Development of storage and retrieval algorithms for automated parking systems

Chao Dou

The University of Toledo

Follow this and additional works at: http://utdr.utoledo.edu/theses-dissertations

Recommended Citation
Dou, Chao, "Development of storage and retrieval algorithms for automated parking systems" (2012). Theses and Dissertations. 302.
http://utdr.utoledo.edu/theses-dissertations/302

This Thesis is brought to you for free and open access by The University of Toledo Digital Repository. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of The University of Toledo Digital Repository. For more information, please see the repository’s About page.
A Thesis

entitled

Development of Storage and Retrieval Algorithms for Automated Parking Systems

by

Chao Dou

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the

Master of Science Degree in Engineering

________________________________________________________________________
Dr. Gursel Serpen, Committee Chair

________________________________________________________________________
Dr. Lawrence Thomas, Committee Member

________________________________________________________________________
Dr. Matthew Franchetti, Committee Member

________________________________________________________________________
Dr. Patricia R. Komuniecki, Dean
College of Graduate Studies

The University of Toledo

May 2012
An Abstract of
Development of Storage and Retrieval Algorithms for Automated Parking Systems

by

Chao Dou

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the Master of Science Degree in Engineering

The University of Toledo
May 2012

This thesis presents development of a complete suite of informed search algorithms to manage multiple concurrent requests, in real time and in a dynamic context, for storage and retrieval of robotic load-carrying carts for a fully-automated and driving-free parking lots or storage warehouses. A set of informed search algorithms including D* Lite and A* with domain-specific heuristics, and the uninformed search algorithm Uniform Cost Search are integrated for path search and planning in a completely-automated framework. The problem domain is considered as a rectangular array of parking or storage cells with several cells allocated to entry-exit points such as elevators: the storage topology does not have any driving lanes other than an allocation of blank cells where all storage is conceived to be on moveable carts. It is further assumed that the entire floor can be fully occupied with the exception of blank cells, which need to be leveraged to form temporary passageways for carts on the move for storage or retrieval. The number of blank cells is determined to maximize the storage or parking capacity and yet must be large enough to facilitate to serve the multiple and concurrent storage and retrieval requests in real time. Multiple carts are considered to be potentially moving in a layout where each moving cart...
will likely make a change to the environment by relocating carts in its way as it moves. Strategies for storage in the parking lot or the warehouse to facilitate a quick completion through following a path that is as close as possible to the optimal or shortest path are proposed.

A software simulator based on multi-threaded Java code was developed to perform empirical testing and validation of the performance of the proposed integration framework for the set of path search and storage strategy algorithms. A parking lot with 400 (20×20), 800 (20×40), 1200 (30×40), and 1600 (40×40) parking or storage spots was considered. A small percentage of parking spots were reserved as available blank cells to facilitate movement of robotic carts carrying the car to its storage or retrieval destination location. A typical business day scenario where morning rush hour that fills the parking lot to its maximum capacity at its conclusion and the evening rush hour that nearly empties the entire parking lot from a fully-occupied state was considered. Multiple concurrent and combination of storage and retrieval requests were generated. The performance effect of immobilized carts that form fixed obstacles on the parking floor was considered. The performance of the proposed system was assessed and evaluated using a number of performance metrics that included the actual path length, real-time response of the search and planning algorithms, the combined memory cost of the search processes, and the ability to serve multiple requests.

Simulation results indicate that the automated parking and retrieval system presented in this thesis is feasible and practical. The actual path lengths measured through the number
of movements per request is close to the computed shortest path length, which means the system provides a nearly optimal path for each request. The system provides a quick response during the path planning process even in the presence of tens of concurrent storage and retrieval requests and numerous immobilized carts to make it possible for deployment in real-time environments. The simulation study results further indicated that the developed system could handle over 100 concurrent requests with manageable process memory cost.

The simulation study indicates that the set of algorithms developed are suitable for fully-automated and robotic parking floors to serve tens of concurrent storage-retrieval requests in real time with manageable computing resources under real-life scenarios.
Acknowledgements

This thesis would not have been possible without the support from many people. I would like to acknowledge the guidance from my thesis advisor and committee chairman, Dr. Gursel Serpen. This thesis would not have completed without his suggestions. I will also thank my graduate committee members, Dr. Larry Thomas and Dr. Matthew Franchetti for all the encouragements.

I acknowledge all my classmates, especially my co-workers in the lab, Jiakai Li, Linqian Liu and Zhenning Gao for all the support to me. I also thank the College of Engineering, The University of Toledo, providing financial aid and all the research facilities.

I would like to thank my family members, especially my parents, Jianfeng Li and Jiande Dou for the financial support and encouragements theses years. Without their encouragements, I would not have finished the degree.
# Table of Contents

Abstract iii  
Acknowledgements vi  
Table of Contents vii  
List of Tables x  
List of Figures xi  
List of Abbreviations xiii  
List of Symbols xiv  

1 Introduction 1  
1.1 Motivation ................................................................. 1  
1.2 Current Parking Technology ............................................. 2  
1.3 Problem Statement .......................................................... 6  

2 Background 7  
2.1 Related Problem Domains ................................................... 7  
2.1.1 Robot Soccer ............................................................ 7  
2.1.2 Computer Games .......................................................... 12  
2.1.3 Warehouse Storage and Retrieval Systems ........................... 16  
2.2 Summary ........................................................................... 21  

3 Proposed Methodology 22  
3.1 Problem Specifications ....................................................... 22
Simulation Raw Data .............................................................. 91
UML Diagrams .............................................................................. 107
Pesudocode for Auxiliary Functions ............................................. 110
Simulation Code ......................................................................... 115
List of Tables

4.1 Percentages of Requests for a 24-hour Period in a Workday ......................... 54
A.1 Performance for 7% Blank Cells and 20×20 Topology ................................. 92
A.2 Performance for 13% Blank Cells and 20×20 Topology ............................... 93
A.3 Performance for 20% Blank Cells and 20×20 Topology ............................... 94
A.4 Performance for 7% Blank Cells and 20×40 Topology ................................. 95
A.5 Performance for 13% Blank Cells and 20×40 Topology ............................... 96
A.6 Performance for 20% Blank Cells and 20×40 Topology ............................... 97
A.7 Performance for 7% Blank Cells and 30×40 Topology ................................. 98
A.8 Performance for 13% Blank Cells and 30×40 Topology ............................... 99
A.9 Performance for 20% Blank Cells and 30×40 Topology ............................... 100
A.10 Performance for 7% Blank Cells and 40×40 Topology ............................... 101
A.11 Performance for 13% Blank Cells and 40×40 Topology ............................. 102
A.12 Performance for 20% Blank Cells and 40×40 Topology ............................. 103
A.13 Performance for High Concurrent Demand (13% Blank Cells and 40×40 Topology) .................................................................................................................. 104
A.14 Performance for 1% Immobilized Cells (13% Blank Cells and 40×40 Topology) .................................................................................................................. 105
A.15 Performance for High Concurrent Demand and 1% Immobilized Cells (13% Blank Cells and 40×40 Topology) ......................................................... 106
List of Figures

3-1 20×20 Parking Lot Layout for Automated Parking ............................................ 23
3-2 Pseudocode of D* Lite Path Search Algorithm for Parking Problem ............... 28
3-3 Unblocking Procedure Illustrated ....................................................................... 30
3-4 Pseudocode for Unblock Procedure ................................................................... 32
3-5 Pseudocode for findBlank() Function which
  Implements the UCS Algorithm ........................................................................... 34
3-6 Illustration of How UCS Locates Blank Cells .................................................... 35
3-7 Pseudocode for moveBlank() Function which
  Implements the A* Algorithm .......................................................................... 37
3-8 Flow Diagram for unblock() Procedure .............................................................. 39
3-9 Function Call Diagram for unblock() Procedure ............................................... 39
3-10 Storage Strategy Based on Zone Priority and Manhattan Distance Measure ... 41
3-11 Different Options for Distribution of Priority Zones ......................................... 44
4-1 An Example for the Interface of the Application (20×20 cell topology) ............ 48
4-2 Processing of Request through Threads in Simulator ......................................... 52
4-3 Control Flow for Storage Process ....................................................................... 55
4-4 Control Flow for Retrieval Process ................................................................... 56
4-5 Capacity Utilization Curve for 6AM-to-6PM Period .......................................... 59
4-6 20×40 Topology Description .............................................................................. 63
4-7 30×40 Topology Description .............................................................................. 63
4-8 40×40 Topology Description .............................................................................. 64
4-9 Manhattan Distances for Cells in First Quadrant of 10×10 Elevator Zone
   (Darkest Cells Indicate Elevators and Gray Cells Indicate Load-unload
   Cells) .................................................................................................................. 67
4-10 Optimal Path Length vs. Blank Cell Percentages in Four Topologies .......... 68
4-11 Actual Number of Movements vs. Blank Cell Percentages in Four
   Topologies ......................................................................................................... 70
4-12 Optimal Path vs. Actual Number of Movements for 7% Blank Cells ......... 71
4-13 Optimal Path vs. Actual Number of Movements for 13% Blank Cells ........ 72
4-14 Optimal Path vs. Actual Number of Movements for 20% Blank Cells ........ 73
4-15 Search Time vs. Blank Cell Percentages in Four Topologies ...................... 74
4-16 Memory Utilization vs. Blank Cell Percentages in Four Topologies .......... 76
4-17 Performance Profile for Increased Concurrent Demand vs. Lighter
   Concurrent Demand .......................................................................................... 78
4-18 Performance Profile with and without Immobilized Cells ......................... 80
4-19 Performance Profile for High Concurrency Demand and Immobilized Cells ...
   ............................................................................................................................... 81
A-1 UML Use Case Diagram ................................................................................... 107
A-2 UML Class Diagram .......................................................................................... 108
A-3 UML Sequence Diagram for Retrieval Request ............................................. 109
A-4 UML Sequence Diagram for Storage Request ............................................... 109
A-5 Pseudocode for Support Functions Being Used in D* Lite Implementation .... 112
A-6 Pseudocode for Other Functions Being Used in Unblock Procedure .......... 114
### List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS/RS</td>
<td>Automated Storage and Retrieval Systems</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JDK</td>
<td>Java(TM) Developer’s Kit</td>
</tr>
<tr>
<td>MAPS</td>
<td>Modular Automated Parking Systems</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>PGI</td>
<td>Parking Guidance and Information</td>
</tr>
<tr>
<td>PGS</td>
<td>Parking Guidance Systems</td>
</tr>
<tr>
<td>PRM</td>
<td>Probabilistic Roadmap</td>
</tr>
<tr>
<td>RRT</td>
<td>Rapidly-exploring Random Trees</td>
</tr>
<tr>
<td>RTA*</td>
<td>Real-Time A*</td>
</tr>
<tr>
<td>RTAA*</td>
<td>Real-Time Adaptive A*</td>
</tr>
<tr>
<td>UCS</td>
<td>Uniform Cost Search</td>
</tr>
<tr>
<td>VMS</td>
<td>Variable Message Sign</td>
</tr>
</tbody>
</table>
List of Symbols

\( b \) ....................................... Branch factor of a search tree
\( blanks \) ................................. A list of blank vertexes which are closest to the target vertex in unblock procedure
\( Closed \) .................................. A list of vertexes which have been evaluated in D* Lite and unblock procedure
\( d \) ......................................... Maximum depth of a search tree
\( destinations \) ............................ A list of vertexes which are next to the target vertex and have the smallest Manhattan distance to the D* Lite goal vertex
\( f \) ......................................... A value which \( f = g + h \)
\( fringe \) ..................................... A list of vertexes which are being expanded in the function \( findBlank() \) in unblock procedure using in unblock procedure
\( g \) ......................................... Actual cost from current node to the start in D* Lite and A*
\( h \) ......................................... An estimate cost from the current node to the goal in D* Lite and A*
\( m \) ......................................... Number of columns of a parking layout
\( N \) ......................................... Number of vertices in tree \( T \)
\( n \) ......................................... Certain node in RTA* algorithm or number of rows of a parking layout
\( O \) ......................................... Time and space complexity
\( Open \) .................................... A list of vertexes which are being evaluated in D* Lite and unblock procedure
\( q \) ......................................... Randomly picked point in RRT algorithm
\( R \) ......................................... A Roadmap in PRM algorithm
\( S \) ......................................... Number of generated milestones in PRM algorithm or a set of vertexes in D* Lite and unblocking procedure
\( s, s', s'', u', u'', s_1, s_2 \) ....  Certain vertexes using in D* Lite and unblock procedure. Some of them may have specified meaning in specific functions
\( s_{\text{blank}} \) .............................. The chosen blank vertex in unblock procedure
\( s_{\text{curr}} \) ............................... The current position of the target vertex in D* Lite and unblock procedure
\( s_{\text{desti}} \) ............................... The chosen destination position for \( s_{\text{blank}} \)
\( s_{\text{goal}} \) ................................. The goal vertex in D* Lite
\( s_{\text{start}} \) ............................... The start vertex in D* Lite
\( T \) ......................................... A random tree in RRT algorithm
\( x \) ......................................... Number of movements in RTA* algorithm, which is equal to \( d \) or a certain node in A* algorithm
Chapter 1

Introduction

1.1 Motivation

Parking is a challenging and high-cost endeavor in metropolitan areas. This high cost is associated with the search time and parking fees for desirable parking options. Finding an available parking spot might take a very long time leading to missed appointments, traffic violations, or even accidents. Pollution due to exhaust, congestion due to driving around for the search, and illegal parking due to lack of other options are also possibilities. An additional scenario is parking in one of very few available but very high cost spots.

Building multi-story parking lots has been one viable option for some time. However, a multistory parking environment is quite different from the flat ones. Usually, the driver may not know where the available parking spot is. He/she may need to circle around the floors until you see one. Sometimes the driver may waste a long time searching. Even though some of the parking lots have sensors, it is only helpful in a very limited context. Because this environment is quite dynamic at times, what is necessary, which is obtaining the parking availability information in real-time, is often not available.
Given that new space cannot be claimed in congested areas, one option is to create new spaces through re-claiming the space allocated as roads within a multistory parking structure. A fully automatic robotic system can save people in many ways, which may include time, fuel, and unnecessary driving among others. Additionally, a fully-automated system will reduce the pollution due to exhaust during the search-and-wait time. Costumers don’t need to park themselves. They can simply place their cars in the ground floor and leave, and then pick up their vehicles in the same place. This likely provides a more pleasant parking experience given that the parking lots or structures are polluted with exhaust, are noisy due to traffic, pose potential danger for accidents, and are typically unattractive spaces in general. As such, a fully automated robotic multistory parking system is an attractive option to explore a potential solution for the kind of parking problem most often encountered in highly congested major metropolitan areas.

1.2 Current Parking Technology

In the industry, the most widely used system is Parking Guidance Systems (PGS) [1]. PGS systems are designed to provide parking space information to the drivers where occupancy levels are low. Parking Guidance and Information (PGI) systems are based on gathering signals from sensors to supply the drivers with information for parking availability. The systems contain traffic monitoring service, communication service and variable-message sign (VMS) service. A VMS (or Changeable Message Signs) is an electronic traffic sign usually used on roadways to give drivers information about special events. VMS shows the traffic congestion, accidents, incidents, roadwork zones, or speed
limits on a specific highway segment [2].

Information on the available parking spaces is obtained from sensors by counting the number of cars entering and exiting or by comparing the tickets issued at machines or cash registers to the capacity of the system. This information is sent to the computer where all the processing takes place. Availability is usually signaled as "full" or "empty," but in some cases the exact number of spaces is given. However showing actual numbers may cause a problem because when the number is small, drivers tend not to enter because they think that all of the remaining spaces will be taken by cars already in the parking structure. This assumption may be erroneous and possibly cause a waste of available space.

Robotic Parking Systems' patented Modular Automated Parking System (MAPS) (http://www.roboticparking.com/downloads/brochures/robotic_parking_reprint_GE.pdf) integrates computerization with automatic lifts and pallets to store and retrieve vehicles in multi-level, modular garages (units are standard in size and design, but can be arranged together in different ways). After loading onto a pallet in the entrance floor, the vehicle will be transported to the upper floor and leave the lift. The pallet is then automatically transported through the garage to an available parking spot, which is assigned by the control system, and then unloads the car. The pallet moves on the designed tracks which are connected with every parking spot. For retrieval, the driver inserts his parking label into the reader sensor, signaling the system to retrieve his vehicle to the entrance floor. The retrieval process is the nearly the same as storage. Cars arrive in an average of two
minutes, facing forward for an easy exit from the park. In this system, aisles are still needed for moving the pallets up and down, left and right. However, a fully-automated parking system may not need those gangways: nearly an entire floor can be seen as parking space, and only some space would be reserved for facilitating access to the entire parking lot. Consequently, such a parking system would bring more parking capacity and land utilization, which are critical considerations for parking in metro areas.

Cecelia Camacho [3] patented another kind of automated parking system. It also contains elevators and platforms for parking. However, the cars are parked on a large floor just like the normal parking areas. A floor is separated into a large number of parking spots but without gangways. While requiring for storage, the driver will park the car in a waiting area and get a card. Then the system will determine an available parking spot for the car, and the elevator will take the car, which is carried by a transporter with wheels, up to the target floor. When the elevator stops, the transporter will take the car to the spot. While retrieving, the order of the operation is reversed. The system scans the driver’s card and moves the particular car into the elevator. Then the elevator takes the car down to the waiting area so that the driver can get his/her car. There will always be some blank spots on each floor. These blank spots will help when the car is blocked by other cars, for both the storing and retrieving processes. The blank spot will move next to the target car and then the target car can move one step towards the elevator or the parking spot. This process will repeat until the car reaches its destination. While this patented system is similar with the system which will be presented in this thesis, the patent does not, however, detail how the storage for parking or retrieval processes will be managed. More
specifically, the patent simply fails to present the multiple and concurrent path planning algorithms for dynamic environments which is a challenging research venue. We have developed algorithms to address all aspects of floor management for storage and retrieval and run simulations to demonstrate their feasibility and performance. We formulated a strategy for assigning the parking spots on a given floor for incoming cars. Path planning algorithms, such as A*, D* Lite and Uniform Cost Search are implemented and integrated within a unified framework to guide the cars from a starting point to their destination, both during the storage and retrieval processes. For the situations when there is no connected path for a car being stored or retrieved, we formulated a procedure to unblock the obstacles in the path. All of these algorithms were implemented for multiple concurrent requests and in a dynamic environment. The entire suite of algorithms was implemented in a multi-threaded context to validate the feasibility of the overall solution developed.

Parking Guidance and Information (PGI) systems, the most widely used system, is successful in reducing the time and cost of the parking problem. Pollution from exhaust and congestion due to entry or exit queues are reduced, but not completely addressed. Robotic parking systems currently in use offer a much better solution in the way of much more enhanced space utilization, drastic reduction in pollution and congestion, reduced traffic activity etc. However, significant space at a given floor is allocated for the lifts. It is desirable to improve the space utilization of current robotic parking systems so that more parking spaces for a given parking structure can be claimed.
1.3 Problem Statement

The research reported in this thesis aims to develop an intelligent suite of algorithms to manage storage and retrieval processes in a fully automated parking system on a single floor of a robotic parking structure that allocates the entire floor space for parking with the exception of elevators or access entryways and some minimal blank space. Specific goals are to manage multiple storage and retrieval requests concurrently in real time in a dynamic environment where multiple concurrent requests might cause obstructions for each other; develop accurate heuristics for real-time planning algorithms for near-optimal plans; and develop efficient storage, retrieval, and re-arrangement strategies.
Chapter 2

Background

2.1 Related Problem Domains

Storage and retrieval of multiple cars in real time in a robotic and fully automated parking lot have noteworthy similarities to a number of problems in other domains including robotic soccer and game playing. These problem domains along with search, path planning and miscellaneous management algorithms in use, are discussed next.

2.1.1 Robot Soccer

This problem domain entails teams of autonomous robots acting in adversarial, dynamic environments. In a robot soccer game, each team has five autonomously moved robots to play a competitive soccer game in a 5 ft by 9 ft playing field without any human input. Each robot, which is a soccer player, has one or more external cameras to provide a global view of the playing field. Robots aim to kick the ball into the opponents’ goal and defend their own goal. The robot will be three or four-wheeled omnidirectional mobile, with bounded acceleration and a maximum velocity [4, 5]. The environment is dynamic, with up to ten agents moving at a given time, and unpredictable; it is not known with
certainty how the opponent robots may move, and the motion of the ball may not be predicted precisely. Furthermore, there is a constant nontrivial probability of collision with other robots, which may require re-planning of the path of the colliding robots.

Although there might be different ways to fulfill the intelligent computing requirements of this problem, in one approach, each team has a central computer to manage the communication and perform the computation. The computer receives the imaging camera signal from each robot to locate all ten “players” and the ball on the field. Also, additional sensors measure the velocity of all ten robots and the ball. Next, the computer integrates this information, plans the path for each robot, and sends this information back to each robot for navigation. All this has to happen in real-time if real robots are being employed for the game.

Path planning is the main problem that must be addressed in real time. There are several constraints for the path planning algorithms in this domain: perhaps the most limiting one is the fact that not only should the algorithm be in real time, but it should also have the ability to re-plan, re-using parts of the existing plan in highly dynamic and unpredictable environments.

The A* family of algorithms are the most commonly used in the domain of mobile robot motion planning. The A* algorithm follows a path of the lowest known cost, keeping a sorted priority queue of alternate path segments along the way. In each iteration, a node which has the lowest value \( f(x) = g(x) + h(x) \), where \( g(x) \) is the cost from start node
to the current node $x$, and $h(x)$ indicates the estimated cost from current node to the goal) is expanded, and the successors are added to the priority queue. The algorithm terminates when the goal node is expanded [6, 7]. A* is a basic path planning algorithm which is used in static environments. D* is an extension of the A* which is adapted in dynamic environments. The D* algorithm builds on A* to create a variant which only recalculates portions of the problem where costs change, achieving significant speedup for domains where the environment changes slowly over time [8, 9].

To implement an A*/D* algorithm as a path planner for robot soccer, the play arena is represented as a grid-based environment. The more the number of grids employed, the more agile the robot is likely to be. However, based on the complexity of an A*/D* algorithm as the number of grids increases, the time and space cost will be increase exponentially. This has a direct adverse effect, making this algorithm non-suitable for large search spaces (e.g. too many grid lines) for real-time implementations. Accordingly, much recent work has centered on the idea of randomized sampling for approximation to the optimal path.

The Probabilistic Roadmap (PRM) algorithm is a multi-query planner. It operates in two phases. First, a roadmap, designated as $R$, is pre-computed by repeatedly sampling the configuration space at random. If a sample is collision-free, this sample is selected as a milestone. The process will repeat until $S$ milestones are generated. Each of two adjacent milestones is seen as a pair, and the algorithm connects pairs of milestones that are not
too far apart by simple paths and retains those which test collision-free as local paths. Then the milestones are connected as a network, which is called a roadmap [10-12].

In order to implement the algorithm as a planner of the robotic soccer problem, the following aspects are important. First, the position of the robot (as start state) and the position of the ball or opponent’s position (as goal state) must be added as milestones and then, several free-space points are picked randomly as other milestones. Second, to achieve real-time computation, the number of milestones must be kept at a minimum. Third, although the basic algorithm itself is not suitable for dynamic environments, it is possible to incorporate additional functionality to it to make it appropriate for dynamic environments. One such functionality is the collision detection, which will make it possible for the algorithm to re-plan a new path when a collision is detected.

The performance of the algorithm depends on the number of generated milestones $S$. The space complexity is directly based on it as $O(S)$. In order to generate $S$ milestones, the time cost is $O(S)$. To determine which two milestones can be seen as pairs, the algorithm should compare the distances between each two milestones to see if they are close enough. Therefore, the time complexity should be

$$C_S^2 = \frac{S!}{2 \times (S-2)!} = \frac{S \times (S-1) \times (S-2) \times S \times \ldots \times 2 \times 1}{2 \times (S-2) \times (S-3) \times \ldots \times 2 \times 1} = \frac{S \times (S-1)}{2} = O(S^2).$$

The total time complexity is $O(s) + O(s^2) = O(s^2)$. The real time cost is uncertain because it can be affected by other aspects such as the number of obstacles and the number of local paths.
Rapidly-exploring Random Trees (RRT) algorithm is a probabilistic complete, non-optimal, kinodynamic path planner used to explore large state spaces [13]. This algorithm is widely used because it is efficient in finding a path in large areas, or high dimensional spaces. The steps are as follows: first, pick an initial tree $T$ which only contains a single point, and then iterate: picking a random point $q$ uniformly in the free space, looking for the nearest point from $q$ in $T$ with a particular heuristic function and extending the point towards $q$. The extension operation is to connect a point in the existing tree $T$ to the point $q$ with a series of points and add all of the new points into $T$. If a point, which is sufficiently close to the goal point, is expended, the algorithm will terminate.

The RRT algorithm can be converted into a planner for the robot soccer problem as follows. In order to avoid hitting obstacles in the environment, the extending operation should check if the generated point is collision free. If the generated point is in the area of an obstacle, this point should be abandoned and another point should be regenerated. Second, a heuristic determines which point in the tree $T$ is the nearest one towards $q$. Euclidean distance is being used as the heuristic function [14].

