5-2015

FORMAL SPECIFICATION AND REFINEMENT OF THE NAVIGATION TASKS OF AUTONOMOUS ROBOTS

Eman Rabiah Rabiah

United Arab Emirates University

College of Information Technology

Software Development Track

FORMAL SPECIFICATION AND REFINEMENT OF THE NAVIGATION TASKS OF AUTONOMOUS ROBOTS

Eman Rabiah

This thesis is submitted in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering

Under the Supervision of Professor Boumediene Belkhouche

May 2015

Declaration of Original Work

I, Eman Rabiah, the undersigned, a graduate student at the United Arab Emirates University (UAEU), and the author of this thesis entitled "*Formal Specification and Refinement of the Navigation Tasks of Autonomous Robots*", hereby, solemnly declare that this thesis is an original research work that has been done and prepared by me under the supervision of Professor Boumediene Belkhouche, in the College of Information Technology at UAEU. This work has not been previously formed as the basis for the award of any academic degree, diploma or a similar title at this or any other university. The materials borrowed from other sources and included in my thesis have been properly cited and acknowledged.

Student's Signature ____________________ Date ______________

Approval of the Master Thesis

This Master Thesis is approved by the following Examining Committee Members:

1) Advisor (Committee Chair): Boumediene Belkhouche

Title: Professor

Department of Software Development

College of Information Technology

Signature ______________________ Date ______________

2) Member: Mamoun Awad

Title: Associate Professor

Department of Software Development

College of Information Technology

Signature ______________________ Date ______________

3) Member (External Examiner): Rabeb Mizouni

Title: Assistant Professor

Department of Computer engineering

Institution: Khalifa University

Signature ______________________ Date ______________

This Master Thesis is accepted by:

Dean of the College of Information Technology: Dr. Shayma Al Kobaisi

Signature ______________________________ Date _______________

Dean of the College of Graduate Studies: Professor Nagi T. Wakim

Signature ______________________________ Date _______________

Copy __________ of __________

Abstract

Autonomous robots are hybrid systems whose role in our daily life is becoming increasingly critical. They are tasked with various activities requiring reliability, safety, and correctness of their software-controlled behavior. Formal methods have been proved effective in addressing development issues associated with these software qualities. However, even though autonomous robot navigation is a primordial function, there is no research dealing with enhancing reliability of the navigation algorithms. Thus, our focus is to investigate this type of algorithms, and specifically path planning, a fundamental and critical functionality supporting autonomy. We formally address the issue of enhancing reliability of the widely-used A* path planning algorithm. In our stepwise refinement process, we capture successively more concrete specifications by transforming a high-level specification into an equivalent executable program. To elaborate an initial representation of the A* algorithm, we express it in an abstract and intuitive, yet formal, description. We use traditional mathematical concepts, such as sets, functions and predicate logic to capture this description. In the next step, we use the Z specification language to effect the transformation from the mathematical description into Z schemas. The resulting specification is completely formal. Subsequently, we use the formal theory of refinement in Z to generate the implementation specification. This stage involves both data and operation refinement and is carried out in several basic sub-steps. A Java-based simulation prototype that mirrors the implementation specification is developed in order to demonstrate the effectiveness of our software development approach.

Keywords: Z, formal specification, path planning, A* algorithm, autonomous robots, formal refinement, equivalent implementation, simulation, obstacle avoidance, navigation tasks.

Title and Abstract (in Arabic)

التوصيف الرسمي و التحسين الرسمي لمهام ملاحة الروبوتات المستقلة

الملخص

تعددت الاستخدامات للربوتات المستقلة في عصرنا الحالي ، ومن المتوقع أن يتزايد الاستخدام لهذه الأنظمة الذكية بشكل أكبر في المستقبل القريب. هناك العديد من الطرق التي من الممكن استخدامها للحصول على برمجيات موثوقة ومنها الطرق الرسمية التي أثبتت كفاءتها وفعاليتها وقدرتها، في الكثير من الأبحاث السابقة، على بناء برمجيات موثوقة خالية من الأخطاء . في هذه الأطروحة قمنا باستخدام أحد هذه الطرق وهي (لغة زاد) لتوصيف سلوك الروبوت لايجاد أقصر طريق بين نقطتين ، وباستخدام هذه اللغة حصلنا على نموذج رياضي مجرد للمشكلة ، ومن ثم قمنا باستخدام أحد طرق التحسين الرسمية التي ساعدتنا علي تحويل هذا النموذج الرياضي المجرد الي شيفرة برمجية قابلة للاختبار. وبناء على ذلك تم اختبار هذه الشيفرة البرمجية التي أسفرت عن نتائج أثبتت صحة الطريقة المتبعة في هذا البحث.

