years from now.

The Rules of the Simulation

A very life-true performance of the model is extremely hard to capture. Like all good simulation models, even well-known parking choice models [Thompson, Richardson 1996] take only a fraction of real-life variables into account. Despite the simplifications however, a carefully crafted simulation model can tell us a lot about real-life phenomena. The important task for studying a specific system in a simulation model is to isolate the critical variables from the less critical ones. At one extreme, one could technically model all possible phenomena that might directly or remotely influence a system, but people of long experience [Minsky, 2006] argue that not all such variables are important for understanding the higher level reactions of the system studied. If one chose the path of carefully modeling all possible variables, then one would be doomed to replicating reality and never able to study higher level alternative scenarios of reality. In the opposite extreme, the assumptions made in a simulation might simplify too much the reality and hence provide unreasonable evidence. I think that a good simulation should neither attempt to mimic reality nor make groundless assumptions, but rather analyze a carefully chosen set of underlying dynamics influencing reality.

We could look at street-parking as a structure involving multiple participants, and a physical environment where the system operates. The overall behavior of the street-parking system is then a combination of three categories of variables: 1) the physical and legal environment that drivers operate in 2) a driver's individual behavior 3) dynamics of group behavior (Fig. 8).

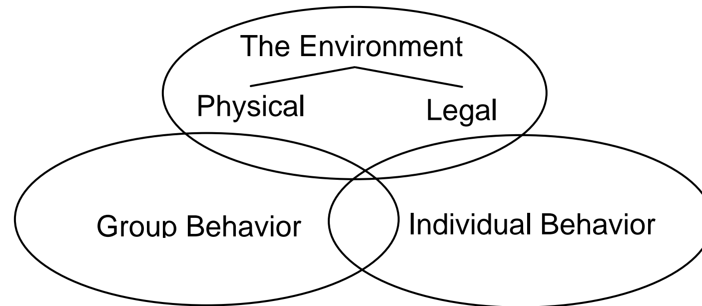


Figure 8 The three categories of variables in the street-parking system.

This distinction between group behavior and individual behavior is similar to the division between microeconomics and macroeconomics. In order to study performance, it is crucial to distinguish which is our viewpoint of evaluation. Features that might provide great benefit to individual behavior might at the same time jeopardize group behavior and vice versa. This, in fact, is one of the central outcomes of this thesis and I shall come back to this idea in the concluding chapter. For now, I would like to emphasize that the simulations primarily experiment with improving the performance of group behavior (reducing the overall search time of cars) by introducing a behavioral modification (the guidance system) at the individual level.

To a limited extent, I shall also experiment with modifying the physical and legal environment in which street-parking operates. The environmental modifications that I am interested in are 1) an introduction of dynamic pricing as part of the legal parking policy, 2) a physical modification in the amount of parking spaces a neighborhood should have and 3) alterations in the traffic layout to accommodate queuing cars.

As a result of intervening in those two categories of the search for parking (the Individual Behavior and the Environment), I shall study the resulting search time

efficiency for the group. The main question that I am trying to address with these simulations is: *What are the critical conditions, under which one strategy performs more efficiently (i.e. reducing search-times) than another*?

The methodology I chose was to intuitively build a simple search model and evaluate how it works. Based on the first strategy, I devised a different strategy and compared the search efficiency of the two models. Then followed the third and the forth and so on. Hence, I started with relatively simple simulations of the existing parking system and gradually built up complexity by adding variables one step at a time.

What seems natural in everyday life, actually requires a vast amount of common sense and is therefore exceedingly hard to capture in a computer program. Computers are efficient processors of information, capable of finding optimal solutions rather quickly. However, the sensing capacity of humans is far greater than that of computers. Thus, computers can potentially compete faster in tedious calculations, but they currently have very limited perceptive capacity compared to people, which strongly undermines any definitive assumption of the superiority of a computational system over human decision making. Chess playing programs are a sufficient illustration of this- players of great experience can still occasionally defeat any computer programs. Many have experienced the inferiority of a modern computer in a GPS navigation system in a car, providing wrong information, instructing too much or not instructing enough. It would be incongruous to propose that everyone should adopt a computational guidance system. In order to avoid common sense conflicts, the application of the real-time parking guidance system should be complementary to the existing method of parking search and not replace it. Compatibility with traditional ways of finding parking and downward compatibility towards a simpler technological system are important conditions in order not jeopardize the functioning of the existing cognitive processes. The guidance system ought to only add beneficial strategies

that can increase an individual's search capacity and not take away the existing ones.

In addition, for most people, the psychological barrier and the learning curve of a new technology can be diminished, if previous habitual methods are preserved in parallel with a new technology[14]. Whether or not to use the system should be decided by individuals and not imposed by law. However, I shall try to show that some aspects of the proposed system provide great time and financial benefits to individual users, which could themselves provide enough of an incentive for users to join the system.

The next section will first explain some general rules that all of my parking models follow. Each distinct model also has some uniquely specific rules, which are explained further below under appropriate headings.

[14] This has been elegantly demonstrated by [Mackay, W.E. 2000] and [Samad, Weyrauch 2000]

General Rules

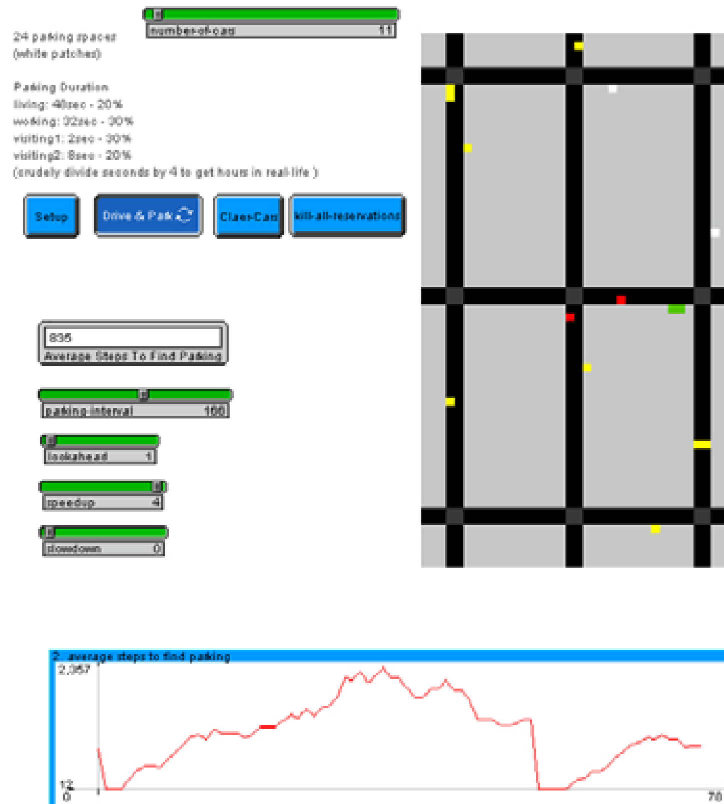


Figure 9 Graphical user interface of the Star Logo simulation model

First, a chosen number of cars (see the “number-of-cars” slider in the interface window) is created and dispersed randomly on the black patches of the screen, which represent the driving lanes of the road. The cars are divided into 4 categories: 1) “living” 2) “working” 3) “visiting1” and 4)”visiting2”, which represent the main kinds of drivers in a real-life neighborhood, depending if they live, work or visit a neighborhood. The four categories differ by duration that each member spends in a parking-space at a time. The times chosen are the following: 48 seconds for “living”, 32 seconds for “working”, 2 seconds for “visiting1” and 8 seconds for “visiting2”. These times are proportionally set to

match the parking durations that the main kinds of drivers normally use (12h for "living", 8h for "working", 2h and ½h for brief visitors). I chose the values based on a consultation with urban transportation specialists at MIT[15]. I tried to estimate values that would correspond to the commonly known classes of drivers in the real-world. I estimated that 20% of cars belong to people who live in the neighborhood, 30% to workers, 30% to visitors and 20% to brief visitors.

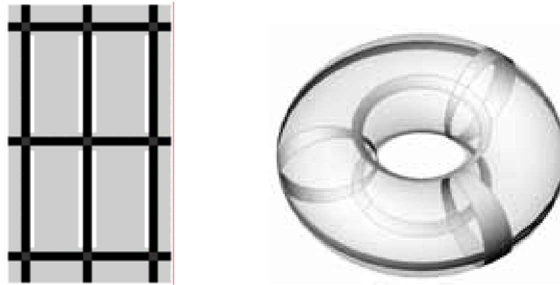


Figure 10 Layout of the street-network. In the model, cars can "wrap" out of the picture on one side and re-enter from the opposite side, creating an infinite torus shaped topological continuum.

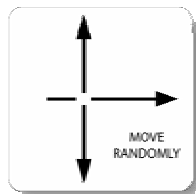
When a car is in "Drive" mode and arrives at an intersection, it can either go straight, take a right turn or a left turn. In the *Traditional* parking simulator, the decision of turning right, left or continuing straight is a random one and in a simplified way, it simulates how the search for parking works if we do not know where vacant spots are located. We shall see later, in the case of "Intel" models, that these intersection decisions are what guide a car to the closest parking spot, if cars know such information. When a car arrives at the end of the screen it will re-enter from the opposite side of the screen, performing a "wrap". This is equivalent to using the street grid as a topological torus, where one can never drive out of the scene. Hence, instead of introducing new incoming cars from outside the scene, the same cars illustrate the searching and thoroughfare traffic in a repetitive sequence. The number of cars in a given simulation is constant at

[15] Consultation with Mikel Murga and Chrisopher Zegras.

all times. The cars are also programmed to consider each others presence- they have to slow down if a car is directly ahead of them. Accordingly, they can speed up if there is no-one ahead until they reach the speed-limit. If the "look-ahead" slider is set to 2 instead of 1, then cars will decelerate according to two cars ahead. The "speedup" and "slowdown" sliders control how quickly a car accelerates or decelerates.

An output monitor on the screen counts the average number of steps to find parking for all searching cars. This value is recorded at every iteration and also stored in memory for later analysis. The plot window at the bottom of the screen graphs this average number of steps.

Traditional Parking Search Model



In the case of Traditional parking, once the drivers have been assigned to a certain group, they start driving and looking for a parking spot. The procedure[16] calls each car to first find a parking spot. If there is no parking spot next to the car, the car will keep driving until it finds a vacant spot right next to itself and parks there. Once a car has found a spot, it will stay there for a time period that is characteristic to its category. Once pulled out of parking, a car will drive around for a chosen time-period until it starts looking for a parking spot again. This time period is determined by a slider on the screen named "parking-interval." By default the interval is set to 36 units, which allows cars to randomly drive

[16] The Source Codes to all the models can be found in the Appendix and at http://web.mit.edu/asevtsuk/www/thesis

sufficiently far away from the previous spot. Modifying this variable will have a direct impact on the availability of parking spots: the smaller the parking-interval, the more often every car starts looking for parking again, the bigger the demand.

The initial model is set to have 6 available parking places (white patches), but different quantities were tested. The default amount of cars is 6 or 24, generating a demand that is either equal to or four times greater than supply.

The simplified pseudo code of an agent's procedures in this model looks like the following:

```
Repeat Infinitely[
      Park[
            Turn the color of the agent to yellow
            Repeat until you find parking[
If there is a vacant spot next to you, then park, else
make a step forward]
if you are at an intersection[
choose to go right, straight or left randomly]
If parking was satisfied, turn color back to red]
Drive without parking for X period]
      End
```

Intel **Parking Search Models**

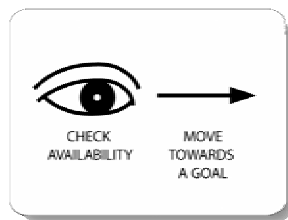
The four distinct *intelligent* parking simulations that were tested are named Intel_1, Intel_3, Intel_5 and Intel_7. Their particular differences are described in the following section.

By an “intelligent” parking model, I refer to the real-time guidance scenario, where agents have access to additional information that helps them find the closest curb space. As I noted in the previous chapter, the key feature that allows agents to have intelligent knowledge in choosing the closest parking spot, is a list of all vacant parking spaces, which all agents can monitor at all times. This list shows the X and Y coordinates of all white patches (parking spaces), which have no agents occupying them. This list (called "aa")is continuously updated at 0.5 second intervals, and saved as a global variable. "Aa" is not structured in any significant order, its elements are added and deleted as vacant spots happen to appear or disappear.

Unlike the traditional parking simulation, cars in intelligent simulations do not make random decisions at intersections, looking for any available parking spots. Instead, before starting a search, they evaluate where the closest vacant spot lies and then start heading towards it. To evaluate the closest spot, an agent sums the horizontal distance and the vertical distance from its own position to each spot in the “aa” list one by one, memorizes the shortest option, and chooses the closest spot as its destination. If the spot is in the opposite direction than the agent is heading towards, then a length of ½ of the block perimeter is added to the distance, because the agent is first forced to drive around the block before heading towards the spot. As the following descriptions show, this parking guidance information is used differently in the four models. However, the way a car orients itself towards a chosen spot is common in all three: if the chosen spot lies ahead of the agent on the same street (along an orthogonal axis- up, down, left or right) then it keeps moving straight until it reaches the spot and then parks on either the right or left side of the road. If the agent is moving towards the spot, but the spot is in nearby blocks in the left or right side sectors (seen from an intersection), then an agent turns left or right accordingly. If the agent is moving further away from the spot, then it takes a left or right turn at the first possible intersection depending on which side the destination spot is. These simple rules

allow cars to take the shortest possible route to their chosen spot in the given traffic grid. Similarly as in the simulation of existing parking, if cars on the edges of the screen "wrap" around and re-enter from the opposite side, then the same rules continue to apply, even though the spot that was passed may now appear ahead of the car.

Intel_ 1 Parking Search Model



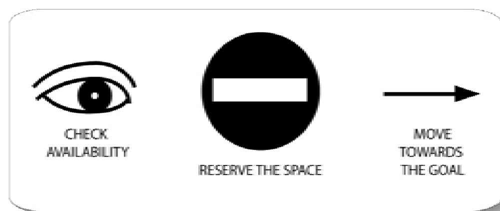
In the Intel_1 strategy, when agents enter the parking cycle and start looking for a vacant parking space, then they first assess which spot in the “aa” list is closest to them. Once the closest spot is chosen, the agent makes a step towards that spot along the shortest calculated path. The size of the step is the same as the size of an agent as illustrated in the interface above. (Each iteration all agents can move only one step.) At the beginning of the next step, the agent calculates the closest available spot again, and if the same spot appears closest, then the agent continues its way towards that spot. However, if that spot appears to have been taken by another agent or if another spot that is closer has been vacated in the meantime, then the agent will change its destination to the newer nearest option. Thus, if along the way of driving towards a specific destination, an agent encounters another occasional vacant space, then it is allowed to immediately occupy that space, despite the fact that some other agent might have that space as a destination and might be driving towards it. The agents who loose a destination in such a manner will re-evaluate their destinations at the following step and choose a new destination. Hence agents have a real-time knowledge of where vacant spots lie.

If an agent reaches its destination and the destination is still unoccupied, then the agent parks there for the time period specified by its class (living, working, visiting, visiting2).If no vacant spots are available in the aa list, then agents roam around randomly (taking random decisions at intersection points whether to go straight, left or right). This procedure is repeated at every step.

The simplified pseudo code of an agent's procedures in this model resembles the following:

```
Repeat Infinitely[
      Park[
            Turn the color of the agent to yellow
            Repeat until you find parking[
Find the closest parking space from the "aa" list,
memorize it as a destination
If there is a vacant spot next to you, then park, else
make a step towards the chosen spot]
If parking was satisfied, turn color back to red]
Drive without parking for X period]
      End
```

***Intel_3* Parking Search Model**



The *Intel_ 3* system differs from the previous model primarily by its introduction of a policy of reservations. A reservation restricts random passers-by from parking at that space. Technically, in the model this means that a virtual reservation agent is created at the chosen destination, which keeps other cars

from being able to use that space for parking while the agent who made the reservation is driving to it. In this model, before agents drive off towards a parking spot, they again consult the closest option from the “aa” list. The agent’s manipulation of the “aa” list, however, differs significantly from the previous model. This is due to the condition, that reservations have to be mutually exclusive. That is, the same reservation should not be given to several cars. Hence, when an agent first queries the “aa” list, it needs to check if the list is available for consulting. If not, then it waits until the list becomes available. Once the list is available, then the agent blocks the list from other agents’ access and then checks if there are any vacant parking spaces available in the list. After choosing an available spot, or choosing nothing if the list is empty, the agent saves changes to the list and unblocks it for others to use. This is similar to the way on-line shops with multiple simultaneous clients function. The sequence of steps a computer makes for a buyer is the following:

1) Check if the list of goods is available
2) Wait until the list of goods is available
3) Access the list of goods and lock it from others
4) Check if there is anything to buy from the goods
5) Perform a purchase and exit the list or simply exit the list
6) Save the changes to the list of goods
7) Unlock the list of goods for others to use.

These steps are necessary in a real-time market to ensure that a single good is not sold to many customers.

If an agent finds vacant spots in the list, then it chooses the closest spot as its destination in the same way as in the previous model. Unlike Intel_1, agents here do not re-evaluate their destinations at every step; they keep the same destination until they park. In addition, it can also put a “reservation” on that spot. Reserving

the spot, the agent asks the “aa” list to eliminate that spot from being offered as a vacant space to other agents. During the approach, the agent who made the reservation uses a verification procedure at every parking space along the way to check whether the space corresponds to its reservation. Only if a match is found between the reserved spot and the reserver, can the agent park at the spot. However, if the agent happens to pass another unoccupied space along the way to its destination, which does not have a reservation on it, then it is also allowed to occupy that space. In this case, the agent gives up its reservation at its initial destination, which then is put back on the “aa” list for everyone to use.

The simplified pseudo code of an agent’s procedures in this model looks like the following:

```
Repeat Infinitely[
      Park[
            Turn the color of the agent to yellow
            Wait until the "aa" list becomes available
            lock the "aa" list
      Find a parking space from the "aa" list
Eliminate the chosen space from the "aa" list or exit
if the list is empty.
      Unlock the "aa" list

      Repeat until you find parking[
If there is a vacant and unreserved spot next to you,
then park, and if you had a reservation, then cancel
it. Else make a step towards your reserved spot.
If no reservation could be made, make a random
decision on the next intersection and then try to find
a vacant space from the "aa" list again]

If parking was satisfied [
Turn color back to red
```

```
Drive without parking for X period]
      End
```

***Intel_5* Parking Search Model**

The Intel_5 model tries to unify strategies of both Intel_1 and Intel_3. Similar to Intel_1, agents are exposed to the real-time list of all available spaces at every step of the search. The model also tries to take advantage of the mutually exclusive allocation policy of Intel_3- it tries to avoid the allocation of one space to several cars. This is achieved, however, without a reservation policy.

In addition to the real-time vacancy list that Intel_1 used, agents in Intel_5 have access to significantly better information about the traffic situation. Namely, when an agent consults the real-time vacancy list, then it does not directly choose the closest target space and start driving towards it, but also evaluates if any other agents have the same destination, and if so, how far they are from that destination. Only if the agent is closest to the target spot among all competitors, will it start driving towards that spot.

If not, then it will try the next closest spot in the “aa” list and do the same evaluation again. If necessary, then this can continue until all the spots in the list have been tested. If an agent at an intersection is not the closest competitor to any spot, then it will make a random decision and continue checking the list at the next step. This evaluation procedure is repeated at every step.

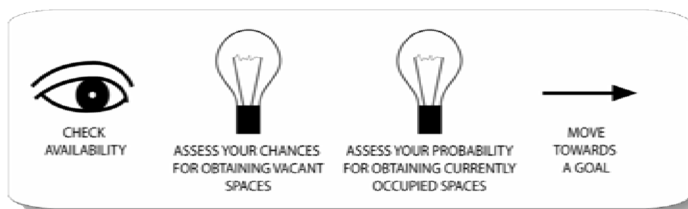
In case of a situation where a newly arrived searcher might appear closer to a parking space that an agent had calculated for its destination at a previous step, then the closer newcomer has priority over the space. This is because no reservations are used. However, as soon a new searcher appears in the scene and impacts the allocation solution of the previous scene, then all agents recalculate their destinations at once, without any further driving towards their last goals. Agents, whose targets haven't attracted any closer competitors, will continue to drive towards their previous goals as planned.

The simplified pseudo code of Intel_5 resembles the following.

```
Repeat Infinitely[
      Park[
            Turn the color of the agent to yellow
            Repeat until you find parking[
Find the closest parking space from the "aa" list,
check if any other agents have the same target and
check how far the competitors are from the target.
If you are the closest of all competitors, then make a
step towards the chosen spot. If not, try the second
closest spot from the "aa" list. If necessary, repeat
this until all elements of the "aa" list are
exhausted. If you are at an intersection and are not
closest to any spots or if "aa" is empty, take a
random decision]
      If there is a vacant spot next to you, then park
If parking was satisfied, turn color back to red]
Drive without parking for X period]
      End
```

Similar to all previous models, agents are also able to park at any occasionally vacated closer spaces along the way to their target, should such spaces appear. If the real-time vacancy list appears empty, then agents at intersections make a random decision and query the list again at the next step.

Intel_7 Parking Search Model



Lastly, the Intel_7 search model introduces a small, but conceptually significant addition to the Intel_5 model. In situations where supply of parking spaces is large enough to satisfy all demanding cars, Intel_7 functions identically to Intel_5. The additional feature it introduces appears useful only in a case of over demand.

Instead of repeating most of the Intel_5 procedure above, let's assume that Intel_7 follows the exact same steps until a situation appears where the searching agent is not the closest competitor to any available spot in the scene or when the vacancy list simply appears empty. We saw in all 3 previous Intel models that in such a case agents at traffic intersections took a momentary random decision and checked the vacancy list again at the following step. However, this is certainly not the way drivers would react in reality. Instead, drivers use various techniques for guessing where the potential vacant spots might appear. The better the experience or information that a driver can use in the face of such uncertainty, the better the chances for taking a tactical decision. If no clues from the environment or prior experience offer certainty, then people could choose

probabilistically. In a similar manner, the basic idea of Intel_7 is that as long as there are vacant spots available, agents should cooperatively allocate them between each other, just like in Intel_5. However, if no vacant spots are available, then agents should be able to guess intelligently, where to go searching with probability, instead of roaming randomly.

In order to achieve this, accurate information about the current occupancies of all parking spots have to be acquired from the environment. Hence, I propose that the parking sensors that were introduced in the last chapter, embedded in asphalt between two parking spots, also record the duration of stay of each car that occupies them. This information is collected in a second real-time list called “bb” and fed back to the server for probabilistic analysis of upcoming vacancies. “”BB” is a global variable similar to “AA”, it contains the X and Y coordinates of every occupied space, available for all agents in the scene to consult at any time.

As the program can keep track of the parking duration of occupied spaces, then a probability of an occupied space freeing up can be calculated. I have been using 4 classes of cars: living, working, visiting, visiting2. They differ by the time they spend on a parking spot ("living" cars park 48 seconds, "working" 32 sec, "visiting" 8 sec, and "visiting2" 2 sec). 20% of all cars are living, 30% working, 30% visiting and 20% visiting2. These were chosen to roughly correspond to different classes of real-life drivers. It follows that if a spot is occupied, then the probability that the owner of the car belongs to one of the groups is:

"living group" - 0.2
"working" - 0.3
"visiting" – 0.3
"visiting2"- 0.2

A timer records the time of occupancy of each parking space in seconds.
If the timer (T) on an occupied spot is $0 < T < 2$, then the chances that the spot vacates in less than or equal to 2 second is 0.2.
If the timer (T) on an occupied spot is $2 < T < 8$, then the chances that the spot vacates in less than or equal to 6 seconds is 0.375.
If the timer (T) on an occupied spot is $8 < T < 32$, then the chances that the spot vacates in less than or equal to 24 second is 0.6.
If the timer (T) on an occupied spot is $32 < T < 48$, then the chances that the spot vacates in less than or equal to 16 second is 1 (100%).