Assume that $N$ is the number of vertices in the tree. For the time cost, we divided the whole process into four parts. First, pick $N$ random samples, for which the time cost is $O(N)$. Second, in order to find the nearest node already in the tree, we should compare each node’s distance, which is $\sum_{i=1}^{N} (i-1)\times T_{dist} \approx O(N^2)$. Third, expansion time, which is not relevant to the nodes already in the tree, and may cost more time by collision check,
can also be seen as $O(N)$. Finally, time complexity of adding the nodes into the tree is obviously $O(N)$. Then the total time cost is $O(N) + O(N^2) + O(N) + O(N) \approx O(N^2)$.

2.1.2 Computer Games

Computer or video games provide computer-controlled opponents. Incorporating human-like behaviors to the in-game “robots” is the major challenge and most readily addressed by the artificial intelligence field through planning, decision-making, prediction and adaptation. Regardless of the game (e.g. first-person-shooter, racing, real-time strategy, etc.), there is a need for computer opponents to perform autonomous movements and navigation. Accordingly, path planning, in most cases for multi robots or computer opponents, obviously is one of the focus areas to respond to this need. Certainly, real-time and dynamic environments impose the most significant constraints because path planning problem becomes very challenging.

Originally, the A* search algorithm was used in finding paths for computer games, since A* will always find the minimum cost path from a start point to an end. However, basic A* algorithm has a number of limitations among which is the computational complexity, which precludes its employment as is without any enhancements or improvements particularly in a real-time context. Modern games, in general, always have large game maps. A larger map means a larger number of nodes in the search tree (or higher complexity of the algorithm). This directly affects the performance of the algorithm, and
real-time constraints might be violated, although some modifications to the basic A* algorithm may be able to make it relevant for certain types of real-time environments.

The Real-time A* (RTA*) algorithm is widely used in real-time strategy games, because of its efficiency and dynamic planning property [15]. To implement A* search in dynamic environment, we have to perform a loop. In each step of the loop, the robot makes one movement, and the new position of the robot should be updated as the new start for the next step. Simply repeating minimum search for each move ignores information from previous searches and results in infinite loops. In addition, since actions are committed based on limited information often the best move, may be due to undo the previous move. For each move, the $f(n) = g(n) + h(n)$ value of each neighbor of the current state is computed where is now the cost of the edge from the current state to the neighbor, instead of from the initial state. The problem solver moves to the neighbor with the minimum $f(n)$ value, and stores with the previous state the best $f(n)$ value among the remaining neighbors. This represents the $h(n)$ value of the previous state from the perspective of the new current state. This is repeated until a goal is reached. To determine the $h(n)$ value of a previously visited state, the stored value is used, whereas for a new state the heuristic evaluator is called. Note that the heuristic evaluator may employ a minimum look-ahead search with branch-and-bound as well [15].

Real-Time Adaptive A* is another version of Real-Time A* search algorithm. If one needs to perform several A* processes with the same goal but different starts, the algorithm will make the heuristic more informed to speed up the future steps of A* searches. It updates the heuristic function with an intermediate value, which is relative to
the g value of a state to be expanded and the estimated goal distance from the current start to the goal. It uses the information from each step of A* search and makes the heuristic better for future searches [16].

Assume \( b \) as the branch factor of a search tree, and \( d \) is the maximum depth of the search tree. For the worst case, the goal state is the last one being generated, the total number of generated nodes is: \( 1 + b + b^2 + \ldots + b^d = O(b^d) \), so the time and space complexity are all \( O(b^d) \).

RTA* and RTAA* are the algorithms developed from original A*. As in RTA* and RTAA*, the A* algorithm is being implemented \( x \) times \((x \) is the total number of movements from start state to goal state which is equal to \( d \)) Therefore, the time complexity for RTA* and RTAA* is \( O(x \times b^d) = O(b^{d+1}) \) and space complexity will be still \( O(b^d) \) because the memory will be cleared after each A* operation.

According to the time complexity, A* and other derived algorithms such as RTA* and RTAA* can be real-time and optimal while the environment is not large. If the environment is large enough, the cost will be extremely expansive because the complexity increases exponentially.

The original A* algorithm is not efficient in a dynamic environment. If an obstacle moves onto a path that was identified by A*, the plan will fail because A* lacks the facilities for replanning needed in those kinds of circumstances. RTA* and RTAA* are adapted for
dynamic environments because they replan the path after each movement to avoid moving obstacles.

The path planning process in the automated parking system is similar to that of robot soccer and computer games because they all need path planning algorithms to take an autonomous or controlled mobile cart or vehicle from a start location to a goal location. However, robot soccer and computer games have some differences with automated parking. In robot soccer and computer games, the robots can be anywhere on the playground. In other words, there are not fixed start and goal points for the robots. In automated storage and retrieval systems, the parking cars tend to be located in specific locations unless and until their locations are minimally altered as a side effect of a nearly path for vehicle being stored or retrieved. Note that which robots are moving continuously, so that the path planning would have to be redone on a continuous basis throughout the entire game for robot soccer and computer games. However, in an automated parking system, the path will not be altered dramatically, with the exception of where there might obstacles on the already planned path due to other moving vehicles if concurrent requests are being served in the environment. Otherwise, the path planning will need to be performed only twice, once for storage and another one for retrieval, with minor readjustments if the planned path is disturbed by other moving carts.
2.1.3 Warehouse Storage and Retrieval Systems

The automated storage and retrieval systems (AS/RS) are widely used in warehousing and other automated systems. The main advantages of the usage of AS/RS are savings in time and labor costs, reducing space, increasing reliability and so on [17]. To design an AS/RS system, there are many issues to be considered including physical issues and control issues. The AS/RS system is a storage and retrieval system that uses fixed paths and moving machines running on them with automation [17]. The components of most AS/RS systems are racks, cranes, aisles and input-output (I/O) points. Racks are the locations used for accommodating goods; cranes are the automated moving machines that carrying the loads; aisles are the spaces that cranes can move through; and I/O-points are the locations where the incoming goods are picked up as well as where retrieved goods are dropped off.

For a particular design problem, the total capacity of the warehouse is given. Therefore, the physical decisions such as number and locations of I/O points, number of aisles, height and length of the racks, and lengths of aisles are deduced based on the capacity and the total space of the warehouse. However, even other aspects are given, the length and height of the racks need to be well designed. This is because the cranes can move both vertically and horizontally, and vertical movements are harder compared to horizontal movements (move slower or spend more power). A good balance between height and length helps the system become more efficient. For each AS/RS system, there
needs to be more than one I/O point. The smart design of their locations and number helps the storage and retrieval process easily and quickly.

Control issues decide what activities the AS/RS should make. The issues contain storage assignment, batching, dwell-point, and sequencing. A storage assignment is used for determining the location in which the incoming load should be stored. Typically, there are five methods [17, 18]: dedicated storage assignment, random storage assignment, closest open location storage assignment, full-turnover-based storage assignment and class-based storage assignment. For the dedicated storage assignment, every load type has its fixed location in the warehouse. The main disadvantage of this type of assignment is that the space requirement is high but the space utilization is low. Even it has advantages, such as well managed storage layout, it is too similar to the non-automated systems. Random storage assignment means that all of the empty locations have the same opportunity to be utilized for storage. It is similar to the closest open location storage assignment, which uses the first encountered available location to store. They are similar because the retrieval process is not concerned as a factor while they are making storage decision. The main disadvantage for both of random storage assignment and closest open location storage assignment may make retrieval process more complex. The full-turnover-based storage assignment policy determines the storage locations for goods based on their demand frequencies. The high turnover frequency products are stored near the I/O points and low turnover frequency ones are located further. Clearly, these frequencies would have to be known in advance. However, the main problem with this assignment method is that the turnover frequencies may change constantly, as well as the
types of products. If frequency changes or new products come into the warehouse, a large amount of products needs to change locations based on the new frequency list. To avoid large number of repositioning of goods and lower space requirement, the *class-based assignment* can be implemented. It combines the *full-turnover-based* and *random* storage assignments into one. The whole warehouse is divided into several areas (zones). Each product is assigned to a zone based on its demand frequency. The *random storage assignment* determines where to locate the good within a zone. This method increases the efficiency of the system and lowers the space requirement. There are three things to be concerned before performing *class-based assignment*: the number of classes (zones), number of products in each zone, and locations of the zones. It was determined through simulation studies that the performance of the *full-turnover-based* and *class-based* assignments is far better than either the *random* or the *closest open location* assignments [17, 18].

The batching approach is to execute multiple orders within one tour of a crane. The main benefit of batching is reducing the time for each order. However, the number of a batch is limited by the capacity of a single crane and the respond time. Therefore, how to perform the batching process to execute orders as many as possible and minimize the time is a concern. First, an order is selected as the first order (seed) of a batch. There are several heuristics selecting the seed, for instance, the order with largest/smallest number of locations to be visited and the order with largest/smallest volume [17,19], and the highest percentage of capacity of crane [17,20], etc. Then, allocate other orders to the batch using order addition rule, which uses largest number of common locations [17,21] or geometric
similarities [17,22] as heuristic. Finally, check if the batch is complete. The batch is done while the capacity is reached, respond time limitation is reached, or all orders are complete. If the batch is complete, go to the first step to restart a new batch, else go to the second step to allocate other orders.

Dwell-points are the locations where the empty cranes are located. Good locations of the empty cranes help to reduce the good’s travel time. There are 4 methods to locate [17,23]. The first method is to always put them at the input points, or always put them at the midpoint of the rack. The second method suggests that if a storage request is completed, put the crane at the input point. The third method proposes the following: if a retrieval request is completed, then put it at the output position, or if a storage request is completed, then place the crane at the last storage position. According to the fourth method, if a retrieval request is complete, then it should be positioned at the output point.

Sequencing manages the order of requests being executed. The storage requests are not time-critical, so usually, the storage requests are based on the first-come-first-serve principle. The sequence of retrieval requests is far more complex because the list of the retrieval requests is changing continually. There are two ways [17,24]. First, select a block of most urgent requests (either retrieval or storage), after sequencing them, then select the next block of requests from the new list. It is called block sequencing. The other way is called dynamic sequencing. The list of requests will be sequenced every time when a new request is added. We can sequence the list based on a first-come-first-serve
rule. Also, we can serve the retrieval request based on their estimated completion time, the one with the shortest completion time will be served first.

The performance measurement evaluates the design of an AS/RS. The following aspects are the main performance measurements [17]:

- Time cost for each request;
- Number of requests handled in a period of time;
- Total time spend for a particular numbers of requests;
- Waiting times of goods to be storage/retrieval;
- Number of requests waiting to be executed.

The warehouse storage and retrieval system has some common attributes with automated parking and retrieving systems. The AS/RS utilizes lanes or passageways for transporting the goods or merchandise which is similar to how some automated parking systems operate. The path planning process will only be implemented when there is a storage or retrieval request for both systems. Both systems utilize fixed locations to deposit goods or cars, and I/O points, such as entrances and exits in AS/RS and elevators in automated parking systems. In both systems, storage assignment needs to be completed before the storage process starts. There are also noteworthy differences. In fully automated parking systems, there will be no real aisles. The entire parking lot is allocated as parking spots with the exception of several empty spaces to be used for creating passageways for moving robotic carts. As a consequence, the path planning algorithms are very complex and needs to deal with multiple concurrent requests in highly dynamic settings.
2.2 Summary

This chapter presented a survey of autonomous robotic systems in use in industry for parking and warehouse storage. In all of the systems surveyed, a significant percentage of the parking or storage space is allocated to pathways or driving lanes. There is currently no fully-automated parking or warehouse storage system deployed or being built, where the parking or warehouse floor is entirely dedicated to storing cars or goods without any driving or access lanes. It was found that there is a US patent that describes a fully automated parking structure that does not utilize any type of driving lanes and which allocates possibly more than 90% of the space for parking. However the same patent simply describes the structure, topology, and the operation aspects of such a system, and does not detail any of the search, path planning, space management algorithms, and others.
Chapter 3

Proposed Methodology

3.1 Problem Specifications

Automation of storage and retrieval processes and strategies for a single-story robotic parking structure will be accomplished. The automated parking system is expected to respond to and satisfy multiple storage and retrieval requests in a timely fashion in a real time context for a realistic parking garage scenario as typically found major metro areas with high population counts and congested city layout.

The parking space is considered as a rectangular area as shown in Figure 3-1. There will be no roadways. There will be multiple entry-exit cells onto the parking space which will be designated as elevator spots. Each elevator is twice as larger as a parking space (presented as gray cells with “E” in Figure 3-1). Vehicles will be mounted onto robotic and mobile pallets which will carry the vehicle to its destination which may be either an available and open parking spot or the loading-unloading spot (presented as cells with “L”) adjacent to an elevator. The number of elevators spots will be correlated with the total number of parking cells or spots on the floor: it is reasonable to assume one elevator
for every 50 parking spots. Each elevator spot will have an adjacent spot that will be always unoccupied except to load or unload the elevator.

A certain and small percentage of the cells should be kept empty for the processing of storage and retrieval requests. The goal is to keep the number of blank cell count as small as possible without compromising the real-time response of the system. The overall management algorithm should be able to use these empty cells to complete possibly multiple and concurrent storage or retrieval requests through an optimal path in real time. Definition of real time is subject how long a typical customer in a given locale is expected to wait before he or she considers the fulfillment of his/her request as late.
Normally there will not be too many pending requests at any given moment, except some busy periods like:

- Morning rush hour: the layout will be empty at the beginning of this period, before large number of storage requests will come in with a small number of retrieval requests, and the layout will be filled near the full capacity at the end of this time period.
- Evening rush hour: the layout will be near full at the beginning of this period, before large number of retrieval requests will come in with few concurrent storage requests, and the layout will be near empty at the conclusion of this time period.

3.2 Thesis Research and Study Scope

The scope of this thesis is limited to design, development, and testing through simulation of a suite of algorithms to address mainly storage and retrieval processes. Algorithms related to background tasks such as relocating parked cars or unoccupied spots or to continuously adjust the state of the parking floor to improve response times of storage and retrieval processes are not considered.

The design, layout and organization of the parking structure for the study is limited to one floor in the shape of rectangle, has one size storage or parking cells, and parking floor has centrally-located elevators at a ratio of one elevator per approximately 50 parking spots or cells.
The set of algorithms that address storage and retrieval planning, storage strategies, and background tasks are presented in the next sections.

3.3 Search Algorithms for Path Planning

The problem being addressed is akin to a concurrent planning of multiple paths with different start and target states in dynamic environments. Accordingly, it is not possible to preplan the entire path in advance and execute in a batch framework. It is possible that the side effects of relocating one car (towards either its designated parking spot or an elevator) will cause the topology of obstacles to change on the fly. Therefore, the path planning algorithm needs to be able to continuously re-plan in an incremental fashion while preserving a large portion of the original plan from previous re-planning attempt to the next.

The car parking management system explores a number of different search algorithms based on the phases during the parking or retrieving process. These phases and associated search algorithms are as follows:

- path planning with the D* Lite algorithm [8-9], and
- unblocking or removing the obstacles in the path of the car being relocated through uniform cost search (UCS) [6] and A* search [6] algorithms.
3.3.1 D* Lite Path Search Algorithm

D* Lite [8-9] is an incremental search algorithm for dynamic environments. It is used as the basic path planning algorithm for both the car storage and retrieval. The problem domain is a two-dimensional area partitioned into rectangular subareas by gridlines. Each subarea is considered as a parking spot or cell and the entire parking area may be represented as a graph where subareas correspond to vertices and the neighborhood relationship between any two subareas define the existence of a non-directional edge between the two corresponding vertices. Accordingly, the terminology used for the search algorithms in this manuscript is given as follows:

- $S$ is the set of all vertexes;
- $s$, $u$, $s'$ are certain vertexes in $S$;
- $s_{\text{start}}$ is the start vertex of the problem;
- $s_{\text{goal}}$ is the goal vertex of the problem;
- $s_{\text{curr}}$ is the vertex indicating the current location, and it is a global value being used in the entire D* Lite search process;
- $c(s,s')$ is the cost of moving from $s$ to $s'$;
- $g(s)$ is the cost from $s_{\text{goal}}$ to $s$;
- $rhs$ is a value where

$$ rhs(s) = \begin{cases} 0, & \text{if } s = s_{\text{goal}} \\ \min_{s' \in \text{pred}(s)} [c(s, s') + g(s')], & \text{otherwise} \end{cases} $$
- $h(s,s')$ is the estimate cost from vertex $s$ to $s'$. It needs to satisfy
  
  $$h(s_{\text{start}},s_{\text{start}})=0 \text{ and } h(s_{\text{start}},s') \leq h(s_{\text{start}},s') + c(s,s')$$

- $\text{pred}(s)$ return a list with all the predecessors of $s$;

- $\text{Open}$ is a priority list of vertices being evaluated with the value $key$, and

- The priority of vertex $s$ is $key(s)=[k_1(s),k_2(s)]$, which has two components

  where $k_1 = \min(g(s),rhs(s))+h(s_{\text{start}},s)$ and $k_2 = \min(g(s),rhs(s))$.

The function of D* Lite, presented in Figure 3-2, first calls $\text{init}()$ to initialize the $g$ and $rhs$ values for all the vertices and set the current vertex as the start vertex. Then, it computes the shortest path from the goal vertex towards the current vertex in the reverse direction. Afterwards, the function will perform a loop while the current vertex has not reached the goal. If it fails to find a path, then it will call the unblock procedure to find a way out. If a path can be found, then it will determine the new current vertex as follows: choose the successor state as the new current state if its “cost-so-far” value is the least. After selecting the new current vertex, it will update the position of the mobile (robotic platform) from the previous current to the new current position. The $\text{computePath}()$ function which computes the shortest path will be called again with the new current state and the goal state. The loop will continue until the goal is reached. The pseudocode for D* Lite implementation for the parking problem is presented in Figure 3-2.
Function *computePath()* is the most important function within the entire D* Lite algorithm. This function computes the shortest path from the goal vertex towards the current vertex. It expands locally inconsistent vertices (for which \( g \neq \text{rhs} \)) according to their priorities. If the vertex \( s \) is over-consistent (\( g(s) > \text{rhs}(s) \)), it holds the \( \text{rhs} \) value as the \( g \) value, and calls the function *update()* for all the predecessors of \( s \); if the vertex \( s \) is under-consistent (\( g(s) < \text{rhs}(s) \)), then it sets \( g(s) = \infty \), and calls *update()* for all the predecessors of \( s \) and \( s \) itself. This function will be terminated as the current vertex is locally consistent or the priority list is empty. If \( g(s_{\text{curr}}) = \infty \) after the search is terminated, it means the algorithm failed to find a path from the current vertex to the goal vertex. The pseduocode for *ComputePath()* is given in Figure A-5 in the Appendix.

Function *update(u)* is used to update the \( g \) and \( \text{rhs} \) value of \( u \) after each visit while function *calculateKey(s)* is used to calculate the key value for each vertex \( s \). Function *init()* is used to initialize the search problem by setting the \( g \) values and \( \text{rhs} \) values as \( \infty \).
for each vertex and the $g$ value of the goal vertex as 0. Also, the priority list is cleared in this function. Pseudocodes are presented in Figure A-5 in the Appendix.

3.3.2 Unblock Procedure – UCS and A* Search Algorithms

It is conceivable that the car being relocated might end up being surrounded by other parked or immobilized cars during the parking or retrieval process. In such a case, it is necessary to identify the closest empty spot or cell and move it next to the car being relocated in such a way as to facilitate advancement of the car being relocated towards to goal (or target) spot, which could be a parking cell or the loading-unloading cell next to the closest elevator. An unblock procedure is implemented when the car being stored or retrieved is surrounded by other parked cars on all sides; one or more sides can also be boundaries of the parking lot. The main idea of the procedure is to move a blank cell next to the blocked cell where the car currently is, and use the blank cell to help the blocked cell out. Figure 3-3 shows how the procedure works for the case of retrieval. Referring to Figure 3-3(a), the cell with “s” is the car being relocated; the cell with ”E” is the destination elevator for the car; the cells marked with ”o” are the occupied cells.
The unblocking procedure contains five steps as follows and presented in Figure 3-3, where one quadrant in the parking floor layout is considered and a row and column numbering scheme based on positive integers with the top left cell being at coordinates of (1,1) for (row, column) format will be used for ease of referencing in the following discussion:

1. Find the blank cell(s) that are closest as measured by their Manhattan distances to the current location of the car being relocated, which is at position (3, 2), using the uniform cost search (UCS) algorithm: closest blank cells are illustrated with “gray” color in Figure 3-3(b) and their positions are (1, 1) and (2, 4). These locations of the blank cells identified for relocation are to be as close as (using
Manhattan distance measure) to the position of the car being relocated (which is ensured by the UCS), and also closest (again using the Manhattan distance measure) to the destination elevator.

2. Determine the desired relocation position(s) for the blank cell(s) as marked by a “d” in Figure 3-3(c); the relocation positions of desired blank cells are (4, 2) and (3, 3). The desired relocation locations must be a) adjacent to the car being relocated; b) on the shortest path from the car being relocated to the destination elevator where the blank cell relocation Manhattan distance is equal to one less the Manhattan distance of the car being relocated.

3. If the desired location belongs to an immobilized cart, then there are two cases to consider as follows. In one case there are one or more desired relocation cells on an alternate path for the car being relocated with the same Manhattan distance value as the path on which the immobilized cell is currently located. If not, then search for a new relocation location that has a Manhattan distance to the destination elevator that is longer by one cell. If still not successful, keep searching for relocation locations (4 in total since it must be adjacent to the position of the car being relocated) while first considering those locations with the smallest Manhattan distances to the destination elevator.

4. If there are multiple desired destinations and(or) multiple blank cells found, just like the situation in Figures 3-3(b) and 3-3(c), select a suitable pair of blank and destination cells among all the combinations of blank and destination cells, with the smallest Manhattan distance from a blank cell to a relocation destination. As in Figure 3-3(d), the blank cell identified is at position (2, 4) and the destination
(which must be on the shortest path from the car being relocated to the destination
elevator as measured by its Manhattan distance) is determined to be (3, 3).

5. Plan the path for the blank cell guiding it to the desired position next to the
blocked cell using the A* search algorithm, which is illustrated in Figure 3-3(e);
the blank cell relocates from position (2, 4) to position (3, 3).

6. Move the car at position (3, 2) to the blank cell at position (3, 3) which was just
relocated next to it as shown in Figure 3-3(f), and return the new position of the
(blocked) car.

These steps for the unblock procedure are implemented in pseudocode as presented in
Figure 3-4.

```
function unblock(s1, s2)
    init();
    findBlank(s1);
    findDestination(s1)
    match();
    moveBlank(sblank, sdestination);
    move s1 to sdest;
    return sdest;
```

Figure 3-4. Pseudocode for Unblock Procedure

The uniform cost search (UCS) algorithm, which uses the Manhattan distance as the cost
function value, is implemented in the function findBlank(s). This function is tasked to
find the blank cells, which are the closest to the cell (vertex) marked by s. The main
construct in this function is implemented through a while loop. At first, the Open list only
contains the cell $s$. The loop will continue as long as no blank cell has been detected. While in the loop, it expands all the cells in the Open list, and places the successors in the fringe if they are already not in the Closed. Once all cells in the Open list are expanded, the Open list is cleared. Next all the blank cells or vertices in the fringe are added to the blanks list and replace the Open list with all the elements in fringe. There is also a Closed list to record every visited vertex: the idea is to avoid revisiting a previously visited cell. While a cell is expanded to generate its successors, it will be added to the Closed list if it is not already in Closed. The terminology and the set of invoked functions used for the UCS algorithm are given as follows:

- $s_1$ is the start vertex;  
- $s'$ is an element in Open list;  
- $s''$ is a successor of $s'$;  
- $u'$ is an element in fringe list;  
- Open is a list of vertexes which are being expanded;  
- Closed is a list of vertexes which have already been expanded;  
- fringe is a list of vertexes which are the being evaluated;  
- blanks is a list of blank vertexes that are the closest to the start state;  
- list.add(s) adds state $s$ into the list;  
- list.isEmpty() returns true if the list is empty, else returns false;  
- state.isBlank() returns true if the state is a blank one, else returns false;

The pseudocode for the function findBlank() is presented in Figure 3-5.
Figure 3-5. Pseudocode for `findBlank()` Function which Implements the UCS Algorithm

```
function findBlank(s₁)
    Open.add(s₁);
    while (blanks.isEmpty())
        for all s' ∈ Open
            for all s'' ∈ succ(s')
                if (s'' ∉ Closed)
                    fringe.add(s'');
                    Closed.add(s'');
        Open.clear();
        for all u' ∈ fringe
            if (u'.isBlank()) blanks.add(u');
            Open.add(u');
```

Figure 3-6 shows how the UCS as implemented within the `findBlank()` function works. The cell with “s” is the current position of the car being relocated and the cells with “o” indicate those cells that are occupied as shown in Figure 3-6(a). During the first iteration of the UCS loop, the highlighted cells are generated as the successors and are checked if any of them is blank as illustrated in Figure 3-6(b). Because no blank cell is detected, the search process will repeat and therefore redo the loop. Following the third iteration of the loop, see Figures 3-6(c) and 3-6(d), three blank cells are located at positions (5, 1), (6, 2), and (2, 4). They are added into the list `blanks`, and the function `findBlank()` exits.
The function \textit{moveBlank()}, which is being called in the \textit{unblock()} function, attempts to move the selected blank cell towards the selected or desired position next to the current cell which has the car being relocated. A* search algorithm is used to identify the path of movement. This process will build an optimal path from the location of the blank spot to its destination. In A* algorithm, here is an important value $f(s) = g(s) + h(s)$. The value $g(s)$ is the actual cost from the start state to the state $s$. The value $h(s)$ is the heuristic function, which estimates the cost from the state $s$ to the goal state. The Manhattan distance will be used as heuristic function in this process. At first, the blank spot $s_{blank}$ is added into an \textit{Open} list. Then the program will enter a loop until a path is found or the \textit{Open} list is empty. In each iteration of the loop, first it will find the node $s'$ with the
lowest $f$ value among the cells in the $Open$ list. If $s'$ is not equal to $s_{dest}$, $s'$ will be removed from $Open$ and added into $Closed$, if it is not already in the $Closed$. After that, $s'$ will be expanded and all its successors will be added to $Open$ except the ones which are already in $Open$. If $s'$ is equal to $s_{dest}$, a path will be built and the blank cell will move along the path. If there is(are) another blank cell(s) contained within the path (caused by other storage or retrieval requests), when the current blank cell hits another blank cell, then the second one will be the current moving blank cell and keep moving along the path.