كلمات مفتاحية: لغة زاد ، طرق التحسين الرسمية ، الطرق الرسمية ، الروبوتات المستقلة ، أقصر مسار ، تجنب العوائق.

Acknowledgements

I would like to express the deepest appreciation to my committee chair, Professor Boumediene Belkhouche, for his guidance and constant help during the whole stages of working on this dissertation. He continually and persuasively conveyed a spirit of adventure in regard to research and scholarship, and an excitement in regards to teaching. Without his supervision and constant help this dissertation would not have been possible. My special appreciation and thanks extended to Dr. Mamoun Awad (my internal reviewer) and Dr. Rabeb Mizouni (my external reviewer) for their support and help.

Finally, I would like to dedicate this dissertation to my parents who taught me that education has to have always the first priority in my life and to my loyal supporters, my beautiful sister and my brothers.

Dedication

This thesis is dedicated to my family for their endless love and encouragement, and dedicated to my Professor Boumediene Belkhouche who had a positive impact on both my personal and academic life.

Table of Contents

Title i

Declaration of Original Work ii

Copyright iii

Approval of the Master Thesis iv

Abstract vi

Title and Abstract (in Arabic) vii

Acknowledgments viii

Dedication ix

Table of Contents x

List of Tables xiii

List of Figures xiv

Chapter 1: Introduction 1
- 1.1 Motivations 1
- 1.2 Literature review 3
  - 1.2.1 Path planning 3
  - 1.2.2 A* algorithm 4
  - 1.2.3 Formal methods 6
  - 1.2.4 Z specification language 8
  - 1.2.5 Formal refinement 9
  - 1.2.6 Formal specifications and verifications of autonomous robots . 10
- 1.3 Potential contributions and limitations of the study 13
- 1.4 Outline of the thesis 14

Chapter 2: Semi-formal specification 16
- 2.1 Definitions 16
- 2.2 Operations 22
  - 2.2.1 Set the target position 22
  - 2.2.2 Set the robot position 23
  - 2.2.3 Set obstacle Position 24
  - 2.2.4 Free robot Position 25
  - 2.2.5 Free target Position 25
  - 2.2.6 Free obstacle Position 26
  - 2.2.7 Free border pair 26
  - 2.2.8 Evaluate the best pair 26
  - 2.2.9 Search pairs no path 27
  - 2.2.10 Construct path 27

2.2.11 Search pairs 28
2.2.12 Refine diagonal neighbors 29
2.2.13 Delete diagonal neighbors 35
2.2.14 Refine neighbors from obstacles and border 35
2.2.15 Delete neighbors from obstacles and border 35
2.2.16 Calculate H Value 36
2.2.17 Calculate G Current 36
2.2.18 Calculate G neighbor 36
2.2.19 Calculate F value 37
2.2.20 Calculate Cost Value 37
2.2.21 Evaluate neighbors 37
2.2.22 Follow path 38
2.2.23 Move down 41
2.2.24 Move up 41
2.2.25 Move right 41
2.2.26 Move left 42
2.2.27 Move up right 42
2.2.28 Move up left 42
2.2.29 Move down right 43
2.2.30 Move down left 43
2.3 Example of finding the shortest path 44

Chapter 3: Formal specification 52
3.1 Definitions 52
3.2 System status 53
3.3 Operations 64
3.4 Proof of some properties 74

Chapter 4: Formal refinement 77
4.1 Data refinement 78
4.2 Forwards simulation 79
4.3 Data refinement of the formal specification 80
4.4 Operations refinement of the formal specification 89
4.5 Summary 96

Chapter 5: Simulation 97
5.1 Execution Scenarios 98
5.1.1 Setting the environment 98
5.1.2 Path planning operations 101
5.1.3 The robot movements 111
5.2 Summary 114