To choose which spot to go to, the timer T is weighed according to the distance of the given spot from the driver. If a *vacant* spot is 25 steps away from the car, then the value of that spot is still 1 because the agent has already determined that it is the closest in the competition for that spot, and the spot is assumed to be vacant until this agent arrives there. Hence, vacant spots always have the highest value (1).

However, if there are currently no available spots, then instead of roaming randomly, an agent can guess which way to go with probability. If an occupied spot is 25 steps away from the car, then the agent first has to query what the timer T on that parking spot shows. If the timer is $8 < T < 32$, then the value of that spot is 0.6 / abs(25 - 24) = 0.6. That is the probability of it vacating in 24 seconds divided by absolute value of the distance minus the time till vacating. Ideally the driver would like that spot to vacate just a little less than in 25 steps, right when it arrives there. In this example, chances are 0.6 that the spot will be vacated by the time the car gets there. Similarly to evaluating the competition with vacant spots that we saw above, agents here can also assess if someone else is targeting the same occupied spot and only choose the occupied spot as a target if they are the closest competitor to it. This evaluation procedure is again repeated at every step.

```
Repeat Infinitely[
      Park[
            Turn the color of the agent to yellow
            Repeat until you find parking[
Find the closest parking space from the "aa" list,
check if any other agents have the same target and
check how far the competitors are from the target.
If you are the closest of all competitors, then make a
step towards the chosen spot. If not, try the second
next closest spot and repeat this until all elements
of the "aa" list are exhausted. If you are closest to
none or if "aa" is empty, then calculate the
probabilities of obtaining currently occupied spaces
and evaluate your chances of obtaining them compared
to other competitors. Memorize the option with the
highest probability value and make a step towards it]
      If there is a vacant spot next to you, then park
If parking was satisfied, turn color back to red]
Drive without parking for X period]
      End
```

By introducing additional information about the current occupancy times of taken spaces into the search process, this model attempts to reduce uncertainty for agents with no assured goals.

Critical Variables

We saw in the ethnographic description of a real-life parking situation in the last chapter that there are a great number of different cognitive and physical activities unfolding simultaneously when one tries to park. Personal behavior is directly

influenced by physical environmental factors that can cause changes in the strategies a driver uses for the search of parking. For instance, if a person notices that there is an unusual amount of traffic in a given neighborhood, then he might immediately decide to settle on the first available parking space, even if it is somewhat inconvenient and the person wouldn't do so under conditions of light demand. I shall outline below some of those critical factors that play a role in the overall performance of the street parking system. These factors comprise a mixture of individual, group and environmental variables, which I find most significant for reducing search times. The list could be potentially infinite, but for intuitive and technical reasons, I have mainly tested the simulation models under the following variable conditions:

1. The balance between the number of searchers (demand) and available spaces (supply). The models use either 6 cars and 6 parking spots or 24 cars and 6 parking spots.
2. Connectivity and size of the street grid. The two grids tested were a 3 x 3 and a 5 x 5 rectangular street network with topologically connected edges.
3. Distribution of parking spots on the grid. The models use two types of distributions: dispersed (parking spaces equally distributed across the field), and concentrated (all parking spaces located on a single street segment).
4. Real-time Information. The *Intel* models look at four different ways of using real-time information about parking spaces, about competing cars, about placing reservations and about the total group performance of all cars.

Additional variables that can equally affect the performance of the models (duration of parking; management of queuing; dynamic pricing) are touched upon in text in Chapter 3.

Optimum Strategy versus *Satisficing*[17] Strategy

In chess there are hundreds of different opening strategies[18]. Each opening also has a corresponding defense, which is known to be the most efficient strategy to react to a particular opening. There is no one best strategy for all openings, but instead a particular one for each case. The initial procedures of the game can be predicted in advance for several steps if both players follow well-known strategies. When one of the players makes an unpredicted move, then the real game begins. Players no longer know for certain which move is optimal, but are rather forced to strategize which moves would be most beneficial. To do so, a trained player can compute solutions to many possible scenarios and choose the most satisfying move. However, after a certain threshold it takes too much calculation to predict the best strategy. Theoretically, it is possible to calculate the ultimate optimal defense strategy for any situation, but the number of required computation is around 10ˇ120, well beyond any human, or computer capacity. The ultimate optimal strategy is therefore practically impossible to predict and instead, players use the move that seems most promising. In the subsequent moves, strategies in chess have to be readjusted in real-time as the game evolves according to the particular responses from both opponents.

Similarly to chess, optimal space allocation in parking requires good computation. In urban settings, variables are arguably even more complex than in chess. It is impossible to define an absolutely optimal strategy for space allocation in a particular situation, because all possible outcomes are beyond existing computational capacity. A solution can only be optimal to the given variables it accounts for. No urban system functions as neatly as a machine, performing its task in a clearly optimal manner. The variables in real life are far

[17] Term used by Herbert Simon, *The Sciences of the Artificial*, first published 1969.
[18] List of Chess Openings, Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/List_of_chess_openings

too complex for that. We can, however, outline a *satisficing* strategy, which based on limited computation, proves to work better than other strategies. Unlike chess openings, real time parking strategies can function as differential equations, they perform best within a certain range of variables. One strategy is not merely useful for a particular setting of cars and parking spots, but rather for multiple settings within certain limits. When variables exceed these limits, then another strategy becomes more efficient.

The choice of a space allocation strategy should therefore depend on circumstances of the situation. In an ideal allocation system, different circumstances require different strategies, which are substitutable in real-time. The decision to switch from one strategy to another itself can require a lot of knowledge. Even more computational energy is necessary to compare a situation with two different strategies simultaneously. This leads me to state the obvious: if commuting within a city were as complex as playing chess, where one has to evaluate the next best move after each step, then it is clearly unlikely that people would ever accept to deal with such complexity. Simple and repetitive methods of commuting in the city are desirable, because instead of solving combinatorial problems of optimal travel paths at every step, we can commute by memory, and instead think of various other things at the same time.

Today, we base our commuting decisions mostly on the immediate perception of the environment around us and analyze only a few of the obvious alternative commuting options. With the example of Kahneman's work, I suggested in chapter one that this can lead to erroneous decisions. However, for computers calculating thousands of combinations at every step is not too ambitious. Assessing the parking circumstances of a situation in real-time, calculating a *satisficing* solution and adjusting the search strategy for space allocation accordingly is a relatively simple task for any modern computer. The messy and extensive calculation process can thus be performed within a small computational

device, a person's cellular phone for instance, and all that the owner of the device ever has to see is the *satisficing* solution to his problem proposed in simple graphics on a screen.

Hazards of the simulation

There are several real-life variables that the models described above do not account for. First, the model simulating the existing parking system asks agents to take completely random decisions on traffic intersections when looking for a parking space. As we saw in Chapter one, in real-life such decisions are not completely random, drivers are capable of applying previous experience and skills in decision making. A simple example is that drivers might not turn back and search for spaces on the same street they have already covered several times. In a random choice model this can happen with a fairly high probability. However, we also saw in Laurier's ethnographic study how the driver eventually did end up repeating the search on the same streets several times and even chose the final parking on one of the streets she initially passed and declined. Hence, it is not unlikely that such repetitions do also occur in real life. This aspect of decision making under conditions of uncertainty is addressed to a limited extent in the *Intel_7* model, which adopts a probabilistic decision making approach, instead of a random one.

Also other, more complex behaviors can help the driver sense vacant or about to become vacant spaces. Many of these qualities are not only hard to capture in a computational model, but remain unclear to us at the cognitive level. A list of some of such activities was presented in chapter one. Human decision making under conditions of uncertainty in street-parking deserves a whole paper on its own. The way I chose to deal with the lack of such knowledge, was to use the same functions of vacant space detection in all models. In other words, the

limitation of only noticing immediately proximate spots in the existing parking model was also used for the detection of randomly vacant parking spots in all other models. Hence, I tried to only compare the added value of the real-time guidance system in relation to random search. Due to the complexity of the task, the added value of experience and intuition had to be unfortunately cancelled out from the models. Hopefully, more of these features can be added to the models in the future.

The Intelligent models assume that the accuracy of reporting vacancies and occupancies of parking spaces is close to perfect. In the real-world, drivers often park on the edges of parking spaces, between two adjacent spaces etc. Such occurrences could cause false reports in the system that tracks the precise availability of free spaces. This uncertainty remains an issue in the models and for the sake of clarity I kept the reporting system faultless. As technology advances over time, it is not unlikely to suspect that such deficiencies could also be eliminated in the real-world.

Additional errors could emerge from people tricking the system, attempting to block parking sensors on the ground in order to avoid public announcement of a space. Also the opposite could occur- false unblocking of sensors, in order to encourage more visitors to drive to a commercial area for example. Vandalism could possibly disable parts of the system, causing bogus information and malfunction in the overall system management. In addition to deliberate cheating of the system, a much more important concern is people's irrational behavior- tt would certainly be wrong to anticipate, that all drivers, who use the real time guidance system, always follow the guidance suggestions. Instead, as with many other technologies, it is likely to expect that people would distrust the system or simply believe in their own intuition more than the suggestions made by the system. This could be accounted for in future models by using a small random error algorithm.

The Intel- 1, 3, 5 and 7 models all suppose that every car has access to the real-time list of available spots and actively uses this list. In a real-world application it is probable that only some people would want to use the guidance system, some would experience technical difficulty and some might not be able to afford it. Hence the results of the system that appear from the model can be too good to be true. However, my intent with the simulation models is not replicating reality, but studying the critical conditions from a more theoretical point of view in order to outline the qualities of different search strategies. Nevertheless, it is likely that if the system provides significant and easy to use aid in locating a convenient parking space, then this is a good enough incentive for the majority of drivers to use the system. A true to life usage ratio could be used in the model, if a credible user study were carried out first.

Perhaps the most serious shortcoming in the simulation models is that agents' adaptive behavior is not complex enough compared to the behavior of drivers in the real world. In the *Traditional* and *Intel_1* models, when an agent starts looking for parking, then the behavioral rules that guide its decisions are the same from the start of the task to the accomplishment of the task- agents only have one strategy. In chapter one I emphasized the importance of adaptive behavior that makes drivers revert to many different goals and strategies if searching takes too long. The Intel_7 model addresses this shortcoming to some degree, by using at least 3 different strategies (i.e. informed competition for vacant spaces, using probability for competing for currently occupied spaces, and using random search if nothing is available) that an agent can use, depending on the environmental conditions. Nevertheless, the agents in the simulation models can only adapt to the outer environment, they have no built-in means to change strategies of the search based on their inner credit assignment. The agents have no capacity of learning nor assessing the efficiency of their strategies, they

simply test one strategy after another in a hierarchical order. I try to compensate for that lack by addressing the issue in text instead of models.

Such are the assumptions with which the simulation models were built. The next chapter will analyze the outcomes the simulations of these assumptions generated.

Chapter Three

Results

We saw in the previous chapter, all models used six parking spaces. The variables that were changed in different simulations were the number of cars (6 and 24, while the number of parking spaces was always kept constantly at 6), different street grid sizes (3x3 and 5x5) and a different distribution of the six spots (one on every street and all on one street segment). The variable that was changed most of all was the type of real-time information that was available to agents, as well as the specific strategy of using it. The four different guidance strategies (Inlte_1, 3, 5 and 7) were presented in the last Chapter.

	Number of Steps			
Model name and Properties	**Average**	**Median**	**Standard Deviation**	**Max.**
"Traditional" 6cars on 3x3 grid	**35.9801**	29	29.01499	162
"Traditional" 24cars on 3x3 grid	**58.8676**	52	38.56921	252
"Traditional" 6cars, 5x5 grid	**133.486**	107	94.19525	535
"Traditional" 24cars, 5x5 grid	**164.038**	159	37.29277	300
"Intel_1" 6cars on 3x3 grid	**18.9373**	16	13.82078	81
"Intel_1" 24cars on 3x3 grid	**69.5191**	71	20.65161	111
"Intel_1" 6cars on 5x5 grid	**29.4841**	30	10.67552	55
"Intel_1" 24cars on 5x5 grid	**120.731**	122	30.60324	179

“Intel_3” 6cars on 3x3 grid	**17.1136**	14	12.58576	67
“Intel_3” 24cars on 3x3 grid	**214.771**	211.5	30.43529	289
“Intel_3” 6cars on 5x5 grid	**27.9552**	28	13.27635	59
“Intel_3” 24cars on 5x5 grid	**183.894**	181	21.04165	240
“Intel_5” 6cars on 3x3 grid	**14.1429**	11	12.84207	74
“Intel_5” 24cars on 3x3 grid	**46.4671**	46	11.24772	93
“Intel_5” 6cars on 5x5 grid	**30.6998**	23	30.24971	143
“Intel_5” 24cars on 5x5 grid	**81.5912**	79	18.20295	148
“Intel_7” 6cars on 3x3 grid	**20.1343**	16	14.40481	76
“Intel_7” 24cars on 3x3 grid	**54.945**	54	12.34717	93
“Intel_7” 6 cars on 5x5 grid	**36.2814**	34	19.99611	108
“Intel_7” 24 cars on 5x5 grid	**62.423**	67	17.38108	88
“Traditional” 6cars on 3x3 grid concentrated	**240.737**	208	148.9806	940
“Traditional” 24cars on 3x3 grid concentrated	**284.687**	277	79.99469	554
“Traditional” 6cars, 5x5 grid concentrated	**636.851**	619	242.8204	1281
“Traditional” 24cars, 5x5 grid concentrated	**764.374**	752	158.0438	1130
“Intel_1” 6cars on 3x3 grid concentrated	**30.0871**	29	15.78522	84
“Intel_1” 24cars on 3x3 grid concentrated	**45.5967**	43	15.41761	93

"Intel_1" 6cars on 5x5 grid concentrated	**49.959**	50	18.1443	105
"Intel_1" 24cars on 5x5 grid concentrated	**59.226**	69	23.70496	119
"Intel_3" 6cars on 3x3 grid concentrated	**33.5031**	25	28.77843	174
"Intel_3" 24cars on 3x3 grid concentrated	**61.3806**	50	42.75369	272
"Intel_3" 6cars on 5x5 grid concentrated	**52.5776**	44	34.56188	182
"Intel_3" 24cars on 5x5 grid concentrated	**252.835**	254	80.20153	404
"Intel_5" 6cars on 3x3 grid concentrated	**23.547**	22	13.14438	64
"Intel_5" 24cars on 3x3 grid concentrated	**31.4074**	31	15.6449	64
"Intel_5" 6cars on 5x5 grid concentrated	**35.0155**	36	11.32714	65
"Intel_5" 24cars on 5x5 grid concentrated	**70.1508**	71	15.58593	97
"Intel_7" 6cars on 3x3 grid concentrated	**42.6401**	42	16.79825	121
"Intel_7" 24cars on 3x3 grid concentrated	**69.3122**	69	21.75266	124
"Intel_7" 6cars on 5x5 grid concentrated	**57.7118**	52	19.4395	143
"Intel_5" 24cars on 5x5 grid concentrated	**82.4779**	85	18.1882	118

Number of parking spots in all models is 6. On the upper half of the table, the six spots were distributed uniformly around the street network. In the lower half of the table, where indicated "concentrated", all six spots were concentrated on one single street.
These simulations were measured over aprox. 3000 and 10 000 iterations.

Figure 11 Comparison of the average number of steps to find a parking space in simulation models.

In most cases, the least amount of average steps to find parking was achieved with the *Intel_5* strategy. Though this might seem logical and self-explanatory, I shall nonetheless try to describe why this is the case.

Effects of Grid Size

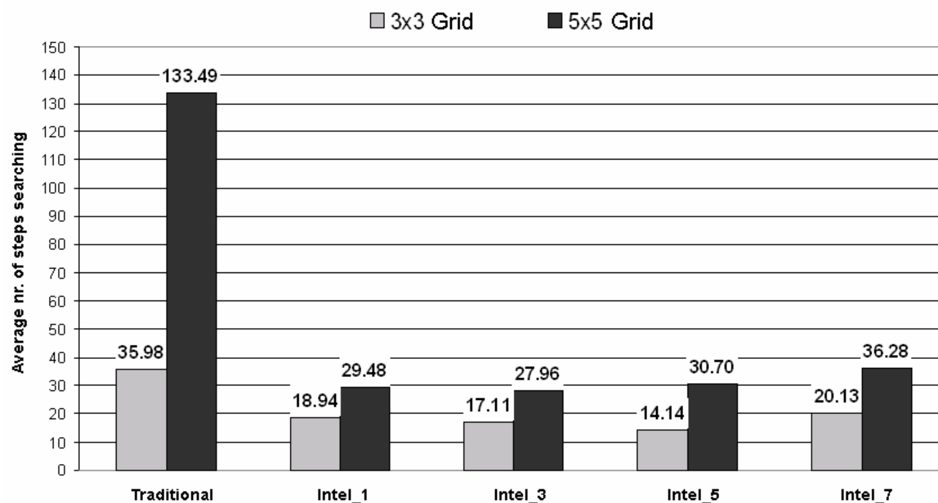


Figure 12 Comparison search efficiencies on a 3x3 and a 5x5 grid with 6 equally distributed parking spaces.

First, it is easy to understand how a random search in the *Traditional* model performed least efficiently in most cases (not all!). In a 3 x 3 grid, there are nine intersections, each of which offers 3 choices (right, straight, left). In total, there are close to 19 683 (3ˇ9 = 19 683) different ways to travel through this grid before repeating a same street segment twice. This gives random search high chances of guiding the driver wrongly before he stumbles upon a street that has a vacant space. However, when parking spaces were distributed so that every street (one street in a 3 x 3 grid is composed of 3 segments) had a parking spot on it, then the success rate of random search increased considerably- it took only 35 steps on average to stumble upon a vacant space (the longer side of one block in

the model is 22 steps long). A highly connected street grid could orient a driver to search fruitlessly on several streets before finding the right street by chance.

The *Intel_1* strategy performed considerably better with 6 cars than random search did on a 3 x 3 grid, but still less efficiently than the *Intel_5* and Intel_3 strategies. The explanation to this is that in *Intel_1,* all cars possess knowledge about the location of vacant spaces, but as they are not aware if there might be other drivers driving towards the same target, then for a certain time period multiple cars might be heading towards the same goal. Only one of those drivers, the closest one, is destined to reach the spot first and occupy it. Hence other competing agents in such a scenario are always driving in vain until they find out that their target spot disappears from the vacancy list when the closest car reaches it. At this point these other agents will try to find a new destination, but under high demand conditions, even then they might have bad luck and loose the next destination to someone else again. If drivers are not aware of competitor's presence and if spots are not reserved, then in case of a competition, all but the closest car will travel in vain until the spot disappears from the vacancy list. This shortcoming becomes less remarkable in a larger grid. While a larger grid makes random search harder, the Intel strategies still guide the agents towards the area where parking spots are located. In a large grid, longer travel distances reduce the possibility of several cars arriving at a destination at the same time. By the time the second or third car with the same target destination arrives at a spot, the first car might already have left. A comparison of the Traditional and Intel_1 models on 3x3 and 5x5 grids confirmed this: In the former case Intel_1 was almost twice as efficient, while in the latter case Intel_1 performed already three times more efficiently.

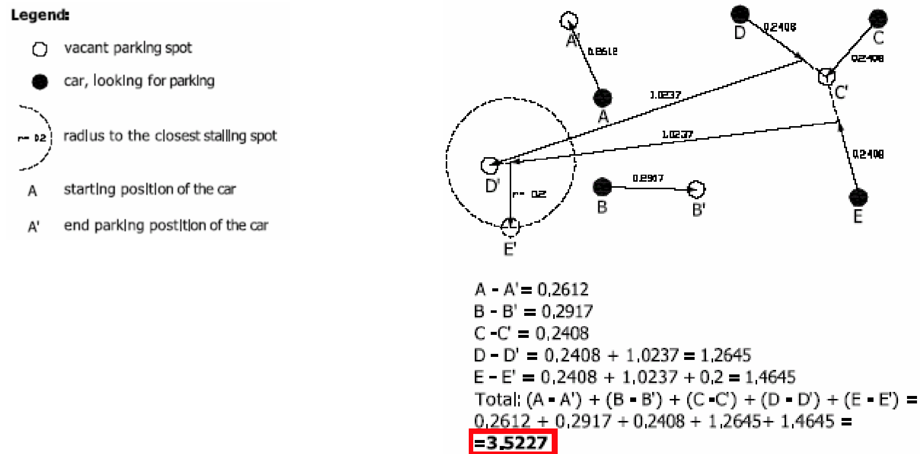


Figure 13 Graphic calculation of the *Intel_1* search strategy.

The table above shows that *Traditional* (random) search performed better than *Intel_1* in the high demand situation with 24 cars and 6 spaces on a 3x3 grid, where spaces were distributed across the grid uniformly. However this result is not characteristic to all situations of high demand. In fact, the results can reverse in a situation where the urban grid is larger. For instance, on a 5 x 5 grid, Intel_1 performs more efficiently than Traditional. In the case of a large grid, random search exponentially accumulates additional possible paths of driving. In a situation of a 5 x5 grid, the simulations of 24 cars and 6 spaces showed that with random search (*Traditional*) it took an average of 164 steps to park, whereas *Intel_1* took 120.

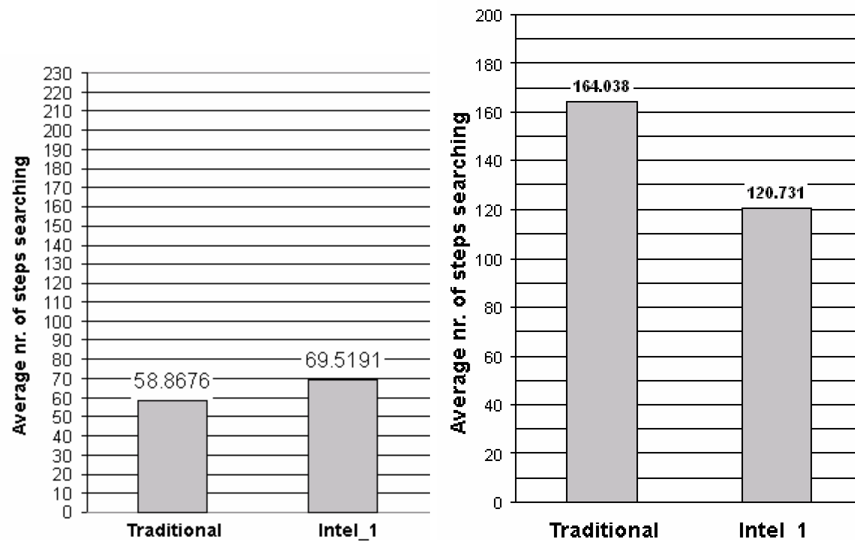


Figure 14 The average number of searching steps of 24 cars on a 3x3 street grid (on the left) and Figure 15 The average number of searching steps of 24 cars on a 5x5 street grid (on the right)

This reversal of efficiency is caused by the additional combinations of driving paths in a 5 x 5 grid, which decrease the efficiency of random navigation. For *Intel_1*, the increase in grid size does not play an important role, as the drivers still choose a specific target, and navigate towards it along the shortest route. If a parking spot is five intersections away, for instance, then the driver will still take the path that is shortest along those five intersections. He will not get caught in all the other approximately 25ˇ9 (2.9 x 10^17) possible paths along the way, which can happen in a random search. Hence, in addition to the precise conditions of demand and supply, the success of the *Traditional* (random) search strategy, is inversely related to the connectivity of the street network: It works well in a small grid, but terribly in a large grid. Real-life neighborhood street networks often have many more streets than 3 x 3 or 5 x 5; the number of traveling combinations that is added with each n x n grids grows exponentially thus also decreasing the efficiency of random search accordingly.