The function $buildPath(s_1, s_2)$ is used to construct the actual path searched by A*. A back pointer is used for tracking the path. The destination node $s_2$ is first added into a list called $path$. Then a loop is started. For each iteration, the predecessor of $s_2$ will be added into $path$ and $s_2$ will be replaced by its predecessor. The loop ends when $s_1$ and $s_2$ is the same cell and an optimal path is constructed. The pseudocode for the $moveBlank()$ function that implements that A* search algorithm is shown in Figure 3-7.
Other functions being called in the main function are defined as follows (see pseudocode in Figure A-6 in appendix for detail):

- Function `init()` is to initialize all the lists being used in the unblock procedure;

- Function `findDestination(s)` looks for cells which are the successors of the cell `s` and have the least cost towards the goal cell;

- In the function `match()`, a pair of blank cells (among the results of `findBlank()` and destination cell(among the results of `findDestination()`) are being selected as the start and end states for the function `moveBlank()`, with the smallest value of the following heuristic function: the Manhattan distance from the blank

Figure 3-7. Pseudocode for `moveBlank()` Function which Implements the A* Algorithm
blank cell to the destination cell plus the Manhattan distance from the
destination cell to the goal cell.

The terminology and invoked functions used in moveBlank() and buildPath() are given as follows:

- \( s \) is an element in Open list;
- \( s' \) is the element in Open list with the smallest \( f \) value;
- \( s'' \) is one of the successors of \( s' \);
- \( s_1 \) is the start vertex for buildPath();
- \( s_2 \) is the goal vertex for buildPath();
- Open is a list of vertexes which are being expanded;
- Closed is a list of vertexes which have already been expanded;
- \( s_{\text{blank}} \) is the current position of the chosen blank state and \( s_{\text{desti}} \) is the chosen
destination for the blank;
- \( \text{list.add}(s) \) adds state \( s \) into the list;
- \( \text{list.isEmpty}() \) returns true if the list is empty, else returns false;
- \( \text{state.isBlank}() \) returns true if the state is a blank one, else returns false;

The flow and function call diagrams for the unblock() procedure are given in Figures 3-8 and 3-9.
Figure 3-8. Flow Diagram for `unblock()` Procedure

Figure 3-9. Function Call Diagram for `unblock()` Procedure
3.4 Storage Strategy

The storage strategy is used for deciding where the incoming car should be parked. The main idea behind the strategy is to minimize the obstruction for subsequent storage and retrieval actions. The strategy is best explained through an illustrated example which is presented in Figures 3-10 and 3-11. The entire parking lot is partitioned into zones: each elevator is associated with a particular zone surrounding it. It is assumed that another algorithm (which is not considered within the scope of the study entailed by this thesis) monitors the available capacity of each zone and identifies the zone for storing a car based on this available capacity evaluation. Once the car is unloaded from a given elevator the storage strategy presented herein decides where exactly in that zone this car should be stored.

In Figure 3-10, the cells with “E” are the elevators and the ones with “L” are the loading and unloading areas (cells) for each elevator. The cells are assigned one of two priority levels: the gray areas have the higher priority and the white ones have the lower priority as shown in Figures 3-10(b) and (c). The incoming storage requests will first be assigned to the higher priority cells and then the lower ones.

Additionally, each cell is assigned a cost value as measured by its Manhattan distance from the closest elevator in its zone. To minimize the number of obstructions, it is necessary to first store the cars as far as possible from the elevator because farther locations will not block future incoming cars, especially when the higher priority spots
are all claimed during the rush hour when storage requests are at a peak. Therefore, the storage strategy will return the location of a cell with the highest priority and the largest Manhattan distance (highest cost) as the destination location for a car waiting in the load-unload cell next to an elevator.

Initially the entire floor is empty. As cars arrive for parking, they are first stored in a specific cell in the gray zone based on the cell’s priority and cost so that the parked cars will not block the path of future incoming cars as shown in Figure 3-10(b). After the gray zone is full, if the cars still keep arriving for parking, they will have to be parked in the white zone as shown in Figure 3-10(c) which might block the shortest paths for any subsequent retrieval requests. Also, the sequence of parking in the same priority zone is a
descending order based on the Manhattan distances from the loading and unloading cell (presented as the numbers in the cells in both Figures 3-10(b) and 3-10(c)). If multiple cells have the same Manhattan distance value, then the selection is random.

The storage strategy can be generalized as follows. The idea is to designate columns or rows of cells for storage while other columns or rows as passageways so that any storage and retrieval request can be fulfilled without encountering any obstructions. Noting that the distribution of the high priority cells and the low priority cells could be different with the distribution shown in Figure 3-10. However, the distribution must be performed with the same policy. Typically each passageway column or row is surrounded by two storage columns or rows with one on each side.

The main expectation from the storage strategy is to be able to store the maximum number of cars starting from any given state without encountering any obstacles in the form of already-parked cars. Accordingly, the definition of priority zones needs to be carefully accomplished for a given topology. To illustrate the factors at play in this process, Figure 3-11 presents two specific cases where one of the two zone designation lead to a better arrangement for the metric of concern. The cell with “L” at location (5, 3) presents a start cell for storage and destination cell for retrieval. The gray cells are the higher priority cells. If the higher priority cells are spread as in Figure 3-11 (a), the two cells at the top corners at positions (1, 1) and (1, 6) will be blocked for retrieval after all of the gray cells are occupied. Therefore, this distribution is suboptimal. As in Figure 3-11(b), after all the gray cells are occupied, there will also be a clear path for each of them
to be retrieved. From this state, any further incoming cars for parking will lead to obstructions for subsequent retrieval requests. However, this particular zone designations lead to an optimal layout.

Sometimes, however, according to the topology and the location of the elevator, the assignment of the priorities could be more complex and a more careful consideration is needed. As in Figures 3-11(c) and 3-11(d), the location of the loading cell has changed to position (3, 6), and consequently we have multiple choices of distribution of the priorities.

What needs to be done is to find a better one which will also minimize the number of obstructions after the high priority cells are occupied. For example in Figure 3-11(c), assume all the high priority cells (colored in gray) are occupied. According to the storage strategy, the incoming cars will be assigned to the remaining cells starting with the highest Manhattan distance (highest priority) and in a decreasing order of their Manhattan distances from the loading cell, where the numbers in the cells represent their respective Manhattan distances to the loading cell. However, after occupying the cells with “6” and “5” in Figure 3-11(c), the cell with Manhattan distance value of “4” at position (3,2) is blocked. So we need to modify the priority assignment and initial loading as shown in Figure 3-11(d) to avoid that situation.
In this chapter, we presented the methodologies we used for the storage and retrieval processes. We use D* Lite as our main path planning algorithm that guides a storage-retrieval request from a start point to a goal point while there is a clear path for the target. The unblock procedure which utilizes the A* and Uniform Cost Search algorithms helps the blocked cells keep moving towards the goal. Meanwhile, to minimize the number of obstructions for the storage during rush hours, a heuristic storage strategy is implemented.
Chapter 4

Simulation Study

4.1 Introduction

In order to demonstrate the feasibility and performance of proposed algorithms for an automated parking system, a simulation software application with multithreading was developed in Java.

This simulation is developed on a Windows desktop computer with the main hardware configuration as listed below:

- CPU: Intel® Core™ 2 6600 @ 2.40GHz
- Memory: 2.00GB DDR2 RAM

The operating system on this computer is Microsoft® Windows® XP® Professional with Service Pack 3. The application is developed in Java language, and the Java version is Java™ SE (Standard Edition) 1.6.0-24. The IDE (Integrated Development Environment) being used for the simulation is NetBeans 6.9.1.
4.1.1 Parking Space Layout

The parking space is represented as a 2-D grid based layout. There are a total of $m$ columns and $n$ rows. Therefore, there are $m \times n$ cells or parking spots in one parking area. These spots can be classified into three different categories in the parking area:

- **Elevators**: each elevator is constructed with two adjacent spots. Each elevator has its own position, and they are not movable. The number of elevators in the entire parking area depends on the number of spaces. In general, each elevator works for 50 or so spaces. The elevators will always be regarded as obstacles, which mean the mobile carts with moving “cars” cannot go through an elevator or park at the elevator.

- **Loading Areas**: each elevator has a particular loading area next to it and each loading area contains one spot. The loading areas are the starting point of storing a car and the ending point of retrieving a car. In the storage process, the car will appear at one loading area which means the car is unloaded from the elevator, and then begin to move. And in the retrieval process, the car will arrive at one loading area and then is loaded into the elevator. These spaces can be made part of the path but cannot be occupied as a parking spot.

- **Parking Spots**: The rest part of the parking area is all parking spots. They can be empty or occupied. The empty spots are not parked; cars are ready to be parked and cars can move onto it. Once they are parked, they will be regarded as obstacles. Remarkably, not all the sparking spots can be parked. There is always
some reasonable number of parking spots that should be empty. Because if a moving car is blocked by others, they need the empty cells to perform the unblocking procedure. The minimum number of empty spots is 7% of the total number of parking spots as default.

Figure 4-1 is an example of parking space with the size of 20×20 cells. The total number of cells is 400, so the number of elevators is 400/50=8. Since each elevator occupies two spaces, the elevators cost a total of 16 cells or spaces. As each elevator has a loading area, the number of loading areas is 8 which cost an additional 8 spaces. The number of parking spots is then calculated as 400-16-8=376. Out of these 376 parking spots, at least 19 spots (which is 5% of 376) must always be open or empty. So the total capacity of the parking floor is 376-19=357, and the utilization rate for parking is 357/376≈89.3%. Consider a normal parking lot, a single parking spot takes a maximum 9 ft wide and a 20 ft long and the area of a parking spot is approximately 180 ft\(^2\) [27]. Take the No.1 parking lot in Beijing Capital International Airport (BCIA) as an example. The parking lot is about 23854 m\(^2\) (roughly 256,761 ft\(^2\)) and contains 588 parking spots [28]. Therefore, the total area used for parking is 588×180=105,840 ft\(^2\). And the utilization rate used for parking is 105,840/256,761≈41.2%. Compare to the utilization ratio of our automated parking system, it gives an obvious result that our automated parking system is much better than normal parking lots in the aspect of space utilization ratio.
4.1.2 Application Description

This application is designed using a Model-View-Controller (MVC) framework [25]. The view part is mainly used for representing the main interface and the status of the data in model and the data changes. The model takes the responsibility of storing data structures and can notify the view(s) that its state has changed. Controller will accept user’s actions such as click buttons, and make appropriate responses. Also, the controller can communicate with the model and the view.

For this application, there is only one view class. The main interface is shown in Figure 4-1. There are a number of text labels and two buttons in the panel. The black labels
represent the elevators in the parking area; the labels with “load” are the loading areas for the elevator. Each elevator has one corresponding loading area. The blank labels denote the parking spots. The controller defines the appropriate action while user clicks the buttons. The button “Start” starts the program and simulates the situation in a real parking space, such as cars being stored and retrieved. The button “Quit” stops the program and returns the control to the operating system.

The layout matrix and some global variables are stored in the model as the fundamental data structures. The layout matrix is a two-dimensional static array, whose size depends on the size of the parking space. A pair of coordinates can locate one spot or cell on the panel as well as one element in the matrix. Each element in the matrix corresponds to a spot on the panel with the same pair of coordinates. And the value of an element in the matrix represents the state of the corresponding space: if value is 10, then the spot is in the area of an elevator; if value is 0, the spot is available for parking and; if value is 1, the spot is already occupied; if value is 2, the spot has been assigned but the car is still on the move; if value is 3, the spot (must be blank) is in use in an unblock procedure; if value is 4, the spot is a loading cell next to the elevator.
4.1.3 Concurrent Multi-Robot Navigation

An automated parking system is a multi-robot, and concurrent system. In this simulation of automated parking system, multiple cars may need to move in the zone simultaneously. In order to achieve it, multi-threaded programming technology is a good choice. A thread of execution is the smallest unit of processing that can be scheduled by an operating system. Multi threads are allowed to exist and run under the same process, and these threads share the resources with its owner process. However, one thread can run independently without interfering with each other. Threads may also require mutually-exclusive operations while they are accessing the shared resources to avoid data conflict such as accessing data which is being modified or modifying the same data simultaneously. Careless use of multi thread may lead to serious system failure and unexpected error. For a multi-core computer, the CPU can process multiple threads at any given time and the cores will switch between threads while processing.

In this simulation, each storage or retrieval request generates a thread. Each thread is then responsible for moving a car using the suite of path planning algorithms that include D* Lite, A* and UCS towards its destination position. The thread exits as soon as the car being relocated reaches its target location. If there are multiple requests for storage and retrieval at any given time, multiple threads will be created with one thread allocated per each unique request. This makes it possible multiple cars to be moved in the same zone concurrently. Since the paths may cross, thread access to shared resources, which are layout matrix and some global variables, must be managed through the mutual exclusion
(mutex) mechanism. While a thread is performing an operation (such as reading or writing) on the shared data, this data will be locked and will not be released until the thread completes the operation in progress. The maximum number of threads running under a process depends on the size of physical memory. By default, each thread has 1 MB stack space, and one process can own a maximum of 2 GB memory. Thus, theoretically, each process can create at most 2048 threads. Although the actual number may be smaller than 2048, it is obviously enough for this simulation.

The overall process of the simulation is shown as in Figure 4-2. After the program is started, the request generator keeps generating one of two kinds of requests, storage or retrieval following a certain schedule as shown in Table 4.1. Each request is associated with either storage or retrieval of a car. The requests include the start and destination location of the car. Then each request will lead to creation and launching of a program thread to serve its needs. The thread will invoke path search and planning algorithms to guide the car to its destination.
4.1.4 Request Generator

The request generator is used for generating the requests for storage and retrieval following a predetermined schedule that mimics a busy parking structure in a downtown metro area with very high population density. There are three kinds of requests which are storage, retrieval, and no request or idle. A “storage” request simulates that a car is being stored in the parking lot; a “retrieval” request simulates that a car is leaving the parking lot; and an “idle” request simulates the idle time without any request. The generator can
create one request each half second, and the three kinds of requests are generated with different probabilities in different periods.

There are four different time periods: the first period presents the morning rush hour from 6 AM to 8 AM; the second period stands for the late morning and afternoon from 8 AM to 4 PM; the third period is the rush hour in the afternoon from 4 PM to 6 PM; and the fourth period represents from 6 PM to 6 AM of the next day.

- Initially and prior to the start of the morning rush hour, the parking area is nearly empty at or about 5% capacity utilization. With the initiation of the morning rush hour period, most of the cars are being stored and only a small number of cars are being retrieved. The generator will create requests with the following probabilities: 90% are storage, 5% are retrieval, and 5% are idle. This period will terminate once the percentage of occupied spots reach 95%.

- In the second period that starts at 8 AM and lasts until 4 PM, the percentages of storage and retrieval requests generated are minimal. The percentages for storage and retrieval requests are both 5%, and the remaining 90% are for the idle requests.

- In the third period which spans from 4 PM to 6 PM, the situation is just the opposite when compared to the first period in that 90% of all the requests are retrieval and 5% of the requests are storage. Also, there are 5% idle requests. During this period the number of parked cars on the floor will decrease. Once the number reduces to 5% of the total capacity of the parking area, this period is over.
In the fourth period, the situation will be similar with the second period, but the percentage of storage and retrieval request will be smaller, about 1% each.

Following the fourth period, the generator will go to the first period and redo the whole process until the application is terminated. Table 4.1 presents the different percentages of requests in different periods as well as the capacity utilization along with the increase or decrease in the occupancy rates.

<table>
<thead>
<tr>
<th>Time Period</th>
<th>Storage Requests</th>
<th>Retrieval Requests</th>
<th>Idle</th>
<th>Parking Capacity Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM Rush Hours (6 am to 8 am)</td>
<td>90%</td>
<td>5%</td>
<td>5%</td>
<td>Increasing from 5% to 95%</td>
</tr>
<tr>
<td>Day Time (8 am to 4 pm)</td>
<td>5%</td>
<td>5%</td>
<td>90%</td>
<td>Stays around 95%</td>
</tr>
<tr>
<td>PM Rush Hours (4 pm to 6 pm)</td>
<td>5%</td>
<td>90%</td>
<td>5%</td>
<td>Decreasing from 95% to 5%</td>
</tr>
<tr>
<td>Night Time (6 pm to 6 am)</td>
<td>1%</td>
<td>1%</td>
<td>98%</td>
<td>Stays around 5%</td>
</tr>
</tbody>
</table>

4.1.5 Storage Process

The storage process facilitates movement of a car from a certain loading area to an available parking spot. It runs as a thread. If there are empty spots on the floor, the car will be first unloaded onto a certain loading area associated with the elevator that transported that same car as the starting point of the car. The process will then, in conjunction with the storage strategy, identify an available parking spot for the car (refer to section entitled “Storage Strategy”). After determining the start and destination location for the car, the path planning algorithm will be activated and will help move the
car to the destination step by step. During the move, if the car is blocked by other cars, the unblocking procedure will help the car keep moving. Once the car arrives at the destination, this process ends. Figure 4-3 shows the control flow for the storage process.

4.1.6 Retrieval Process

A retrieval process will facilitate movement of a car from its parking spot to the closest loading area, and it runs as a thread. The program will identify (randomly for simulation purposes) a parked car to retrieve. Its current location is the starting point. Then the retrieval process will identify the closest loading area as the destination cell or spot. The remainder of the process is the same as the storage process. The D* Lite algorithm and unblock procedure will lead the car until the car arrives at the destination. Figure 4-4 shows the control flow for this process.
4.2 Performance Testing and Evaluation

Testing will be performed to assess and measure performance aspects of the proposed design that manages the inflow and outflow of traffic in the single-story parking structure around the clock. It was established that there would be periods of intense parking activity associated with rush hours in the mornings and evenings as well as increased activity around noon. The performance of the proposed design as to how it manages the time periods with intense requests is of interest. An important parameter that affects the storage and retrieval path planning algorithms is the number and topological distribution of unmovable carts. Effect of this parameter on the performance of the system will also need to be assessed.
4.3 Testing Scenario

The testing scenario shall entail a typical weekday or workday profile. There are distinct periods of activity for a given 24-hour day. Representative activity periods would include morning rush hours, a subsequent low activity period until evening rush hour, increased activity period at evening rush hours, and followed by low-level activity throughout the evening, night and early morning. These activity periods are listed and described below.

Period 1. The single-story garage will be loaded to its nearly full capacity, i.e. 95% of the capacity, starting around 6 AM and possibly ending around 8 AM, with large number of storage actions and few retrieval actions.

Period 2. There will be few storage and retrieval actions during the 8 AM to 4 PM period

Period 3. There will be a rush of car retrieval requests starting at 4 PM and possibly ending around 6 PM when the garage will practically be emptied, i.e. less than 5% of parking capacity is likely to be in use.

Period 4. There will be few and sporadically distributed (in time) storage and retrieval requests between the times of 6 PM and 6 AM (the next day).

Further elaborations on periods 1 and 3 are made to clarify each one while the entire capacity utilization curve for the 24-hour period is shown in Figure 4-5.

Period 1 - Starts from a practically empty parking structure, until 95% of the capacity is
occupied as is typical for morning rush between 6 AM and 8 AM. As full loading is being achieved, sporadic but few retrieval requests are expected to occur, and thus, will also need to be simulated. The loading occurring during this 2-hour phase can be modeled after a quadratic-rise curve. This period will end once the garage is 95% full.

Period 3 - This phase assumes the parking structure is nearly full (i.e., 95% of capacity utilized). Requests for retrieval starts to come in around 4 PM and keeps increasing in a steady phase until 6 PM. This period will be terminated as soon as the parking lot is practically empty, i.e. 5% of capacity is occupied. Conceivably, there will be sporadic storage requests during this period. Note that it is typically expected that much less storage requests are likely during Period 3 as compared to retrieval requests during Period 1.
4.4 Performance Measures and Assessment

It is necessary to develop a set of performance measures to be able to profile the performance of the developed system. Some of the obvious ones would require ensuring that the storage and retrieval can be accomplished with the minimal number of cart movements, and the cart carrying the vehicle to be stored or retrieved would follow the shortest path to its destination. One would also be interested in making sure that the planning application software is fast enough to deal with storage and retrieval requests during peak times so that it can be deployed potentially for a real time context. It is also of interest to determine the maximum number of concurrent storage or retrieval requests that can be served as measured by the maximum number of concurrent storage-retrieval
threads in existence at any given time during the entire simulation. Also ability of the designed system to operate in the presence of failure of carts, such as breakdown and consequent immobility, is of interest. Memory space requirements of the overall planning process are also of interest. These points are formulated into performance measures to be leveraged for performance profiling of the software system that implements the suite of search and management algorithms presented earlier.

Performance measures will include the following:

a) Actual number of movements and shortest (optimal) path lengths for a given storage or retrieval request.

b) Search and planning time.

c) Application memory space utilization.

d) Maximum number of concurrent storage-retrieval threads.

e) Fault tolerance to failures due to immobilized carts.

The optimal path length is measured in terms of the Manhattan distance. For each storage or retrieval request, the first step is to identify the starting point and destination point. For a storage request, the starting point is a loading area next to an elevator and the destination is the parking spot assigned by storage strategy. For a retrieval request, the starting point is the location of car to be retrieved and the destination is the closest loading area next to the destination elevator. The optimal path is then the Manhattan distance between the starting and destination cells. After the simulation terminates, an average of the optimal path lengths (Manhattan distances) for all the requests is computed. The actual number of movements measures the total number of movements for a given
request. There are two aspects contributing to this measure: one is the movements of the
car being stored or retrieved, and the movements of the blank cell, which causes already-
parked carts to be relocated as it moves towards a spot on the shortest path from the car
being relocated to the destination elevator loading cell.

Search time measures the time cost for the time it takes to perform a search for path
planning. The search time must be small enough to ensure that the storage or retrieval
request can be satisfied in real time. Each storage and retrieval request leads to creation
of a dedicated thread. As soon as the thread is launched, a CPU clock measurement is
taken by this thread. The same thread will also record the time just before it exits and the
difference between two clock times is used to measure the total thread execution time.
The thread will call appropriate suite of path planning algorithms and execute them
which will effectively lead to a finite “search time”. During the course of lifetime of a
“Thread.sleep()” function is called with certain time interval value. This function will
make the thread suspend for a specified time (measured in milliseconds) to model the
actual time it takes to move a car from one position to its next. These sleep() times are
deducted from the total thread execution time to determine the net search and planning
time, which is used as the measurement values reported in the simulation study for time.

The space cost measures the maximum memory being used during the running of the
search algorithms. This number will be determined through monitoring the maximum
memory request by the process that executes must be reasonable and feasible to avoid
system crash caused by lack of memory. And to measure the memory cost, we used a
tool from Windows(R) 2003 Server(TM) Resource Kit called “memmonitor”. This is a
command-line tool used for monitoring the memory usage of a specified process. The
monitor should run all the time during the execution of the simulation program and we
pick the “PeakWSSize” from the last shown result set as the result we need.
“PeakWSSize” is short for “Peak Workspace Size”, and it represents the maximum value
of the memory usage during the running of the monitored process. Therefore, we need
this value to measure the maximum memory cost of the simulation.

4.5 Control Parameters

A number of parameters will be controlled to assess and evaluate the scaling properties of
the planning application. These parameters are the topology in terms of its dimensions
and corresponding cell count, and the percentage of blank cells. The topology of the floor
layout (20×40, 30×40 and 40×40) will be varied as follows and illustrated in Figure 4-6
through 4-8:

- 20×20
- 20×40
- 30×40
- 40×40
Figure 4-6. 20×40 Topology Description

Figure 4-7. 30×40 Topology Description
For each floor layout topology, the percentage of number of unoccupied (or blank) cells will be set as 7%, 13%, and 20%. The number of immobilized carts to determine the fault tolerance of the operation of the proposed system is set at 1%. Also the simulation was found to operate reliably for, and therefore can handle, up to about 64% of the theoretical value of maximum number of threads running concurrently.
4.6 Simulation Results

For each of the four different parking layout topologies (20×20, 20×40, 30×40 and 40×40), three different blank cell counts are considered as 7%, 13% and 20%. Thus we need to run the simulation for 12 different cases. There are additional cases as well: one scenario considers the situation where a number of carts are immobilized for a number of reasons including breakdown; a second scenario considers the case there is a heavy load for storage and retrieval requests leading to launching a high number of threads; and a third scenario combines those two situations. The simulation study is repeated 30 times for each case to facilitate computation of mean and standard deviation values based on the assumption of normal distribution. For each simulation run, four performance metrics are employed as follows:

- Optimal path length per request: record all the optimal path length for all the requests and compute the average,
- Actual number of movements per request: record all the actual movements for all the requests and compute the average,
- Search time per request: record all the search time for all the requests and compute the average, and
- Max memory utilization: record all the memory cost for all the requests and compute the average during the entire simulation run time.
The results as raw measurement data are shown in the Appendix in Tables A.1 through A.15 in tabular format. Normal probability density curves are generated for each case, and presented in Figures 4-10 through 4-19.

Figure 4-10 shows the average optimal path length (as Manhattan distance) per request for three different blank cell percentages in all four topologies. These plots are generated based on 30 simulation runs per case where storage-retrieval requests are randomly generated while also following the 24-hour load-unload schedule presented in Figure 4-5. For a given topology, as the number of blank cell percentage increases, the optimal path length will also increase. This is a consequence caused by the storage strategy which assigns furthest available locations with respect to the elevator locations in a given zone for incoming storage requests. Initially when the parking lot is lightly loaded, the optimal path length will be high and will decrease as more and more storage requests come in since as the parking lot starts to fill from the furthest locations towards the elevators per the storage strategy, the average optimal path length value will decrease.

Assume that the farthest location from the elevator has the longest distance value of $m$, and the total number of cars (parking capacity) is $n$. In a given zone, there will be multiple locations with the same Manhattan distance value that will vary in the interval $[m, 1]$ for all requests. Assuming the parking lot is filled to full capacity, then the average Manhattan distance is approximately $3m/4$ for a storage-retrieval request. As shown in Figure 4-9, for the cells in the first quadrant of $10\times10$ elevator zone, the average Manhattan distance is approximately $0.75\times6=4.5$. This value correlates highly with the
empirically determined values found through the simulation as reported in Figure 4-10. Average optimal path length values do not change appreciably as the topologies change since structure of so-called elevator zones are preserved exactly from one topology to another. Also, note that in all topology-specific plots, the average value increases as the percentage of blank cells increases although the increase is no more than 0.25 overall.

Figure 4-9. Manhattan Distances for Cells in First Quadrant of 10×10 Elevator Zone (Darkest Cells Indicate Elevators and Gray Cells Indicate Load-unload Cells)
Figure 4-11 shows the average actual number of movements per request for different blank cell percentages and different parking floor topologies. The curves for each single topology indicate that the average actual number of movements will increase if the blank cell percentage decreases, which has the opposite tendency with optimal path length. This situation is caused by the obstacles along the Manhattan path of a storage-retrieval request since the unblock procedure will need to be invoked to clear the path of the obstacles. If fewer of blank cells are available, then during period of high capacity utilization, there will be comparatively more obstacle relocation through repeated calls to
the unblock procedure and hence affecting the Manhattan distances of the relocated carts. Especially during the retrieval rush-hour period, unblock procedure will be invoked continually, each time to be able to move the cart one step which is the same as relocating to the next cell on the path. Relocating the obstacle requires a blank cell to be position in place of the obstacle and hence requiring typically multiple movements. Also for lower percentage of blank cells, it is more likely that average the position of an available blank cell will be further compared to the case where the percentage of blank cells is larger. As a result, the more unblock procedure performs, the longer the average actual number of movements is. Fewer blank cells will result in more invocations of the unblock procedure, and will also lead to longer actual movements on the average. Figure 4-1 shows that the topology does not affect the actual number of movements by an appreciable amount, and the reason is the same as it is for the optimal path length.
Figures 4-12 through 4-14 compare the average optimal path length to the average actual movements with a certain blank cell percentage throughout four different topologies. At least two important observations can be made. First, the actual number of movements will always be longer than the optimal path length in every simulation case. The optimal path length is the lower bound on the actual movements, which would only hold if there were no obstacles on the Manhattan path and the unblock procedure did not need to be invoked. The fact is that the unblock procedure is invoked a number of times for many of the storage-retrieval requests. The results show that the average actual number of
movements is about two more movements longer than the average optimal path length. The second observation is that, comparing the distances between the two curves throughout different blank cell percentages for a specific topology, the two curves get closer as the blank percentage increases. The explanation is as follows. With the increase of the blank cell percentage, the actual number of movements will be shorter according to data in Figure 4-11, while the optimal path length will be longer according to data in Figure 4-10. Therefore, the difference between them is smaller.

Figure 4-12. Optimal Path vs. Actual Number of Movements for 7% Blank Cells
Figure 4-13. Optimal Path vs. Actual Number of Movements for 13% Blank Cells
Plots in Figure 4-15 show the average search time per request measured in milliseconds with different blank cell percentages and topologies. If we compare Figure 4-15 with Figure 4-11, we can detect that each curve in Figure 4-15 correlates with the corresponding one in Figure 4-11. This leads to the conclusion that the search time is directly impacted by the actual number of movements. The time complexity of D* Lite is $O(b^d)$ [8], and those of A*, and UCS are also $O(b^d)$ [6], in which $b$ is the branching factor and the $d$ is the path length. If the $b$ is fixed (which is the case for all simulation cases), then the search time will only depend on the path length, and the longer the path is, the
more time is needed for searching. That is the reason why search time is directly related to the number of actual movements. Another observation from Figure 4-15 concerns the real-time property of the automated parking system. For the largest topology and smallest blank cell percentage (40×40 and 7% blank cell), the time spent for search is about 80 to 88 milliseconds for each request, and each request results in about an average of 6.4 to 7 movements. Thus, the search time for a single movement will be around 12.5 milliseconds, which offers a quick response time for real-time path planning purposes.

Figure 4-15. Search Time vs. Blank Cell Percentages in Four Topologies
Figure 4-16 presents the results for the maximum memory utilization during the simulation run. We can observe from those plots that for a fixed blank cell percentage, the mean value of the maximum memory utilization will increase as the topology becomes larger. The reason is that if the topology is greater, there will be more elevators, and more elevators will result in higher concurrency, which means more threads will be run simultaneously. Accordingly, higher concurrency will lead to more memory utilization. However, the memory usage does not vary significantly among four topologies. The memory utilizations are all in an acceptable range, which is around 31 MB. Therefore, the memory utilization is not expected to impact negatively the performance of the designed system as validated through the simulation study.
It is also of interest to identify the effect of increase in storage-retrieval demand that is concurrent on the performance metrics. Since each storage-retrieval thread requires exactly one blank cell for the unblock procedure, the maximum number of concurrent storage-retrieval thread that can exist is given by the number of available blank cells for a given topology and blank cell count. For the 40×40 topology and 13% blank cells, the number of blank cells is 195 which is the maximum number of concurrent threads that can be in existence any time during the operation of the automated parking system management software. An exploratory study indicated that the software did not execute
its functionality reliably for more than 125 concurrent threads. Accordingly, a case study was performed for up to 125 concurrent threads for the 40×40 topology with 13% blank cells. Results for the actual movements, optimal path, search time, and memory utilization are presented in Figure 4-17.

Actual number of movements with increased concurrent demand is greater than that for the lighter concurrent demand (which employed no more than 31 threads). The increase in the number of movements is approximately 1. The dynamic replanning in the presence of concurrently moving carts and therefore affecting each others’ movements or paths is the main reason for this increase although it does not appear to be large enough to cause a major concern even in a real time context. In conjunction with the increased actual number of movements for increased concurrent demand situation, the search time also, as before, increases by 10 ms from approximately a mean value of 82 ms to 92 ms. The memory utilization increases also from a mean value of 32 MB to a mean value of 35.5 MB which is not significant by any means.
Another point of exploration is how the immobilized carts will affect the performance of the proposed design. This case assumes there are immobilized cells uniformly distributed across the parking floor. The locations of the immobilized cells are generated through 15 (1% of the capacity) randomly picked numbers in the range from 0 to 1599. Each randomly-picked number $n$ would be converted to a pair of coordinates ($n/40, n \% 40$) as every parking location is represented by its row and column coordinates as $(x, y)$ on the parking layout. If such a randomly determined location is claimed by an elevator, or a
loading area, then a new random drawing is executed which continues until an available cell is identified through this process.

Figure 4-18 shows the results for the $40 \times 40$ topology and 13% blank cells where 1% of the cells are immobilized. The actual number of movements increases by approximately 2.5 cell movements per request for this size elevator zone, which is significant. Along with this increase in the actual movements, the search time increases as before. The increase in the memory utilization is slight and probably does not warrant any further consideration. The nature of immobilized cell is that if they reside on the Manhattan path of a cart in motion, then the cart will have to go around the immobilized cell to reach its destination. This introduces significant extra movements. Those immobilized carts that are closer to the elevators will block more Manhattan paths than those that are away from the elevators. The impact on the performance brought by the immobilized cells depends on the distribution of those cells. The farther from the elevator, the lower the adverse impact will be.
The final case study is when two potentially negative factors, namely the immobilized cells and high concurrency, both co-exist in the environment. It is of interest to determine how the performance will be affected if there is high demand for requests and 1% of the cells are occupied by immobilized carts. Simulation results for this case are presented in Figure 4-19. For comparison purposes, plots for the three other scenarios are also superimposed in Figure 4-19: these are plots for the case with no high concurrency or immobilized carts, the case for high concurrency, and the case for immobilized carts. For the actual movements and search time plots (top left and top right, respectively), the
“High Concurrency with Immobilized Cells” curve correlates more closely with the “Immobilized Cells Only” curve. This indicates that immobility in the environment is the primary factor adversely affecting the performance. This means the performance for the actual movements and the search time will be impacted significantly by the existence of immobilized cells, but not by the degree of concurrency. From the memory utilization plot at the bottom left, the “High Concurrency with Immobilized Cells” curve is close to the “High Concurrency Only” curve and far away from the other two curves, which indicates that the utilization of the memory is highly correlated to the degree of concurrency but not to the existence of immobilized cells.

Figure 4-19. Performance Profile for High Concurrency Demand and Immobilized Cells
4.7 Overview and Review of Findings

In this section, we present an overview of the major findings for the entire scope of the simulation study.

- The actual number of movements per request will always be longer by up to 49% than the optimal path length for these specific topology layouts. Within the same topology, the actual number of movements will increase as the percentage of blank cells decreases. Also, the actual movements will not be impacted significantly by the size of topology due to modular construction;

- Time cost for path planning (search time) is directly associated to the actual number of movements. The search time is small and approximately 82 ms, which suggests a quick response by the system, and perhaps more importantly, the design can be deployed for real time environments;

- The memory space requirement of the system is relatively low in general. Even when there are 64% of max concurrent requests, its value is no more than 36 MB. The memory utilization mainly correlates with the degree of concurrency in the environment. If the concurrent demand is high, then the design requires more memory resources, but the amount always stays within reasonable boundaries for today’s technology.

- Through special scenarios (high concurrency demand, immobilized cells, and both combined), it became apparent that if immobilized cells exist, they will result
in notable deterioration for the two performance measures, namely the actual
number of movements and the search time.

In light of and within the context of the simulation study presented, the design appears
feasible for real time deployment in an industrial-grade environment.
Chapter 5

Conclusions and Recommendations for Future Study

In this thesis, we presented a design, development, simulation and performance assessment and an evaluation of storage and retrieval algorithms for an automated parking and retrieving system. Differing from other automated parking systems, there are no aisles contained in the parking floor, which will bring larger parking capacity, making the path planning process more complex. Moreover, the system should be concurrent and real-time, so multiple algorithms construct a whole path planning system, including D* Lite replanning and unblock procedure, such as D* Lite algorithm, A* algorithm and Uniform Cost Search algorithm.

We also designed a storage strategy according to the practical parking situation. We also developed a simulation program to demonstrate the feasibility of our proposed system on a multi-threaded basis. The simulation is based on a four-period-scenario, including storage rush hours from 6AM to 8AM, few requests (storage and retrieval) from 8AM to 4PM, retrieval rush hours from 4PM to 6PM and rare requests (storage and retrieval) from 6PM to 6AM on the next morning. The simulation runs through 4 different topologies (20×20, 20×40, 30×40 and 40×40) and 3 different blank cell counts (7%, 13% and 20%). Four major performance metrics, optimal path length per request, actual
number of movements per request, search time per request and max memory utilization are measured. Throughout comparison of optimal path length and number of actual movements, we can indicate that the number of actual movements will be up to 49% more than the optimal path length. The search time is directly correlative to the number of path length, and the values are between about 70 ms to 90 ms, which is acceptable for our real time demand. The memory utilization is about 32 MB which we also considered as feasible. Furthermore, some special cases, high concurrency demand (up to 64% of the max number threads), running with immobilized cells and two of them combined, have also been implemented. Those special cases show that the high concurrency demand will lead to significant impact to the memory utilization (from 32 MB to 36 MB) and immobilized cells will affect the actual number of movements(from about 6 movements to about 9 movements) and search time(from maximum 90ms to maximum 110ms). To sum up, the simulation demonstrates that the proposed system is feasible for our concurrent, real time, and dynamic requirements.

The research could be extended in the future from the following aspects:

- Extend the single-story parking layout to a multi-story parking layout. To do this, more research should be done such as elevator schedule strategy, check elevator availabilities and so on.

- In our current system, all the parking spots are in the same size. However, based on the different kind of cars (sedans, SUVs, mini vans, trucks, etc.), the size of a parking spot should match the largest car size. To minimize the land utilization
rate and enhance the capacity, we could design another topology containing cells with different sizes for different kind of cars.

- Depending on the simulation result, the immobilized cells will impact the performance greatly. If we can reduce that effect by improving methodologies or algorithms, that will be a very important improvement for the system.
References


[16] Sven Koenig and Maxim Likhachev, “Real-Time Adaptive A*”, in Proc. of the


[26] Wikipedia®, ”thread”, Internet:


Appendix A

Simulation Raw Data
Table A.1. Performance for 7% Blank Cells and 20×20 Topology

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time(ms)</th>
<th>Memory Utilization(KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.57</td>
<td>6.93</td>
<td>86.39</td>
<td>31596</td>
</tr>
<tr>
<td>2</td>
<td>4.51</td>
<td>6.34</td>
<td>83.71</td>
<td>31452</td>
</tr>
<tr>
<td>3</td>
<td>4.58</td>
<td>6.6</td>
<td>82.04</td>
<td>31476</td>
</tr>
<tr>
<td>4</td>
<td>4.56</td>
<td>6.81</td>
<td>84.65</td>
<td>31580</td>
</tr>
<tr>
<td>5</td>
<td>4.52</td>
<td>6.49</td>
<td>80.45</td>
<td>29576</td>
</tr>
<tr>
<td>6</td>
<td>4.54</td>
<td>6.32</td>
<td>78.76</td>
<td>31392</td>
</tr>
<tr>
<td>7</td>
<td>4.51</td>
<td>6.23</td>
<td>77.18</td>
<td>29680</td>
</tr>
<tr>
<td>8</td>
<td>4.54</td>
<td>6.31</td>
<td>78.69</td>
<td>31300</td>
</tr>
<tr>
<td>9</td>
<td>4.57</td>
<td>6.53</td>
<td>81.57</td>
<td>31404</td>
</tr>
<tr>
<td>10</td>
<td>4.61</td>
<td>6.91</td>
<td>85.14</td>
<td>31356</td>
</tr>
<tr>
<td>11</td>
<td>4.57</td>
<td>6.94</td>
<td>84.76</td>
<td>31444</td>
</tr>
<tr>
<td>12</td>
<td>4.55</td>
<td>6.86</td>
<td>85.55</td>
<td>31256</td>
</tr>
<tr>
<td>13</td>
<td>4.54</td>
<td>6.51</td>
<td>80.72</td>
<td>30504</td>
</tr>
<tr>
<td>14</td>
<td>4.5</td>
<td>6.43</td>
<td>76.68</td>
<td>31540</td>
</tr>
<tr>
<td>15</td>
<td>4.44</td>
<td>6.18</td>
<td>76.71</td>
<td>31436</td>
</tr>
<tr>
<td>16</td>
<td>4.57</td>
<td>6.76</td>
<td>84.19</td>
<td>31716</td>
</tr>
<tr>
<td>17</td>
<td>4.58</td>
<td>6.76</td>
<td>83.43</td>
<td>31392</td>
</tr>
<tr>
<td>18</td>
<td>4.55</td>
<td>6.69</td>
<td>82.92</td>
<td>29944</td>
</tr>
<tr>
<td>19</td>
<td>4.58</td>
<td>6.85</td>
<td>83.1</td>
<td>31568</td>
</tr>
<tr>
<td>20</td>
<td>4.57</td>
<td>6.91</td>
<td>85.74</td>
<td>31416</td>
</tr>
<tr>
<td>21</td>
<td>4.53</td>
<td>6.55</td>
<td>81.49</td>
<td>31524</td>
</tr>
<tr>
<td>22</td>
<td>4.58</td>
<td>6.39</td>
<td>78.66</td>
<td>31348</td>
</tr>
<tr>
<td>23</td>
<td>4.57</td>
<td>6.86</td>
<td>85.51</td>
<td>31388</td>
</tr>
<tr>
<td>24</td>
<td>4.57</td>
<td>6.66</td>
<td>88</td>
<td>31292</td>
</tr>
<tr>
<td>25</td>
<td>4.54</td>
<td>6.56</td>
<td>80.6</td>
<td>31360</td>
</tr>
<tr>
<td>26</td>
<td>4.55</td>
<td>6.84</td>
<td>84.19</td>
<td>31348</td>
</tr>
<tr>
<td>27</td>
<td>4.53</td>
<td>6.69</td>
<td>83.06</td>
<td>35184</td>
</tr>
<tr>
<td>28</td>
<td>4.47</td>
<td>6.42</td>
<td>78.64</td>
<td>31520</td>
</tr>
<tr>
<td>29</td>
<td>4.51</td>
<td>6.35</td>
<td>78.82</td>
<td>29564</td>
</tr>
<tr>
<td>30</td>
<td>4.56</td>
<td>6.61</td>
<td>81.48</td>
<td>31676</td>
</tr>
<tr>
<td>AVG</td>
<td>4.55</td>
<td>6.61</td>
<td>82.09</td>
<td>31307.7</td>
</tr>
<tr>
<td>STDEV</td>
<td>0.035</td>
<td>0.225</td>
<td>3.035</td>
<td>949.216</td>
</tr>
</tbody>
</table>

92
Table A.2. Performance for 13% Blank Cells and 20×20 Topology

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.62</td>
<td>6.02</td>
<td>74.56</td>
<td>31932</td>
</tr>
<tr>
<td>2</td>
<td>4.64</td>
<td>6.29</td>
<td>78.14</td>
<td>29560</td>
</tr>
<tr>
<td>3</td>
<td>4.69</td>
<td>6.64</td>
<td>82.73</td>
<td>29528</td>
</tr>
<tr>
<td>4</td>
<td>4.71</td>
<td>6.71</td>
<td>83.76</td>
<td>29412</td>
</tr>
<tr>
<td>5</td>
<td>4.71</td>
<td>6.53</td>
<td>81.35</td>
<td>29944</td>
</tr>
<tr>
<td>6</td>
<td>4.67</td>
<td>6.69</td>
<td>83.65</td>
<td>29960</td>
</tr>
<tr>
<td>7</td>
<td>4.63</td>
<td>6.21</td>
<td>76.67</td>
<td>31056</td>
</tr>
<tr>
<td>8</td>
<td>4.6</td>
<td>6.34</td>
<td>84.63</td>
<td>31092</td>
</tr>
<tr>
<td>9</td>
<td>4.55</td>
<td>6.4</td>
<td>79.31</td>
<td>31172</td>
</tr>
<tr>
<td>10</td>
<td>4.67</td>
<td>6.34</td>
<td>78.84</td>
<td>31072</td>
</tr>
<tr>
<td>11</td>
<td>4.66</td>
<td>6.49</td>
<td>79.98</td>
<td>30628</td>
</tr>
<tr>
<td>12</td>
<td>4.62</td>
<td>6.26</td>
<td>78.43</td>
<td>31276</td>
</tr>
<tr>
<td>13</td>
<td>4.59</td>
<td>5.9</td>
<td>72.98</td>
<td>29392</td>
</tr>
<tr>
<td>14</td>
<td>4.61</td>
<td>6.75</td>
<td>84.26</td>
<td>31092</td>
</tr>
<tr>
<td>15</td>
<td>4.6</td>
<td>6.06</td>
<td>75.77</td>
<td>29220</td>
</tr>
<tr>
<td>16</td>
<td>4.61</td>
<td>6.17</td>
<td>76.3</td>
<td>31167</td>
</tr>
<tr>
<td>17</td>
<td>4.63</td>
<td>6.39</td>
<td>79.61</td>
<td>31200</td>
</tr>
<tr>
<td>18</td>
<td>4.61</td>
<td>6.42</td>
<td>80.02</td>
<td>31132</td>
</tr>
<tr>
<td>19</td>
<td>4.66</td>
<td>6.47</td>
<td>80.69</td>
<td>29352</td>
</tr>
<tr>
<td>20</td>
<td>4.6</td>
<td>6.38</td>
<td>79.37</td>
<td>31176</td>
</tr>
<tr>
<td>21</td>
<td>4.62</td>
<td>6.6</td>
<td>82.25</td>
<td>31064</td>
</tr>
<tr>
<td>22</td>
<td>4.58</td>
<td>6.38</td>
<td>78.9</td>
<td>31024</td>
</tr>
<tr>
<td>23</td>
<td>4.7</td>
<td>7.18</td>
<td>89.49</td>
<td>31056</td>
</tr>
<tr>
<td>24</td>
<td>4.71</td>
<td>6.94</td>
<td>86.78</td>
<td>31260</td>
</tr>
<tr>
<td>25</td>
<td>4.61</td>
<td>5.98</td>
<td>74.88</td>
<td>31112</td>
</tr>
<tr>
<td>26</td>
<td>4.57</td>
<td>6.12</td>
<td>76.19</td>
<td>31172</td>
</tr>
<tr>
<td>27</td>
<td>4.64</td>
<td>6.29</td>
<td>77.58</td>
<td>31272</td>
</tr>
<tr>
<td>28</td>
<td>4.67</td>
<td>6.47</td>
<td>80.35</td>
<td>29728</td>
</tr>
<tr>
<td>29</td>
<td>4.72</td>
<td>6.62</td>
<td>88.65</td>
<td>31124</td>
</tr>
<tr>
<td>30</td>
<td>4.65</td>
<td>6.82</td>
<td>85.4</td>
<td>29312</td>
</tr>
</tbody>
</table>

| AVG    | 4.64                         | 6.43                        | 80.38            | 30598.23                |
| STDEV  | 0.045                        | 0.290                       | 4.132            | 783.447                 |