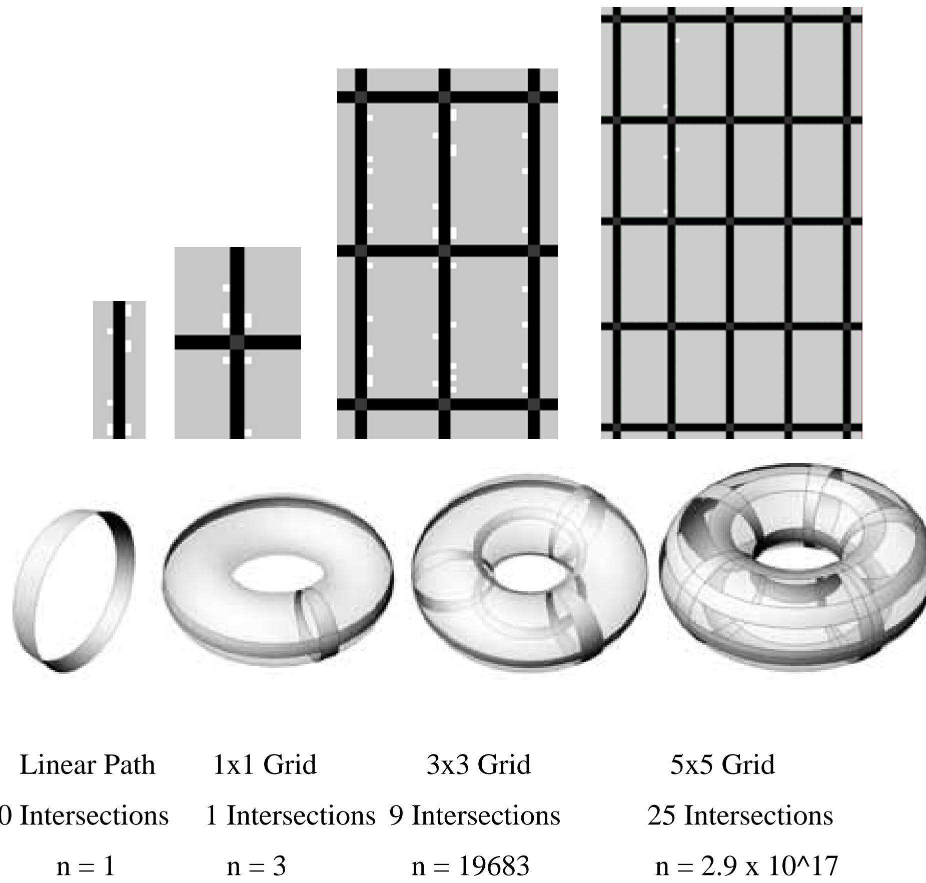


Figure 16 Comparison of the number of alternative travel paths on different grid sizes. The calculations show the approximate number of different travel paths, without repeating any street segment twice and assuming that the search continues till it runs into a dead end.

Effects of Demand and Supply Balance

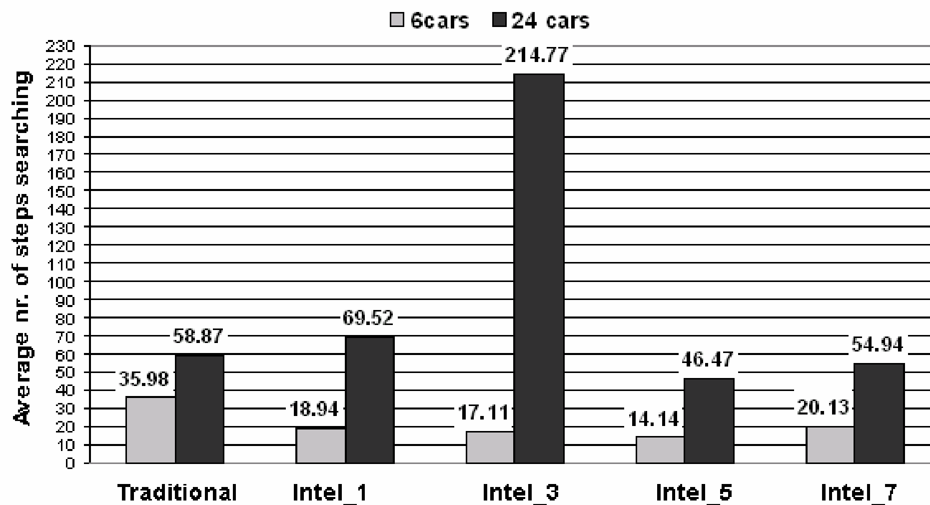


Figure 17 A comparison of strategy performances with 24 cars on a 3x3 grid with 6 geographically dispersed parking spaces.

Interestingly enough, under high demand, random search (*Traditional*) with 24 cars and 6 spaces can perform better than the *Intel_1* strategy. This can be explained again by the fact that *Intel_1* drivers often drive purposefully in vain. While randomness distributes competitors uniformly in the area, Intel_1 can lead many of them towards wrong goals clustered in certain areas. If one space is available in Intel_1and 3 cars compete for it, then all 3 will start moving towards that spot, but only one is destined to succeed. (See Figure 2 above around the vacant parking spot ***C'***). A situation of high demand in *Intel_1* thus reminds me of a poorly coordinated soccer game, where all players storm towards the ball at every step, not realizing that only the closest player will be able to take it. If this keeps repeating in a cycle, then all players will run long distances aimlessly. Ironically this is often the case with young schoolchildren playing soccer. A herd of the kids usually runs after the ball simultaneously without any clear

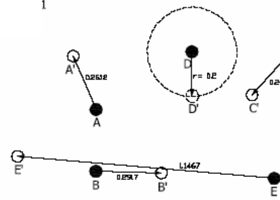
cooperation strategy. Under high demand, the *Intel_1* strategy behaves similarly. The simulation findings show that the amount of driving that result can be even greater than with random search. Random search distributes the demand equally across the grid. In case of soccer, as children mature and their thinking becomes more sophisticated, then they stop running after the ball all at once. Instead, they learn to allocate the ball to the closest players cooperatively. Everyone sees that one of the kids is clearly closest to the ball, and usually others will not run for it. As mental computation is arguably more advanced in older children's minds, then this example demonstrates how higher computational capacity can result in strategies of more efficient commuting.

The *Intel_3* strategy tries to address this shortcoming by allowing cars to make reservations for the spaces they choose. This seems to work fairly well when demand is equal to supply or lower. All cars who find a destination in the real-time vacancy list are guaranteed to have that particular space held for them when they arrive. The reservation blocks all cars, except the one that made the reservation, from parking at the spot. In a situation of demand and supply equilibrium, Intel_3 performed better than Intel_1 and Traditional. However, a poorly strategic reservation system with no knowledge of drivers' locations, might assign a reservation to a driver who is not necessarily the closest competitor for that space. This becomes especially apparent in a situation of over demand. For instance, if only one space is currently available in the neighborhood and two cars are searching for parking, and the reservations mechanism has no way to estimate who is closer to the spot, then chances are 50% that the space might be allocated to the further driver, hence reducing the efficiency of the strategy. The problem becomes even more critical, if there are more than two cars counting on one space. The graphic example below demonstrates that a reservations system with 5 cars and 5 spaces, which is not aware of cars' locations and hence does not strategically allocate spaces to the closest cars, would probably allocate spaces to further cars, thus increasing the

overall search times. However, as this system still assumes that cars only try to reserve the spots that are closest to them, then two of the spots in the scheme below are always allocated to cars **A** and **B** because they are the only ones competing for them. Instead of 5! (120) different allocation possibilities, only 3! (6) are left. This means that in the simplistic scheme below, only one out of six times would the system allocate the spots in the most efficient way. In five times out of six, the unknowledgeable reservation policy would allocate the three parking spaces in a less efficient manner, forcing cars to drive longer distances than necessary.

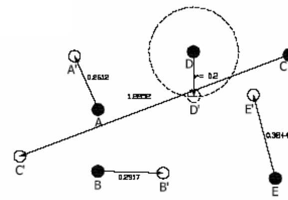
Even though reservations can be an efficient method to ensure that individual drivers do not approach target spaces in vain, unless the system has a clever allocation strategy, reservations can often go to drivers who are not closest to these spaces. That is the case in a first-come-first-serve reservation policy. However, even if this error were fixed, and the reservation system given accurate methods for matching cars with closest spaces in a sustainable way, then a serious deficiency still remains. This deficiency is caused by the fact that in a situation of high over demand, there are by definition many cars in the area and the balance of demand and supply varies constantly. New searchers appear frequently. As new demand may appear on any street, then the newcomers are likely to occasionally appear closer to vacant spaces than the drivers who are driving towards them with reservations. Because these spaces are reserved, they cannot be occupied by these occasional passers-by and hence the overall turnover of parking spaces decreases. In other words, the concept of reserving under high demand is meant to prohibit the use of a particular spot from everyone but the reserver, which by definition reduces turnover. This was confirmed in the 6 spots, 24 cars simulations, where *Traditional* search required an average of 58 steps to find parking while *Intel_3* required 214.

Scenario 1: Among the cars C, D and E, who all compete for the same closest space, C gets the first reservation and E gets the subsequent reservation to the space on the far left.



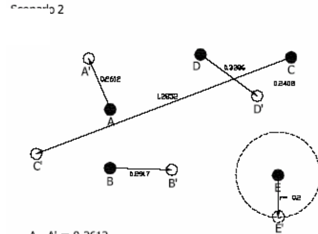
A - A' = 0,2612
B - B' = 0,2917
C -C' = 0,2
D - D' = 0,2408
E - E' = 1,1467
Total: (A - A') + (B - B') + (C -C') + (D - D') + (E - E') =
0,2612 + 0,2917 + 0,2 + 0,2408 + 1,1467 =
=2,1404

Scenario 3: Among the cars C, D and E, who all compete for the same closest space, E gets the first reservation and C gets the subsequent reservation to the space on the far left.



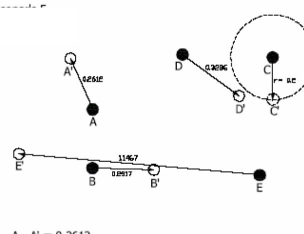
A - A' = 0,2612
B - B' = 0,2917
C -C' = 1,2852
D - D' = 0,2
E - E' = 0,3844
Total: (A - A') + (B - B') + (C -C') + (D - D') + (E - E') =
0,2612 + 0,2917 + 1,2852 + 0,2 + 0,3844 =
=2,4225

Scenario 2: Among the cars C, D and E, who all compete for the same closest space, D gets the first reservation and C gets the subsequent reservation to the space on the far left.



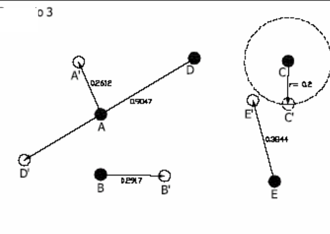
A - A' = 0,2612
B - B' = 0,2917
C -C' = 1,2852
D - D' = 0,3286
E - E' = 0,2
Total: (A - A') + (B - B') + (C -C') + (D - D') + (E - E') =
0,2612 + 0,2917 + 1,2852 + 0,3286 + 0,2 =
=2,3667

Scenario 4mong the cars C, D and E, who all compete for the same closest space, D gets the first reservation and E gets the subsequent reservation to the space on the far left.



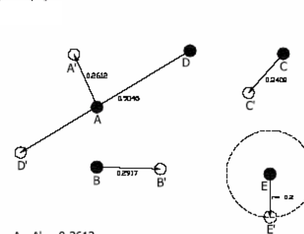
A - A' = 0,2612
B - B' = 0,2917
C -C' = 0,2
D - D' = 0,3286
E - E' = 1,1467
Total: (A - A') + (B - B') + (C -C') + (D - D') + (E - E') =
0,2612 + 0,2917 + 0,2 + 0,3286 + 1,1467 =
=2,2282

Scenario 5o **3**: Among the cars C, D and E, who all compete for the same closest space, E gets the first reservation and D gets the subsequent reservation to the space on the far left.



A - A' = 0,2612
B - B' = 0,2917
C -C' = 0,2
D - D' = 0,9047
E - E' = 0,3844
Total: (A - A') + (B - B') + (C -C') + (D - D') + (E - E') =
0,2612 + 0,2917 + 1,2852 + 0,3286 + 0,2 =
=2,042

Scenario 6 \mong the cars C, D and E, who all compete for the same closest space, D gets the first reservation and E gets the subsequent reservation to the space on the far left.



A - A' = 0.2612
B - B' = 0.2917
C -C' = 0.2408
D - D' = 0.9047
E - E' = 0.2
Total: (A - A') + (B - B') + (C -C') + (D - D') + (E - E') =
0,2612 + 0,2917 + 0,2408 + 0,9047 + 0,2 =
=1,8984

Average: 2.
Best: 1.8984

Figure 18 Graphic calculation of the Intel_3 strategy. In the case of a first-come-first serve reservation policy, this example has 6 different outcomes, of which only one is optimal.

An analogy to this situation can be found in restaurant reservations. If the demand for a particular restaurant is high, then a line forms at its door. If some tables have been reserved by expected visitors in advance, then the people appearing at the door are not allowed to occupy those reserved tables. If this were not so, and the people at the door were immediately allowed to seize vacant tables, then the overall turnover of the tables would be greater. Hence, in situations where demand surpasses supply, reservations on street parking are mostly likely to decrease the overall efficiency of the system. Of course other aspects should be considered, for instance, reservations could be highly priced and used to collect money for some public good related to urban transportation, but this is a different point.

Effects of the Geographic Distribution of Parking Spaces

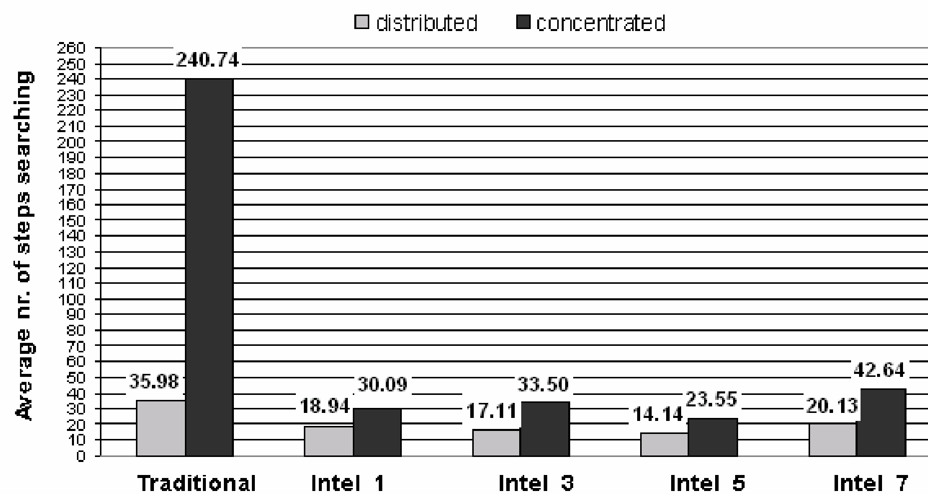


Figure 19 A comparison search efficiencies with uniformly distributed or locally concentrated 6 parking spaces on a 3 x 3 grid.

A third crucial factor that determines the efficiency of a strategy is the location of the parking spots in a given grid. In the first examples above, the six spaces were dispersed equally apart, so that one was found on every street. Such a distribution greatly enhances the chances of finding a parking space with random search. However, if parking spaces are all concentrated in a specific limited area, then it becomes much less probable to find a space by random search. This was clearly illustrated in the results of the simulation model. In a uniformly distributed parking field with 6 spots and 6 cars, heuristic search required 35 steps on average. When all six parking spaces were situated next to each other on a single street segment, then the average amount of steps raised to 240. In a 5 x5 grid the corresponding number raised from 133 to 636.

Though this seems clearly intuitive, it is important to emphasize that a concentrated location of parking spots in a given area can greatly diminish the odds of finding a space with random search and therefore increase the value of a real-time guidance system.

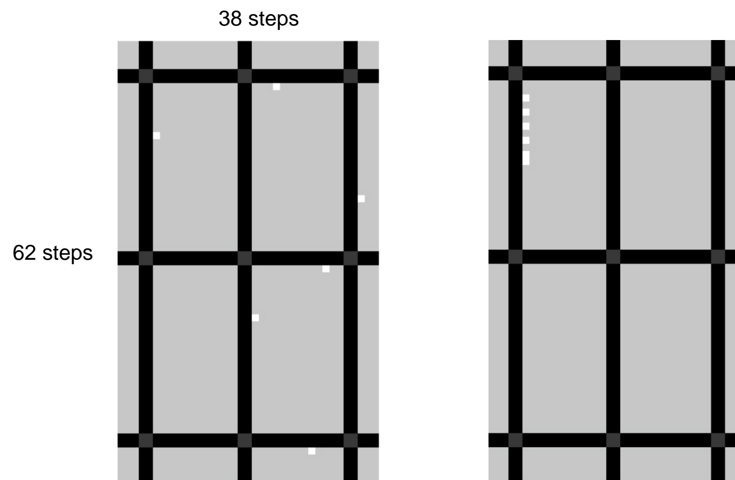


Figure 20 Six parking spaces on a 3 x 3 grid distributed uniformly and concentrated.

The location of parking spots also plays an important function in the *Intel_1* strategy, more so than any of the other *Intel* strategies. If spots are concentrated in close proximity within a limited geographic area, then the abundant decisions to drive towards a single spot that 10 drivers are competing for, as illustrated by the children's soccer example above, is not so useless anymore for Intel_1 participants. Instead, following a false lead that might not be fruitful in the first round, will still lead the driver towards the right area where all the parking spaces are located. This effect proved to be so useful, that even under low demand, *Intel_1* performed more efficiently than Intel_3 with a concentrated parking-space distribution (which was not the case with the dispersed spaces simulation). In a condition of high demand, this was even further apparent: in case of dispersed locations of parking spots, it took *Intel_1* with 24 cars on a 3 x 3 grid an average of 69 steps to find parking, versus only 45 steps when the parking spots under same conditions were concentrated on a single street.

Synthesis of Strategies

The general highest efficiency achieved by the *Intel_5* and *Intel_7* strategies can be credited to several cooperating features of these strategies. Intel_5 avoids the shortcomings that appeared with the reservations in Intel_3. In fact, due to the capacity to assess which parking spot the agent is likely to reach first, reservations become unnecessary. Only in cases of special demand should reservations be made and the price charged for the convenience (in compensation for the inconvenience caused to others) should be accordingly high.

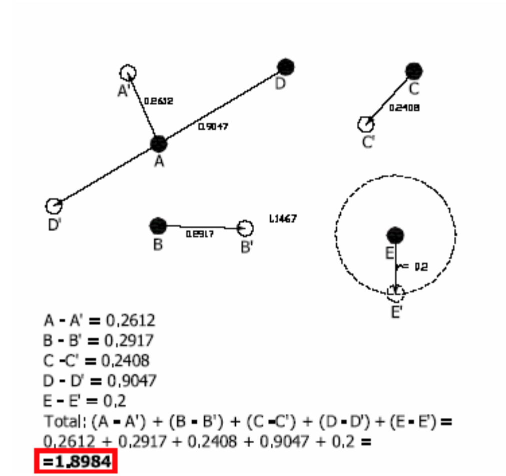


Figure 21 Graphic calculation of the Intel_5 search strategy.

The table shows that Intel_7 performed better than Intel_5 only in the most challenging situation- on a 5x5 grid with 24 distributed spaces. However, the differences between the two models were not large and it is very likely that with a different initial position of agents, the advantages could reverse. In situations of low demand or small grid size, the performance differences of the two models are too small for clear conclusions.

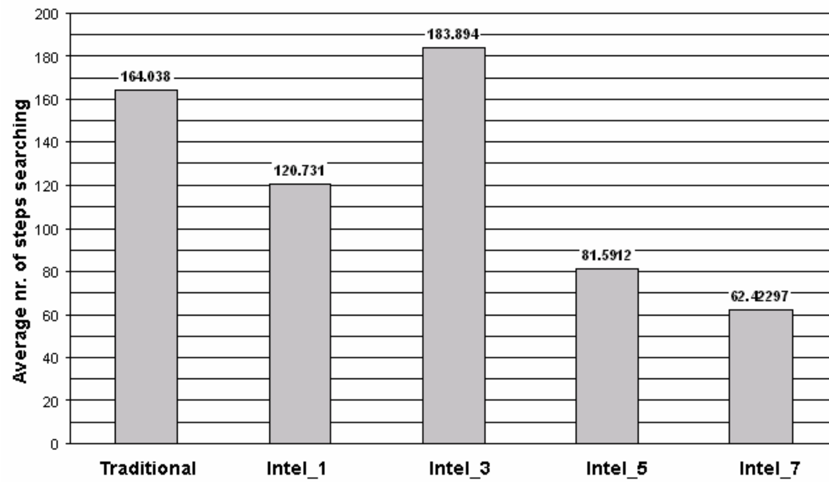


Figure 22 Comparison of search strategies on a 5x5 grid with 24 uniformly distrinuted spots.

Compared to the deficiencies we indicated for Intel_1, Intel_5 computes the parking destination not only based on the locations of a vacant spot, but also based on an assessment of obtaining the spot in case of a competition. Referring again to the example of a soccer game, participants in Intel_5 have information and computational capacity to understand that chasing a ball that someone else is closer to, is useless. As cars will only follow a destination that they are almost sure of obtaining, then the collective storming for a vacancy, which *Intel_1* faced, is avoided.

I deliberately said almost in order to include the possibility of new cars appearing for a search during the time when an agent drives towards its destination. As the parking system is dynamic, changing in real-time, then new demand can appear while the previous demand situation is being solved. When car A is driving towards a spot that it has calculated itself to be closest to, there can be a new car B entering the area, who also wants to park. When car A did its calculation on the previous step, car B was not part of the scene yet, and could therefore not be accounted for. But when B appears and happens to be located closer to the destination that A is driving towards, then B is likely to reach the spot first. This is why the Intel_5 strategy needs to re-evaluate its target, as well as the chances of obtaining the target, at every step. In the example of cars A and B above, car A will automatically know when B appears and will therefore stop following a target that B is closer to. This is one of the efficiency advantages of Intel_5 and Intel_7 over Intel_3, but also a humanly inconvenient aspect of the Intel_5 and Intel_7 strategies: practically, it can happen that a given driver is forced to choose its parking destination more than once, changing it along the way if newly arrived searchers intervene with his plans.

The occasional loss of a target space due to newly appearing demand is characteristic to all models, but from the group efficiency point of view, Intel_5 has the most capable means for coping with the situation. In Intel_1 a similar

scenario is also constantly caused by over demand. When a particular street has fewer parking spaces to offer than cars that need to be accommodated, then target destinations can also be lost to current competitors (in *Intel_1* and *Traditional).* Unlike other strategies, in *Intel_5* an agent is immediately aware if its search is likely to be fruitless. This is an important and powerful advantage of Intel_5 and leads us to suggest that due to this knowledge, queuing cars could be accommodated in a different way than they are today, cruising and polluting in dense traffic aimlessly. I will come back to this point in a later section dedicated to queuing.

Simulation Conclusions

The list below recaps the effects of the studied variables in the simulation models.

- The size of the grid determines the number of possible paths on the grid. As the grid size increases, possible commuting paths increase exponentially, reducing the effectiveness of a random search and increasing the value of a guidance system. If multiple agents drive towards the same destination, then a large grid also increases the distances that all but the closest driver cross in vain.

- Randomness distributes agents evenly across the field, whereas guided strategies direct agents towards the available parking spots. If no cooperation between agents happens, then agents can agglomerate around a single spot. If they are unable to seize the spot, then they can drive around the block in circles, which can extensively increase cruising.

- A uniform distribution of spaces across the street network increases the efficiency of random search, while a concentrated distribution greatly decreases the efficiency of random search. The opposite is true for a guided search. When agents are guided towards the agglomeration of parking spots concentrated in one area, then their chances of seizing an arbitrary vacant spot around their original target increases. Good examples of this effect in real-life are structured parking garages containing several vacant spaces. If drivers know the location of a garage, then they can approach the garage hoping to park at a specific space in the garage, but if the specific spot happens to be taken, then chances are good that another spot will be vacant in the same facility.

- Reservations can be useful in low or equilibrium demand conditions, but in high demand conditions they increase overall cruising. However, from the point of view of group efficiency, under all circumstances, reservations are only efficient if they have a strategic allocation system, distributing spots to the agents that are closest to them. Strategic allocation becomes especially important when the competition for spots increases. By definition, a reservation policy prohibits newly arriving passers-by from seizing a reserved space; this reduces the turnover of spaces and can produce great inefficiencies in allocation under high demand.

- If agents have information about their competitors as well as their location in relation to parking spots, then agents can be guided only towards the parking spots that they can surely occupy before others. This strategy eliminates the common need for reservations as well as cruising in vain. Mutual awareness of each other’s locations also allows newly arriving parkers to immediately enter the allocation pool on equally competitive terms. This means that if a newly arriving searcher happens

to start closer to a vacant parking spot than a previous searcher, then the new agent will have priority over the spot. This reduces overall cruising.

The results of the simulations indicate that the advantages of a more sophisticated strategy appear more clearly under more critical conditions. The advantage of Intel_5 above other strategies was apparently more remarkable in situations of 24 cars and 6 spots than in 6 cars and 6 spots, on a 5x5 street grid more than on a 3x3 street grid and with a concentrated distribution of spots rather than dispersed spots. This is again intuitive: in simpler situations almost any strategy can give a satisfactory result, while in challenging situations simpler strategies fail and intelligent strategies prevail.

As the five strategies (Traditional; Intel_1; Intel_3; Intel_5 and Intel_7) gradually built up in complexity, then the results also showed that the usage of more information and computational power for navigation can give better results. This is coherent with Herbert Simons argument about intelligent systems that I quoted in the very beginning of this thesis: “The behavior of an artificial system may be strongly influenced by the limits of its adaptive capacities- its knowledge and computational powers.” [Simon 1996, p. 29] However, computational capacity and abundant information alone do not automatically result in efficient performance. The fundamental strategic differences between models Intel_1, Intel_3 and Intel_5 demonstrate that for a successful performance, information and computation must be well coordinated. If this is not the case, then even worse results can appear than in a random search technique (*Intel_3* with reservations on a 3 x 3 grid and 24 cars competing for 6 spaces resulted in an average of 214 steps of searching, while Traditional random search under the same conditions required only 58 steps on average).

The use of more general information, that is, taking account the actions of other agents as well as the overall competition situation of the area, gave better results

than a simple use of individual information. Furthermore, a probabilistic search with the Intel_7 model that also accounted for the currently occupied spaces and statistically compared the values of driving to a vacant or a probably vacating spot, achieved even more efficient results. The success of these strategies was enhanced by collaboration between agents. This leads me to suggest that Kahneman's theory about the deficiencies of immediately perceivable decision making might apply to parking indeed. The use of broad and crosscutting information gave better results than the use of narrow individual information.

Intel_9: The Collaborative Equilibrium and Game Theory Model

A 6th model, which was not tested in agent-based simulations could be added to the list of strategies outlined above as potentially even more efficient in the reduction of cruising than Intel_5 or Intel_7. We can call it *Intel_9*. This strategy for street-parking introduces an interesting aspect of Game Theory, namely the *Prisoner's Dilemma (PD).*

In Intel_9 we include another new variable to the parking system: dynamic pricing. As we saw in Chapter 1, pricing is one of the most influencing variables in real–life parking conditions. Donald Shoup has convincingly argued that fair market-rate pricing can alone be a powerful mechanism for reducing demand on street parking. My aim here is slightly different: Assuming that conditions of over demand will always continue to exist in popular areas, I shall try to propose that dynamic pricing could be used as a tool for creating incentives for collaborative behavior in overcrowded street parking. As drivers in situations of over demand are expected to act self-interestedly, then such behavior can be characterized by Game Theory.

The Prisoner's Dilemma (PD) is a classic example of Game Theory, where two or more players react to each other with the goal of maximizing their own profit. I shall quote the description of the classical Prisoner's Dilemma from the Wikipedia encyclopedia[19] as follows:

"Two suspects, A and B, are arrested by the police. The police have insufficient evidence for a conviction, and, having separated both prisoners, visit each of them to offer the same deal: if one testifies for the prosecution against the other and the other remains silent, the betrayer goes free and the silent accomplice receives the full 10-year sentence. If both stay silent, the police can only give both prisoners 6 months for a minor charge. If both betray each other, they receive a 2-year sentence each. Each prisoner must make a choice - to betray the other, or to remain silent. However, neither prisoner knows for sure what choice the other prisoner will make. What will happen?

It can be summarized thusly:

	Prisoner B Stays Silent	*Prisoner B Betrays*
Prisoner A Stays Silent	*Both serve six months*	*Prisoner A serves ten years* *Prisoner B goes free*
Prisoner A Betrays	*Prisoner A goes free* *Prisoner B serves ten years*	*Both serve two years*

The dilemma arises when one assumes that both prisoners only care about minimizing their own jail terms. Each prisoner has two options: to cooperate with his accomplice and stay quiet, or to betray his accomplice and give evidence. The outcome of each choice depends on the choice of the accomplice. However, neither prisoner knows the choice of his accomplice. Even if they were able to talk to each other, neither could be sure that he could trust the other.

[19] The Prisoner's Dilemma was invented by Merril Flood and Melvin Dresher in 1950. Since then, the dilemma has become a widely used model for predicting conditions of uncertainty in economics. http://en.wikipedia.org/wiki/Prisoner%27s_Dilemma

Let's assume the protagonist prisoner is working out his best move. If his partner stays quiet, his best move is to betray as he then walks free instead of receiving the minor sentence. If his partner betrays, his best move is still to betray, as by doing it he receives a relatively lesser sentence than staying silent. At the same time, the other prisoner's thinking would also have arrived at the same conclusion and would therefore also betray.

If reasoned from the perspective of the optimal outcome for the group (of two prisoners), the correct choice would be for both prisoners to cooperate with each other, as this would reduce the total jail time served by the group to one year total. Any other decision would be worse for the two prisoners considered together. When the prisoners both betray each other, each prisoner achieves a worse outcome than if they had cooperated."

Uncertain about the decision of the partner, it is assumed that rational prisoners in a one time PD would decide to betray their partner in order to maximize their own benefits.

If the prisoner's have to repeat such a dilemma multiple times (*Repeated Prisoner's Dilemma*) then the situation changes drastically. "Repetition is a kind of enforcement mechanism, which enables the emergence of cooperative outcomes in equilibrium, when everybody is acting in his best interest."[20] In a repeated game, the optimal solution for a prisoner is not betrayal of the partner anymore, but cooperation instead. This is because in a repeated game, a betrayal of the partner will most likely be responded to with a similar betrayal in the following round. Betrayal in a first round would lead to a constant mutual treachery, where both prisoners eventually realize that the betraying the other also diminishes their own gains. In the extended PD players thus get to know each other, and soon realize that selfish action will only result in a similar

[20] This is the fundamental insight upon which Robert J. Aumann was awarded the 2005 Nobel Prize in economics.

response, which jeopardizes both players. Hence, it is not in their best interest to betray. Instead, the best strategy to react would rather be cooperation in the first round. In Game Theory this is called the folk theorem: *Cooperative outcomes in the outset of the game correspond to equilibrium outcomes in the repeated game*.[21] After the first round, both players can adjust their strategies depending on each other's responses. However, if due to a bad start, a repeating equilibrium of betrayal is achieved, then an unexpected cooperative choice from one of the players can re-establish a cooperative equilibrium. In order to avoid such looping conditions, Nash equilibriums have been proposed. Nash equilibrium is a set of strategies that prevent both player from having an incentive to unilaterally change their actions. "What is maintaining the equilibrium in a repeated game is the threat of punishment, not carrying it out. If you like, call it MAD- mutually assured destruction, the model of the Cold War".[22]

The Intel_9 parking strategy that takes advantage of the Repeated Prisoner's Dilemma could function as follows. The computational features of this strategy are similar to those of Intel_5. At every step agents evaluate their chances of going to each available parking spot in their vicinity, but only start driving to a specific destination if they are the closest of all competitors to the given spot. If nothing is available, then agents act similarly to Intel_7, that is, they navigate probabilistically towards the occupied spaces that might shortly vacate. The important difference with the former strategies is that agents can increase their personal profit even more if they choose to cooperate with other agents. This is how it works: assume that the price of street-parking is dynamically adjusted with the goal of reducing the amount of cruising in a given area. Due to the infrastructure, which is already set up by the guidance system, it is relatively easy for the system to assess the amount of people looking for a parking space as well as the distances they cover in the search process. As the number of parking

[21] Discovered by various people, notably Ariel Rubinstein.
[22] Quotation from Robert J. Aumann's 2005 Nobel Prize in economics award speech.

spaces is finite, then each searching car increases traffic in the area. In order to keep prices low, it is in everybody's interest to reduce cruising on the streets.

When choosing the closest available parking space that a driver can surely occupy, it becomes important to weigh that decision with the overall performance of all searching cars. For example it might happen that a parking space, which is the nearest to driver **A** is also the nearest to driver **D**. But as **A** is closer than **D** to the spot, then similarly to Intel_5, it seems like **A** should get the space. However, if the second closest spaces are taken into account, this might cause **D** to make a large detour to its next best closest space, increasing the overall amount of cruising in the neighborhood and thus increasing the price of parking for everyone. Instead, if **A** decided to give up his closest spot to **D** and take a second closest spot himself, then the total amount of cruising could be reduced considerably and parking would be cheaper for everyone. The incentive to cooperate is the reduced price of parking.

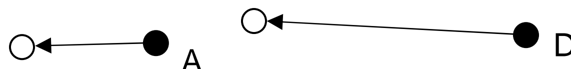


Figure 23 Graphic example of the benefits of collaborative behavior.

Assuming that at least some people collaborate, this strategy can cause less cruising than *Intel_5*. If a driver does not cooperate, then he can earn the

irritation of others and be treated correspondingly. The optimal solution for everyone is a collaborative equilibrium.

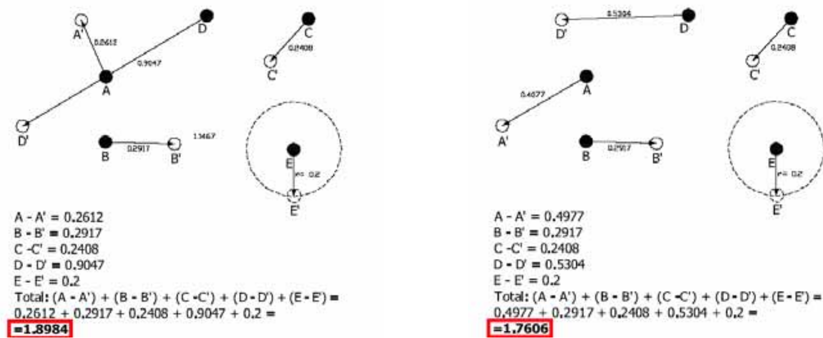


Figure 24 Graphic calculation of Intel_5 and Intel_9 strategies.

A potential issue with this strategy is that wealthier drivers, who might not be sensitive to price raises, would never collaborate. This would cause lower efficiencies in the overall performance and higher prices for all drivers. In a well-performing system, drivers should indicate the spot they are planning to seize before driving to it as well as their distance from it. As long as all agents have accurate information about who is heading where and how far they are from their targets, then a non-collaborative driver will not have a major impact on the performance of the system. If nobody collaborates, then the efficiency of the system is just as good as it was in Intel_7. If agents collaborate, it can only be better. The decision to collaborate or not would depend, among many other factors, on the scale of the price incentive that one can gain.

A solution to this issue could be a personalization of the benefits. If the administrating program of the system recognizes that a person has acted collaboratively, and thereby contributed to the reduction of cruisers, then a lower

price for parking can be offered to that person individually. In the opposite case, a higher price ought to be offered.

Consistently with the previous Intel strategies, all the computation of possible choices should happen in a computer that the driver carries, not in the drivers head. The driver could simply demand 3 different parking solutions: a cheaper collaborative offer, a slightly more expensive non-collaborating offer, and a possibly highly charged reservation option, depending on the overall demand in the reservation area.

The dynamic pricing mechanism could also be used to accommodate different time and location requirements of drivers. If a person is in a great hurry or simply unwilling park anywhere but a specific place, which also happens to be demanded by others, then a higher price can be charged for a priority reservation. Depending on the time availability and financial resources of a driver, the computational interface in the vehicle could include indicators for one's willingness to wait or pay. Drivers in a rush could thus always choose to overpay the less urgent drivers and immediately gain access to their desired areas. Reservations could ensure that the spot remains vacant until the person arrives.

A collaborative equilibrium as proposed by Intel_9 could thus also accommodate individually different needs and charge for the level of service accordingly. Collaborative commuters would be charged least and the prioritized ones most, similar to any other travel industry.

Importance of Efficient Queuing in Real-time Systems.

So far we have been testing strategies that can allocate a finite amount of parking spaces to different cars, dispersed in a finite area. The central question I have

been addressing thus far has been *What is the most satisfycing way of distributing a given amount of parking spaces, so that least searching is required?* I have tried to demonstrate how distant information and mobile computing capacity can help traveling agents find the quickest parking solution as well as reduce the overall cruising and polluting on popular streets. I have not seriously addressed the issue of what to do with surplus cars, which simply cannot be parked at a given moment due to insufficient parking space. In other terms, I haven't dealt with the issue of queuing.

Real-time communication operations introduce new management challenges to the realm of urban planning. In the past, large clustering of people, massive group meetings, protests, parades or open air spectacles appeared relatively rarely in cities. Such events require a considerable amount of organization and preparation. To avoid large scale conflicts in such circumstances, administrative organizations have over centuries developed sophisticated policing techniques to keep things under control. For instance, most public meetings in developed countries require official permits. A prior notice allows administrators enough planning to ensure that the events unroll without conflicts. Because simultaneous mobilization of large amounts of people also exerts unusual demand on public infrastructure and transportation, then adequate preparations are done well in advance to avoid over congestion and queuing.

Real-time communication in urban resource distribution can increase the amount of instant mobilizations drastically, without leaving nearly as much preparation time for authorities and urban system managers to cope with unexpected situations. In digital communication, the importance of time delay caused by physical distances disappears, allowing simultaneous gatherings and demand to appear instantly. This paradigm is well known in information technology and more recently in urban literature under titles like *flash mobs*, *digital communication waves* and *flocking* [Mitchell 2003; Castells 2006]

For distribution of physical resources, such as parking spaces, the phenomenon of instant communication strongly reinforces the need for good system management, particularly queuing.

In traditional systems of urban resource distribution, participants interact relatively slowly. Simultaneous queuing can be absorbed by the slow interaction of participants. For instance, we can imagine a town **A**, which has particular resources and people **B**, **C** and **D**, who live outside of town **A**.

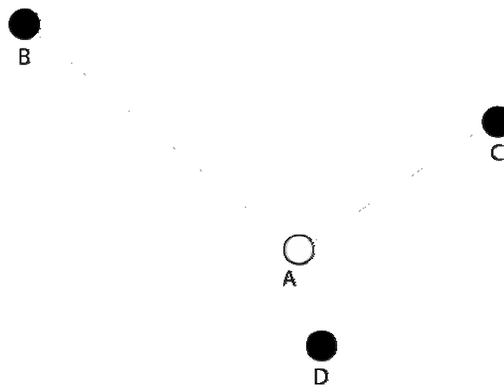


Figure 25 Absorption of physical queues in slow-interaction systems.

People **B**, **C** and **D** are all at different physical distances from town **A**. If all three people develop a need to use a certain resource in town **A** at the same moment, and start driving towards town A simultaneously, then congestion and queuing at point **A** are possibly avoided because it takes **B**, **C** and **D** different times to reach **A**. Assuming it is a relatively quick service, then by the time **B** gets to point **A**, **D** and **C** might have already left. In other words, the potential queuing is absorbed in different travel times.

In real-time communication systems, on the other hand, physical distances do not absorb queuing times. If on the same scheme above, all participants **B**, **C** and **D**

had a real-time communication system to reserve appointments at **A** and their necessities to use the resource at **A** again formed at the same moment, then their request would be received instantaneously at **A** and a queue would have to be managed. In a virtual communication system information flows at the speed of light and physical distances do not absorb simultaneous demands. Hence in any real-time space management system where multiple participants can exert a simultaneous demand on the same resource, queue management becomes a crucial efficiency issue.

In the light of those two scenarios, street parking offers an interesting mixture of both aspects. If street-parking can use real-time guidance technology for making destination choices at a distance, and possibly reservations, then demands from competing participants for the same spot can arrive at the same instant. If cars **B** and **D** compete for the same spot on a virtual reservations screen, then their demands for the spot arrive at the allocation system simultaneously. One of the drivers will have to be accommodated in a queue or redirected to search in another area. However, similarly to the slow interaction scenario, driving to the chosen spot still requires physical travel. Therefore an efficient system should be capable of evaluating which car should have priority for the reservation. A certain amount of queuing can be absorbed by the fact that it takes different cars a different amount of time to reach the destination. In other words, if car **B** reaches the destination much faster than car **D**, then it is possible that by the time car **D** reaches the spot, car **B** has already left and no queuing is necessary. Such decisions could be based on probabilistic and intelligent guessing and past experiences. Unless the reservation is given to the driver who happens to be closer or otherwise capable of reaching the space sooner, the turnovers of spaces can be reduced, just like we saw in the Intel_3 model.

There are multiple ways to manage queues of cars waiting to be parked. Two general categories of queuing in spatial systems like street-parking could be 1) dynamic queuing and 2) static queuing.

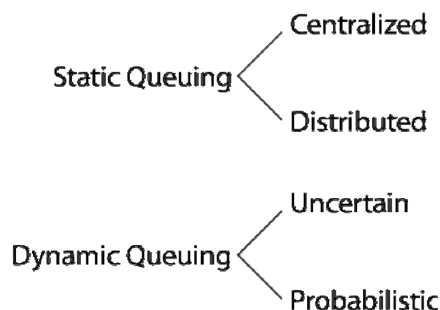


Figure 26 Different Queuing Strategies in Street Parking

Dynamic Queuing

In case of over demand in the current parking system, a relatively large group of people cruise and congest traffic in vain. In other words, cars that are waiting for parking form *dynamic queues* by driving around in traffic and searching for vacating spaces. This sort of queue management forms a serious environmental problem, by creating congestion (up to 30% of all traffic in a rush-hour C.B.D [Shoup 2005]), air pollution and augmenting the risk of traffic accidents.

Despite these effects, real-life dynamic queuing is hardly describable as random. Knowledge from previous experiences and learned intuition help most drivers use intelligent guessing in the search process. The intel_7 model tried to address this issue, by allowing agents to use a probabilistic search strategy. This strategy, derived from the idea of implicit enumeration search[23], does not guide a driver to the closest available space, but makes a prediction, based on limited knowledge,

[23] Developed by William J. Mitchell, Robin Legget

about which area of the search tree is most likely to yield positive results. Hence, the guidance system can inquire knowledge about the occupancy periods of each taken spot, and make intelligent predictions when these occupied spaces might vacate. This strategy implies that dynamical searching (active cruising) can clearly increase the chances of obtaining a parking space sooner. That is, instead of waiting for a vacancy to occur, a driver can already drive towards a soon-to-be-vacated space.

Static Queuing

An alternative method of queuing could be an allocation of special short term stalling spaces for cars that are waiting to park. If drivers would be informed in real time that there are currently no spots available in a neighborhood and assuming that they would act rationally, they could stop searching. Instead of forming a disguised queue in the moving traffic along with passing cars, parkers could use designated stalling areas that specifically accommodate the queue of searchers. Such static queuing could economize the gasoline burnt during the process of fruitless cruising and minimize traffic congestion. When a parking space in the vicinity of the stalling area is vacated, then a driver could notice an appropriate message on his communication device, and drive to it from the stalling space.

Static queues could be managed in either centralized or distributed ways. A centralized static parking queue could provide a common stalling area for several cruising cars, similar to a taxi stop.

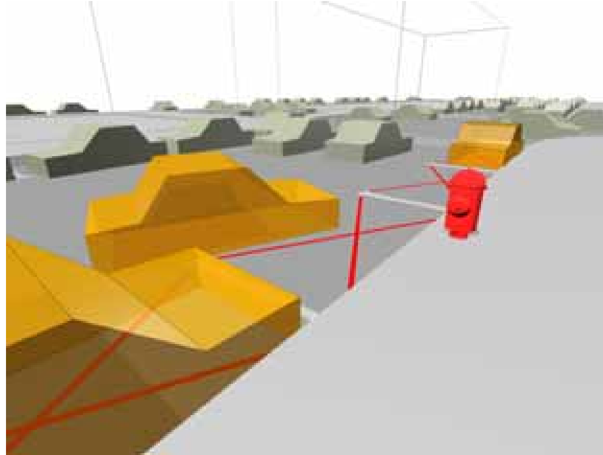


Figure 27 Centralized stalling area, similar in design to a taxi stall on a street corner.

The positive aspect of such stalling is that centralized locations can be easily remembered by drivers, which makes returning to the stall undemanding for frequent parkers. For first time users a collective queuing location could also be easier to find by inquiries from local people or a digital map. Queue management would be simple and straightforward, comparable again to the yellow cab queue: the first car in the row would be allocated the first vacating parking spot in the neighborhood, the next one to the second and so on. The clarity of such a linear allocation system makes it simple to comprehend for all drivers and is analogous with most physical queues that people are accustomed to.

On the other hand, a collective queuing stand can also have negative effects on overall efficiency of the parking system. Drivers would be subjected to unpleasant equity, forced to stand in line with all other drivers, even though their time and financial availability might vary significantly. In Chapter 1 we saw an empirical description of a person driving to a train station in a great hurry. The current parking system offers different services for people with different time availability or willingness to pay. In case of a hurry, a driver can decide to use an

expensive private parking structure or valet parking, achieving the goal in a costly but immediate way. In a collective stalling area, such individual differences would be subject to group attention. Passing other cars in the line would stir up conflicting feelings amongst other members of the line.

Centralized stalling areas would also require a substantial amount of access driving. By definition, collective stalling areas, which assemble searchers from a relatively large geographic extent, should be located at sparser intervals than individual stalling spaces, which creates larger access radiuses. The time and distance spent on driving to the queue from the location where a person starts searching, and then in turn to a vacated parking space in another location, would in most cases cause more driving than distributed stalling.