93
Table A.3. Performance for 20% Blank Cells and 20×20 Topology

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.8</td>
<td>6.21</td>
<td>76.72</td>
<td>31328</td>
</tr>
<tr>
<td>2</td>
<td>4.75</td>
<td>6.19</td>
<td>76.85</td>
<td>31312</td>
</tr>
<tr>
<td>3</td>
<td>4.76</td>
<td>6.25</td>
<td>77.74</td>
<td>31508</td>
</tr>
<tr>
<td>4</td>
<td>4.75</td>
<td>6.37</td>
<td>79.41</td>
<td>31400</td>
</tr>
<tr>
<td>5</td>
<td>4.76</td>
<td>6.15</td>
<td>76.67</td>
<td>31380</td>
</tr>
<tr>
<td>6</td>
<td>4.78</td>
<td>6.31</td>
<td>78.28</td>
<td>31340</td>
</tr>
<tr>
<td>7</td>
<td>4.75</td>
<td>6.31</td>
<td>78.46</td>
<td>31456</td>
</tr>
<tr>
<td>8</td>
<td>4.77</td>
<td>5.9</td>
<td>73.13</td>
<td>31372</td>
</tr>
<tr>
<td>9</td>
<td>4.8</td>
<td>6.02</td>
<td>74.66</td>
<td>31324</td>
</tr>
<tr>
<td>10</td>
<td>4.75</td>
<td>6.08</td>
<td>75.77</td>
<td>31504</td>
</tr>
<tr>
<td>11</td>
<td>4.8</td>
<td>6.15</td>
<td>76.32</td>
<td>31332</td>
</tr>
<tr>
<td>12</td>
<td>4.79</td>
<td>6.36</td>
<td>79.72</td>
<td>31452</td>
</tr>
<tr>
<td>13</td>
<td>4.73</td>
<td>6.13</td>
<td>76.2</td>
<td>31472</td>
</tr>
<tr>
<td>14</td>
<td>4.74</td>
<td>6.13</td>
<td>75.27</td>
<td>31372</td>
</tr>
<tr>
<td>15</td>
<td>4.77</td>
<td>6.12</td>
<td>74.75</td>
<td>31448</td>
</tr>
<tr>
<td>16</td>
<td>4.69</td>
<td>6.29</td>
<td>77.32</td>
<td>31488</td>
</tr>
<tr>
<td>17</td>
<td>4.73</td>
<td>6.23</td>
<td>76.94</td>
<td>29956</td>
</tr>
<tr>
<td>18</td>
<td>4.68</td>
<td>6.19</td>
<td>77.11</td>
<td>31440</td>
</tr>
<tr>
<td>19</td>
<td>4.79</td>
<td>6.36</td>
<td>78.6</td>
<td>31304</td>
</tr>
<tr>
<td>20</td>
<td>4.82</td>
<td>6.35</td>
<td>78.28</td>
<td>31360</td>
</tr>
<tr>
<td>21</td>
<td>4.8</td>
<td>6.51</td>
<td>80.45</td>
<td>31296</td>
</tr>
<tr>
<td>22</td>
<td>4.73</td>
<td>6.16</td>
<td>76.63</td>
<td>31352</td>
</tr>
<tr>
<td>23</td>
<td>4.71</td>
<td>6.02</td>
<td>74.26</td>
<td>31412</td>
</tr>
<tr>
<td>24</td>
<td>4.74</td>
<td>6.19</td>
<td>82.59</td>
<td>30364</td>
</tr>
<tr>
<td>25</td>
<td>4.74</td>
<td>6.11</td>
<td>74.72</td>
<td>28924</td>
</tr>
<tr>
<td>26</td>
<td>4.71</td>
<td>5.96</td>
<td>73.95</td>
<td>31544</td>
</tr>
<tr>
<td>27</td>
<td>4.75</td>
<td>6.27</td>
<td>77.93</td>
<td>29528</td>
</tr>
<tr>
<td>28</td>
<td>4.82</td>
<td>6.22</td>
<td>76.36</td>
<td>31360</td>
</tr>
<tr>
<td>29</td>
<td>4.78</td>
<td>6.34</td>
<td>78.32</td>
<td>31536</td>
</tr>
<tr>
<td>30</td>
<td>4.75</td>
<td>6.42</td>
<td>79.86</td>
<td>31436</td>
</tr>
</tbody>
</table>

**AVG**  
4.76  
6.21  
77.11  
31176.67

**STDEV**  
0.035  
0.137  
2.076  
617.449

94
Table A.4. Performance for 7% Blank Cells and 20×40 Topology

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.52</td>
<td>6.68</td>
<td>83.35</td>
<td>31792</td>
</tr>
<tr>
<td>2</td>
<td>4.47</td>
<td>6.9</td>
<td>86.15</td>
<td>31720</td>
</tr>
<tr>
<td>3</td>
<td>4.45</td>
<td>6.74</td>
<td>84.1</td>
<td>31840</td>
</tr>
<tr>
<td>4</td>
<td>4.48</td>
<td>6.83</td>
<td>85.23</td>
<td>31868</td>
</tr>
<tr>
<td>5</td>
<td>4.54</td>
<td>6.82</td>
<td>85.13</td>
<td>31812</td>
</tr>
<tr>
<td>6</td>
<td>4.59</td>
<td>7.07</td>
<td>88.21</td>
<td>31756</td>
</tr>
<tr>
<td>7</td>
<td>4.55</td>
<td>7.09</td>
<td>86.81</td>
<td>31800</td>
</tr>
<tr>
<td>8</td>
<td>4.49</td>
<td>6.65</td>
<td>83.01</td>
<td>31820</td>
</tr>
<tr>
<td>9</td>
<td>4.49</td>
<td>6.66</td>
<td>82.86</td>
<td>31696</td>
</tr>
<tr>
<td>10</td>
<td>4.49</td>
<td>6.76</td>
<td>84.23</td>
<td>31732</td>
</tr>
<tr>
<td>11</td>
<td>4.47</td>
<td>6.87</td>
<td>85.35</td>
<td>31736</td>
</tr>
<tr>
<td>12</td>
<td>4.47</td>
<td>6.71</td>
<td>83.89</td>
<td>31784</td>
</tr>
<tr>
<td>13</td>
<td>4.52</td>
<td>6.9</td>
<td>85.9</td>
<td>31824</td>
</tr>
<tr>
<td>14</td>
<td>4.48</td>
<td>6.77</td>
<td>84.18</td>
<td>31748</td>
</tr>
<tr>
<td>15</td>
<td>4.5</td>
<td>6.68</td>
<td>83.39</td>
<td>31656</td>
</tr>
<tr>
<td>16</td>
<td>4.53</td>
<td>6.86</td>
<td>85.15</td>
<td>31800</td>
</tr>
<tr>
<td>17</td>
<td>4.51</td>
<td>6.7</td>
<td>83.35</td>
<td>31804</td>
</tr>
<tr>
<td>18</td>
<td>4.51</td>
<td>7</td>
<td>87.26</td>
<td>31848</td>
</tr>
<tr>
<td>19</td>
<td>4.49</td>
<td>6.71</td>
<td>83.35</td>
<td>31700</td>
</tr>
<tr>
<td>20</td>
<td>4.51</td>
<td>6.62</td>
<td>82.6</td>
<td>31880</td>
</tr>
<tr>
<td>21</td>
<td>4.49</td>
<td>6.74</td>
<td>83.91</td>
<td>31688</td>
</tr>
<tr>
<td>22</td>
<td>4.54</td>
<td>6.69</td>
<td>83.23</td>
<td>31756</td>
</tr>
<tr>
<td>23</td>
<td>4.55</td>
<td>6.83</td>
<td>84.61</td>
<td>31760</td>
</tr>
<tr>
<td>24</td>
<td>4.55</td>
<td>6.79</td>
<td>84.71</td>
<td>31716</td>
</tr>
<tr>
<td>25</td>
<td>4.52</td>
<td>6.82</td>
<td>84.58</td>
<td>31672</td>
</tr>
<tr>
<td>26</td>
<td>4.53</td>
<td>6.7</td>
<td>83.24</td>
<td>31844</td>
</tr>
<tr>
<td>27</td>
<td>4.5</td>
<td>6.69</td>
<td>82.13</td>
<td>31832</td>
</tr>
<tr>
<td>28</td>
<td>4.51</td>
<td>6.87</td>
<td>83.57</td>
<td>31888</td>
</tr>
<tr>
<td>29</td>
<td>4.52</td>
<td>6.88</td>
<td>85.56</td>
<td>31680</td>
</tr>
<tr>
<td>30</td>
<td>4.52</td>
<td>6.65</td>
<td>82.84</td>
<td>31680</td>
</tr>
<tr>
<td>AVG</td>
<td>4.51</td>
<td>6.79</td>
<td>84.40</td>
<td>31771.07</td>
</tr>
<tr>
<td>STDEV</td>
<td>0.030</td>
<td>0.120</td>
<td>1.432</td>
<td>66.233</td>
</tr>
</tbody>
</table>
Table A.5. Performance for 13% Blank Cells and 20×40 Topology

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.59</td>
<td>6.8</td>
<td>84.8</td>
<td>31868</td>
</tr>
<tr>
<td>2</td>
<td>4.58</td>
<td>6.63</td>
<td>82.75</td>
<td>31812</td>
</tr>
<tr>
<td>3</td>
<td>4.62</td>
<td>6.51</td>
<td>81.03</td>
<td>31804</td>
</tr>
<tr>
<td>4</td>
<td>4.57</td>
<td>6.45</td>
<td>80.46</td>
<td>31769</td>
</tr>
<tr>
<td>5</td>
<td>4.66</td>
<td>6.49</td>
<td>80.79</td>
<td>31752</td>
</tr>
<tr>
<td>6</td>
<td>4.63</td>
<td>6.59</td>
<td>81.87</td>
<td>31820</td>
</tr>
<tr>
<td>7</td>
<td>4.59</td>
<td>6.72</td>
<td>83.74</td>
<td>31816</td>
</tr>
<tr>
<td>8</td>
<td>4.61</td>
<td>6.72</td>
<td>83.7</td>
<td>31825</td>
</tr>
<tr>
<td>9</td>
<td>4.64</td>
<td>6.51</td>
<td>81.4</td>
<td>31732</td>
</tr>
<tr>
<td>10</td>
<td>4.62</td>
<td>6.64</td>
<td>82.91</td>
<td>31612</td>
</tr>
<tr>
<td>11</td>
<td>4.61</td>
<td>6.66</td>
<td>82.38</td>
<td>31784</td>
</tr>
<tr>
<td>12</td>
<td>4.63</td>
<td>6.84</td>
<td>85.35</td>
<td>31748</td>
</tr>
<tr>
<td>13</td>
<td>4.57</td>
<td>6.53</td>
<td>81.75</td>
<td>31732</td>
</tr>
<tr>
<td>14</td>
<td>4.62</td>
<td>6.67</td>
<td>82.4</td>
<td>31984</td>
</tr>
<tr>
<td>15</td>
<td>4.65</td>
<td>6.79</td>
<td>84.93</td>
<td>31756</td>
</tr>
<tr>
<td>16</td>
<td>4.59</td>
<td>6.64</td>
<td>82.65</td>
<td>31804</td>
</tr>
<tr>
<td>17</td>
<td>4.59</td>
<td>6.81</td>
<td>84.99</td>
<td>31668</td>
</tr>
<tr>
<td>18</td>
<td>4.56</td>
<td>6.55</td>
<td>81.63</td>
<td>31772</td>
</tr>
<tr>
<td>19</td>
<td>4.58</td>
<td>6.93</td>
<td>86.33</td>
<td>31720</td>
</tr>
<tr>
<td>20</td>
<td>4.58</td>
<td>6.35</td>
<td>78.99</td>
<td>31676</td>
</tr>
<tr>
<td>21</td>
<td>4.62</td>
<td>6.64</td>
<td>81.6</td>
<td>31700</td>
</tr>
<tr>
<td>22</td>
<td>4.6</td>
<td>6.81</td>
<td>84.39</td>
<td>31792</td>
</tr>
<tr>
<td>23</td>
<td>4.68</td>
<td>6.78</td>
<td>84.69</td>
<td>31836</td>
</tr>
<tr>
<td>24</td>
<td>4.63</td>
<td>6.79</td>
<td>84.03</td>
<td>31746</td>
</tr>
<tr>
<td>25</td>
<td>4.6</td>
<td>6.52</td>
<td>81.04</td>
<td>31960</td>
</tr>
<tr>
<td>26</td>
<td>4.7</td>
<td>6.78</td>
<td>84.4</td>
<td>31760</td>
</tr>
<tr>
<td>27</td>
<td>4.6</td>
<td>6.67</td>
<td>83.08</td>
<td>31800</td>
</tr>
<tr>
<td>28</td>
<td>4.61</td>
<td>6.55</td>
<td>81.63</td>
<td>31824</td>
</tr>
<tr>
<td>29</td>
<td>4.6</td>
<td>6.55</td>
<td>81.76</td>
<td>31624</td>
</tr>
<tr>
<td>30</td>
<td>4.61</td>
<td>6.69</td>
<td>83.34</td>
<td>31752</td>
</tr>
<tr>
<td><strong>AVG</strong></td>
<td><strong>4.61</strong></td>
<td><strong>6.65</strong></td>
<td><strong>82.83</strong></td>
<td><strong>31774.93</strong></td>
</tr>
<tr>
<td><strong>STDEV</strong></td>
<td><strong>0.032</strong></td>
<td><strong>0.133</strong></td>
<td><strong>1.677</strong></td>
<td><strong>79.405</strong></td>
</tr>
</tbody>
</table>
Table A.6. Performance for 20% Blank Cells and 20×40 Topology

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.71</td>
<td>6.62</td>
<td>82.18</td>
<td>31772</td>
</tr>
<tr>
<td>2</td>
<td>4.71</td>
<td>6.31</td>
<td>78.59</td>
<td>31784</td>
</tr>
<tr>
<td>3</td>
<td>4.72</td>
<td>6.61</td>
<td>82.44</td>
<td>31588</td>
</tr>
<tr>
<td>4</td>
<td>4.76</td>
<td>6.6</td>
<td>82.12</td>
<td>31920</td>
</tr>
<tr>
<td>5</td>
<td>4.75</td>
<td>6.58</td>
<td>81.96</td>
<td>31780</td>
</tr>
<tr>
<td>6</td>
<td>4.71</td>
<td>6.48</td>
<td>80.71</td>
<td>31768</td>
</tr>
<tr>
<td>7</td>
<td>4.73</td>
<td>6.35</td>
<td>78.91</td>
<td>31876</td>
</tr>
<tr>
<td>8</td>
<td>4.72</td>
<td>6.4</td>
<td>79.78</td>
<td>31664</td>
</tr>
<tr>
<td>9</td>
<td>4.68</td>
<td>6.33</td>
<td>78.93</td>
<td>31792</td>
</tr>
<tr>
<td>10</td>
<td>4.71</td>
<td>6.33</td>
<td>78.97</td>
<td>31816</td>
</tr>
<tr>
<td>11</td>
<td>4.71</td>
<td>6.55</td>
<td>81.91</td>
<td>31916</td>
</tr>
<tr>
<td>12</td>
<td>4.67</td>
<td>6.51</td>
<td>81.23</td>
<td>31756</td>
</tr>
<tr>
<td>13</td>
<td>4.75</td>
<td>6.74</td>
<td>84.06</td>
<td>31804</td>
</tr>
<tr>
<td>14</td>
<td>4.75</td>
<td>6.44</td>
<td>80.19</td>
<td>31552</td>
</tr>
<tr>
<td>15</td>
<td>4.68</td>
<td>6.54</td>
<td>81.71</td>
<td>31888</td>
</tr>
<tr>
<td>16</td>
<td>4.74</td>
<td>6.53</td>
<td>81.4</td>
<td>31540</td>
</tr>
<tr>
<td>17</td>
<td>4.75</td>
<td>6.67</td>
<td>83.04</td>
<td>31928</td>
</tr>
<tr>
<td>18</td>
<td>4.75</td>
<td>6.64</td>
<td>82.78</td>
<td>31908</td>
</tr>
<tr>
<td>19</td>
<td>4.72</td>
<td>6.42</td>
<td>80.18</td>
<td>31824</td>
</tr>
<tr>
<td>20</td>
<td>4.75</td>
<td>6.65</td>
<td>82.65</td>
<td>31840</td>
</tr>
<tr>
<td>21</td>
<td>4.75</td>
<td>6.63</td>
<td>82.75</td>
<td>31828</td>
</tr>
<tr>
<td>22</td>
<td>4.73</td>
<td>6.6</td>
<td>82.39</td>
<td>31840</td>
</tr>
<tr>
<td>23</td>
<td>4.75</td>
<td>6.6</td>
<td>82.45</td>
<td>31852</td>
</tr>
<tr>
<td>24</td>
<td>4.76</td>
<td>6.67</td>
<td>83.11</td>
<td>31748</td>
</tr>
<tr>
<td>25</td>
<td>4.71</td>
<td>6.39</td>
<td>80.22</td>
<td>31612</td>
</tr>
<tr>
<td>26</td>
<td>4.71</td>
<td>6.81</td>
<td>84.89</td>
<td>31764</td>
</tr>
<tr>
<td>27</td>
<td>4.74</td>
<td>6.41</td>
<td>79.76</td>
<td>31748</td>
</tr>
<tr>
<td>28</td>
<td>4.73</td>
<td>6.74</td>
<td>84.02</td>
<td>31856</td>
</tr>
<tr>
<td>29</td>
<td>4.74</td>
<td>6.33</td>
<td>82.11</td>
<td>31668</td>
</tr>
<tr>
<td>30</td>
<td>4.66</td>
<td>6.32</td>
<td>78.56</td>
<td>31868</td>
</tr>
</tbody>
</table>

**AVG**

<table>
<thead>
<tr>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.73</td>
<td>6.53</td>
<td>81.47</td>
<td>31783.33</td>
</tr>
</tbody>
</table>

**STDEV**

<table>
<thead>
<tr>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.027</td>
<td>0.141</td>
<td>1.700</td>
<td>105.335</td>
</tr>
</tbody>
</table>
Table A.7. Performance for 7% Blank Cells and 30×40 Topology

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.52</td>
<td>6.4</td>
<td>80.16</td>
<td>32164</td>
</tr>
<tr>
<td>2</td>
<td>4.58</td>
<td>6.61</td>
<td>81.89</td>
<td>32240</td>
</tr>
<tr>
<td>3</td>
<td>4.53</td>
<td>6.43</td>
<td>80.33</td>
<td>31412</td>
</tr>
<tr>
<td>4</td>
<td>4.54</td>
<td>6.6</td>
<td>82.35</td>
<td>32248</td>
</tr>
<tr>
<td>5</td>
<td>4.53</td>
<td>6.64</td>
<td>82.53</td>
<td>32248</td>
</tr>
<tr>
<td>6</td>
<td>4.58</td>
<td>6.61</td>
<td>82.35</td>
<td>32284</td>
</tr>
<tr>
<td>7</td>
<td>4.51</td>
<td>6.68</td>
<td>83.36</td>
<td>32208</td>
</tr>
<tr>
<td>8</td>
<td>4.53</td>
<td>6.49</td>
<td>80.88</td>
<td>32240</td>
</tr>
<tr>
<td>9</td>
<td>4.5</td>
<td>6.6</td>
<td>82.27</td>
<td>32188</td>
</tr>
<tr>
<td>10</td>
<td>4.53</td>
<td>6.53</td>
<td>83.57</td>
<td>32308</td>
</tr>
<tr>
<td>11</td>
<td>4.56</td>
<td>6.56</td>
<td>80.36</td>
<td>32128</td>
</tr>
<tr>
<td>12</td>
<td>4.54</td>
<td>6.44</td>
<td>78.89</td>
<td>32208</td>
</tr>
<tr>
<td>13</td>
<td>4.55</td>
<td>6.75</td>
<td>84.31</td>
<td>32248</td>
</tr>
<tr>
<td>14</td>
<td>4.53</td>
<td>6.47</td>
<td>81.1</td>
<td>32160</td>
</tr>
<tr>
<td>15</td>
<td>4.56</td>
<td>6.71</td>
<td>83.59</td>
<td>31820</td>
</tr>
<tr>
<td>16</td>
<td>4.53</td>
<td>6.56</td>
<td>81.05</td>
<td>32088</td>
</tr>
<tr>
<td>17</td>
<td>4.53</td>
<td>6.66</td>
<td>83.01</td>
<td>32060</td>
</tr>
<tr>
<td>18</td>
<td>4.53</td>
<td>7.06</td>
<td>87.97</td>
<td>31180</td>
</tr>
<tr>
<td>19</td>
<td>4.54</td>
<td>6.61</td>
<td>82.26</td>
<td>32124</td>
</tr>
<tr>
<td>20</td>
<td>4.56</td>
<td>6.72</td>
<td>83.49</td>
<td>32236</td>
</tr>
<tr>
<td>21</td>
<td>4.52</td>
<td>6.4</td>
<td>79.69</td>
<td>32136</td>
</tr>
<tr>
<td>22</td>
<td>4.57</td>
<td>6.68</td>
<td>84.56</td>
<td>32304</td>
</tr>
<tr>
<td>23</td>
<td>4.58</td>
<td>6.77</td>
<td>84.12</td>
<td>32292</td>
</tr>
<tr>
<td>24</td>
<td>4.5</td>
<td>6.41</td>
<td>79.96</td>
<td>32204</td>
</tr>
<tr>
<td>25</td>
<td>4.56</td>
<td>6.75</td>
<td>84.21</td>
<td>32252</td>
</tr>
<tr>
<td>26</td>
<td>4.52</td>
<td>6.51</td>
<td>81.03</td>
<td>32272</td>
</tr>
<tr>
<td>27</td>
<td>4.51</td>
<td>6.63</td>
<td>82.5</td>
<td>32228</td>
</tr>
<tr>
<td>28</td>
<td>4.52</td>
<td>6.6</td>
<td>81.72</td>
<td>32324</td>
</tr>
<tr>
<td>29</td>
<td>4.56</td>
<td>6.53</td>
<td>81.17</td>
<td>32268</td>
</tr>
<tr>
<td>30</td>
<td>4.51</td>
<td>6.56</td>
<td>81.95</td>
<td>32208</td>
</tr>
<tr>
<td><strong>AVG</strong></td>
<td><strong>4.54</strong></td>
<td><strong>6.60</strong></td>
<td><strong>82.22</strong></td>
<td><strong>32142.67</strong></td>
</tr>
<tr>
<td><strong>STDEV</strong></td>
<td><strong>0.023</strong></td>
<td><strong>0.136</strong></td>
<td><strong>1.818</strong></td>
<td><strong>247.363</strong></td>
</tr>
</tbody>
</table>
Table A.8. Performance for 13% Blank Cells and 30×40 Topology

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.67</td>
<td>6.63</td>
<td>82.69</td>
<td>32112</td>
</tr>
<tr>
<td>2</td>
<td>4.63</td>
<td>6.33</td>
<td>78.93</td>
<td>32236</td>
</tr>
<tr>
<td>3</td>
<td>4.75</td>
<td>6.52</td>
<td>81.41</td>
<td>32256</td>
</tr>
<tr>
<td>4</td>
<td>4.62</td>
<td>6.21</td>
<td>77.27</td>
<td>32156</td>
</tr>
<tr>
<td>5</td>
<td>4.62</td>
<td>6.28</td>
<td>78.27</td>
<td>32256</td>
</tr>
<tr>
<td>6</td>
<td>4.64</td>
<td>6.29</td>
<td>78.43</td>
<td>32472</td>
</tr>
<tr>
<td>7</td>
<td>4.65</td>
<td>6.37</td>
<td>79.41</td>
<td>32188</td>
</tr>
<tr>
<td>8</td>
<td>4.62</td>
<td>6.36</td>
<td>79.31</td>
<td>32116</td>
</tr>
<tr>
<td>9</td>
<td>4.63</td>
<td>6.27</td>
<td>78.03</td>
<td>32464</td>
</tr>
<tr>
<td>10</td>
<td>4.62</td>
<td>6.48</td>
<td>82.9</td>
<td>32316</td>
</tr>
<tr>
<td>11</td>
<td>4.65</td>
<td>6.31</td>
<td>78.56</td>
<td>32200</td>
</tr>
<tr>
<td>12</td>
<td>4.63</td>
<td>6.2</td>
<td>77.34</td>
<td>32256</td>
</tr>
<tr>
<td>13</td>
<td>4.62</td>
<td>6.46</td>
<td>80.71</td>
<td>32184</td>
</tr>
<tr>
<td>14</td>
<td>4.67</td>
<td>6.7</td>
<td>83.51</td>
<td>32244</td>
</tr>
<tr>
<td>15</td>
<td>4.62</td>
<td>6.26</td>
<td>77.79</td>
<td>32292</td>
</tr>
<tr>
<td>16</td>
<td>4.67</td>
<td>6.6</td>
<td>82.27</td>
<td>31736</td>
</tr>
<tr>
<td>17</td>
<td>4.66</td>
<td>6.42</td>
<td>79.85</td>
<td>32144</td>
</tr>
<tr>
<td>18</td>
<td>4.62</td>
<td>6.19</td>
<td>77.25</td>
<td>32164</td>
</tr>
<tr>
<td>19</td>
<td>4.59</td>
<td>6.12</td>
<td>75.91</td>
<td>32208</td>
</tr>
<tr>
<td>20</td>
<td>4.64</td>
<td>6.8</td>
<td>84.7</td>
<td>32276</td>
</tr>
<tr>
<td>21</td>
<td>4.65</td>
<td>6.37</td>
<td>79.12</td>
<td>32352</td>
</tr>
<tr>
<td>22</td>
<td>4.64</td>
<td>6.15</td>
<td>78.49</td>
<td>32236</td>
</tr>
<tr>
<td>23</td>
<td>4.67</td>
<td>6.41</td>
<td>80.26</td>
<td>32956</td>
</tr>
<tr>
<td>24</td>
<td>4.67</td>
<td>6.41</td>
<td>79.29</td>
<td>32216</td>
</tr>
<tr>
<td>25</td>
<td>4.62</td>
<td>6.33</td>
<td>78.83</td>
<td>32312</td>
</tr>
<tr>
<td>26</td>
<td>4.62</td>
<td>6.14</td>
<td>76.44</td>
<td>32192</td>
</tr>
<tr>
<td>27</td>
<td>4.68</td>
<td>6.34</td>
<td>78.88</td>
<td>32236</td>
</tr>
<tr>
<td>28</td>
<td>4.66</td>
<td>6.63</td>
<td>82.66</td>
<td>32232</td>
</tr>
<tr>
<td>29</td>
<td>4.65</td>
<td>6.55</td>
<td>81.43</td>
<td>32204</td>
</tr>
<tr>
<td>30</td>
<td>4.65</td>
<td>6.63</td>
<td>82.67</td>
<td>32228</td>
</tr>
</tbody>
</table>