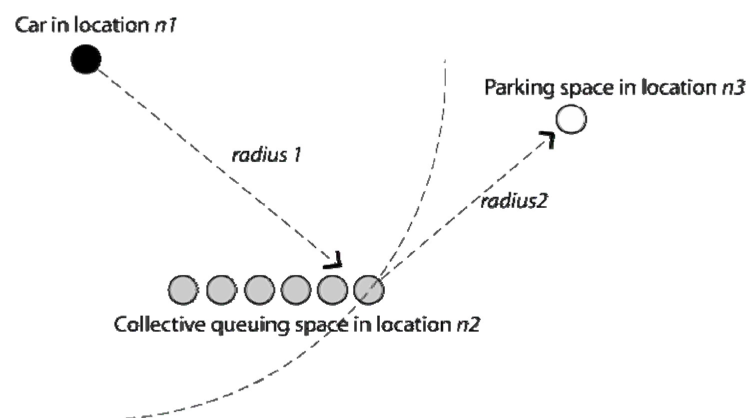


Figure 28 Functioning of collective queuing for street-parking.

Furthermore, if the queues are long, then considerable lengths of street space would be consumed in a single location, potentially rendering an entire street into an unpleasant row of buzzing cars and leaving no parking spaces for inhabitants or business owners.

Decentralized stalling offers an alternative. The idea could function more similarly to packet routing of data over the Internet, where individual pieces can take different paths of the network, avoiding clustering and congestion in centralized bottle-necks. A potentially exploitable resource for distributed stalling is fire hydrant spaces, currently banned from use. Fire hydrant spaces are otherwise unusable as parking spaces and their exploitation as stalling spaces would not affect the number of current parking spaces. Cruisers, who have been notified that there are absolutely no available parking spaces at a given time, could either drive to another neighborhood or use the fire hydrant spaces for temporary stalling. Drivers should not be allowed to leave the seat while stalling at a hydrant, facing big penalties for violations. In addition to fire hydrants, additional stalling spaces could be allocated dynamically, depending on the current need through controllable signage on the ground. Using the communication infrastructure set up by the guidance system, the current over demand in a given area could be approximated momentarily and stalling spaces allocated accordingly.

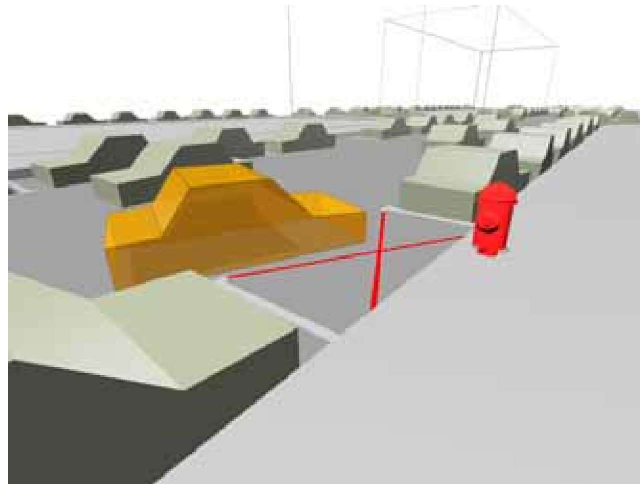


Figure 29 Allocation of temporary stalling spaces at fire hydrants and other designated spaces according to demand.

Access radiuses to local fire hydrants and other designated individual stalling spaces would be much smaller than to collective stalling areas. Hence, in total, less distance would need to be traveled between the location where a driver starts searching for parking, the location of the stalling space and the eventual location of a parking space.

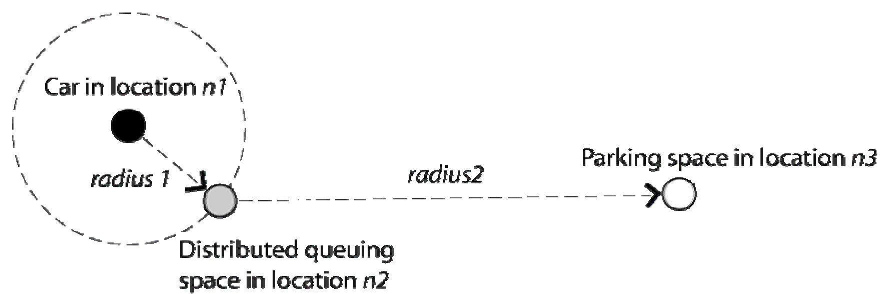


Figure 30 Functioning of distributed queuing for street-parking.

Whereas distributed queuing could also disperse the quantity of queued cars into a larger geographic area without creating overly intense stalling lines on specific streets, it would therefore also be harder for individual drivers to locate these dispersed spaces in a neighborhood. Using the guidance software on a personal communication device could again aid in such a search. It is quite likely that similarly to the collective stalling spaces, local inhabitants and signage could also help guide drivers to appropriate sites.

More importantly, distributed queuing at individual stalling spaces could flexibly accommodate different time and financial constraints of drivers in the queue. In case of urgency or simply willingness to pay higher fees for priority parking, drivers in scattered stalling spaces would not be subject to uncomfortable

situations, where distinguishing themselves in front of all the drivers in the queue might irritate others.

Though my arguments seem to suggest that distributed queuing might be quite beneficial for reducing overall search times, more research on these assumptions is certainly needed. Specifically, critical conditions need to be outlined to indicate when stalling is better than dynamical probabilistic search. For instance, if the occupation turnover of parking spots in an area is relatively rapid, then driving to a stalling space, and subsequently from the stalling space to the parking spot, might increase the total search time. Under conditions of rapid turnover, a probabilistic search, or even a random search for that matter, could yield higher efficiencies for reducing overall search times. This, and many other similar uncertainties outlined in this thesis, require additional experimental research. But rather than only indicating the poor value of the current findings, I believe that such controversy adequately demonstrates the multilayered complexity of the use of real-time information in a dynamic allocation of urban resources.

Chapter Four

Conclusions

The question that I have been exploring is one of efficiency from a highly rational point of view. This efficiency is the overall reduction of searching time for street parking, that is, efficiency from the point of view of the group. Faced with challenges of sustainable development, group efficiency is becoming an important task for planners in the 21st century. Somewhat counter intuitively, a system that might be beneficial for an individual agent can undermine the efficiency of the group and *vice versa*. For instance, the simulation models showed that a reservation policy for street-parking, which clearly benefits an individual, can in fact cause great inefficiencies at a group level. Choosing and retaining the collective viewing angle has been an important part of this thesis, since I believe it is here that the emerging technology for urban systems needs most attention.

Clearly, cruising for curb parking produces severe pollution and congestion today. Donald Shoup has adequately illustrated this story [Shoup, 2005]. However, excessive cruising is not only caused by the low price of street-parking, but also by an inefficient match between the supply and the demand for parking spaces. Street-parking, like many other urban resources, is often times in over demand and it is likely to remain so in popular areas. Urban resource allocation should not be mistaken for a classical economic equilibrium case where supply is supposed to balance demand. Such an equilibrium condition of urban resources rarely occurs. As the metropolitan population in the world is growing faster than ever, it would be tremendously dangerous to produce enough urban resources that satisfy classical demand/supply equilibriums for everyone.

Hence, instead of producing more supplies, the important challenge is to deal with the over demand, with optimal allocation and queuing when there is little to allocate but demanders are many.

I have tried to argue that that the efficiency of the current street-parking system could be improved in at least two aspects:

1. By providing broader information to drivers, than is currently available to them in their immediate visual surroundings.

2. By using combinatorial and probabilistic calculations on a computer to enhance decision making with the available information.

I have proposed that a computational guidance system can be used to balance these shortcomings. Personal mobile communication devices could exchange and process enough information to find satisfying solutions to combinatorial problems of commuting more efficiently than intuitive searching today. However, currently the sensing capacity of humans is far greater than of computers. Computers can do statistical calculations faster than human beings, but they have very limited capacity to sense information from the environment. Nevertheless, even with the limited but strategic group information, the simulation models that were presented suggest that overall parking search times can be diminished by at least a factor of two, depending on environmental conditions, with the aid of a digital guidance system. Rather than betting on one or the other, a seamless collaboration between a digital avatar and cognitive intuition can result in a more optimal search process.

The comparison of simulations that used narrow individual information, and those that took into account group behavior and broader statistical information, showed that highest efficiencies can be achieved through collaborative behavior,

combining a wide range of information. The simulation results also suggested that an increase in information and computational capacity does not automatically lead to more efficient search results- a well planned coordination strategy is indispensable for good results. As expected, clever strategies worked better under critical circumstances. In simpler situations almost any strategy could give a satisfactory result, while in challenging environmental conditions simpler strategies failed and intelligent strategies prevailed.

A satisfying strategy is highly dependent on the external variables of the environment that it has to operate in. The causal effects of the few important environmental variables that I have outlined are 1) street grid size, 2) demand/supply ratio, 3) parking spot distribution and 4) alternative methods of guidance information. In addition to the external variables, search decisions are also affected by internal stimuli of the driver that allow him to freely switch between different goals and strategies with no apparent external environmental changes. The current intuitive searching behavior can provide many clues for a system designer to achieve a more efficient and humanly pleasant guidance system. Specifically beneficial for the efficiency of the search would be factors such as goal knowledge and goal switching capacity.

I have also tried to emphasize, that an introduction of digital optimization systems to the physical realm of urban resource allocation brings about a set of important real-time management issues that have been far less crucial in both the digital realm and the urban realm so far. In digital information networks information travels at the speed of light and overlapping demand queues are solved in a fraction of a second. Information systems generally do not need to consider the physical distances between the remote parts of the network to manage queues. In traditional urban systems, on the other hand, interaction between people and places is slow and a potential queuing for a unique space can often be avoided due to different travel times of people in the physical world.

However, the use of digital communication for urban space allocation has to account for the physical efforts involved in relocating people and resources, as well as good queuing management. Unless carefully planned, an electronic system could cause severely wasteful allocation. This makes the digital allocation system susceptible to laws of physics.

The issues I have been outlining in this thesis are not solely characteristic to parking management, but to virtually any digital space allocation system. The study of an efficient parking strategy has merely provided a slice of many more general issues that the introduction of real-time information systems creates for urban planners. The potentially affected domains are wide and cross-disciplinary, ranging from real-estate values, public and private transportation management, temporary space allocation, the distribution of goods and services, etc.[24] It is yet to be seen how important the role of real-time information in urban economies will turn out to be, but there are reasons to believe that city planners should pay close attention and participate in this development. Currently, most real-time information technology is being pioneered by the private sector. Given that the clientele of the private sector is essentially composed of profit seeking individuals, it is natural that the technology is focused on the individual and that personal interests dominate. This thesis has tried to challenge this direction. It has tried to demonstrate that certain co-operative behavior amongst the agents in the system can lead to a better group outcome as well as higher individual gains than purely individual competition. Real-time awareness of other competitors combined with a clever decision making strategy increases the general competitiveness of the individual in the environment in which it operates. On the other hand, if well planned, the resulting competitive collaboration does not jeopardize group efficiency but rather improves it. The collaborative behavior between agents, that I have been exploring, is not achieved by centralized

[24] A supplementary list of similar case studies to street-parking can be found at web.mit.edu/asevtsuk/www/thesis

planning, quite the opposite. It is achieved by taking advantage of innovative technology, competition and Game Theory in order to provide incentives for collaborative behavior amongst profit seeking individuals. I believe that it is precisely such group performance, which masks the immediately visible gains for the private sector (but does indeed contain them), that mostly needs the attention of planners. Undoubtedly, a further understanding and debate around the issues of group performance of real-time allocation systems can eventually also shift the focus of private technology companies towards a more universal understanding of their impacts on the cities of tomorrow.

References

Anjali Mahendra, 2000, *Congestion Prices in Cities of the Developing World*: Exploring Prospects in Mexico City, MIT Urb. Studies M.C.P. Thesis.

Axelrod Robert and Cohen D. Michael (2000), *Harnessing Complexity, Organizational Implications of a Scientific Frontier*, Basic Books.

Axhausen, K.W. and J.W. Polak (1991) *Choice of parking: Stated preference experiments*, Transportation, 18 (1) 59-81.

Axhausen, K.W. and J.W. Polak (1996) *A disaggregate model of the effects of parking guidance systems*, D. Hensher, J. King and T. Oum 'World Transport Research', 1, 139-149, Elsevier, Amsterdam

Axhausen, K.W., J.W. Polak, M. Boltze and J. Puzicha (1994) *Effectiveness of the parking guidance system in Frankfurt/Main*, Traffic Engineering and Control, 35 (5) 304-309

Batty, Michael (2005) Cities and Complexity, MIT Press.

Benjamin, Walter, (1999) *The Arcades Project*, translated by Howard Eiland and Kevin McLaughlin (Cambridge, Massachusetts and London, England: The Belknap Press of Harvard University Press).

Borenstein, Severin (2005) *The Long-Run Efficiency of Real-Time Electricity Pricing*. Center for the Study of Energy Markets, University of California.

Manuel Castells, Mireia Fernandez-Ardevol, Jack Linchuan Qiu, and Araba Sey, *Electronic Communication and Socio-Political Mobilisation: A New Form of Civil Society*, in Global Civil Society 2005/2006, London: Sage, 2006, pages 266-287.

Carrera Fabo (2003) *City Knowledge: An Emergent Information Infrastructure for Sustainable Urban Maintenance, Management and Planning*. PhD Thesis, MIT.

Clinch Peter, Kelly Andrew (2003), *Testing the sensitivity of parking behavior and modal choice to price of on-street parking*, University College Dublin, environmental studies research series ESRS 03/03.

Dawson Michael, (2005) *Minds and Machines: Connectionism and Psychological Modeling*, Blackwell Publishers.

Flaxman, M. *Using Virtual Cities to Plan Real Cities: Alternative Futures for Hangzhou, China*. SIGGRAPH 2002 Conference Abstracts and Applications. Computer Graphics Annual Conference Series, Sketches and Applications section.

Hilton, Ian C., *Holding place technique and the removal of car-park queues*, Traffic Engineering + Control, April 1989.

Jacobs, Jane (1961) *The Death and Life of Great American Cities* , Vintage Books, New York.

Kahneman Daniel (Editor), Slovic Paul (Editor), Tversky Amos (Editor), 1982 *Judgment under Uncertainty : Heuristics and Biases.* Cambridge University Press.

Laurier, E. (2003) *Searching for a parking space*. Originally presented at Espace, Inter / Action & Cognition ARCo in Paris 2003.

Lynch, Kevin (1984) *Good City Form*, MIT Press.

Mackay, W.E. (2000) *Is Paper Safer? The Role of Paper Flight Strips in Air Traffic Control*. ACM/Transactions on Computer-Human Interaction. Vol. 6 (4), pp. 311-340.

Mackay, W.E. (2000) *Responding to cognitive overload: Co-adaptation between users and technology*. Intellectica. Vol. 30 (1), pp. 177-193.

Mackay, W.E. (2001) *Does Tutoring Really Have to be Intelligent?*. To appear in *ACM/CHI2001 Extended Abstracts*, Seattle, WA.

Maya Abou Zeid, 2001, *Models and Algorithms for the Optimization of the Traffic Flows and Emissions Using Dynamic Routing and Pricing*, MIT M.S. in Transportation Thesis.

Metcalf Sara and Mark Paich (2005) *Spatial Dynamics of Social Network Evolution*, Department of Geography University of Illinois at Urbana-Champaign, Presented at the 23rd International Conference of the System Dynamics Society
July 19, 2005

Minsky Marvin, 1988, *The Society of Mind*, Touchstone Press.

Minsky Marvin, 2006, *The Emotion Machine*, MIT course Society of Mind course materials.

Morris Joan, 2001, *A Simulation-based Approach to Dynamic Pricing*, MIT Media A & S SM Thesis.

William J. Mitchell, *ME++ The Cyborg Self and the Networked City*. Cambridge, MA, MIT Press, 2003

Nishimura Masahiro, 1996, *Congestion Pricing for Air Pollution Reduction - Environmental Evaluation of Pollution-adjusted-rate Pricing and Comparisonn with Other Strategies*, MIT Urban Studies M.C.P. Thesis.

Ormerod Paul, 2005. *Why Most Things Fail*, Pantheon Books.

Polak, J.W., I.C. Hilton, K.W. Axhausen and W. Young (1991) *Parking guidance and information systems: A European review*, The Parking Professional, (2) 16-34

Rauterberg, M (1994) *About the Relationship between Incongruity, Complexity and Information: Design Implications for Man-Machine Systems*, Wolf Rauch / Frenz Strohmeier / Harald Hiller / Christian Schlög (Hg.), Schriften zur Informationswissenschaft, Band 16, 1994.

Samad T.; Weyrauch J., *Automation, Control and Complexity. An Integrated Approach*, 2000 Wiley & Sons Ltd.

Shoup Donald (2005) *The High Cost of Free Parking*, APA Planners Press.

Shoup Donald and Menville Michael (2005), *Parking, People and Ethics*, Journal of Urban Planning and Development, Volume 131 nr. 4, Dec. 2005, ASCE

Simon Herbert, 1996, *The Sciences of the Artificial* (3rd Edition), MIT Press.

Thompson, R.G. and A.J. Richardson (1998). *A parking search model*, Transportation Research, Part A, Vol. 32, 159-70, Pergamon

Thompson, R.G., K. Takada and S. Kobayakawa (1999). *Understanding the demand for access information*, Transportation Research, Part C, Vol. 6, 231-45, Pergamon.

Thompson, R.G., K. Takada and S. Kobayakawa (2000). *Optimision of parking guidance and information systems display configurations*, Transportation Research, Part C, Vol. 9, 69-85, Pergamon, Elsevier.

Townsend, Anthony, 2003. *Wired / unwired : The urban geography of digital networks* PhD Thesis, MIT.

Watson, Rod. 1999. *Driving in Forests and Mountains : A Pure and Applied Ethnography.* Ethnographic Studies 3:50-60.

List of Illustrations

Chapter 1

Figure 1 Cruising in the 20th century. Source: *The High Cost of Free Parking* (2005); D. Shoup

Figure 2 Parking in central business districts. Source: *The High Cost of Free Parking* (2005); D. Shoup

Figure 3 Map of off-street parking lots around MIT. Shaded areas indicate multi-story structures.

Figure 4 Aerial view of parking coverage north of Vassar Street at MIT. Google Maps.

Figure 5 Financial benefits of cruising. Source: *The High Cost of Free Parking*, Shoup 2005.

Figure 6 Low cost street-parking in popular areas is often filled to almost full capacity, making it difficult for drivers to find the remaining few hidden parking spaces. Beacon Hill, Boston. Photo: Andres Sevtsuk.

Figure 7 Illustration of the guidance system for street-parking.. Andres Sevtsuk.

Chapter 2

Figure 8 The three categories of variables in the street-parking system.

Figure 9 Layout of the street-network. In the model, cars can "wrap" out of the picture on one side and re-enter from the opposite side, creating an infinite torus shaped topological continuum.

Figure 10 Graphical user interface of the Star Logo simulation model.

Chapter 3

Figure 11 Comparison of the average number of steps to find a parking space in simulation models.

Figure 12 Comparison search efficiencies on a 3x3 and a 5x5 grid with 6 equally distributed parking spaces.

Figure 13 Graphic calculation of the Intel_1 search strategy.

Figure 14 The average number of searching steps of 24 cars on a 3x3 street grid

Figure 15 Average number of searching steps of 24 cars on a 5x5 street grid

Figure 16 Comparison of the number of alternative travel paths on different grid sizes. The calculations show the approximate number of different travel paths, without repeating any street segment twice and assuming that the search continues till it runs into a dead end.

Figure 17 A comparison of strategy performances with 24 cars on a 3x3 grid with 6 geographically dispersed parking spaces.

Figure 18 Graphic calculation of the Intel_3 strategy. In the case of a first-come-first serve reservation policy; this example has 6 different outcomes, of which only one is optimal.

Figure 19 Comparison of search efficiencies with uniformly distributed or locally concentrated 6 parking spaces on a 3 x 3 grid.

Figure 20 Six parking spaces on a 3 x 3 grid distributed uniformly and concentrated.

Figure 21 Graphic calculation of the Intel_5 search strategy.

Figure 22 Comparison of search strategies on a 5x5 grid with 24 uniformly distrinuted spots.

Figure 23 Graphic calculation of the benefits of collaborative behavior.

Figure 24 Graphic calculation of Intel_5 and Intel_9 strategies.

Figure 25 Absorption of physical queues in slow-interaction systems.

Figure 26 Different Queuing Strategies in Street Parking

Figure 27 Centralized stalling area, similar in design to a taxi stall on a street corner.

Figure 28 Functioning of collective queuing for street-parking.

Figure 29 Allocation of temporary stalling spaces at fire hydrants and other designated spaces according to demand.

Figure 30 Functioning of distributed queuing for street-parking.

Appendix

Source codes for Star-Logo simulation models of street-parking.