**AVG** 4.64 6.39 79.75 32248.00

**STDEV** 0.029 0.174 2.222 179.234
Table A.9. Performance for 20% Blank Cells and 30×40 Topology

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.76</td>
<td>6.2</td>
<td>78.1</td>
<td>31544</td>
</tr>
<tr>
<td>2</td>
<td>4.75</td>
<td>6.24</td>
<td>77.76</td>
<td>32372</td>
</tr>
<tr>
<td>3</td>
<td>4.76</td>
<td>6.11</td>
<td>75.5</td>
<td>32608</td>
</tr>
<tr>
<td>4</td>
<td>4.76</td>
<td>6.09</td>
<td>74.77</td>
<td>32232</td>
</tr>
<tr>
<td>5</td>
<td>4.78</td>
<td>6.27</td>
<td>78.12</td>
<td>32372</td>
</tr>
<tr>
<td>6</td>
<td>4.73</td>
<td>6.23</td>
<td>77.06</td>
<td>32236</td>
</tr>
<tr>
<td>7</td>
<td>4.82</td>
<td>6.3</td>
<td>77.93</td>
<td>32272</td>
</tr>
<tr>
<td>8</td>
<td>4.77</td>
<td>6.23</td>
<td>77.79</td>
<td>32264</td>
</tr>
<tr>
<td>9</td>
<td>4.76</td>
<td>6.34</td>
<td>78.25</td>
<td>32180</td>
</tr>
<tr>
<td>10</td>
<td>4.76</td>
<td>6.2</td>
<td>77.14</td>
<td>32232</td>
</tr>
<tr>
<td>11</td>
<td>4.74</td>
<td>6.25</td>
<td>77.85</td>
<td>32228</td>
</tr>
<tr>
<td>12</td>
<td>4.76</td>
<td>6</td>
<td>74.77</td>
<td>32176</td>
</tr>
<tr>
<td>13</td>
<td>4.72</td>
<td>6.18</td>
<td>76.44</td>
<td>32248</td>
</tr>
<tr>
<td>14</td>
<td>4.75</td>
<td>6.35</td>
<td>80.56</td>
<td>32284</td>
</tr>
<tr>
<td>15</td>
<td>4.75</td>
<td>6.13</td>
<td>76.04</td>
<td>32492</td>
</tr>
<tr>
<td>16</td>
<td>4.74</td>
<td>6.27</td>
<td>77.75</td>
<td>32300</td>
</tr>
<tr>
<td>17</td>
<td>4.76</td>
<td>6.19</td>
<td>76.23</td>
<td>32228</td>
</tr>
<tr>
<td>18</td>
<td>4.75</td>
<td>6.1</td>
<td>75.99</td>
<td>32304</td>
</tr>
<tr>
<td>19</td>
<td>4.73</td>
<td>6.29</td>
<td>77.91</td>
<td>32304</td>
</tr>
<tr>
<td>20</td>
<td>4.76</td>
<td>6.38</td>
<td>78.74</td>
<td>32304</td>
</tr>
<tr>
<td>21</td>
<td>4.76</td>
<td>6.08</td>
<td>75.66</td>
<td>32304</td>
</tr>
<tr>
<td>22</td>
<td>4.78</td>
<td>6.28</td>
<td>77.3</td>
<td>32284</td>
</tr>
<tr>
<td>23</td>
<td>4.72</td>
<td>6.32</td>
<td>78.64</td>
<td>32440</td>
</tr>
<tr>
<td>24</td>
<td>4.75</td>
<td>6.27</td>
<td>78.21</td>
<td>32296</td>
</tr>
<tr>
<td>25</td>
<td>4.76</td>
<td>6.09</td>
<td>77.69</td>
<td>32424</td>
</tr>
<tr>
<td>26</td>
<td>4.72</td>
<td>6.13</td>
<td>75.51</td>
<td>32224</td>
</tr>
<tr>
<td>27</td>
<td>4.76</td>
<td>6.23</td>
<td>79.66</td>
<td>32332</td>
</tr>
<tr>
<td>28</td>
<td>4.75</td>
<td>6.82</td>
<td>84.53</td>
<td>32292</td>
</tr>
<tr>
<td>29</td>
<td>4.72</td>
<td>6.26</td>
<td>78.02</td>
<td>32268</td>
</tr>
<tr>
<td>30</td>
<td>4.74</td>
<td>6.27</td>
<td>79.24</td>
<td>32288</td>
</tr>
<tr>
<td>AVG</td>
<td>4.75</td>
<td>6.24</td>
<td>77.64</td>
<td>32277.73</td>
</tr>
<tr>
<td>STDEV</td>
<td>0.021</td>
<td>0.141</td>
<td>1.874</td>
<td>163.364</td>
</tr>
<tr>
<td>Sim Run</td>
<td>Optimal Movements per Thread</td>
<td>Actual Movements per Thread</td>
<td>Search Time (ms)</td>
<td>Memory Utilization (KB)</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------</td>
<td>-----------------------------</td>
<td>------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>1</td>
<td>4.52</td>
<td>6.82</td>
<td>84.77</td>
<td>31908</td>
</tr>
<tr>
<td>2</td>
<td>4.51</td>
<td>6.65</td>
<td>82.71</td>
<td>33944</td>
</tr>
<tr>
<td>3</td>
<td>4.52</td>
<td>6.72</td>
<td>83.96</td>
<td>33128</td>
</tr>
<tr>
<td>4</td>
<td>4.55</td>
<td>6.78</td>
<td>84.67</td>
<td>33072</td>
</tr>
<tr>
<td>5</td>
<td>4.51</td>
<td>6.59</td>
<td>82.16</td>
<td>33108</td>
</tr>
<tr>
<td>6</td>
<td>4.54</td>
<td>6.64</td>
<td>82.87</td>
<td>33076</td>
</tr>
<tr>
<td>7</td>
<td>4.53</td>
<td>6.62</td>
<td>82.63</td>
<td>33160</td>
</tr>
<tr>
<td>8</td>
<td>4.53</td>
<td>6.61</td>
<td>82.33</td>
<td>33104</td>
</tr>
<tr>
<td>9</td>
<td>4.55</td>
<td>6.73</td>
<td>83.28</td>
<td>32996</td>
</tr>
<tr>
<td>10</td>
<td>4.5</td>
<td>6.87</td>
<td>83.54</td>
<td>33160</td>
</tr>
<tr>
<td>11</td>
<td>4.5</td>
<td>6.74</td>
<td>83.93</td>
<td>33160</td>
</tr>
<tr>
<td>12</td>
<td>4.55</td>
<td>6.76</td>
<td>82.97</td>
<td>33112</td>
</tr>
<tr>
<td>13</td>
<td>4.54</td>
<td>6.93</td>
<td>86.23</td>
<td>33084</td>
</tr>
<tr>
<td>14</td>
<td>4.52</td>
<td>6.69</td>
<td>83.47</td>
<td>32936</td>
</tr>
<tr>
<td>15</td>
<td>4.53</td>
<td>6.74</td>
<td>83.8</td>
<td>32869</td>
</tr>
<tr>
<td>16</td>
<td>4.53</td>
<td>6.61</td>
<td>82.07</td>
<td>33088</td>
</tr>
<tr>
<td>17</td>
<td>4.51</td>
<td>6.84</td>
<td>85.06</td>
<td>33336</td>
</tr>
<tr>
<td>18</td>
<td>4.53</td>
<td>6.8</td>
<td>84.9</td>
<td>32904</td>
</tr>
<tr>
<td>19</td>
<td>4.5</td>
<td>6.59</td>
<td>82.24</td>
<td>32988</td>
</tr>
<tr>
<td>20</td>
<td>4.54</td>
<td>6.81</td>
<td>84.7</td>
<td>32960</td>
</tr>
<tr>
<td>21</td>
<td>4.52</td>
<td>6.92</td>
<td>86.15</td>
<td>32976</td>
</tr>
<tr>
<td>22</td>
<td>4.52</td>
<td>6.79</td>
<td>84.12</td>
<td>32568</td>
</tr>
<tr>
<td>23</td>
<td>4.53</td>
<td>6.78</td>
<td>84.39</td>
<td>31968</td>
</tr>
<tr>
<td>24</td>
<td>4.5</td>
<td>6.6</td>
<td>81.73</td>
<td>33016</td>
</tr>
<tr>
<td>25</td>
<td>4.54</td>
<td>6.8</td>
<td>84.61</td>
<td>33236</td>
</tr>
<tr>
<td>26</td>
<td>4.56</td>
<td>6.83</td>
<td>84.78</td>
<td>33136</td>
</tr>
<tr>
<td>27</td>
<td>4.57</td>
<td>6.87</td>
<td>85.18</td>
<td>33004</td>
</tr>
<tr>
<td>28</td>
<td>4.51</td>
<td>6.73</td>
<td>83.63</td>
<td>31872</td>
</tr>
<tr>
<td>29</td>
<td>4.48</td>
<td>6.7</td>
<td>83.53</td>
<td>32252</td>
</tr>
<tr>
<td>30</td>
<td>4.53</td>
<td>6.55</td>
<td>81.45</td>
<td>32912</td>
</tr>
</tbody>
</table>

**AVG** 4.53 6.737 83.73 32934.43

**STDEV** 0.020 0.103 1.229 425.051
Table A.11. Performance for 13% Blank Cells and 40×40 Topology

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.65</td>
<td>6.88</td>
<td>85.77</td>
<td>31996</td>
</tr>
<tr>
<td>2</td>
<td>4.63</td>
<td>6.77</td>
<td>84.45</td>
<td>31956</td>
</tr>
<tr>
<td>3</td>
<td>4.66</td>
<td>6.64</td>
<td>82.45</td>
<td>32980</td>
</tr>
<tr>
<td>4</td>
<td>4.62</td>
<td>6.5</td>
<td>81.13</td>
<td>32484</td>
</tr>
<tr>
<td>5</td>
<td>4.62</td>
<td>6.56</td>
<td>81.99</td>
<td>31904</td>
</tr>
<tr>
<td>6</td>
<td>4.64</td>
<td>6.64</td>
<td>82.72</td>
<td>33008</td>
</tr>
<tr>
<td>7</td>
<td>4.63</td>
<td>6.37</td>
<td>79.51</td>
<td>33084</td>
</tr>
<tr>
<td>8</td>
<td>4.64</td>
<td>6.61</td>
<td>82.33</td>
<td>32908</td>
</tr>
<tr>
<td>9</td>
<td>4.64</td>
<td>6.61</td>
<td>82.26</td>
<td>33036</td>
</tr>
<tr>
<td>10</td>
<td>4.59</td>
<td>6.44</td>
<td>80.28</td>
<td>31920</td>
</tr>
<tr>
<td>11</td>
<td>4.62</td>
<td>6.5</td>
<td>81.16</td>
<td>31084</td>
</tr>
<tr>
<td>12</td>
<td>4.64</td>
<td>6.57</td>
<td>82.14</td>
<td>31948</td>
</tr>
<tr>
<td>13</td>
<td>4.64</td>
<td>6.62</td>
<td>82.48</td>
<td>31980</td>
</tr>
<tr>
<td>14</td>
<td>4.61</td>
<td>6.48</td>
<td>80.75</td>
<td>31888</td>
</tr>
<tr>
<td>15</td>
<td>4.6</td>
<td>6.66</td>
<td>83.18</td>
<td>32312</td>
</tr>
<tr>
<td>16</td>
<td>4.61</td>
<td>6.46</td>
<td>80.56</td>
<td>31888</td>
</tr>
<tr>
<td>17</td>
<td>4.62</td>
<td>6.62</td>
<td>82.81</td>
<td>32952</td>
</tr>
<tr>
<td>18</td>
<td>4.64</td>
<td>6.58</td>
<td>81.94</td>
<td>32040</td>
</tr>
<tr>
<td>19</td>
<td>4.65</td>
<td>6.64</td>
<td>82.83</td>
<td>31920</td>
</tr>
<tr>
<td>20</td>
<td>4.63</td>
<td>6.63</td>
<td>82.74</td>
<td>32248</td>
</tr>
<tr>
<td>21</td>
<td>4.66</td>
<td>6.71</td>
<td>83.77</td>
<td>31916</td>
</tr>
<tr>
<td>22</td>
<td>4.59</td>
<td>6.42</td>
<td>80.21</td>
<td>33024</td>
</tr>
<tr>
<td>23</td>
<td>4.66</td>
<td>6.65</td>
<td>82.95</td>
<td>31980</td>
</tr>
<tr>
<td>24</td>
<td>4.62</td>
<td>6.46</td>
<td>80.7</td>
<td>32268</td>
</tr>
<tr>
<td>25</td>
<td>4.64</td>
<td>6.63</td>
<td>82.7</td>
<td>31976</td>
</tr>
<tr>
<td>26</td>
<td>4.62</td>
<td>6.78</td>
<td>84.55</td>
<td>31920</td>
</tr>
<tr>
<td>27</td>
<td>4.64</td>
<td>7</td>
<td>87.83</td>
<td>31888</td>
</tr>
<tr>
<td>28</td>
<td>4.65</td>
<td>6.75</td>
<td>93.99</td>
<td>33060</td>
</tr>
<tr>
<td>29</td>
<td>4.63</td>
<td>6.54</td>
<td>81.29</td>
<td>32984</td>
</tr>
<tr>
<td>30</td>
<td>4.63</td>
<td>6.75</td>
<td>84.11</td>
<td>31940</td>
</tr>
<tr>
<td>AVG</td>
<td>4.63</td>
<td>6.62</td>
<td>82.85</td>
<td>32283.07</td>
</tr>
<tr>
<td>STDEV</td>
<td>0.018</td>
<td>0.136</td>
<td>2.682</td>
<td>518.990</td>
</tr>
<tr>
<td>Sim Run</td>
<td>Optimal Movements per Thread</td>
<td>Actual Movements per Thread</td>
<td>Search Time (ms)</td>
<td>Memory Utilization (KB)</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------</td>
<td>-----------------------------</td>
<td>------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>1</td>
<td>4.71</td>
<td>6.29</td>
<td>78.46</td>
<td>31872</td>
</tr>
<tr>
<td>2</td>
<td>4.69</td>
<td>6.49</td>
<td>80.99</td>
<td>31908</td>
</tr>
<tr>
<td>3</td>
<td>4.7</td>
<td>6.29</td>
<td>78.41</td>
<td>32580</td>
</tr>
<tr>
<td>4</td>
<td>4.74</td>
<td>6.86</td>
<td>85.22</td>
<td>33008</td>
</tr>
<tr>
<td>5</td>
<td>4.73</td>
<td>6.47</td>
<td>80.56</td>
<td>32964</td>
</tr>
<tr>
<td>6</td>
<td>4.71</td>
<td>6.28</td>
<td>79.94</td>
<td>32320</td>
</tr>
<tr>
<td>7</td>
<td>4.75</td>
<td>6.41</td>
<td>79.87</td>
<td>31984</td>
</tr>
<tr>
<td>8</td>
<td>4.71</td>
<td>6.34</td>
<td>78.3</td>
<td>32196</td>
</tr>
<tr>
<td>9</td>
<td>4.75</td>
<td>6.61</td>
<td>81.69</td>
<td>32980</td>
</tr>
<tr>
<td>10</td>
<td>4.74</td>
<td>6.5</td>
<td>80.57</td>
<td>33056</td>
</tr>
<tr>
<td>11</td>
<td>4.73</td>
<td>6.64</td>
<td>82.72</td>
<td>32492</td>
</tr>
<tr>
<td>12</td>
<td>4.72</td>
<td>6.5</td>
<td>80.83</td>
<td>32016</td>
</tr>
<tr>
<td>13</td>
<td>4.73</td>
<td>6.57</td>
<td>81.95</td>
<td>32748</td>
</tr>
<tr>
<td>14</td>
<td>4.72</td>
<td>6.24</td>
<td>77.51</td>
<td>31960</td>
</tr>
<tr>
<td>15</td>
<td>4.73</td>
<td>6.69</td>
<td>83.13</td>
<td>33048</td>
</tr>
<tr>
<td>16</td>
<td>4.74</td>
<td>6.5</td>
<td>81.21</td>
<td>32252</td>
</tr>
<tr>
<td>17</td>
<td>4.73</td>
<td>6.31</td>
<td>78.35</td>
<td>32264</td>
</tr>
<tr>
<td>18</td>
<td>4.75</td>
<td>6.42</td>
<td>79.78</td>
<td>32540</td>
</tr>
<tr>
<td>19</td>
<td>4.75</td>
<td>6.4</td>
<td>79.18</td>
<td>33040</td>
</tr>
<tr>
<td>20</td>
<td>4.76</td>
<td>6.63</td>
<td>82.72</td>
<td>32068</td>
</tr>
<tr>
<td>21</td>
<td>4.73</td>
<td>6.49</td>
<td>80.59</td>
<td>31992</td>
</tr>
<tr>
<td>22</td>
<td>4.7</td>
<td>6.33</td>
<td>79.02</td>
<td>32024</td>
</tr>
<tr>
<td>23</td>
<td>4.73</td>
<td>6.5</td>
<td>81.14</td>
<td>31984</td>
</tr>
<tr>
<td>24</td>
<td>4.74</td>
<td>6.32</td>
<td>78.6</td>
<td>31952</td>
</tr>
<tr>
<td>25</td>
<td>4.71</td>
<td>6.66</td>
<td>82.44</td>
<td>31928</td>
</tr>
<tr>
<td>26</td>
<td>4.75</td>
<td>6.39</td>
<td>79.68</td>
<td>32276</td>
</tr>
<tr>
<td>27</td>
<td>4.74</td>
<td>6.42</td>
<td>80.03</td>
<td>31972</td>
</tr>
<tr>
<td>28</td>
<td>4.73</td>
<td>6.48</td>
<td>80.56</td>
<td>33132</td>
</tr>
<tr>
<td>29</td>
<td>4.74</td>
<td>6.63</td>
<td>82.29</td>
<td>31924</td>
</tr>
<tr>
<td>30</td>
<td>4.74</td>
<td>6.33</td>
<td>78.88</td>
<td>32508</td>
</tr>
<tr>
<td><strong>AVG</strong></td>
<td><strong>4.73</strong></td>
<td><strong>6.47</strong></td>
<td><strong>80.49</strong></td>
<td><strong>32366.27</strong></td>
</tr>
<tr>
<td><strong>STDEV</strong></td>
<td><strong>0.017</strong></td>
<td><strong>0.145</strong></td>
<td><strong>1.732</strong></td>
<td><strong>429.807</strong></td>
</tr>
</tbody>
</table>
Table A.13. Performance for High Concurrent Demand
(13% Blank Cells and 40×40 Topology)

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.61</td>
<td>7.45</td>
<td>92.92</td>
<td>36265</td>
</tr>
<tr>
<td>2</td>
<td>4.6</td>
<td>7.27</td>
<td>90.63</td>
<td>36548</td>
</tr>
<tr>
<td>3</td>
<td>4.65</td>
<td>7.64</td>
<td>94.05</td>
<td>36672</td>
</tr>
<tr>
<td>4</td>
<td>4.59</td>
<td>7.16</td>
<td>88.3</td>
<td>36254</td>
</tr>
<tr>
<td>5</td>
<td>4.59</td>
<td>7.48</td>
<td>92.95</td>
<td>36076</td>
</tr>
<tr>
<td>6</td>
<td>4.63</td>
<td>7.62</td>
<td>93.98</td>
<td>36184</td>
</tr>
<tr>
<td>7</td>
<td>4.58</td>
<td>7.5</td>
<td>92.97</td>
<td>35508</td>
</tr>
<tr>
<td>8</td>
<td>4.61</td>
<td>8.04</td>
<td>97.52</td>
<td>36432</td>
</tr>
<tr>
<td>9</td>
<td>4.6</td>
<td>7.62</td>
<td>93.97</td>
<td>36324</td>
</tr>
<tr>
<td>10</td>
<td>4.57</td>
<td>7.56</td>
<td>93.26</td>
<td>36676</td>
</tr>
<tr>
<td>11</td>
<td>4.6</td>
<td>7.61</td>
<td>93.94</td>
<td>36568</td>
</tr>
<tr>
<td>12</td>
<td>4.55</td>
<td>7.62</td>
<td>93.96</td>
<td>36572</td>
</tr>
<tr>
<td>13</td>
<td>4.6</td>
<td>7.59</td>
<td>93.65</td>
<td>36284</td>
</tr>
<tr>
<td>14</td>
<td>4.61</td>
<td>8.16</td>
<td>98.23</td>
<td>36264</td>
</tr>
<tr>
<td>15</td>
<td>4.57</td>
<td>7.41</td>
<td>92.76</td>
<td>36464</td>
</tr>
<tr>
<td>16</td>
<td>4.61</td>
<td>7.68</td>
<td>94.12</td>
<td>35808</td>
</tr>
<tr>
<td>17</td>
<td>4.6</td>
<td>7.81</td>
<td>95.03</td>
<td>36048</td>
</tr>
<tr>
<td>18</td>
<td>4.62</td>
<td>7.57</td>
<td>93.32</td>
<td>36268</td>
</tr>
<tr>
<td>19</td>
<td>4.57</td>
<td>7.51</td>
<td>92.97</td>
<td>36424</td>
</tr>
<tr>
<td>20</td>
<td>4.59</td>
<td>7.51</td>
<td>92.96</td>
<td>35588</td>
</tr>
<tr>
<td>21</td>
<td>4.62</td>
<td>7.79</td>
<td>94.91</td>
<td>36152</td>
</tr>
<tr>
<td>22</td>
<td>4.58</td>
<td>7.47</td>
<td>92.98</td>
<td>36416</td>
</tr>
<tr>
<td>23</td>
<td>4.64</td>
<td>8.28</td>
<td>99.03</td>
<td>36820</td>
</tr>
<tr>
<td>24</td>
<td>4.56</td>
<td>7.42</td>
<td>92.87</td>
<td>35928</td>
</tr>
<tr>
<td>25</td>
<td>4.57</td>
<td>7.44</td>
<td>92.9</td>
<td>36340</td>
</tr>
<tr>
<td>26</td>
<td>4.61</td>
<td>7.68</td>
<td>94.05</td>
<td>36220</td>
</tr>
<tr>
<td>27</td>
<td>4.6</td>
<td>7.71</td>
<td>94.36</td>
<td>36864</td>
</tr>
<tr>
<td>28</td>
<td>4.63</td>
<td>7.52</td>
<td>93.01</td>
<td>36180</td>
</tr>
<tr>
<td>29</td>
<td>4.62</td>
<td>7.8</td>
<td>94.93</td>
<td>35764</td>
</tr>
<tr>
<td>30</td>
<td>4.58</td>
<td>7.88</td>
<td>95.43</td>
<td>36324</td>
</tr>
<tr>
<td>AVG</td>
<td>4.60</td>
<td>7.63</td>
<td>93.87</td>
<td>36274.50</td>
</tr>
<tr>
<td>STDEV</td>
<td>0.024</td>
<td>0.239</td>
<td>2.004</td>
<td>327.460</td>
</tr>
</tbody>
</table>
Table A.14. Performance for 1% Immobilized Cells
(13% Blank Cells and 40×40 Topology)