Intel_1

Observer Procedures:

```
globals [aa East-Free West-Free Total-Free List-Ready?]
patches-own [EastSide? xcoord ycoord number]
breeds [living working visiting1 visiting2]


to track-vacant
	loop[
	set Total-Free (count-patches-with [pc = white and count-
turtles-here = 0 ] )
	output Total-Free]
end


to track-vacant-all ; here we generate a list called "aa" which
tracks the vacant spots on the east sides of roads
	; we need to keep them separate so that when a car estimates
its driving distance, it knows if it needs to go around the block
or not
	setList-Ready? false
	let [:a (count-patches-with [pc = white and count-turtles-
here = 0] )]
	setTotal-Free (:a)
	ask-patches [if pc = white and count-turtles-here = 0 [
	repeat Total-Free [
	if (number? xcor) [let [:xpos xcor]]
	if (number? ycor) [let [:ypos ycor]]
	let [:bb (list xcor ycor)]

	set aa make-list 0 (East-Free)
	set aa insert 1 aa :bb
	]]]
	setList-Ready? true
	;show aa ; for debugging only to see if "aa" works
end


to setup
	ct
	crt number-of-cars
	ask-turtles [setCounter 0 setshape cross setc red setspeed 1
setSpeedLimit 1
	if (who <= (number-of-cars / 5)) [setbreed living]
	if (who > (number-of-cars / 5) and who <= (number-of-cars /
2)) [setbreed working]
	if (who > (number-of-cars / 2) and who <= (number-of-cars /
1.25)) [setbreed visiting1]
```

```
	if (who > (number-of-cars / 1.25)) [setbreed visiting2]
		if breed = living [setParkTime 48]
		if breed = working [setParkTime 32]
		if breed = visiting1 [setParkTime 2]
		if breed = visiting2 [setParkTime 8]

	loop [ifelse ((pc-at 0 0) = black and count-turtles-here = 1
) [stop] [
	ifelse (random 2) = 0 [seth 0] [ifelse (random 2) = 0 [seth
90][ifelse (random 2) = 0 [seth 180] [seth 270]]]
	fd (int random 25)
	ifelse (random 2) = 0 [seth 0] [ifelse (random 2) = 0 [seth
90][ifelse (random 2) = 0 [seth 180] [seth 270]]]
	fd (int random 25) ]]]
	ask-patches-with [pc = white] [
	let [:x xcor] set xcoord :x let [:y ycor] set ycoord :y set
number (:x * :y); assign each patch an x and y coordinate valeu
and a unique number.
	]
	starttrackingall
	startcountaverage
end


to clear-cars
	ct
end


to count-steps-to-find-parking
output (average-of-turtles-with [color = yellow] [steps-to-find-
parking])
end


to stop-it
	stoptrackingall
	stopcountaverage
	stopDrive&Park
end


to count-average ; this is for statistical analysis: paste these
number into excel and calculate the mean, median and standard
deviation
	show (round average-of-turtles-with [color = yellow] [steps-
to-find-parking]) wait 1
end
```

Turtle Procedures:

```
turtles-own [speed SpeedLimit ParkTime Parked? steps-to-find-
parking Direction CoordX CoordY Counter MyDist]


to check-patches-after-park
  if (pc-ahead = 7) or (pc-ahead = 9) [
	rt 90 check-patches-after-park
	]
end
```

```
to check-side
       ask-patch-at CoordX CoordY [output EastSide?]
end


to choose-nearest-spot
; here extract the itam in the aa list, then extract ech item's x
and y and check the patche's distance from the turtle (for all
patches)
; create a new variable named :dist which indicates the distance
to the nearest free spot and a variable named "item-find"
indicating the number of the element in the aa list
; the aa list signifies place on the east side of roads, the bb
contains spots from the west side of the road.

       wait-until [List-Ready? = true]
       let [:nullcheck (length aa)]
       ifelse (:nullcheck > 0) [
       setMyDist 2000 ; initialize a distance that is bigger than
any on screen dist, so that a new dist will always be smaller
       let [:CoordX 0]
       let [:CoordY 0]
       let [:Dir 90]
       let [:k 1] ; loop through every elemnt in the "number" list
starts with 1
       let [:aacopy (copy-list aa)] ; make a copy of the "aa", so
if the real aa changes in length, their's remain the same until
end of counting
       let [:aasize (length :aacopy)]
       repeat :aasize [ ;check only for free spots, don't waste RAM
       let [:item-number (item :k :aacopy)] ; extract the first
(eventually each) element from the aa list
       let [:item-numberX (item 1 :item-number)] ; extract the X
value of aa item
       let [:item-numberY (item 2 :item-number)] ; extract the Y
value of the aa item
       let [:a (round(:item-numberX - xcor))]
       let [:b (round(:item-numberY - ycor))]
       ifelse (heading = 0 and :item-numberY < ycor) or (heading =
90 and :item-numberX < xcor) or (heading = 180 and :item-numberY >
ycor) or (heading = 270 and :item-numberX > xcor)[
       let [:distnew ((abs :a) + (abs :b) + 45)]] ; if the
destination spot is in the opposite direction, the add 1/2
(average) block loop (22 + 8) to the dist
       [let [:distnew ((abs :a) + (abs :b))]]

       if (:distnew < MyDist ) [let[:kchosen :k] setMyDist
:distnew]
       ; CoordX and CoordY are turtle-own variables, which remember
which parking spot the turtles zoomed onto, and will keep that
until a turtles goes to that spot.
       if :k <= :aasize [set :k (:k + 1)]
       ]

       set :item-number (item :kchosen :aacopy) ; extract the
memorized smallest distance element from the :aacopy list

       set :item-numberX (item 1 :item-number) ; extract the X
value
       set :item-numberY (item 2 :item-number) ; extract the Y
value
       set :a (round (:item-numberX - xcor))
```

```
       set :b (round (:item-numberY - ycor))

       ifelse (heading = 0 and :item-numberY < ycor) or (heading =
90 and :item-numberX < xcor) or (heading = 180 and :item-numberY >
ycor) or (heading = 270 and :item-numberX > xcor)[
       setMyDist ( (abs :a) + (abs :b) + 45)] ; if the destination
spot is in the opposite direction, the add 1/2 (average) block
loop (22 + 8) to the dist
       [setMyDist ((abs :a) + (abs :b))]
       set CoordX (:item-numberX) set CoordY (:item-numberY)
       setDirection (towards-nowrap CoordX CoordY)

       ] [setCoordX 0 setCoordY 0 setMyDist 1000]
end


to park ; has to be done so that agent will look for parking until
found

       if (Parked? = false)[
       setc yellow
       ;ask-patch-at CoordX CoordY [setpc green]
       ;ask-patch-at CoordX CoordY [setpc white]
       loop[
       choose-nearest-spot
              ; here set the car to break or accelerate according
to other cars
              ifelse (count-turtles-towards heading 1) > 0     ;if
there is a turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-towards heading 1
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at heading 1) > 0
              [setspeed speed-of one-of-turtles-towards heading 2
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]

       ; here is how the actual parking move happens
       ; first, parking at your own side of the road and then
parking at the opposite side of the road, parking on horisontal
streets is also allowed.
       ; augment the counter, which counts the program iterations
during which a car parks (instead of real time)
       ifelse ((pc-at 0 0) = 0 and (pc-at 1 0) = 9 and (count-
turtles-at 1 0) = 0) or ((pc-at 0 0) = 0 and (pc-at (-1) 0) = 9
and (count-turtles-at (-1) 0) = 0) or
       ((pc-at 0 0) = 0 and (pc-at -2 0) = 9 and (count-turtles-at
-2 0) = 0) or ((pc-at 0 0) = 0 and (pc-at 2 0) = 9 and (count-
turtles-at 2 0) = 0) or
       ((pc-at 0 0) = 0 and (pc-at 0 1) = 9 and (count-turtles-at 0
1) = 0) or ((pc-at 0 0) = 0 and (pc-at 0 (-1)) = 9 and (count-
turtles-at 0 (-1)) = 0) or
       ((pc-at 0 0) = 0 and (pc-at 0 -2) = 9 and (count-turtles-at
0 -2) = 0) or ((pc-at 0 0) = 0 and (pc-at 0 2) = 9 and (count-
turtles-at 0 2) = 0) or
       ((pc-at 0 0) = 9) [

       ; here we augment the parking counter and check if the
counter is full, in which case a car leaves
```

```
       if ((pc-at 0 0) = 9 and (pc-at (-1) 0) = 0) [ setCounter
Counter + 1 if Counter > ParkTime [setCounter 0 seth 270 fd 1 seth
0 setParked? true stop]]
       if ((pc-at 0 0) = 9 and (pc-at 1 0) = 0) [ setCounter
Counter + 1 if Counter > ParkTime [setCounter 0 seth 90 fd 1 seth
180 setParked? true stop]]
       if ((pc-at 0 0) = 9 and (pc-at 0 (-1)) = 0) [setCounter
Counter + 1 if Counter > ParkTime [setCounter 0 seth 180 fd 1 seth
270 setParked? true stop]]
       if ((pc-at 0 0) = 9 and (pc-at 0 1) = 0) [setCounter Counter
+ 1 if Counter > ParkTime [setCounter 0 seth 0 fd 1 seth 90
setParked? true stop]]

       if ((pc-at 0 0) = 0 and (pc-at 1 0) = 9 and (count-turtles-
at 1 0) = 0) [seth 90 fd 1 set steps-to-find-parking 0 setMyDist 0
setc red ]
       if ((pc-at 0 0) = 0 and (pc-at (-1) 0) = 9 and (count-
turtles-at (-1) 0) = 0) [seth 270 fd 1 set steps-to-find-parking 0
setMyDist 0 setc red ]
       if ((pc-at 0 0) = 0 and (pc-at -2 0) = 9 and (count-turtles-
at -2 0) = 0)[seth 270 fd 2 set steps-to-find-parking 0 setMyDist
0 setc red ]
       if ((pc-at 0 0) = 0 and (pc-at 2 0) = 9 and (count-turtles-
at 2 0) = 0) [seth 90 fd 2 set steps-to-find-parking 0 setMyDist 0
setc red ]
       if ((pc-at 0 0) = 0 and (pc-at 0 1) = 9 and (count-turtles-
at 0 1) = 0) [seth 0 fd 1 set steps-to-find-parking 0 setMyDist 0
setc red ]
       if ((pc-at 0 0) = 0 and (pc-at 0 (-1)) = 9 and (count-
turtles-at 0 (-1)) = 0) [seth 180 fd 1 set steps-to-find-parking 0
setMyDist 0 setc red ]
       if ((pc-at 0 0) = 0 and (pc-at 0 -2) = 9 and (count-turtles-
at 0 -2) = 0)[seth 180 fd 2 set steps-to-find-parking 0 setMyDist
0 setc red ]
       if ((pc-at 0 0) = 0 and (pc-at 0 2) = 9 and (count-turtles-
at 0 2) = 0) [seth 0 fd 2 set steps-to-find-parking 0 setMyDist 0
setc red ]]
       [if (pc-at 0 0) not= white [ ifelse (pc-ahead = 2 and (pc-at
0 0) = 2) [ifelse (random 2) = 0 [leap 3 set steps-to-find-parking
(steps-to-find-parking + 3)] [ifelse (random 2) = 0 [rt 90 fd 2
set steps-to-find-parking (steps-to-find-parking + 2)][fd 1 lt 90
fd 2 set steps-to-find-parking (steps-to-find-parking + 3)]]]
       [check-patches-after-park fd speed set steps-to-find-parking
(steps-to-find-parking + speed)] if (CoordX = 0 and CoordY =
0)[choose-nearest-spot]]]

; here are the rules to guide a car towards a chosen spot,
assuming it can also park on the opposite side of the road.
ifelse (CoordX not= 0 and CoordY not= 0 and MyDist not= 1000) [
;only if you are closest in the competition for a given spot,
drive there, else roam randomly and try again next step
;in reality you should try the second best option here and then
third best and so on!

; for heading = 0, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor < CoordX and ycor < CoordY))[
ifelse (CoordX < (xcor + 3))[fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor < CoordX and ycor > CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
```

```
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor > CoordX and ycor > CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor > CoordX and ycor < CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)][fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]

; for heading = 90, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
< CoordX and ycor < CoordY))[
ifelse (CoordX < (xcor + 3)) [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)][fd 2 set steps-to-find-
parking (steps-to-find-parking + 2)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
< CoordX and ycor > CoordY))[
ifelse (CoordX < (xcor + 3)) [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)] [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
> CoordX and ycor > CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
> CoordX and ycor < CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]

; for heading = 180, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor < CoordX and ycor < CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor < CoordX and ycor > CoordY))[
ifelse (CoordX < (xcor + 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor > CoordX and ycor > CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor > CoordX and ycor < CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]

; for heading = 270, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor < CoordX and ycor < CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor < CoordX and ycor > CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor > CoordX and ycor > CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)] [fd 2 set steps-to-find-
parking (steps-to-find-parking + 2)]]
```

```
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor > CoordX and ycor < CoordY))[
ifelse (CoordX > (xcor - 3)) [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)] [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)]]
][if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random 2) = 0
[leap 3 set steps-to-find-parking (steps-to-find-parking + 3)]
[ifelse (random 2) = 0 [rt 90 fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)][fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]]]
]]
end


to drive ; each egnt will park and then drive for a certain time

       if (pc-at 1 0) = 7 or (pc-at 1 0) = 9 [
       seth 0
       setParked? false

       park

       repeat (parking-interval * 10) [ ; these are turtles driving
up
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
              ifelse (count-turtles-at 0 1) > 0     ;if there is a
turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-at 0 1
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at 0 1) > 0
              [setspeed speed-of one-of-turtles-at 0 2
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]

       if (pc-at (-1) 0) = 7 or (pc-at (-1) 0) = 9 [
       seth 180
       setParked? false
       park
       repeat (parking-interval * 10) [ ; these are turtles driving
up
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
              ifelse (count-turtles-at 0 (-1)) > 0     ;if there is
a turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-at 0 (-1)
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at 0 (-1)) > 0
              [setspeed speed-of one-of-turtles-at 0 (-2)
              decelerate]
              [accelerate]]   ;else accelerate
```

```
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]

       if (pc-at 0 1) = 7 or (pc-at 0 1) = 9 [
       seth 270
       setParked? false
       park
       repeat (parking-interval * 10) [ ; these are turtles driving
up
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
              ifelse (count-turtles-at (-1) 0) > 0     ;if there is
a turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-at (-1) 0
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at (-1) 0) > 0
              [setspeed speed-of one-of-turtles-at (-2) 0
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]

       if (pc-at 0 (-1)) = 7 or (pc-at 0 (-1)) = 9 [
       seth 90
       setParked? false
       park
       repeat (parking-interval * 10) [ ; these are turtles driving
up
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
              ifelse (count-turtles-at 1 0) > 0     ;if there is a
turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-at 1 0
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at 2 0) > 0
              [setspeed speed-of one-of-turtles-at 2 0
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
```

```
             if (pc-at 0 0 )= 2 [fd 1]
end


to accelerate
  setspeed (speed + (speedup / 1000))
end


to decelerate
  setspeed speed - (slowdown / 1000)
end
```

Intel_3

Observer Procedures:

```
globals [aa Total-Free Reservation-Available?]
patches-own [xcoord ycoord number]
breeds [living working visiting1 visiting2 reserving]

to track-vacant-all ; here we generate a list called "aa" which
tracks the vacant spots on the east sides of roads
       wait-until [Reservation-Available? = true]
       if Reservation-Available? = true [
       set Reservation-Available? false
       set aa make-list 0 0
       let [:a (count-patches-with [pc = white and count-turtles-
here = 0])]
       ask-patches [if pc = white and count-turtles-here = 0 [
       let [:xpos xcor]
       let [:ypos ycor]
       let [:bb (list xcor ycor)]
       set aa insert 1 aa :bb
       ]]
       set Reservation-Available? true]
end


to setup
       set-random-seed 100
       set Reservation-Available? true
       ct
       crt number-of-cars
       ask-turtles [setCounter 0 setshape cross setc red setspeed 1
setSpeedLimit 1
       if (who <= (number-of-cars / 5)) [setbreed living]
       if (who > (number-of-cars / 5) and who <= (number-of-cars /
2)) [setbreed working]
       if (who > (number-of-cars / 2) and who <= (number-of-cars /
1.25)) [setbreed visiting1]
       if (who > (number-of-cars / 1.25)) [setbreed visiting2]
             if breed = living [setParkTime 48]
             if breed = working [setParkTime 32]
             if breed = visiting1 [setParkTime 2]
             if breed = visiting2 [setParkTime 8]
```

```
       loop [ifelse ((pc-at 0 0) = black and count-turtles-here = 1
) [stop] [
       ifelse (random 2) = 0 [seth 0] [ifelse (random 2) = 0 [seth
90][ifelse (random 2) = 0 [seth 180] [seth 270]]]
       fd (int random 25)
       ifelse (random 2) = 0 [seth 0] [ifelse (random 2) = 0 [seth
90][ifelse (random 2) = 0 [seth 180] [seth 270]]]
       fd (int random 25) ]]]
       starttrackingall
       startcountaverage
end


to clear-cars
       ct
end


to count-steps-to-find-parking
output (average-of-turtles-with [color = yellow] [steps-to-find-
parking])
end

to stop-it
       stoptrackingall
       stopDrive&Park
       stopcountaverage
end


to count-average ; this is for statistical analysis: paste these
number into excel and calculate the mean, median and standard
deviation
       show (round average-of-turtles-with [color = yellow] [steps-
to-find-parking]) wait 1
end
```

Turtle Procedures:

```
turtles-own [speed SpeedLimit ParkTime Parked? steps-to-find-
parking Direction CoordX CoordY Counter Dist]

to check-patches-after-park
  if (pc-ahead = 7) or (pc-ahead = 9) [
       rt 90 check-patches-after-park
      ]
end


to choose-nearest-spot

; here extract each item in the "aa" list, then extract each
item's x and y and check the patche's distance from the turtle
; use the turtles-own variable Distance which indicates the
distance to the nearest free spot and a variable named ":item-
find" indicating the number of the element in the aa list

wait-until [Reservation-Available? = true]
if Reservation-Available? = true [
```

```
set Reservation-Available? false

       let [:nullcheck (length aa)]
       if (:nullcheck > 0) [
       setDist 2000 ; initialize a distance that is bigger than any
on screen dist, so that a new dist will always be smaller
       let [:k 1] ; k is the counter to loop through every elemnt
in the "number" list starts with 1
       let [:aacopy (copy-list aa)] ; make a copy of the "aa", so
if the real aa changes in length, their's remain the same until
end of counting
       repeat (length :aacopy) [ ;check only for free spots, don't
waste RAM
       let [:item-number (item :k :aacopy)] ; extract the first
(eventually each) element from the aa list
       let [:item-numberX (item 1 :item-number)] ; extract the X
value of aa item
       let [:item-numberY (item 2 :item-number)] ; extract the Y
value of the aa item
       let [:xdist (:item-numberX - (round xcor))]
       let [:ydist (:item-numberY - (round ycor))]
       ifelse (heading = 0 and :item-numberY < ycor) or (heading =
90 and :item-numberX < xcor) or (heading = 180 and :item-numberY >
ycor) or (heading = 270 and :item-numberX > xcor)[
       let [:distnew ((abs :xdist) + (abs :ydist) + 45)]] ; if the
destination spot is in the opposite direction, the add 1/2
(average) block loop (22 + 8) to the dist
       [let [:distnew ((abs :xdist) + (abs :ydist))]]
       ; you're always comparing distnew to the initial dist, of
course you'll just end up choosing the last one...
       if (:distnew < Dist ) [let [:kchosen :k] setDist :distnew]
       if :k <= (length :aacopy) [ set :k (:k + 1)]]

       set :item-number (item :kchosen :aacopy) ; extract the
memorized smallest distance element from the :aacopy list
       set :item-numberX (item 1 :item-number) ; extract the X
value
       set :item-numberY (item 2 :item-number) ; extract the Y
value
       set :xdist (:item-numberX - (round xcor))
       set :ydist (:item-numberY - (round ycor))

       ifelse (heading = 0 and :item-numberY < ycor) or (heading =
90 and :item-numberX < xcor) or (heading = 180 and :item-numberY >
ycor) or (heading = 270 and :item-numberX > xcor)[
       set Dist ((abs :xdist) + (abs :ydist) + 45)] ; if the
destination spot is in the opposite direction, the add 1/2
(average) block loop (22 + 8) to the dist
       [set Dist ((abs :xdist) + (abs :ydist))]
       set CoordX (:item-numberX) set CoordY (:item-numberY)
       setDirection (towards-nowrap CoordX CoordY)
       ask-patch-at :xdist :ydist [sprout [setbreed reserving setc
green setshape cross]]
       let [:bb (list CoordX CoordY)]
       set aa (remove-element :bb aa)]
       set Reservation-Available? true
       ]
end


to park ; has to be done so that agent will look for parking until
found

       if breed not= reserving[
```

```
       if (Parked? = false)[
       setc yellow
       ; the idea here is that if an agent has found a destination
from the aa list, then it will keep driving there until it parks,
if not, it will roam randomly once and then try the aa list again
       choose-nearest-spot
       loop[
; First check if there is a parking spt next to you. here is how
the actual parking move happens
       ; first, parking at your own side of the road and then
parking at the opposite side of the road, parking on horisontal
streets is also allowed.
       let [:roundx (round xcor)
       :roundy (round ycor)]

       ifelse ((pc-at 0 0) = 0 and (pc-at 1 0) = 9 and (count-
turtles-at 1 0) = 0) or ((pc-at 0 0) = 0 and (pc-at (-1) 0) = 9
and (count-turtles-at (-1) 0) = 0 ) or
       ((pc-at 0 0) = 0 and (pc-at -2 0) = 9 and (count-turtles-at
-2 0) = 0 ) or ((pc-at 0 0) = 0 and (pc-at 2 0) = 9 and (count-
turtles-at 2 0) = 0 ) or
       ((pc-at 0 0) = 0 and (pc-at 0 1) = 9 and (count-turtles-at 0
1) = 0 ) or ((pc-at 0 0) = 0 and (pc-at 0 (-1)) = 9 and (count-
turtles-at 0 (-1)) = 0 ) or
       ((pc-at 0 0) = 0 and (pc-at 0 -2) = 9 and (count-turtles-at
0 -2) = 0 ) or ((pc-at 0 0) = 0 and (pc-at 0 2) = 9 and (count-
turtles-at 0 2) = 0 ) or

       ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY)) and
(((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and (pc-at
0 0) = 0 and (pc-at 1 0) = 9 and (count-reserving-at 1 0) = 1) or
((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY)) and
(((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and (pc-at
0 0) = 0 and (pc-at (-1) 0) = 9 and (count-reserving-at (-1) 0) =
1 ) or
       ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY)) and
(((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and (pc-at
0 0) = 0 and (pc-at -2 0) = 9 and (count-reserving-at -2 0) = 1 )
or ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY)) and
(((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and (pc-at
0 0) = 0 and (pc-at 2 0) = 9 and (count-reserving-at 2 0) = 1 ) or
       ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY)) and
(((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and (pc-at
0 0) = 0 and (pc-at 0 1) = 9 and (count-reserving-at 0 1) = 1 ) or
((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY)) and
(((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and (pc-at
0 0) = 0 and (pc-at 0 (-1)) = 9 and (count-reserving-at 0 (-1)) =
1 ) or
       ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY)) and
(((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and (pc-at
0 0) = 0 and (pc-at 0 -2) = 9 and (count-reserving-at 0 -2) = 1 )
or ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY)) and
(((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and (pc-at
0 0) = 0 and (pc-at 0 2) = 9 and (count-reserving-at 0 2) = 1 )[

       if ((pc-at 0 0) = 0 and (pc-at 1 0) = 9 and (count-turtles-
at 1 0) = 0) [ seth 90 fd 1 kill one-of-reserving-at (CoordX -
(round xcor)) (CoordY - (round ycor)) set CoordX 0 set CoordY 0
setDirection 0 setParked? true set steps-to-find-parking 0 setc
red]

       if ((pc-at 0 0) = 0 and (pc-at (-1) 0) = 9 and (count-
turtles-at (-1) 0) = 0) [seth 270 fd 1 kill one-of-reserving-at
(CoordX - (round xcor)) (CoordY - (round ycor)) set CoordX 0 set
```

```
CoordY 0 setDirection 0 setParked? true set steps-to-find-parking
0 setc red]
       if ((pc-at 0 0) = 0 and (pc-at -2 0) = 9 and (count-turtles-
at -2 0) = 0) [seth 270 fd 2 kill one-of-reserving-at (CoordX -
(round xcor)) (CoordY - (round ycor)) set CoordX 0 set CoordY 0
setDirection 0 setParked? true set steps-to-find-parking 0 setc
red]
       if ((pc-at 0 0) = 0 and (pc-at 2 0) = 9 and (count-turtles-
at 2 0) = 0) [seth 90 fd 2 kill one-of-reserving-at (CoordX -
(round xcor)) (CoordY - (round ycor)) set CoordX 0 set CoordY 0
setDirection 0 setParked? true set steps-to-find-parking 0 setc
red]
       if ((pc-at 0 0) = 0 and (pc-at 0 1) = 9 and (count-turtles-
at 0 1) = 0) [seth 0 fd 1 kill one-of-reserving-at (CoordX -
(round xcor)) (CoordY - (round ycor)) set CoordX 0 set CoordY 0
setDirection 0 setParked? true set steps-to-find-parking 0 setc
red]
       if ((pc-at 0 0) = 0 and (pc-at 0 (-1)) = 9 and (count-
turtles-at 0 (-1)) = 0) [seth 180 fd 1 kill one-of-reserving-at
(CoordX - (round xcor)) (CoordY - (round ycor)) set CoordX 0 set
CoordY 0 setDirection 0 setParked? true set steps-to-find-parking
0 setc red]
       if ((pc-at 0 0) = 0 and (pc-at 0 -2) = 9 and (count-turtles-
at 0 -2) = 0 ) [seth 180 fd 2 kill one-of-reserving-at (CoordX -
(round xcor)) (CoordY - (round ycor)) set CoordX 0 set CoordY 0
setDirection 0 setParked? true set steps-to-find-parking 0 setc
red]
       if ((pc-at 0 0) = 0 and (pc-at 0 2) = 9 and (count-turtles-
at 0 2) = 0 ) [seth 0 fd 2 kill one-of-reserving-at (CoordX -
(round xcor)) (CoordY - (round ycor)) set CoordX 0 set CoordY 0
setDirection 0 setParked? true set steps-to-find-parking 0 setc
red]

       if ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY))
and (((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and
(pc-at 0 0) = 0 and (pc-at 1 0) = 9 and (count-reserving-at 1 0) =
1) [seth 90 fd 1 set CoordX 0 set CoordY 0 setDirection 0 kill
one-of-reserving-here setParked? true set steps-to-find-parking 0
setc red]
       if ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY))
and (((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and
(pc-at 0 0) = 0 and (pc-at (-1) 0) = 9 and (count-reserving-at (-
1) 0) = 1) [seth 270 fd 1 set CoordX 0 set CoordY 0 setDirection 0
kill one-of-reserving-here setParked? true set steps-to-find-
parking 0 setc red]
       if ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY))
and (((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and
(pc-at 0 0) = 0 and (pc-at -2 0) = 9 and (count-reserving-at -2 0)
= 1)[seth 270 fd 2 set CoordX 0 set CoordY 0 setDirection 0 kill
one-of-reserving-here setParked? true set steps-to-find-parking 0
setc red]
       if ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY))
and (((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and
(pc-at 0 0) = 0 and (pc-at 2 0) = 9 and (count-reserving-at 2 0) =
1) [seth 90 fd 2 set CoordX 0 set CoordY 0 setDirection 0  kill
one-of-reserving-here setParked? true set steps-to-find-parking 0
setc red]
       if ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY))
and (((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and
(pc-at 0 0) = 0 and (pc-at 0 1) = 9 and (count-reserving-at 0 1) =
1) [seth 0 fd 1 set CoordX 0 set CoordY 0 setDirection 0 kill one-
of-reserving-here setParked? true set steps-to-find-parking 0 setc
red]
```

```
	if ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY))
and (((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and
(pc-at 0 0) = 0 and (pc-at 0 (-1)) = 9 and (count-reserving-at 0
(-1)) = 1) [seth 180 fd 1 set CoordX 0 set CoordY 0 setDirection 0
kill one-of-reserving-here setParked? true set steps-to-find-
parking 0 setc red]
	if ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY))
and (((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and
(pc-at 0 0) = 0 and (pc-at 0 -2) = 9 and (count-reserving-at 0 -2)
= 1)[seth 180 fd 2 set CoordX 0 set CoordY 0 setDirection 0 kill
one-of-reserving-here setParked? true set steps-to-find-parking 0
setc red]
	if ((((:roundy + 4) > CoordY) and ((:roundy - 4) < CoordY))
and (((:roundx + 4) > CoordX) and ((:roundx - 4) < CoordX)) and
(pc-at 0 0) = 0 and (pc-at 0 2) = 9 and (count-reserving-at 0 2) =
1) [seth 0 fd 2 set CoordX 0 set CoordY 0 setDirection 0 kill one-
of-reserving-here setParked? true set steps-to-find-parking 0 setc
red]]
	[if (pc-at 0 0) not= white [ ifelse (pc-ahead = 2 and (pc-at
0 0) = 2) [ifelse (random 2) = 0 [leap 3 set steps-to-find-parking
(steps-to-find-parking + 3)] [ifelse (random 2) = 0 [rt 90 fd 2
set steps-to-find-parking (steps-to-find-parking + 2)][fd 1 lt 90
fd 2 set steps-to-find-parking (steps-to-find-parking + 3)]]]
	[check-patches-after-park fd speed set steps-to-find-parking
(steps-to-find-parking + speed)] if (CoordX = 0 and CoordY =
0)[choose-nearest-spot]]]

	;here we augment the parking counter and check if the
counter is full, in which case a car leaves. When leaving, a car
must also step 1 step away from the spot in order not to park
again.
	if (breed not= reserving and Parked? = true and (pc-at 0 0)
= white) [
	ifelse Counter > ParkTime [
	if ((pc-at (-2) 0) = black and (pc-at (-1) 0) = black and
(pc-at 0 0) = white) [kill one-of-reserving-here seth 270 fd 1
seth 0 setCounter 0 stop]
	if ((pc-at 2 0) = black and (pc-at 1 0) = black and (pc-at 0
0) = white) [kill one-of-reserving-here seth 90 fd 1 seth 180
setCounter 0 stop]
	if ((pc-at 0 (-2)) = black and (pc-at 0 (-1)) = black and
(pc-at 0 0) = white) [kill one-of-reserving-here seth 180 fd 1
seth 270 setCounter 0 stop]
	if ((pc-at 0 2) = black and (pc-at 0 1) = black and (pc-at 0
0) = white) [kill one-of-reserving-here seth 0 fd 1 seth 90
setCounter 0 stop]]
	[set Counter (Counter + 1)]]

; Now, check if you are on an intersection. Here are the rules to
guide a car towards a chosen spot, assuming it can also park on
the opposite side of the road.
ifelse (CoordX not= 0 and CoordY not= 0)[

; for heading = 0, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor < CoordX and ycor < CoordY))[
ifelse (CoordX < (xcor + 3))[fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor < CoordX and ycor > CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
```

```
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor > CoordX and ycor > CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor > CoordX and ycor < CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)][fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]

; for heading = 90, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
< CoordX and ycor < CoordY))[
ifelse (CoordX < (xcor + 3)) [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)][fd 2 set steps-to-find-
parking (steps-to-find-parking + 2)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
< CoordX and ycor > CoordY))[
ifelse (CoordX < (xcor + 3)) [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)] [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
> CoordX and ycor > CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
> CoordX and ycor < CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]

; for heading = 180, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor < CoordX and ycor < CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor < CoordX and ycor > CoordY))[
ifelse (CoordX < (xcor + 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor > CoordX and ycor > CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor > CoordX and ycor < CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]

; for heading = 270, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor < CoordX and ycor < CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor < CoordX and ycor > CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor > CoordX and ycor > CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)] [fd 2 set steps-to-find-
parking (steps-to-find-parking + 2)]]
```

```
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor > CoordX and ycor < CoordY))[
ifelse (CoordX > (xcor - 3)) [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)] [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)]]]

[if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random 2) = 0
[leap 3 set steps-to-find-parking (steps-to-find-parking + 3)]
[ifelse (random 2) = 0 [rt 90 fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)][fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]]]

; if Reserved? was false on the  previous step, try again and then
enter the park proc again.
             ; here set the car to break or accelerate according
to other cars
             ifelse (count-turtles-towards heading 1) > 0     ;if
there is a turtle 1 space ahead, decelerate
             [setspeed speed-of one-of-turtles-towards heading 1
                    decelerate]
                    [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                    [ifelse (count-turtles-at heading 1) > 0
             [setspeed speed-of one-of-turtles-towards heading 2
             decelerate]
             [accelerate]]   ;else accelerate
             [accelerate]]
             if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
             if speed > SpeedLimit [setspeed SpeedLimit]
]]]
end


to drive ; each egnt will park and then drive for a certain time

       if (pc-at 1 0) = 7 or (pc-at 1 0) = 9 [
       seth 0
       setParked? false

       park

       repeat (parking-interval * 10) [ ; these are turtles driving
up
             if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
             ifelse (count-turtles-at 0 1) > 0     ;if there is a
turtle 1 space ahead, decelerate
             [setspeed speed-of one-of-turtles-at 0 1
                    decelerate]
                    [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                    [ifelse (count-turtles-at 0 1) > 0
             [setspeed speed-of one-of-turtles-at 0 2
             decelerate]
             [accelerate]]   ;else accelerate
             [accelerate]]
             if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
             if speed > SpeedLimit [setspeed SpeedLimit]
             fd speed
             ]
             ]
```

```
       if (pc-at (-1) 0) = 7 or (pc-at (-1) 0) = 9 [
       seth 180
       setParked? false
       park
       repeat (parking-interval * 10) [ ; these are turtles driving
up
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
              ifelse (count-turtles-at 0 (-1)) > 0     ;if there is
a turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-at 0 (-1)
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at 0 (-1)) > 0
              [setspeed speed-of one-of-turtles-at 0 (-2)
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]

       if (pc-at 0 1) = 7 or (pc-at 0 1) = 9 [
       seth 270
       setParked? false
       park
       repeat (parking-interval * 10) [ ; these are turtles driving
up
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
              ifelse (count-turtles-at (-1) 0) > 0     ;if there is
a turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-at (-1) 0
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at (-1) 0) > 0
              [setspeed speed-of one-of-turtles-at (-2) 0
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]

       if (pc-at 0 (-1)) = 7 or (pc-at 0 (-1)) = 9 [
       seth 90
       setParked? false
       park
       repeat (parking-interval * 10) [ ; these are turtles driving
up
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
```

```
              ifelse (count-turtles-at 1 0) > 0     ;if there is a
turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-at 1 0
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at 2 0) > 0
              [setspeed speed-of one-of-turtles-at 2 0
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
end

to accelerate
  setspeed (speed + (speedup / 1000))
end

to decelerate
  setspeed speed - (slowdown / 1000)
end
```

Intel_5

Observer Procedures:

```
globals [aa East-Free West-Free Total-Free List-Ready?]
patches-own [EastSide? xcoord ycoord number]
breeds [living working visiting1 visiting2]


to track-vacant
       loop[
       set Total-Free (count-patches-with [pc = white and count-
turtles-here = 0 ] )
       output Total-Free]

end


to track-vacant-all ; here we generate a list called "aa" which
tracks the vacant spots on the east sides of roads
       ; we need to keep them separate so that when a car estimates
its driving distance, it knows if it needs to go around the block
or not
       setList-Ready? false
       let [:a (count-patches-with [pc = white and count-turtles-
here = 0] )]
       setTotal-Free (:a)
```

```
       ask-patches [if pc = white and count-turtles-here = 0 [
       repeat Total-Free [
       if (number? xcor) [let [:xpos xcor]]
       if (number? ycor) [let [:ypos ycor]]
       let [:bb (list xcor ycor)]

       set aa make-list 0 (East-Free)
       set aa insert 1 aa :bb
       ]]]
       setList-Ready? true
       ;show aa ; for debugging only to see if "aa" works
end


to setup

       ct
       crt number-of-cars
       ask-turtles [setPark-Counter 0 setshape cross setc red
setspeed 1 setSpeedLimit 1
       if (who <= (number-of-cars / 5)) [setbreed living]
       if (who > (number-of-cars / 5) and who <= (number-of-cars /
2)) [setbreed working]
       if (who > (number-of-cars / 2) and who <= (number-of-cars /
1.25)) [setbreed visiting1]
       if (who > (number-of-cars / 1.25)) [setbreed visiting2]
              if breed = living [setParkTime 48]
              if breed = working [setParkTime 32]
              if breed = visiting1 [setParkTime 2]
              if breed = visiting2 [setParkTime 8]

       loop [ifelse ((pc-at 0 0) = black and count-turtles-here = 1
) [stop] [
       ifelse (random 2) = 0 [seth 0] [ifelse (random 2) = 0 [seth
90][ifelse (random 2) = 0 [seth 180] [seth 270]]]
       fd (int random 25)
       ifelse (random 2) = 0 [seth 0] [ifelse (random 2) = 0 [seth
90][ifelse (random 2) = 0 [seth 180] [seth 270]]]
       fd (int random 25) ]]]
       ask-patches-with [pc = white] [
       let [:x xcor] set xcoord :x let [:y ycor] set ycoord :y set
number (:x * :y); assign each patch an x and y coordinate valeu
and a unique number.
       ]
       starttrackingall
       startcountaverage
end


to clear-cars
       ct
end


to count-steps-to-find-parking
output (average-of-turtles-with [color = yellow] [steps-to-find-
parking])
end


to stop-it
       stoptrackingall
       stopcountaverage
```

```
       stopDrive&Park
end


to count-average ; this is for statistical analysis: paste these
number into excel and calculate the mean, median and standard
deviation
       show (round average-of-turtles-with [color = yellow] [steps-
to-find-parking]) wait 1
end
```

Turtle Procedures:

```
turtles-own [speed SpeedLimit ParkTime Parked? steps-to-find-
parking Direction CoordX CoordY MyDist Total-Min-Dist Aacopy
Search-Counter Park-Counter]


to check-patches-after-park
  if (pc-ahead = 7) or (pc-ahead = 9) [
       rt 90 check-patches-after-park
      ]
end


to check-side
       ask-patch-at CoordX CoordY [output EastSide?]
end


to try-all-choices
       let [:nullcheck2 (length Aacopy)] ifelse :nullcheck2 not= 0
[
       set Search-Counter 1
       setMyDist 2000
       repeat :nullcheck2 [ ;check only for free spots, don't waste
RAM
       let [:item-number (item Search-Counter Aacopy)] ; extract
the first (eventually each) element from the aa list
       let [:item-numberX (item 1 :item-number)] ; extract the X
value of aa item
       let [:item-numberY (item 2 :item-number)] ; extract the Y
value of the aa item
       let [:a (round(:item-numberX - xcor))]
       let [:b (round(:item-numberY - ycor))]
       ifelse (heading = 0 and :item-numberY < ycor) or (heading =
90 and :item-numberX < xcor) or (heading = 180 and :item-numberY >
ycor) or (heading = 270 and :item-numberX > xcor)[
       let [:distnew ((abs :a) + (abs :b) + 45)]] ; if the
destination spot is in the opposite direction, the add 1/2
(average) block loop (22 + 8) to the dist
       [let [:distnew ((abs :a) + (abs :b))]]

       if (:distnew < MyDist ) [let[:kchosen Search-Counter]
setMyDist :distnew]
       ; CoordX and CoordY are turtle-own variables, which remember
which parking spot the turtles zoomed onto, and will keep that
until a turtles goes to that spot.
       if Search-Counter <= :nullcheck2 [set Search-Counter
(Search-Counter + 1)]
```

```
       ]

       set :item-number (item :kchosen Aacopy) ; extract the
memorized smallest distance element from the :aacopy list
       set :item-numberX (item 1 :item-number) ; extract the X
value
       set :item-numberY (item 2 :item-number) ; extract the Y
value
       set :a (round (:item-numberX - xcor))
       set :b (round (:item-numberY - ycor))

       ifelse (heading = 0 and :item-numberY < ycor) or (heading =
90 and :item-numberX < xcor) or (heading = 180 and :item-numberY >
ycor) or (heading = 270 and :item-numberX > xcor)[
       setMyDist ( (abs :a) + (abs :b) + 45)] ; if the destination
spot is in the opposite direction, the add 1/2 (average) block
loop (22 + 8) to the dist
       [setMyDist ((abs :a) + (abs :b))]

        let [:a (CoordX) :b (CoordY)]
       setTotal-Min-Dist ( min-of-turtles-with [CoordX = :a and
CoordY = :b] [MyDist]) ;show Total-Min-Dist; ask from all turtles
who has the same target, what the min dist is and save it for
later
        ifelse (MyDist <= Total-Min-Dist)[
       set CoordX (:item-numberX) set CoordY (:item-numberY)
       setDirection (towards-nowrap CoordX CoordY)]
        [set Aacopy (remove-element :item-number Aacopy) try-all-
choices]]
       [setCoordX 0 setCoordY 0 setMyDist 1000]
end


to choose-nearest-spot

; here extract the itam in the aa list, then extract ech item's x
and y and check the patche's distance from the turtle (for all
patches)
; use a  turtles-own variable named MyDist which indicates the
distance to the nearest free spot and a variable named ":kchosen"
indicating the number of the element in the aa list


       wait-until [List-Ready? = true]
       let [:nullcheck (length aa)]
       ifelse (:nullcheck > 0) [
       setMyDist 2000 ; initialize a distance that is bigger than
any on screen dist, so that a new dist will always be smaller
       let [:CoordX 0]
       let [:CoordY 0]
       let [:Dir 90]
       setAacopy (copy-list aa) ; make a copy of the "aa", so if
the real aa changes in length, their's remain the same until end
of counting

 try-all-choices

       ] [setCoordX 0 setCoordY 0 setMyDist 1000]
end
```

```
to park ; has to be done so that agent will look for parking until
found

       if (Parked? = false)[
       setc yellow
       ;ask-patch-at CoordX CoordY [setpc green]
       ;ask-patch-at CoordX CoordY [setpc white]
       loop[
       choose-nearest-spot
              ; here set the car to break or accelerate according
to other cars
              ifelse (count-turtles-towards heading 1) > 0     ;if
there is a turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-towards heading 1
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at heading 1) > 0
              [setspeed speed-of one-of-turtles-towards heading 2
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]

       ; here is how the actual parking move happens
       ; first, parking at your own side of the road and then
parking at the opposite side of the road, parking on horisontal
streets is also allowed.
       ; augment the counter, which counts the program iterations
during which a car parks (instead of real time)
       ifelse ((pc-at 0 0) = 0 and (pc-at 1 0) = 9 and (count-
turtles-at 1 0) = 0) or ((pc-at 0 0) = 0 and (pc-at (-1) 0) = 9
and (count-turtles-at (-1) 0) = 0) or
       ((pc-at 0 0) = 0 and (pc-at -2 0) = 9 and (count-turtles-at
-2 0) = 0) or ((pc-at 0 0) = 0 and (pc-at 2 0) = 9 and (count-
turtles-at 2 0) = 0) or
       ((pc-at 0 0) = 0 and (pc-at 0 1) = 9 and (count-turtles-at 0
1) = 0) or ((pc-at 0 0) = 0 and (pc-at 0 (-1)) = 9 and (count-
turtles-at 0 (-1)) = 0) or
       ((pc-at 0 0) = 0 and (pc-at 0 -2) = 9 and (count-turtles-at
0 -2) = 0) or ((pc-at 0 0) = 0 and (pc-at 0 2) = 9 and (count-
turtles-at 0 2) = 0) or
       ((pc-at 0 0) = 9) [

       ; here we augment the parking counter and check if the
counter is full, in which case a car leaves
       if ((pc-at 0 0) = 9 and (pc-at (-1) 0) = 0) [ setPark-
Counter Park-Counter + 1 if Park-Counter > ParkTime [setPark-
Counter 0 seth 270 fd 1 seth 0 setParked? true stop]]
       if ((pc-at 0 0) = 9 and (pc-at 1 0) = 0) [ setPark-Counter
Park-Counter + 1 if Park-Counter > ParkTime [setPark-Counter 0
seth 90 fd 1 seth 180 setParked? true stop]]
       if ((pc-at 0 0) = 9 and (pc-at 0 (-1)) = 0) [setPark-Counter
Park-Counter + 1 if Park-Counter > ParkTime [setPark-Counter 0
seth 180 fd 1 seth 270 setParked? true stop]]
       if ((pc-at 0 0) = 9 and (pc-at 0 1) = 0) [setPark-Counter
Park-Counter + 1 if Park-Counter > ParkTime [setPark-Counter 0
seth 0 fd 1 seth 90 setParked? true stop]]

       if ((pc-at 0 0) = 0 and (pc-at 1 0) = 9 and (count-turtles-
at 1 0) = 0) [seth 90 fd 1 set steps-to-find-parking 0 setc red
setMyDist 0]
```

```
       if ((pc-at 0 0) = 0 and (pc-at (-1) 0) = 9 and (count-
turtles-at (-1) 0) = 0) [seth 270 fd 1 set steps-to-find-parking 0
setc red setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at -2 0) = 9 and (count-turtles-
at -2 0) = 0)[seth 270 fd 2 set steps-to-find-parking 0 setc red
setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at 2 0) = 9 and (count-turtles-
at 2 0) = 0) [seth 90 fd 2 set steps-to-find-parking 0 setc red
setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at 0 1) = 9 and (count-turtles-
at 0 1) = 0) [seth 0 fd 1 set steps-to-find-parking 0 setc red
setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at 0 (-1)) = 9 and (count-
turtles-at 0 (-1)) = 0) [seth 180 fd 1 set steps-to-find-parking 0
setc red setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at 0 -2) = 9 and (count-turtles-
at 0 -2) = 0)[seth 180 fd 2 set steps-to-find-parking 0 setc red
setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at 0 2) = 9 and (count-turtles-
at 0 2) = 0) [seth 0 fd 2 set steps-to-find-parking 0 setc red
setMyDist 0]]
       [if (pc-at 0 0) not= white [ ifelse (pc-ahead = 2 and (pc-at
0 0) = 2) [ifelse (random 2) = 0 [leap 3 set steps-to-find-parking
(steps-to-find-parking + 3)] [ifelse (random 2) = 0 [rt 90 fd 2
set steps-to-find-parking (steps-to-find-parking + 2)][fd 1 lt 90
fd 2 set steps-to-find-parking (steps-to-find-parking + 3)]]]
       [check-patches-after-park fd speed set steps-to-find-parking
(steps-to-find-parking + speed)] if (CoordX = 0 and CoordY =
0)[choose-nearest-spot]]]


; here are the rules to guide a car towards a chosen spot,
assuming it can also park on the opposite side of the road.
;in reality you should try the second best option here and then
third best and so on!

ifelse (CoordX not= 0 and CoordY not= 0 and MyDist not= 1000) [
;only if you are closest in the competition for a given spot,
drive there, else roam randomly and try again next step

; for heading = 0, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor < CoordX and ycor < CoordY))[
ifelse (CoordX < (xcor + 3))[fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor < CoordX and ycor > CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor > CoordX and ycor > CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor > CoordX and ycor < CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)][fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]

; for heading = 90, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
< CoordX and ycor < CoordY))[
```

```
ifelse (CoordX < (xcor + 3)) [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)][fd 2 set steps-to-find-
parking (steps-to-find-parking + 2)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
< CoordX and ycor > CoordY))[
ifelse (CoordX < (xcor + 3)) [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)] [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
> CoordX and ycor > CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
> CoordX and ycor < CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]

; for heading = 180, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor < CoordX and ycor < CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor < CoordX and ycor > CoordY))[
ifelse (CoordX < (xcor + 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor > CoordX and ycor > CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor > CoordX and ycor < CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]

; for heading = 270, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor < CoordX and ycor < CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor < CoordX and ycor > CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor > CoordX and ycor > CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)] [fd 2 set steps-to-find-
parking (steps-to-find-parking + 2)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor > CoordX and ycor < CoordY))[
ifelse (CoordX > (xcor - 3)) [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)] [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)]]
][if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random 2) = 0
[leap 3 set steps-to-find-parking (steps-to-find-parking + 3)]
[ifelse (random 2) = 0 [rt 90 fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)][fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]]]
]]
end
```

```
to drive ; each egnt will park and then drive for a certain time

       if (pc-at 1 0) = 7 or (pc-at 1 0) = 9 [
       seth 0
       setParked? false
       park
       repeat (parking-interval * 10) [ ; these are turtles driving
up
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
              ifelse (count-turtles-at 0 1) > 0     ;if there is a
turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-at 0 1
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at 0 1) > 0
              [setspeed speed-of one-of-turtles-at 0 2
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]

       if (pc-at (-1) 0) = 7 or (pc-at (-1) 0) = 9 [
       seth 180
       setParked? false
       park
       repeat (parking-interval * 10) [ ; these are turtles driving
up
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
              ifelse (count-turtles-at 0 (-1)) > 0     ;if there is
a turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-at 0 (-1)
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at 0 (-1)) > 0
              [setspeed speed-of one-of-turtles-at 0 (-2)
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]

       if (pc-at 0 1) = 7 or (pc-at 0 1) = 9 [
       seth 270
       setParked? false
       park
       repeat (parking-interval * 10) [ ; these are turtles driving
up
```

```
            if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
            ifelse (count-turtles-at (-1) 0) > 0     ;if there is
a turtle 1 space ahead, decelerate
            [setspeed speed-of one-of-turtles-at (-1) 0
                  decelerate]
                  [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                  [ifelse (count-turtles-at (-1) 0) > 0
            [setspeed speed-of one-of-turtles-at (-2) 0
            decelerate]
            [accelerate]]   ;else accelerate
            [accelerate]]
            if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
            if speed > SpeedLimit [setspeed SpeedLimit]
            fd speed
            ]
            ]

      if (pc-at 0 (-1)) = 7 or (pc-at 0 (-1)) = 9 [
      seth 90
      setParked? false
      park
      repeat (parking-interval * 10) [ ; these are turtles driving
up
            if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
            ifelse (count-turtles-at 1 0) > 0     ;if there is a
turtle 1 space ahead, decelerate
            [setspeed speed-of one-of-turtles-at 1 0
                  decelerate]
                  [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                  [ifelse (count-turtles-at 2 0) > 0
            [setspeed speed-of one-of-turtles-at 2 0
            decelerate]
            [accelerate]]   ;else accelerate
            [accelerate]]
            if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
            if speed > SpeedLimit [setspeed SpeedLimit]
            fd speed
            ]
            ]
            if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
            if (pc-at 0 0 )= 2 [fd 1]
end


to accelerate
  setspeed (speed + (speedup / 1000))
end


to decelerate
  setspeed speed - (slowdown / 1000)
end
```