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.68</td>
<td>11.11</td>
<td>136.55</td>
<td>32792</td>
</tr>
<tr>
<td>2</td>
<td>4.6</td>
<td>8.45</td>
<td>108.34</td>
<td>33612</td>
</tr>
<tr>
<td>3</td>
<td>4.57</td>
<td>8.62</td>
<td>110.62</td>
<td>32504</td>
</tr>
<tr>
<td>4</td>
<td>4.62</td>
<td>9.5</td>
<td>126.25</td>
<td>33056</td>
</tr>
<tr>
<td>5</td>
<td>4.64</td>
<td>8.36</td>
<td>107.33</td>
<td>32676</td>
</tr>
<tr>
<td>6</td>
<td>4.58</td>
<td>8.76</td>
<td>114.02</td>
<td>32568</td>
</tr>
<tr>
<td>7</td>
<td>4.58</td>
<td>9.74</td>
<td>127.32</td>
<td>32212</td>
</tr>
<tr>
<td>8</td>
<td>4.64</td>
<td>8.35</td>
<td>107.25</td>
<td>32681</td>
</tr>
<tr>
<td>9</td>
<td>4.64</td>
<td>8.61</td>
<td>110.03</td>
<td>32732</td>
</tr>
<tr>
<td>10</td>
<td>4.63</td>
<td>8.46</td>
<td>108.65</td>
<td>32128</td>
</tr>
<tr>
<td>11</td>
<td>4.6</td>
<td>8.47</td>
<td>108.7</td>
<td>32076</td>
</tr>
<tr>
<td>12</td>
<td>4.62</td>
<td>7.06</td>
<td>86.89</td>
<td>32356</td>
</tr>
<tr>
<td>13</td>
<td>4.64</td>
<td>8.36</td>
<td>107.32</td>
<td>33132</td>
</tr>
<tr>
<td>14</td>
<td>4.59</td>
<td>8.59</td>
<td>110.25</td>
<td>32064</td>
</tr>
<tr>
<td>15</td>
<td>4.59</td>
<td>10.27</td>
<td>132.21</td>
<td>33280</td>
</tr>
<tr>
<td>16</td>
<td>4.56</td>
<td>8.47</td>
<td>108.6</td>
<td>32486</td>
</tr>
<tr>
<td>17</td>
<td>4.62</td>
<td>8.76</td>
<td>114.06</td>
<td>32755</td>
</tr>
<tr>
<td>18</td>
<td>4.61</td>
<td>8.54</td>
<td>109.89</td>
<td>32474</td>
</tr>
<tr>
<td>19</td>
<td>4.57</td>
<td>8.77</td>
<td>114.08</td>
<td>33124</td>
</tr>
<tr>
<td>20</td>
<td>4.64</td>
<td>8.45</td>
<td>108.62</td>
<td>32487</td>
</tr>
<tr>
<td>21</td>
<td>4.61</td>
<td>8.58</td>
<td>110.02</td>
<td>32566</td>
</tr>
<tr>
<td>22</td>
<td>4.61</td>
<td>8.79</td>
<td>114.12</td>
<td>32486</td>
</tr>
<tr>
<td>23</td>
<td>4.57</td>
<td>8.63</td>
<td>110.65</td>
<td>33147</td>
</tr>
<tr>
<td>24</td>
<td>4.62</td>
<td>8.31</td>
<td>99.17</td>
<td>32245</td>
</tr>
<tr>
<td>25</td>
<td>4.63</td>
<td>8.46</td>
<td>108.65</td>
<td>32697</td>
</tr>
<tr>
<td>26</td>
<td>4.62</td>
<td>8.66</td>
<td>110.7</td>
<td>32612</td>
</tr>
<tr>
<td>27</td>
<td>4.58</td>
<td>8.55</td>
<td>110.32</td>
<td>32784</td>
</tr>
<tr>
<td>28</td>
<td>4.66</td>
<td>8.96</td>
<td>118.03</td>
<td>33012</td>
</tr>
<tr>
<td>29</td>
<td>4.62</td>
<td>8.47</td>
<td>108.53</td>
<td>32672</td>
</tr>
<tr>
<td>30</td>
<td>4.59</td>
<td>8.5</td>
<td>109.21</td>
<td>32546</td>
</tr>
<tr>
<td><strong>AVG</strong></td>
<td><strong>4.61</strong></td>
<td><strong>8.72</strong></td>
<td><strong>111.88</strong></td>
<td><strong>32665.40</strong></td>
</tr>
<tr>
<td><strong>STDEV</strong></td>
<td><strong>0.029</strong></td>
<td><strong>0.690</strong></td>
<td><strong>9.241</strong></td>
<td><strong>367.74</strong></td>
</tr>
</tbody>
</table>
Table A.15. Performance for High Concurrent Demand and 1% Immobilized Cells
(13% Blank Cells and 40×40 Topology)

<table>
<thead>
<tr>
<th>Sim Run</th>
<th>Optimal Movements per Thread</th>
<th>Actual Movements per Thread</th>
<th>Search Time (ms)</th>
<th>Memory Utilization (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.48</td>
<td>8.48</td>
<td>108.56</td>
<td>35972</td>
</tr>
<tr>
<td>2</td>
<td>4.53</td>
<td>8.63</td>
<td>110.08</td>
<td>35704</td>
</tr>
<tr>
<td>3</td>
<td>4.57</td>
<td>9.8</td>
<td>128.4</td>
<td>35772</td>
</tr>
<tr>
<td>4</td>
<td>4.57</td>
<td>8.78</td>
<td>113.65</td>
<td>36084</td>
</tr>
<tr>
<td>5</td>
<td>4.54</td>
<td>8.72</td>
<td>112.96</td>
<td>35924</td>
</tr>
<tr>
<td>6</td>
<td>4.52</td>
<td>8.57</td>
<td>109.02</td>
<td>35892</td>
</tr>
<tr>
<td>7</td>
<td>4.53</td>
<td>8.1</td>
<td>97.62</td>
<td>36448</td>
</tr>
<tr>
<td>8</td>
<td>4.57</td>
<td>8.75</td>
<td>113.25</td>
<td>35316</td>
</tr>
<tr>
<td>9</td>
<td>4.55</td>
<td>8.34</td>
<td>105.32</td>
<td>36040</td>
</tr>
<tr>
<td>10</td>
<td>4.57</td>
<td>8.33</td>
<td>105.25</td>
<td>36532</td>
</tr>
<tr>
<td>11</td>
<td>4.55</td>
<td>8.05</td>
<td>97.21</td>
<td>35340</td>
</tr>
<tr>
<td>12</td>
<td>4.58</td>
<td>8.78</td>
<td>113.52</td>
<td>36380</td>
</tr>
<tr>
<td>13</td>
<td>4.56</td>
<td>8.98</td>
<td>115.32</td>
<td>36192</td>
</tr>
<tr>
<td>14</td>
<td>4.57</td>
<td>8.29</td>
<td>104.59</td>
<td>36428</td>
</tr>
<tr>
<td>15</td>
<td>4.56</td>
<td>9</td>
<td>120.36</td>
<td>35960</td>
</tr>
<tr>
<td>16</td>
<td>4.53</td>
<td>7.89</td>
<td>93.26</td>
<td>36350</td>
</tr>
<tr>
<td>17</td>
<td>4.57</td>
<td>8.17</td>
<td>97.31</td>
<td>36336</td>
</tr>
<tr>
<td>18</td>
<td>4.55</td>
<td>9.36</td>
<td>122.36</td>
<td>36824</td>
</tr>
<tr>
<td>19</td>
<td>4.57</td>
<td>9.4</td>
<td>122.69</td>
<td>36228</td>
</tr>
<tr>
<td>20</td>
<td>4.58</td>
<td>8.06</td>
<td>97.23</td>
<td>36244</td>
</tr>
<tr>
<td>21</td>
<td>4.51</td>
<td>8.24</td>
<td>102.24</td>
<td>36504</td>
</tr>
<tr>
<td>22</td>
<td>4.54</td>
<td>8.68</td>
<td>110.07</td>
<td>36556</td>
</tr>
<tr>
<td>23</td>
<td>4.55</td>
<td>9.28</td>
<td>122.69</td>
<td>36728</td>
</tr>
<tr>
<td>24</td>
<td>4.56</td>
<td>8.29</td>
<td>104.51</td>
<td>35700</td>
</tr>
<tr>
<td>25</td>
<td>4.58</td>
<td>8.94</td>
<td>118.32</td>
<td>35524</td>
</tr>
<tr>
<td>26</td>
<td>4.56</td>
<td>8.23</td>
<td>103.86</td>
<td>36184</td>
</tr>
<tr>
<td>27</td>
<td>4.54</td>
<td>9.17</td>
<td>120.79</td>
<td>36008</td>
</tr>
<tr>
<td>28</td>
<td>4.58</td>
<td>8.33</td>
<td>104.09</td>
<td>35860</td>
</tr>
<tr>
<td>29</td>
<td>4.57</td>
<td>8.88</td>
<td>111.98</td>
<td>36508</td>
</tr>
<tr>
<td>30</td>
<td>4.58</td>
<td>8.14</td>
<td>98.67</td>
<td>35916</td>
</tr>
</tbody>
</table>

**AVG** 4.55 8.62 109.51 36108.13

**STDEV** 0.024 0.472 9.221 381.479
UML Diagrams

The UML diagrams for the system include a Use Case diagram, a Class Diagram and a Sequence Diagram as presented in Figures A-1 through A-4.

![UML Use Case Diagram](image_url)

Figure A-1. UML Use Case Diagram
Figure A-2. UML Class Diagram
Figure A-3. UML Sequence Diagram for Retrieval Request

Figure A-4. UML Sequence Diagram for Storage Request
Pseudocode for Auxiliary Functions

The terminologies and functions invoked by \textit{dstarLite()} are defined as follows:

- \( S \) is the set of all vertexes;
- \( s, u, s' \) are certain vertexes in \( S \);
- \( s_{\text{start}} \) is the start vertex of the problem;
- \( s_{\text{goal}} \) is the goal vertex of the problem;
- \( s_{\text{curr}} \) is the vertex indicating the current location, and it is a global value being used in the entire D* Lite search process;
- \( c(s,s') \) is the cost of moving from \( s \) to \( s' \);
- \( g(s) \) is the cost from \( s_{\text{goal}} \) to \( s \);
- \( \text{rhs} \) is a value where
  \[
  \text{rhs}(s) = \begin{cases} 
  0, & \text{if } s = s_{\text{goal}} \\
  \min_{s' \in \text{pred}(s)} \{ c(s,s') + g(s') \}, & \text{otherwise}
  \end{cases}
  \]
- \( h(s,s') \) is the estimate cost from vertex \( s \) to \( s' \). It needs to satisfy
  \[
  h(s_{\text{start}},s_{\text{start}}) = 0 \quad \text{and} \quad h(s_{\text{start}},s) \leq h(s_{\text{start}},s') + c(s,s');
  \]
- \( \text{Open} \) is a priority list of vertices being evaluated with the value \textit{key}, and
- The priority of vertex \( s \) is \( \text{key}(s)=[k_1(s),k_2(s)] \), which has two components where
  \[
  k_1 = \min(g(s),\text{rhs}(s)) + h(s_{\text{start}},s) \quad \text{and} \quad k_2 = \min(g(s),\text{rhs}(s)).
  \]
- \( \text{Open.add}(s, k) \) adds state \( s \) into the priority list with the priority value \( k \);
- \( \text{Open.remove}(s) \) removes the state \( s \) from the priority list;
- *Open.top()* returns the smallest key value among all the states in the priority list. If the list is empty, it will return \([\infty, \infty]\);

- *Open.pop()* deletes the state, which has the smallest key value, from the priority list, and return the state;

- *succ* \((s)\) returns a list with all the successors of \(s\);

- *pred* \((s)\) return a list with all the predecessors of \(s\);

- Compare key values: The priorities are compared according to a lexicographic ordering, such as \(k(s)\) is smaller or equal than \(k(s')\), denoted as \(k(s) \leq k(s')\), iff either \(k_1(s) < k_1(s')\) or \(k_1(s) = k_1(s') \text{ and } k_2(s) \leq k_2(s')\).
The pseudocode for other functions being called in the D* Lite main function is shown in Figure A-5.

```plaintext
function calculateKey(s)
    return \[\min(g(s), rhs(s)) + h(s_{start}, s), \min(g(s), rhs(s))\];

function init()
    Open.clear();
    s_{curr} = s_{start};
    for all s \in S \quad g(s) = rhs(s) = \infty;
    rhs(s_{goal}) = 0;
    Open.add(s_{goal}, calculateKey(s_{goal}));

function update(u)
    if (u \neq s_{goal})
        rhs(u) = \min_{u' \in succ(u)} (c(u, u') + g(u'));
    if (u \in Open)
        Open.remove(u);
    if (g(u) \neq rhs(u))
        Open.add(u, calculateKey(u));

function computePath()
    while (Open.top() \neq calculateKey(s_{curr}) or rhs(s_{curr}) \neq g(s_{curr}))
        u = Open.pop();
        if (g(u) > rhs(u))
            if (g(u) = rhs(u))
                for all u' \in pred(u) \quad update(u');
            else
                g(u) = \infty;
                for all u' \in pred(u) \cup \{u\} \quad update(u');
```

Figure A-5. Pseudocode for Support Functions Being Used in D* Lite Implementation

The function `calculateKey(s)` calculates the key value for the state `s`; function `init()` initialize all the useful variables and lists to an original status; function `update(u)` updates the `g` value and `rhs` value of state `u` after each visit of `u`; function `computePath()` calculates the shortest path from the current state to the goal state.
The terminologies and functions invoked by the unblock procedure are defined as follow:

- $s_1$ is the target vertex to be unblocked;
- $s'$ is one of the successors of $s$;
- $s_{goal}$ is the D* Lite goal state;
- $u'$ is an element in blanks list;
- $u''$ is an element in destinations list;
- $Open$ is a list of vertexes which are being expanded;
- $Closed$ is a list of vertexes which have been expanded;
- $fringe$ is a list of vertexes which are the being evaluated;
- $blanks$ is a list of blank vertexes which are the closest to the start state;
- $destinations$ is a list of vertexes which are next to the target vertex and have the lowest Manhattan value to the D* Lite goal;
- $list.add(s)$ adds state $s$ into the list;
- $list.clear()$ clears the list;
The pseudocode for other functions being invoked by the unblock procedure is presented in Figure A-6.

```plaintext
function init()
    Open.clear();
    Closed.clear();
    fringe.clear();
    blanks.clear();
    destinations.clear();
    s_blank = NULL;
    s_desti = NULL;
function findDestinations(s)
    min_h = min_{s \in \text{succ}(s)} h(s', s_{goal});
    for all s' \in \text{succ}(s)
        if (h(s', s_{goal}) == min_h) destinations.add(s');
function match()
    for all s' \in \text{blanks} and u' \in \text{destinations}
        find suitable s' and u' with \min(h(u', s_{goal}) + h(s', u'));
    s_blank = s';
    s_desti = u';
```

Figure A-6. Pseudocode for Other Functions Being Used in Unblock Procedure

The function `init()` clears all the lists which will be used in unblock procedure; function `findDestination(s)` will find the successor(s) of state `s` with the shortest Manhattan distance to the D* Lite goal state; function `match()` will find a suitable pair of blank cell and destination for the blank cell throughout the lists `blanks` and `destinations`. 
Simulation Code

The simulation was developed in Java language. The simulation program contains 7 Java files: view.java, model.java, controller.java, bg.java, dstar.java, node.java, unblocker.java.

To run Java programs, you need to download and install Java(TM) Developer’s Kit (JDK) v1.6.0-24 to prepare for an environment on your computer. After installing JDK, you need to follow the instructions to run the simulation (use Windows(R) XP(TM) for example):

1. Go to the command line mode of your operating system (windows start  run  cmd);

2. Enter the directory that contains those 7 Java files using command “cd “. For example, if the path of the folder is “C:\simulation”, then type command “cd C:\simulation”. Note that if the target directory is “D:\simulation” and the current directory is “C:\”which on the different hard disk partitions, you will first need to go to D: using the command ”d:”, and then “cd D:\simulation”;

3. To compile a Java file, use the “javac finaname” command. The system will compile the file and other files which are referenced by it. The main function is in the “view.java” file, so type the command ”javac view.java”, all the 7 files will be automatically compiled. After compiling, each Java file will generate a corresponding “.class” file, which is ready to run.

4. The “java classfilename” command is used for running the Java program. To run the simulation, type ”java view”, and the program will be started.
Also, using IDE (Integrated Development Environment) will make the compiling and running processes easier. There are several IDEs for Java language such as Eclipse and NetBeans. First download and install the IDE (any version), and create a new Java project. Create 7 java classes named view, model, controller, bg, dstar, node and unblocker. Copy the corresponding code to each class, find and click “run” button on the menu bar. The program will be automatically compiled and run.

The Java code in all seven files is presented next.

**view.java:**

```java
//This class is for building the main interface
import java.awt.Color;
import javax.swing.*;
public class view extends JFrame{

    public JButton start = new JButton("Start");
    public JButton stop = new JButton("Quit");
    public JLabel[][] map;

    // Class constructor, building the interface
    public view(){
        model m = new model();
        controller c = new controller(m, this);
        JPanel con = new JPanel();
        con.setLayout(null);
        int[][] temp=m.map;

        // construct the parking layout
        map= new JLabel[20][20];
        for(int i=0;i<20;i++){
            for(int j=0;j<20;j++){
                map[i][j]=new JLabel("",JLabel.CENTER);
            }
        }
    }
}
```

map[i][j].setBounds(50+j*50, 50+i*25, 50, 25);
map[i][j].setOpaque(true);
map[i][j].setBorder(BorderFactory.createLineBorder(Color.black, 1));

if(temp[i][j]==10){
    map[i][j].setBackground(Color.black);
}
con.add(map[i][j]);
}

//set the loading areas
map[4][3].setText("load");
map[4][6].setText("load");
map[4][13].setText("load");
map[4][16].setText("load");
map[15][3].setText("load");
map[15][6].setText("load");
map[15][13].setText("load");
map[15][16].setText("load");

start.setBounds(300,600,100,25);
stop.setBounds(750,600,100,25);
faster.setBounds(1060, 200, 100, 25);
slower.setBounds(1060, 350, 100, 25);
con.add(start);
con.add(stop);
con.add(slower);
con.add(faster);
start.addActionListener(c);
stop.addActionListener(c);
faster.addActionListener(c);
slower.addActionListener(c);
this.setContentPane(con);
this.pack();

//main method in the program
public static void main(String args[]){
    JFrame window= new view();
    window.setSize(1200,700);
    window.setTitle("Parking 20X20");
    }
windowsetDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
window.setVisible(true);
model.java:

// defining and keeping all the data structures being used
import java.util.ArrayList;

public class model {
    public int[][] map; //the status of the parking layout
    //10 is elevator, 2 is a reserved
    //cell, 0 is an available cell, 1
    //is an occupied cell
    public int speed; //simulate moving time
    public int num; //number of cars
    public int max; //max capacity
    public double bp; //blank percentage
    public int[][] pri; //priorities of cells
    public int[][] zone; //zone numbers of cells
    public int steps; //total steps
    public int op; //total steps of optimal paths
    public int numOfThreads; //total numbers of threads
    public long searchTime; //total search time
    public int[] tpz; //number of threads per zone
    public int activeThreads; //number of active thread running
    //concurrently
    public int atMax; //Maximum number of threads running
    //concurrently

    //class constructor, initialize all defined data
    public model(){
        map= new int[20][20];
        pri= new int[20][20];
        zone= new int[20][20];
        speed=50;
        num=0;
        bp=0.07; //7% of blank cell
        max= (int)((400- 400/50*3)*(1-bp));
        steps=0;
        op=0;
        numOfThreads=0;
        searchTime=0;
        activeThreads=0;
        atMax=0;
        tpz= new int[8];
for(int i=0;i<=7;i++)
    tpz[i]=0;

    //set the locations of the elevators
    for(int i=4;i<=5;i++){
        map[4][i]=10;
        map[5][i]=10;
        map[14][i]=10;
        map[15][i]=10;
        map[4][i+10]=10;
        map[5][i+10]=10;
        map[14][i+10]=10;
        map[15][i+10]=10;
    }

    // set priorities
    for(int i=0;i<20;i++){
        pri[i][0]=2;
        pri[i][19]=2;
    }
    for(int i=2;i<18;i++){
        pri[0][i]=2;
        pri[19][i]=2;
        if(i==4||i==15)
            continue;
        pri[i][2]=1;
        pri[i][8]=1;
        pri[i][11]=1;
        pri[i][17]=1;
    }
    for(int i=3;i<8;i++){
        pri[2][i]=1;
        pri[17][i]=1;
    }
    for(int i=12;i<17;i++){
        pri[2][i]=1;
        pri[17][i]=1;
    }
    for(int i=7;i<13;i++){
        pri[i][3]=1;
        pri[i][5]=1;
        pri[i][6]=1;
        pri[i][13]=1;
    }
pri[i][14]=1;
pri[i][16]=1;
}

for(int i=4;i<=5;i++){
    pri[4][i]=-1;
pri[5][i]=-1;
pri[14][i]=-1;
pri[15][i]=-1;
pri[4][i+10]=-1;
pri[5][i+10]=-1;
pri[14][i+10]=-1;
pri[15][i+10]=-1;
}

pri[4][3]=-1;
pri[4][6]=-1;
pri[4][13]=-1;
pri[4][16]=-1;
pri[15][3]=-1;
pri[15][6]=-1;
pri[15][13]=-1;
pri[15][16]=-1;

pri[2][3]=0;
pri[2][6]=0;
pri[2][13]=0;
pri[2][16]=0;
pri[17][3]=0;
pri[17][6]=0;
pri[17][13]=0;
pri[17][16]=0;

//set zones
for(int z=0;z<4;z++)
    for(int i=0;i<10;i++)
        for(int j=0;j<5;j++){
            zone[i][j+z*5]=z;
        }
for(int z=0;z<4;z++)
    for(int i=10;i<20;i++)
        for(int j=0;j<5;j++){
            zone[i][j+z*5]=z+4;
        }
for(int i=4;i<=5;i++){
    zone[4][i]=-1;
    zone[5][i]=-1;
    zone[14][i]=-1;
    zone[15][i]=-1;
    zone[4][i+10]=-1;
    zone[5][i+10]=-1;
    zone[14][i+10]=-1;
    zone[15][i+10]=-1;
}

//get the simulated moving time
public int getSpeed(){
    return this.speed;
}

//set the simulated moving time
public void setSpeed(int s){
    speed=s;
}

public void setMap(int x,int y,int value){
    this.map[x][y]=value;
}
Controller.java:

//This class defines the responses to the users actions
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.JOptionPane;
import javax.swing.JButton;
public class controller implements ActionListener{
    public model M;
    public view V;

    public controller(model m, view v){
        M=m;
        V=v;
    }

    public void actionPerformed(ActionEvent e){
        bg back= new bg(M,V);
        // if the user hits the button “start”
        if(e.getSource()==V.start){
            back.start();
            V.start.setEnabled(false);
        }

        //if the user hits the button “quit”
        if(e.getSource()==V.stop){
            System.exit(0);
        }
    }
}

bg.java:

/*This class builds the request generator, and it is running as a
thread. The generator generates storage and retrieval requests
based on the 24-hour testing scenario.*/
import java.util.ArrayList;
import java.util.Random;

public class bg extends Thread{
    private model m;
    private view v;

    // constructor, initialize the member variables
    public bg(model mo,view vi){
        m=mo;
        v=vi;
    }

    // the simulation starts
    public void run(){
        period1();
    }
}

synchronized public void period1(){
    System.out.println("P1....");

    // generate different requests by different percentages
    // until capacity reaches 95%
    while(m.num<m.max){
        Random ran= new Random();
        int num= ran.nextInt(100);
        // 90% of requests are storage requests
        if(num<=89) simComing();
        // 5% of requests are storage requests
        else if(num>89&&num<=94) simGoing();
        else continue;
        try{
            Thread.sleep(m.getSpeed());
        }catch(Exception e){}
    }
//period 1 terminated, then go to the next period
period2();

synchronized public void period2(){
    System.out.println("P2....");
    int i=0;                   //number of request being
    //generated in this period

    //generate requests by different percentages. This period
    //ends until 10 requests are generated
    while(i<=9){
        Random ran= new Random();
        int num= ran.nextInt(100);
        // 5% of requests are storage requests
        if(num<=4){
            simComing();
            i+=1;                  //count generated requests
        }
        // 5% of requests are retrieval requests
        else if(num>4&&num<=9){
            simGoing();
            i+=1;
        }
        //90% no request is generated
        else{
            continue;
        }
    }
    try{
        Thread.sleep(m.getSpeed());
    }
    catch(Exception e){}

    period 2 terminated, then go to the next period
    period3();
}

synchronized public void period3(){
    System.out.println("P3....");
    //generate requests by different percentages until the
    //capacity reaches 5%
    while(m.num>=376*0.05){
        Random ran= new Random();
        int num= ran.nextInt(100);
// 5% of requests are storage requests
if(num<=4) simComing();

// 90% of requests are retrieval requests
else if(num>4&&num<=94) simGoing();
else continue;
try{
    Thread.sleep(m.getSpeed());
} catch(Exception e){}

// period 3 terminated, then go to the next period
period4();

synchronized public void period4(){
    System.out.println("P4....");
    int i=0;                      // request counter
    // generate requests with different percentages until 10
    // requests are generated
    while(i<=4){
        Random ran= new Random();
        int num= ran.nextInt(100);
        // 1% of requests are storage
        if(num<=1){
            simComing();
            i+=1;            // count generated requests
        }
        // 1% of the requests are retrieval
        else if(num>1&&num<=2){
            simGoing();
            i+=1;
        }
        else{
            continue;
        }
    try{
        Thread.sleep(m.getSpeed());
    } catch(Exception e){}
}
//After all periods, print out the results
summery();
}

//simulates a storage request
synchronized public void simComing(){
    int ele;                 // elevator number
    node start= new node();
    node goal= new node();

    //randomly pick an elevator for the process
    while(true){
        Random ran= new Random();
        int eleNum=ran.nextInt(8);
        if(m.tpz[eleNum]>(int)(47*m.bp)){
            continue;
        }
        ele=eleNum;
        break;
    }

    int numOfb=0;        //number of blank cells in the lot
    //count the number of blank cells in the zone
    for(int i=0;i<20;i++)
        for(int j=0;j<20;j++){
            if(m.zone[i][j]==ele&&m.map[i][j]==0){
                numOfb++;
            }
        }
    if(numOfb>(int)(47*m.bp)){
        m.num++;
    }
    else
        return;

    //set start location based on the elevator number
    switch(ele){
        case 0:
            start.x=4;
            start.y=3;
            break;
    }
}
case 1:
    start.x=4;
    start.y=6;
    break;

case 2:
    start.x=4;
    start.y=13;
    break;

case 3:
    start.x=4;
    start.y=16;
    break;

case 4:
    start.x=15;
    start.y=3;
    break;

case 5:
    start.x=15;
    start.y=6;
    break;

case 6:
    start.x=15;
    start.y=13;
    break;

case 7:
    start.x=15;
    start.y=16;
    break;

} // assign a location for storage as the goal location
int p=storageStrategy(ele);