Intel_7

Observer Procedures:

```
globals [aa bb Total-Free List-Ready? Occupied-List-Ready?]
breeds [living working visiting1 visiting2]
patches-own [Probability-Timer]

to track-vacant
       loop[
       set Total-Free (count-patches-with [pc = white and count-
turtles-here = 0 ] )
       output Total-Free]
end


to track-vacant-all ; here we generate a list called "aa" which
tracks the vacant spots on the east sides of roads
       wait-until [List-Ready? = true]
       if List-Ready? = true [
       set List-Ready? false
       set aa make-list 0 0
       ask-patches [if pc = white and count-turtles-here = 0 [
       let [:bb (list xcor ycor)]
       set aa insert 1 aa :bb
       ]]
       set List-Ready? true]

       ask-patches-with [pc = white] [ if (count-turtles-here = 0)
[
       setProbability-Timer 0]
       ]
end


to track-occupied-all ; here we generate a list called "bb" which
tracks the ocupied spots for probabilitsic search
       wait-until [Occupied-List-Ready? = true]
       if Occupied-List-Ready? = true [
       setOccupied-List-Ready? false
       set bb make-list 0 0
       ask-patches [if pc = white and count-turtles-here > 0 [
       let [:bb (list xcor ycor)]
       set bb insert 1 bb :bb
       ]]
       setOccupied-List-Ready? true]
end


to setup
       set-random-seed 100
       set List-Ready? true
       setOccupied-List-Ready? true
       set aa make-list 0 0
       set bb make-list 0 0
       ct
       crt number-of-cars
       ask-turtles [setPark-Counter 0 setshape cross setc red
setspeed 1 setSpeedLimit 1
       if (who <= (number-of-cars / 5)) [setbreed living]
       if (who > (number-of-cars / 5) and who <= (number-of-cars /
2)) [setbreed working]
```

```
      if (who > (number-of-cars / 2) and who <= (number-of-cars /
1.25)) [setbreed visiting1]
      if (who > (number-of-cars / 1.25)) [setbreed visiting2]
            if breed = living [setParkTime 48]
            if breed = working [setParkTime 32]
            if breed = visiting1 [setParkTime 2]
            if breed = visiting2 [setParkTime 8]

      loop [ifelse ((pc-at 0 0) = black and count-turtles-here = 1
) [stop] [
      ifelse (random 2) = 0 [seth 0] [ifelse (random 2) = 0 [seth
90][ifelse (random 2) = 0 [seth 180] [seth 270]]]
      fd (int random 25)
      ifelse (random 2) = 0 [seth 0] [ifelse (random 2) = 0 [seth
90][ifelse (random 2) = 0 [seth 180] [seth 270]]]
      fd (int random 25) ]]]
      starttrackingall
      startcountaverage
      startoccupied-all
end


to count-steps-to-find-parking
output (average-of-turtles-with [color = yellow] [steps-to-find-
parking])
end


to stop-it
      stoptrackingall
      stopcountaverage
      stopDrive&Park
      stopoccupied-all
end


to count-average ; this is for statistical analysis: paste these
number into excel and calculate the mean, median and standard
deviation
      show (round average-of-turtles-with [color = yellow] [steps-
to-find-parking]) wait 1
end
```

Turtle Procedures:

```
turtles-own [speed SpeedLimit ParkTime Parked? steps-to-find-
parking Direction CoordX CoordY MyDist MyDist-Occupied Total-Min-
Dist Total-Min-Occupied-Chances Aacopy Bbcopy Search-Counter Park-
Counter Vacant-Chances Occupied-Chances]

to check-patches-after-park
  if (pc-ahead = 7) or (pc-ahead = 9) [
      rt 90 check-patches-after-park
      ]
end
```

```
to try-all-vacant-choices

       let [:nullcheck2 (length Aacopy)] ifelse :nullcheck2 > 0 [
       set Search-Counter 1
       setMyDist 2000
       repeat :nullcheck2 [ ;check only for free spots, don't waste
RAM
       let [:item-number (item Search-Counter Aacopy)] ; extract
the first (eventually each) element from the aa list
       let [:item-numberX (item 1 :item-number)] ; extract the X
value of aa item
       let [:item-numberY (item 2 :item-number)] ; extract the Y
value of the aa item
       let [:a (round(:item-numberX - xcor))]
       let [:b (round(:item-numberY - ycor))]
       ifelse (heading = 0 and :item-numberY < ycor) or (heading =
90 and :item-numberX < xcor) or (heading = 180 and :item-numberY >
ycor) or (heading = 270 and :item-numberX > xcor)[
       let [:distnew ((abs :a) + (abs :b) + 45)]] ; if the
destination spot is in the opposite direction, the add 1/2
(average) block loop (22 + 8) to the dist
       [let [:distnew ((abs :a) + (abs :b))]]

       if (:distnew <= MyDist ) [let[:kchosen Search-Counter]
setMyDist :distnew]
       ; CoordX and CoordY are turtle-own variables, which remember
which parking spot the turtles zoomed onto, and will keep that
until a turtles goes to that spot.
       if (Search-Counter <= (:nullcheck2)) [set Search-Counter
(Search-Counter + 1)]
       ]

       set :item-number (item :kchosen Aacopy) ; extract the
memorized smallest distance element from the :aacopy list
       set :item-numberX (item 1 :item-number) ; extract the X
value
       set :item-numberY (item 2 :item-number) ; extract the Y
value
       set :a (round (:item-numberX - xcor))
       set :b (round (:item-numberY - ycor))

       ifelse (heading = 0 and :item-numberY < ycor) or (heading =
90 and :item-numberX < xcor) or (heading = 180 and :item-numberY >
ycor) or (heading = 270 and :item-numberX > xcor)[
       setMyDist ( (abs :a) + (abs :b) + 45)] ; if the destination
spot is in the opposite direction, the add 1/2 (average) block
loop (22 + 8) to the dist
       [setMyDist ((abs :a) + (abs :b))]

        set :a (CoordX) set :b (CoordY)
       setTotal-Min-Dist ( min-of-turtles-with [CoordX = :a and
CoordY = :b] [MyDist]) ;show Total-Min-Dist; ask from all turtles
who has the same target, what the min dist is and save it for
later
        ifelse (MyDist <= Total-Min-Dist)[
       set CoordX (:item-numberX) set CoordY (:item-numberY)
       setDirection (towards-nowrap CoordX CoordY)
       setVacant-Chances (1 / MyDist)]; this is for comparison with
the occupied spot search- availability probability / distance.
        [set Aacopy (remove-element :item-number Aacopy) try-all-
vacant-choices]]
       [setCoordX 0 setCoordY 0 setMyDist 1000]
end
```

```
to try-all-occupied-choices

       let [:nullcheck3 (length Bbcopy)]
       ifelse (:nullcheck3 > 0) [
       setMyDist-Occupied 2000 ; initialize a distance that is
bigger than any on screen dist, so that a new dist will always be
smaller
       set Search-Counter 1 ; loop through every elemnt in the
"number" list starts with 1
       repeat :nullcheck3 [ ;check only for free spots, don't waste
RAM
       let [:item-number (item Search-Counter Bbcopy)] ; extract
the first (eventually each) element from the aa list
       let [:item-numberX (item 1 :item-number)] ; extract the X
value of aa item
       let [:item-numberY (item 2 :item-number)] ; extract the Y
value of the aa item
       let [:a (round(:item-numberX - xcor))]
       let [:b (round(:item-numberY - ycor))]
       ifelse (heading = 0 and :item-numberY < ycor) or (heading =
90 and :item-numberX < xcor) or (heading = 180 and :item-numberY >
ycor) or (heading = 270 and :item-numberX > xcor)[
       let [:distnew ((abs :a) + (abs :b) + 45)]] ; if the
destination spot is in the opposite direction, the add 1/2
(average) block loop (22 + 8) to the dist
       [let [:distnew ((abs :a) + (abs :b))]]

       if (:distnew <= MyDist-Occupied ) [let [:kchosen Search-
Counter] setMyDist-Occupied :distnew]
       ; CoordX and CoordY are turtle-own variables, which remember
which parking spot the turtles zoomed onto, and will keep that
until a turtles goes to that spot.
       if (Search-Counter <= (:nullcheck3)) [set Search-Counter
(Search-Counter + 1)]
       ]

       set :item-number (item :kchosen Bbcopy) ; extract the
memorized smallest distance element from the :aacopy list
       set :item-numberX (item 1 :item-number) ; extract the X
value
       set :item-numberY (item 2 :item-number) ; extract the Y
value
       set :a (round (:item-numberX - xcor))
       set :b (round (:item-numberY - ycor))

       ifelse (heading = 0 and :item-numberY < ycor) or (heading =
90 and :item-numberX < xcor) or (heading = 180 and :item-numberY >
ycor) or (heading = 270 and :item-numberX > xcor)[
       setMyDist-Occupied ( (abs :a) + (abs :b) + 45)] ; if the
destination spot is in the opposite direction, the add 1/2
(average) block loop (22 + 8) to the dist
       [setMyDist-Occupied ((abs :a) + (abs :b))]
       ;set :a (CoordX) set :b (CoordY)
       ;setTotal-Min-Dist ( min-of-turtles-with [CoordX = :a and
CoordY = :b] [MyDist-Occupied]) ;show Total-Min-Dist; ask from all
turtles who has the same target, what the min dist is and save it
for later
       show (pc-at :a :b)
        if ((Probability-Timer-at :a :b) > 0) and ((Probability-
Timer-at :a :b) < 2) [setOccupied-Chances (0.2 / (abs (int MyDist-
Occupied - 1)))]
       if ((Probability-Timer-at :a :b) > 2) and ((Probability-
Timer-at :a :b) < 8) [setOccupied-Chances (0.375 / (abs (int
```

```
MyDist-Occupied - 6)))] ; 3 is half of six, which is ha max
waiting time.
       if ((Probability-Timer-at :a :b) > 8) and ((Probability-
Timer-at :a :b) < 32) [setOccupied-Chances (0.6 / (abs (int
MyDist-Occupied - 24)))] ; 12 is half of 24, which is ha max
waiting time.
       if ((Probability-Timer-at :a :b) > 32) and ((Probability-
Timer-at :a :b) < 48) [setOccupied-Chances (1 / (abs (int MyDist-
Occupied - 16)))] ; 8 is half of 16, which is ha max waiting time.

        set :a (CoordX) set :b (CoordY)
       setTotal-Min-Occupied-Chances ( min-of-turtles-with [CoordX
= :a and CoordY = :b] [Occupied-Chances])
        ifelse (Occupied-Chances <= Total-Min-Occupied-Chances)[
       setCoordX (:item-numberX) setCoordY (:item-numberY)
       setDirection (towards-nowrap CoordX CoordY)] ;show
"gotooccupied!"]
        [setBbcopy (remove-element :item-number Aacopy) try-all-
vacant-choices]][setCoordX 0 setCoordY 0 setMyDist-Occupied 1000]
end


to choose-nearest-spot
; here extract the itam in the aa list, then extract ech item's x
and y and check the patche's distance from the turtle (for all
patches)
; use a  turtles-own variable named MyDist which indicates the
distance to the nearest free spot and a variable named ":kchosen"
indicating the number of the element in the aa list
       wait-until [(Occupied-List-Ready? = true) and (List-Ready? =
true)]
       if (Occupied-List-Ready? = true and List-Ready? = true) [

       let [:nullcheck (length aa)]
       ifelse (:nullcheck > 0) [
       setMyDist 2000 ; initialize a distance that is bigger than
any on screen dist, so that a new dist will always be smaller
       setAacopy (copy-list aa) ; make a copy of the "aa", so if
the real aa changes in length, their's remain the same until end
of counting
       setBbcopy (copy-list bb); do the same for occupied spots
        try-all-vacant-choices if (CoordX  = 0 and CoordY = 0)
[try-all-occupied-choices]]
       [try-all-occupied-choices]]
end


to park ; has to be done so that agent will look for parking until
found

       if (Parked? = false)[
       setc yellow
       ;ask-patch-at CoordX CoordY [setpc green]
       ;ask-patch-at CoordX CoordY [setpc white]
       loop[
       choose-nearest-spot
             ; here set the car to break or accelerate according
to other cars
             ifelse (count-turtles-towards heading 1) > 0     ;if
there is a turtle 1 space ahead, decelerate
             [setspeed speed-of one-of-turtles-towards heading 1
                   decelerate]
                   [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
```

```
                 [ifelse (count-turtles-at heading 1) > 0
             [setspeed speed-of one-of-turtles-towards heading 2
             decelerate]
             [accelerate]]   ;else accelerate
             [accelerate]]
             if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
             if speed > SpeedLimit [setspeed SpeedLimit]

       ; here is how the actual parking move happens
       ; first, parking at your own side of the road and then
parking at the opposite side of the road, parking on horisontal
streets is also allowed.
       ; augment the counter, which counts the program iterations
during which a car parks (instead of real time)
       ifelse ((pc-at 0 0) = 0 and (pc-at 1 0) = 9 and (count-
turtles-at 1 0) = 0) or ((pc-at 0 0) = 0 and (pc-at (-1) 0) = 9
and (count-turtles-at (-1) 0) = 0) or
       ((pc-at 0 0) = 0 and (pc-at -2 0) = 9 and (count-turtles-at
-2 0) = 0) or ((pc-at 0 0) = 0 and (pc-at 2 0) = 9 and (count-
turtles-at 2 0) = 0) or
       ((pc-at 0 0) = 0 and (pc-at 0 1) = 9 and (count-turtles-at 0
1) = 0) or ((pc-at 0 0) = 0 and (pc-at 0 (-1)) = 9 and (count-
turtles-at 0 (-1)) = 0) or
       ((pc-at 0 0) = 0 and (pc-at 0 -2) = 9 and (count-turtles-at
0 -2) = 0) or ((pc-at 0 0) = 0 and (pc-at 0 2) = 9 and (count-
turtles-at 0 2) = 0) or
       ((pc-at 0 0) = 9) [

       ; here we augment the parking counter and check if the
counter is full, in which case a car leaves
       if ((pc-at 0 0) = 9 and (pc-at (-1) 0) = 0) [ setPark-
Counter Park-Counter + 1 ask-patch-at 0 0 [setProbability-Timer
(Probability-Timer + 1)] if Park-Counter > ParkTime [setPark-
Counter 0 seth 270 fd 1 seth 0 setParked? true stop]]
       if ((pc-at 0 0) = 9 and (pc-at 1 0) = 0) [ setPark-Counter
Park-Counter + 1 ask-patch-at 0 0 [setProbability-Timer
(Probability-Timer + 1)] if Park-Counter > ParkTime [setPark-
Counter 0 seth 90 fd 1 seth 180 setParked? true stop]]
       if ((pc-at 0 0) = 9 and (pc-at 0 (-1)) = 0) [setPark-Counter
Park-Counter + 1 ask-patch-at 0 0 [setProbability-Timer
(Probability-Timer + 1)] if Park-Counter > ParkTime [setPark-
Counter 0 seth 180 fd 1 seth 270 setParked? true stop]]
       if ((pc-at 0 0) = 9 and (pc-at 0 1) = 0) [setPark-Counter
Park-Counter + 1 ask-patch-at 0 0 [setProbability-Timer
(Probability-Timer + 1)] if Park-Counter > ParkTime [setPark-
Counter 0 seth 0 fd 1 seth 90 setParked? true stop]]

       if ((pc-at 0 0) = 0 and (pc-at 1 0) = 9 and (count-turtles-
at 1 0) = 0) [seth 90 fd 1 set steps-to-find-parking 0 setc red
setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at (-1) 0) = 9 and (count-
turtles-at (-1) 0) = 0) [seth 270 fd 1 set steps-to-find-parking 0
setc red setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at -2 0) = 9 and (count-turtles-
at -2 0) = 0)[seth 270 fd 2 set steps-to-find-parking 0 setc red
setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at 2 0) = 9 and (count-turtles-
at 2 0) = 0) [seth 90 fd 2 set steps-to-find-parking 0 setc red
setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at 0 1) = 9 and (count-turtles-
at 0 1) = 0) [seth 0 fd 1 set steps-to-find-parking 0 setc red
setMyDist 0]
```

```
       if ((pc-at 0 0) = 0 and (pc-at 0 (-1)) = 9 and (count-
turtles-at 0 (-1)) = 0) [seth 180 fd 1 set steps-to-find-parking 0
setc red setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at 0 -2) = 9 and (count-turtles-
at 0 -2) = 0)[seth 180 fd 2 set steps-to-find-parking 0 setc red
setMyDist 0]
       if ((pc-at 0 0) = 0 and (pc-at 0 2) = 9 and (count-turtles-
at 0 2) = 0) [seth 0 fd 2 set steps-to-find-parking 0 setc red
setMyDist 0]]
       [if (pc-at 0 0) not= white [ ifelse (pc-ahead = 2 and (pc-at
0 0) = 2) [ifelse (random 2) = 0 [leap 3 set steps-to-find-parking
(steps-to-find-parking + 3)] [ifelse (random 2) = 0 [rt 90 fd 2
set steps-to-find-parking (steps-to-find-parking + 2)][fd 1 lt 90
fd 2 set steps-to-find-parking (steps-to-find-parking + 3)]]]
       [check-patches-after-park fd speed set steps-to-find-parking
(steps-to-find-parking + speed)] if (CoordX = 0 and CoordY =
0)[choose-nearest-spot]]]


; here are the rules to guide a car towards a chosen spot,
assuming it can also park on the opposite side of the road.
;in reality you should try the second best option here and then
third best and so on!

ifelse (CoordX not= 0 and CoordY not= 0 and MyDist not= 1000) [
;only if you are closest in the competition for a given spot,
drive there, else roam randomly and try again next step

; for heading = 0, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor < CoordX and ycor < CoordY))[
ifelse (CoordX < (xcor + 3))[fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor < CoordX and ycor > CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor > CoordX and ycor > CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and ((heading = 0) and
(xcor > CoordX and ycor < CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)][fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]

; for heading = 90, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
< CoordX and ycor < CoordY))[
ifelse (CoordX < (xcor + 3)) [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)][fd 2 set steps-to-find-
parking (steps-to-find-parking + 2)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
< CoordX and ycor > CoordY))[
ifelse (CoordX < (xcor + 3)) [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)] [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
> CoordX and ycor > CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
```

```
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 90 and (xcor
> CoordX and ycor < CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]

; for heading = 180, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor < CoordX and ycor < CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor < CoordX and ycor > CoordY))[
ifelse (CoordX < (xcor + 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor > CoordX and ycor > CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)] [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 180 and
(xcor > CoordX and ycor < CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]

; for heading = 270, supposing that you can also park on the
opposite side of the road
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor < CoordX and ycor < CoordY))[
rt 90 fd 1 set steps-to-find-parking (steps-to-find-parking + 1)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor < CoordX and ycor > CoordY))[
fd 1 lt 90 fd 2 set steps-to-find-parking (steps-to-find-parking +
3)]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor > CoordX and ycor > CoordY))[
ifelse (CoordX > (xcor - 3)) [fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)] [fd 2 set steps-to-find-
parking (steps-to-find-parking + 2)]]
if ((pc-at 0 0) = 2 and (pc-ahead)= 2) and (heading = 270 and
(xcor > CoordX and ycor < CoordY))[
ifelse (CoordX > (xcor - 3)) [rt 90 fd 1 set steps-to-find-parking
(steps-to-find-parking + 1)] [fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)]]
][if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random 2) = 0
[leap 3 set steps-to-find-parking (steps-to-find-parking + 3)]
[ifelse (random 2) = 0 [rt 90 fd 2 set steps-to-find-parking
(steps-to-find-parking + 2)][fd 1 lt 90 fd 2 set steps-to-find-
parking (steps-to-find-parking + 3)]]]]
]]
end


to drive ; each egnt will park and then drive for a certain time

       if (pc-at 1 0) = 7 or (pc-at 1 0) = 9 [
       seth 0
       setParked? false

       park

       repeat (parking-interval * 10) [ ; these are turtles driving
up
```

```
            if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
            ifelse (count-turtles-at 0 1) > 0     ;if there is a
turtle 1 space ahead, decelerate
            [setspeed speed-of one-of-turtles-at 0 1
                  decelerate]
                  [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                  [ifelse (count-turtles-at 0 1) > 0
            [setspeed speed-of one-of-turtles-at 0 2
            decelerate]
            [accelerate]]   ;else accelerate
            [accelerate]]
            if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
            if speed > SpeedLimit [setspeed SpeedLimit]
            fd speed
            ]
            ]

      if (pc-at (-1) 0) = 7 or (pc-at (-1) 0) = 9 [
      seth 180
      setParked? false
      park
      repeat (parking-interval * 10) [ ; these are turtles driving
up
            if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
            ifelse (count-turtles-at 0 (-1)) > 0     ;if there is
a turtle 1 space ahead, decelerate
            [setspeed speed-of one-of-turtles-at 0 (-1)
                  decelerate]
                  [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                  [ifelse (count-turtles-at 0 (-1)) > 0
            [setspeed speed-of one-of-turtles-at 0 (-2)
            decelerate]
            [accelerate]]   ;else accelerate
            [accelerate]]
            if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
            if speed > SpeedLimit [setspeed SpeedLimit]
            fd speed
            ]
            ]

      if (pc-at 0 1) = 7 or (pc-at 0 1) = 9 [
      seth 270
      setParked? false
      park
      repeat (parking-interval * 10) [ ; these are turtles driving
up
            if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
            ifelse (count-turtles-at (-1) 0) > 0     ;if there is
a turtle 1 space ahead, decelerate
            [setspeed speed-of one-of-turtles-at (-1) 0
                  decelerate]
                  [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                  [ifelse (count-turtles-at (-1) 0) > 0
```

```
              [setspeed speed-of one-of-turtles-at (-2) 0
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]

       if (pc-at 0 (-1)) = 7 or (pc-at 0 (-1)) = 9 [
       seth 90
       setParked? false
       park
       repeat (parking-interval * 10) [ ; these are turtles driving
up
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
              ifelse (count-turtles-at 1 0) > 0     ;if there is a
turtle 1 space ahead, decelerate
              [setspeed speed-of one-of-turtles-at 1 0
                     decelerate]
                     [ifelse lookahead = 2      ;if lookahead=2,
check 2 spaces ahead also
                     [ifelse (count-turtles-at 2 0) > 0
              [setspeed speed-of one-of-turtles-at 2 0
              decelerate]
              [accelerate]]   ;else accelerate
              [accelerate]]
              if speed < 0.01 [setspeed 0.01]   ;also adjust speed
based on SpeedLimit and radar
              if speed > SpeedLimit [setspeed SpeedLimit]
              fd speed
              ]
              ]
              if (pc-ahead = 2 and (pc-at 0 0) = 2) [ifelse (random
2) = 0 [leap 3] [ifelse (random 2) = 0 [rt 90 fd 2][fd 1 lt 90 fd
2]]] check-patches-after-park
              if (pc-at 0 0 )= 2 [fd 1]
end


to accelerate
  setspeed (speed + (speedup / 1000))
end


to decelerate
  setspeed speed - (slowdown / 1000)
end
```