// convert a location into coordinates
goal.x=p/20;
goal.y=p%20;

m.map[goal.x][goal.y]=2;
// start path planning process
dstar my=new dstar(m,v,start,goal,0,ele);
my.start();
try{Thread.sleep(m.getSpeed());}
catch(Exception e){}
synchronized public int storageStrategy(int ele){

    int position=0, x=0, y=0;
    ArrayList list= new ArrayList();

    // find the loading area in the zone
    switch(ele){
        case 0:
            x=4;
            y=3;
            break;
        case 1:
            x=4;
            y=6;
            break;
        case 2:
            x=4;
            y=13;
            break;
        case 3:
            x=4;
            y=16;
            break;
        case 4:
            x=15;
            y=3;
            break;
        case 5:
            x=15;
            y=6;
            break;
        case 6:
            x=15;
            y=13;
            break;
        case 7:
            x=15;
            y=16;
            break;
    }
}
//find the empty cells with highest priority in the zone
for(int pri=2;pri>=0;pri--){
    for(int i=0;i<20;i++)
        for(int j=0;j<20;j++)
            if(m.pri[i][j]==pri&&
                m.map[i][j]==0&&m.zone[i][j]==ele){
                //put an available location in list
                list.add(i*20+j);
            }
    if(!list.isEmpty()){
        break;
    }
}

// find the location that is farest from the loading area
int max=0, index=0;
for(int i=0;i<list.size();i++){
    int po=(Integer)list.get(i);
    int tempx= po/20;
    int tempy= po%20;
    int d=Math.abs(x-tempx)+Math.abs(y-tempy);
    if(d>max){
        max=d;
        index=i;
    }
}

return position=(Integer)list.get(index);

//simulates a retrieval request
synchronized public void simGoing(){
    node start=new node();
    node goal= new node();
    int ele;               //elevator number
    // if parking lot is empty, then terminate the process
    if(m.num==0)
        return;
}
/randomly picks a parked car as the retrieved car
Random ran=new Random();
while(true){
    int index=ran.nextInt(400);
    int x=index/20;
    int y=index%20;
    // if this cell has a car parked
    if(m.map[x][y]==1){
        start.x=x;
        start.y=y;
        ele=selectEle(start);
        if(m.tpz[ele]<(int)(47*m.bp)){
            break;
        }
    }
    else
        continue;
}

//select the loading area in the zone as the goal of the
//retrieved car
switch(ele){
case 0:
    goal.x=4;
    goal.y=3;
    break;
case 1:
    goal.x=4;
    goal.y=6;
    break;
case 2:
    goal.x=4;
    goal.y=13;
    break;
case 3:
    goal.x=4;
    goal.y=16;
    break;
case 4:
    goal.x=15;
    goal.y=3;
    break;
case 5:
    goal.x=15;
```java
    goal.y=6;
    break;
    case 6:
        goal.x=15;
        goal.y=13;
        break;
    case 7:
        goal.x=15;
        goal.y=16;
        break;
}

    // start the path planning process
    dstar my=new dstar(m,v,start,goal,1,ele);
    my.start();
    try{Thread.sleep(m.getSpeed());} catch(Exception e){}
}

    // return the zone of a specified cell
    public int selectEle(node n){
        int ele;
        ele=m.zone[n.x][n.y];
        return ele;
    }

    // print out the results
    public void summery(){
        // make sure period 4 is complete
        try{
            Thread.sleep(2000);
        } catch(Exception e){}

        System.out.println("Average steps for each car:
"+(float)m.steps/m.numOfThreads);
        System.out.println("Average optimal steps for each car:
"+(float)m.op/m.numOfThreads);
        long x=(long)m.steps*m.getSpeed()*1000000;
        System.out.println("Total search time: "+(m.searchTime-x)+"ns");
        System.out.println("Average search time : "+
(double)(m.searchTime-x)/m.numOfThreads+"ns");
```
System.out.println("Maximum number of running threads in
in theory: "+ (int)((400-400/50*3)*m.bp));
System.out.println("Maximum number of running threads in
in fact: "+ m.atMax);
dstar.java:

/*This class is the D* Lite path planning algorithm running as a thread. It will guide the target from the start to the goal. And if no path found, it will call the unblocker() to do the unblock procedure */
import java.awt.Color;
import java.util.ArrayList;
import javax.swing.JOptionPane;

public class dstar  extends Thread{
    public ArrayList open;   //a list of nodes being evaluated
    public ArrayList closed; //a list of nodes have been evaluated
    public ArrayList result; //a list of nodes have been travelled
    public ArrayList successors;
    public ArrayList used;  //a list of nodes have been travelled
    public node start;
    public node goal;
    public model m;
    public view v;
    public boolean flag;       //if has a path
    public int robotPosition;  //current position of the car
    public int type;          // coming or going, coming is 0,
                              // going is 1
    public int ele;           // elevator number

    // constructor, initialize the variables
    public dstar(model mo,view vi, node s, node g, int t,int eleNum){
        open= new ArrayList();
        closed= new ArrayList();
        result= new ArrayList();
        successors= new ArrayList();
        used= new ArrayList();
        start=s;
        goal=g;
        m=mo;
        v=vi;
        flag=false;
        type=t;
        ele=eleNum;
        robotPosition=start.x*20+start.y;
synchronized public void run(){
    m.tpz[ele]+=1;                  //adding 1 to the number
    //of running
    //threads in the zone

    m.activeThreads+=1;             //adding 1 to the total
    //number of running
    //threads

    //calculate the max number of concurrent threads
    if(m.activeThreads>m.atMax)
        m.atMax=m.activeThreads;

    long timerS=0,timerT=0;          //timers
    node n= new node();             //adding 1 to the number
    m.numOfThreads++;            //of generated threads
    //during the simulation runs

    //counting the total optimal path length
    m.op+=Math.abs(start.x-goal.x)+Math.abs(start.y-goal.y);

    if(type==0){
        v.map[start.x][start.y].setBackground(Color.red);
        try{
            Thread.sleep(m.getSpeed());
        }
        catch(Exception e){}
    }

    if(type==1){

    }

    //while not reach the goal
    while(start.x!=goal.x||start.y!=goal.y){
        open.clear();
        closed.clear();

        if(m.map[goal.x][goal.y]==0||m.map[goal.x][goal.y]==2){
            open.add(goal);
        }
    }
}
else{                               //wait for
elevator is free
        try{
            Thread.sleep(m.getSpeed());
            continue;
        }
        catch(Exception ex){}
    }
flag=false;
timerS=System.nanoTime();
while(!open.isEmpty()){
    int indexBest=0;
    //record the position the best node in open
    n=(node)open.get(0);
    for(int i=0;i<open.size();i++){
        //get best key-value node off from list open
        if(fValue(((node)open.get(i)))<=fValue(n)){
            n=(node)open.get(i);
            indexBest=i;
        }
    }
    if(isStart(n)){
        flag=true;
        moveNext(n);        //move one step
        try{
            Thread.sleep(m.getSpeed());
        }
        catch(Exception e){
            break;
        }
    }
open.remove(indexBest);
closed.add(n);
expand(n);
for(int i=0;i<successors.size();i++){
    //for each n'
    boolean is_in_closed=false;
    int indexClosed=-1;
    //record the position of existing n' in
    //closed
    for(int j=0;j<closed.size();j++){
if(((node)closed.get(j)).x==((node)successors.get(i)).x&&((node)closed.get(j)).y==((node)successors.get(i)).y){
    //if n' is in the set of closed
    is_in_closed=true;
    indexClosed=j;
}

boolean is_in_open=false;
int indexOpen=-1;
//record the position of existing n' in //closed
for(int j=0;j<open.size();j++){
    if(((node)open.get(j)).x==((node)successors.get(i)).x&&((node)open.get(j)).y==((node)successors.get(i)).y){
        //if n' is in the set of closed
        is_in_open=true;
        indexOpen=j;
    }
}

if(is_in_closed){
    continue;
}
if(is_in_open){
    continue;
}

//add n' to open
open.add(((node)successors.get(i)));

// if no path being found, call unblocker
if(flag==false){
    unblocker myun= new unblocker(start, goal, m, v, used);
    start=myun.run();
    used.clear();
}
timerT=System.nanoTime();
//count total search time
m.searchTime=m.searchTime+(timerT-timerS);

137
if(flag==true){
    //reached the goal
    //if retrieval process is complete, minus 1 from the
    //total number of cars in the lot
    if(type==1){
        m.setMap(goal.x, goal.y, 0);
        v.map[goal.x][goal.y].setBackground(null);
        m.num--;
    }
    //thread terminated
    m.tpz[ele]-=1;
    m.activeThreads-=1;
}

//compute the f-value
synchronized public int fValue(node n){
    int h=0,f=0;
    h=Math.abs(n.x-goal.x)+Math.abs(n.y-goal.y);
    f=h+n.g;
    return f;
}

//if the current node is the start node
public boolean isStart(node n){
    if(n.x==start.x&&n.y==start.y){
        return true;
    }
    else
        return false;
}

//expand a node
synchronized public void expand(node n){
    successors.clear();
    //move up if it could
    node newNode= new node();
    if((n.x>0&&m.map[n.x-1][n.y]==0)||start.x==n.x-1&&start.y==n.y){
        newNode.x=n.x-1;
        newNode.y=n.y;
        newNode.g=n.g+1;
        fValue(newNode);
        successors.add(newNode);
    }
}
newNode.parent=n;
successors.add(newNode);
}

//move down
newNode= new node();

if((n.x<19&&m.map[n.x+1][n.y]==0)||start.x==n.x+1&&start.y==n.y){
    newNode.x=n.x+1;
    newNode.y=n.y;
    newNode.g=n.g+1;
    newNode.parent=n;
    successors.add(newNode);
}

//move right
newNode= new node();

if((n.y<19&&m.map[n.x][n.y+1]==0)||start.x==n.x&&start.y==n.y+1){
    newNode.x=n.x;
    newNode.y=n.y+1;
    newNode.g=n.g+1;
    newNode.parent=n;
    successors.add(newNode);
}

//move left
newNode= new node();
if((n.y>0&&m.map[n.x][n.y-1]==0)||start.x==n.x&&start.y==n.y-1){
    newNode.x=n.x;
    newNode.y=n.y-1;
    newNode.g=n.g+1;
    newNode.parent=n;
    successors.add(newNode);
}

//move one step along the path and count the actual movements
synchronized public void moveNext(node n){
    int x,y;
    v.map[start.x][start.y].setBackground(null);
    m.setMap(start.x,start.y, 0);
    start.x=n.parent.x;
}
start.y=n.parent.y;
start.parent=null;
x=start.x;
y=start.y;
m.setMap(x,y,1);  // a robot is an obstacle
robotPosition=x*20+y;
v.map[x][y].setBackground(Color.red);
m.steps++;
}
node.java:

//This class defines the data structure of a node
public class node {

    public int x;            //x, y are coordinates of the node
    public int y;
    public int g;            //cost value of the node
    public node parent;      // the parent of the node

    public node(){
        x=0;
        y=0;
        g=0;
        parent=null;
    }

    public node(int a, int b){
        x=a;
        y=b;
        g=0;
        parent=null;
    }
}

import java.awt.Color;
import java.util.ArrayList;

public class unblocker{
    private node goal;
    private model m;
    private view v;
    private node start;
    private ArrayList open;   //a list of nodes being evaluated and being used in UCS
    private ArrayList closed; //a list of nodes have been evaluated and being used in UCS
    private ArrayList openA;  //a list of nodes have been evaluated and being used in A*
    private ArrayList closedA;  //a list of nodes have been evaluated and being used in A*
    private ArrayList successors;
    private ArrayList blanks;   //a list of blank cells
    private ArrayList destinations;  //a list of destinations
    private ArrayList used; //a list of nodes have been travelled
    public node b;               //the chosen blank cell
    public node d;               //the chosen destination

    //constructor function
    public unblocker(node s, node g, model rm, view rv, ArrayList l){
        goal= g;
        start= s;
        m= rm;
        v= rv;
        used=l;
        open = new ArrayList();
        closed= new ArrayList();
        openA = new ArrayList();
        closedA= new ArrayList();
        successors= new ArrayList();
        blanks= new ArrayList();
        destinations= new ArrayList();
        b= new node();
    }
}
d = new node();
}

synchronized public node run(){

    //step 1, find closest blank cells to the target
    open.add(start);
    while(true){
        for(int i=0;i<open.size();i++){
            expand((node)open.get(i));
        }

        for(int j=0;j<successors.size();j++){
            int x = ((node)successors.get(j)).x;
            int y = ((node)successors.get(j)).y;
            if(m.map[x][y]==0){
                blanks.add(((node)successors.get(j)));
            }
        }
        //if any blank cell detected, terminate this process
        if(!blanks.isEmpty())
            break;
        //if no blank cell found, expand all the nodes in
        //successor list
        else{
            open.clear();
            for(int i=0;i<successors.size();i++){
                open.add(((node)successors.get(i)));
            }
            successors.clear();
        }
    }

    //step 2, find desired destinations for blank cell
    ArrayList array= new ArrayList();
    int po= start.x*20+start.y;
    if(start.x>0){
        //expand the cell and exclude the loading cells and elevators
        if(po!=103&&po!=106&&po!=113&&po!=116&&po!=323&&po!=326&&po!=333&&
            po!=336){
            if(m.map[start.x-1][start.y]!=10){
                node n= new node();
                n.x=start.x-1;
                n.y=start.y;
                array.add(n);
            }
        }
    }
}
n.g=Math.abs(n.x-goal.x)+Math.abs(n.y-goal.y);
  n.parent=start;
  if(!isInUsed(n)){
    array.add(n);
  }
}
}

if(start.x<19){
  if(po!=63&&po!=66&&po!=73&&po!=76&&po!=283&&po!=286&&po!=293&&po!=296)
    if(m.map[start.x+1][start.y]!=10){
      node n= new node();
      n.x=start.x+1;
      n.y=start.y;
      n.g=Math.abs(n.x-goal.x)+Math.abs(n.y-goal.y);
      n.parent=start;
      if(!isInUsed(n)){
        array.add(n);
      }
    }
}
}

if(start.y>0){
  if(po!=87&&po!=97&&po!=307&&po!=317){
    if(m.map[start.x][start.y-1]!=10){
      node n= new node();
      n.x=start.x;
      n.y=start.y-1;
      n.g=Math.abs(n.x-goal.x)+Math.abs(n.y-goal.y);
      n.parent=start;
      if(!isInUsed(n)){
        array.add(n);
      }
    }
  }
}
}

if(start.y<19){
  if(po!=82&&po!=92&&po!=302&&po!=312){
    if(m.map[start.x][start.y+1]!=10){
      node n= new node();
      n.x=start.x;
      n.y=start.y+1;
      n.g=Math.abs(n.x-goal.x)+Math.abs(n.y-goal.y);
      n.parent=start;
      if(!isInUsed(n)){
        array.add(n);
      }
    }
  }
}
}
node n = new node();
n.x = start.x;
n.y = start.y+1;
n.g = Math.abs(n.x-goal.x)+Math.abs(n.y-goal.y);
n.parent = start;
if(!isInUsed(n)){
    array.add(n);
}
}

//put the suitable destinations in “destinations” list
int min = ((node)array.get(0)).g;
for(int i = 0; i < array.size(); i++){
    if(((node)array.get(i)).g < min){
        destinations.clear();
        destinations.add(((node)array.get(i)));
    }
    else if(((node)array.get(i)).g == min)
    destinations.add(((node)array.get(i)));
}

//step 3, choose the best pair of blank and destination
//with the smallest Manhattan distance between them
int indexB = 0, indexD = 0;
int minh;
minh = ((node)destinations.get(0)).g + Math.abs(((node)blanks.get(0)).x -
    ((node)destinations.get(0)).x) + Math.abs(((node)blanks.get(0)).y -
    ((node)destinations.get(0)).y);
for(int i = 0; i < blanks.size(); i++)
    for(int j = 0; j < destinations.size(); j++){
        int h = ((node)destinations.get(j)).g + Math.abs(((node)blanks.get(i)).x -
            ((node)destinations.get(j)).x) + Math.abs(((node)blanks.get(i)).y -
            ((node)destinations.get(j)).y);
        if(h < minh){
            indexB = i;
            indexD = j;
        }
    }

//set locations of the chosen blank cell and the
//destination
b = (node)blanks.get(indexB);
m.map[b.x][b.y]=3;
d=(node)destinations.get(indexD);
node temp= new node(d.x,d.y);
used.add(temp);
//step 4, move the blank to its destination
return astar(b,d,start);
}

//expand a node in UCS
synchronized public void expand(node n){
    int po = n.x*20+n.y;            //convert location
    //expand up
    if(n.x>0&&po!=103&&po!=106&&po!=113&&po!=116&&po!=323&&po!=326&&po!=333&&po!=336&&m.map[n.x-1][n.y]!=10){
        node newNode= new node();
        newNode.x= n.x-1;
        newNode.y= n.y;
        newNode.parent=n;
        // if the cell has never been expanded before
        if(!isInClosed(newNode)){
            successors.add(newNode);
            closed.add(newNode);
        }
    }
    //expand down
    if(n.x<19&&po!=63&&po!=66&&po!=73&&po!=76&&po!=283&&po!=286&&po!=293&&po!=296&&m.map[n.x+1][n.y]!=10){
        node newNode= new node();
        newNode.x= n.x+1;
        newNode.y= n.y;
        newNode.parent=n;
        if(!isInClosed(newNode)){
            successors.add(newNode);
            closed.add(newNode);
        }
    }
    //expand right
    if(n.y<19&&po!=82&&po!=92&&po!=302&&po!=312&&m.map[n.x][n.y+1]!=10){
        node newNode= new node();
        newNode.x= n.x;
        newNode.y= n.y+1;
        newNode.parent=n;
        if(!isInClosed(newNode)){
            successors.add(newNode);
            closed.add(newNode);
        }
    }
}
newNode.x = n.x;
newNode.y = n.y+1;
newNode.parent = n;
if(!isInClosed(newNode)){
    successors.add(newNode);
    closed.add(newNode);
}
}

// expand left
if(n.y>0&&po!=87&&po!=97&&po!=307&&po!=317&&m.map[n.x][n.y-1]!=10){
    node newNode = new node();
    newNode.x = n.x;
    newNode.y = n.y-1;
    newNode.parent = n;
    if(!isInClosed(newNode)){
        successors.add(newNode);
        closed.add(newNode);
    }
}

// justify if the node has already been expanded
public boolean isInClosed(node n){
    boolean inClosed = false;
    for(int i=0; i<closed.size();i++){
        if(n.x==((node)closed.get(i)).x&&n.y==((node)closed.get(i)).y){
            inClosed=true;
            break;
        }
    }
    if(inClosed)
        return true;
    else
        return false;
}

// justify if the node has already been travelled
public boolean isInUsed(node n){
    boolean inUsed = false;
    for(int i=0; i<used.size();i++){
        if(n.x==((node)used.get(i)).x&&n.y==((node)used.get(i)).y){
            inUsed=true;
            break;
        }
    }
    if(inUsed)
        return true;
    else
        return false;
}
if(n.x== ((node)used.get(i)).x\&\&n.y==
((node)used.get(i)).y){
    inUsed=true;
    break;
}
}
if(inUsed)
    return true;
else
    return false;
}

// move blank cell using A* algorithm
synchronized public node astar(node blank, node desti, node
start){
    //initialize all list being used
    open.clear();
    closed.clear();
    successors.clear();

    node n= new node();
    node curr= new node();
    boolean flag;

    open.add(blank);
    flag=false;
    //while goal is nod found
    while(!open.isEmpty()){  
        int indexBest=0;
        //record the position the best node in open
        n=(node)open.get(0);
        for(int i=0;i<open.size();i++){  
            //get best f-value node off from list open
            if(fValue(((node)open.get(i)))<=fValue(n)){
                n=(node)open.get(i);
                indexBest=i;
            }
        }
    // if path found, move the blank cell and then go to
    // step 5, move the target to the blank cell next to
    // it
    if(isGoal(n, desti)){
        flag=true;
        move(n,blank);
moveCell(desti, start);
    return desti;
}
open.remove(indexBest);
closed.add(n);
expandA(n, start);
for(int i=0;i<successors.size();i++){
    //for each n'
    boolean is_in_closed=false;
    int indexClosed=-1;
    //record the position of existing n' in closed
    for(int j=0;j<closed.size();j++){
        if(((node)closed.get(j)).x==((node)successors.get(i)).x&&((node)closed.get(j)).y==((node)successors.get(i)).y){
            //if n' is in the set of closed
            is_in_closed=true;
            indexClosed=j;
        }
    }
    boolean is_in_open=false;
    int indexOpen=-1;
    //record the position of existing n' in closed
    for(int j=0;j<open.size();j++){
        if(((node)open.get(j)).x==((node)successors.get(i)).x&&((node)open.get(j)).y==((node)successors.get(i)).y){
            //if n' is in the set of closed
            is_in_open=true;
            indexOpen=j;
        }
    }
    if(is_in_closed){
        continue;
    }
    if(is_in_open){
        continue;
    }
    //if n' is not in open or closed, add n' to open
    open.add(((node)successors.get(i)));
// if A* fails
if(flag==false){
    System.out.println("Can't find path!!");
    return null;
}

// expand a node in A* algorithm
synchronized public void expandA(node n, node start){
    successors.clear();

    // expand up if possible
    int po=n.x*20+n.y;
    if(n.x>0&m.map[n.x-1][n.y]!=10&&(n.x-1!=start.x||n.y!=start.y)&&po!=103&po!=106&po!=113&po!=116&po!=323&po!=326&po!=333&po!=336){
        node newNode = new node();
        newNode.x= n.x-1;
        newNode.y= n.y;
        newNode.g=n.g+1;
        newNode.parent=n;
        successors.add(newNode);
    }

    // down
    if(n.x<19&m.map[n.x+1][n.y]!=10&&(n.x+1!=start.x||n.y!=start.y)&po!=63&po!=66&po!=73&po!=76&po!=283&po!=286&po!=293&po!=296){
        node newNode = new node();
        newNode.x= n.x+1;
        newNode.y= n.y;
        newNode.g=n.g+1;
        newNode.parent=n;
        successors.add(newNode);
    }

    // left
    if(n.y>0&m.map[n.x][n.y-1]!=10&&(n.x!=start.x||n.y-1!=start.y)&&po!=87&po!=97&po!=307&po!=317){
        node newNode = new node();
        newNode.x= n.x;
        newNode.y= n.y-1;
        newNode.g= n.g+1;
        newNode.parent=n;
        successors.add(newNode);
successors.add(newNode);
}

//right
if(n.y<19&m.map[n.x][n.y+1]!=10&(n.x!=start.x||n.y+1!=start.y)&
&po!=82&po!=92&po!=302&po!=312){
    node newNode = new node();
    newNode.x= n.x;
    newNode.y= n.y+1;
    newNode.g=n.g+1;
    newNode.parent=n;
    successors.add(newNode);
}

// calculate f-value for a node
synchronized public int fValue(node n){
    int h=0,f=0;
    h=Math.abs(n.x-goal.x)+Math.abs(n.y-goal.y);
    f=h+n.g;
    return f;
}

// if the node is the goal, return true, else return false
public boolean isGoal(node n,node goal){
    if(n.x==goal.x&&n.y==goal.y){
        return true;
    }
    else
        return false;
}

//build a path for blank cell and move it along the path
synchronized public void move(node n,node blank){
    ArrayList path= new ArrayList();
    //build a path
    path.add(n);
    while(n.x!=blank.x||n.y!=blank.y){
        path.add(n.parent);
        n=n.parent;
    }
    path.add(n);
// move along the path
for(int i=path.size()-2;i>=0;i--){
  int x=((node)path.get(i)).x;
  int y=((node)path.get(i)).y;
  if(m.map[x][y]!=0){
    //move the blank
    int temp;
    int tempx=((node)path.get(i+1)).x;
    int tempy=((node)path.get(i+1)).y;

    temp=m.map[x][y];
    m.map[x][y]=m.map[tempx][tempy];
    m.map[tempx][tempy]=temp;

    Color col;
    col=v.map[x][y].getBackground();

    v.map[x][y].setBackground(v.map[tempx][tempy].getBackground());
    v.map[tempx][tempy].setBackground(col);
    try{
      Thread.sleep(m.getSpeed());
    }catch(Exception e){}

    //count the blank cell movements to the actual
    //movements
    m.steps++;
  }
  //if other blank cells comes into the path, keep
  //moving without moving time and actual movement
  //count
  else{
    int temp;
    int tempx=((node)path.get(i+1)).x;
    int tempy=((node)path.get(i+1)).y;

    temp=m.map[x][y];
    m.map[x][y]=m.map[tempx][tempy];
    m.map[tempx][tempy]=temp;
    continue;
  }
}
}
//step 5, move the target to the blank cell next to it
synchronized public void moveCell(node desti, node start){

    int temp;
    temp=m.map[start.x][start.y];
    m.map[start.x][start.y]= m.map[desti.x][desti.y];
    m.map[desti.x][desti.y]=temp;

    m.map[start.x][start.y]=0;

    Color col;
    col=v.map[desti.x][desti.y].getBackground();

    v.map[desti.x][desti.y].setBackground(v.map[start.x][start.y].getBackground());
    v.map[start.x][start.y].setBackground(col);
    try{Thread.sleep(m.getSpeed());}
        catch(Exception e){System.out.println("sleep error");}
    m.steps++;}
}