Chapter 6: Discussion and Conclusion 115

Bibliography 120

Appendix Chapter A: Notations 125

Appendix Chapter B: Z document 126

Appendix Chapter C: Equivalent implementation 158

List of Tables

4.1 The result of the data refinement process 88

5.1 The abbreviation of each robot move that will appear in the simulator 112

List of Figures

1.1 An example of path planning grid 13

2.1 The pair and the map set 17
2.2 Example of undefined neighbors 18
2.3 The range of the border coordinates 18
2.4 Case1: The down-right neighbor will be excluded 29
2.5 Case2: The down-left neighbor will be excluded 30
2.6 Case3: The up-right neighbor will be excluded 31
2.7 Case4: The up-left neighbor will be excluded 32
2.8 Case5: The down-left and the down-right neighbor will be excluded . 32
2.9 Case6: The up-left and the up-right neighbor will be excluded 33
2.10 Case7: The up-right and the down-right neighbor will be excluded . . 34
2.11 Case8: The up-left and the down-left neighbor will be excluded 34
2.12 The directions of the robot moves 39
2.13 The way of determining the next robot movement 40
2.14 The color palette of the different locations in the map 44
2.15 Example of finding the shortest path 45

3.1 The border coordinates 54
3.2 Case1: The down-right neighbor will be excluded 71

4.1 Forwards simulation 80
4.2 Example of data refinement 82

5.1 Setting the robot position 99
5.2 Setting the target position 100
5.3 Setting the obstacle position 101
5.4 An example of finding the shortest path 102
5.5 An example of not finding any possible path 103
5.6 An example of case 1: discarding the down-right neighbor 104
5.7 An example of case 2: discarding the down-left neighbor 105
5.8 An example of case 3: discarding the up-right neighbor 106
5.9 An example of case 4: discarding the up-left neighbor 107
5.10 An example of case 5: discarding the down-left and down-right neighbors 108
5.11 An example of case 6: discarding the up-left and up-right neighbors . 109
5.12 An example of case 7: discarding down-right and up-right neighbors . 110
5.13 An example of case 8: discarding down-left and up-left neighbors . . 111
5.14 The basic moves of the Robot 112
5.15 Test case 1: The robot moves to reach the target 113
5.16 Test case 2: The robot moves to reach the target 114

6.1 The stages of our research 116

Chapter 1: Introduction

1.1 Motivations

"When software fails, it invariably fails catastrophically. This awful quality is a reflection of the lack of continuity between successive system states in executing software. If the preceding state is correct, there is no inherent carry over of even partial correctness into the next succeeding state "[1]. Therein lies the perplexing complexity of software.

The term "Software Engineering" was coined in 1968 to establish a discipline capable of addressing the then-perceived software crisis [2]. Decades later, because of the rising complexities, the challenges for developing correct software systems are growing even bigger as witnessed by the many shortcomings of some highly-visible critical systems. Since then, there has been several examples of software failures that caused catastrophes. For example, in 1996, the European Space Agency Ariane 5 rocket exploded 37 seconds after launch. The cause of the failure was an exception-handling bug that forced a self destruct [3]. Another famous failure was the THERAC-25 radiation machine, which led to the death of three patients. The severity of this kind of system failure requires developers to design, implement, and deploy high confidence systems, which are capable of performing critical tasks reliably with no failures or losses.

Various complex software systems are being continuously deployed to address the ever-increasing needs of the different users. These systems include critical and non-critical systems. Critical systems are subject to stringent properties, such as reliability, high confidence, and safety. Such systems should always perform properly with no failures. Otherwise,catastrophic consequences might ensue.

Several methods are available to help us develop reliable systems, such as: testing, triple modular redundancy, and formal methods. Testing is not a sufficient approach to guarantee that the system is 100% reliable, because the testing process can never completely detect all the defects within software [4]. Triple-modular redundancy, which consists of three systems performing concurrently the same process and voting on the overall output, tries to limit failure through redundancy. If any of these systems fails, the other systems would handle the fault [5], but in case the software itself contains an error, the three systems will have the same error and will behave the same way.

Formal methods are mathematical techniques that are used in specification, development and verification of hardware and software systems. These techniques can contribute in delivering safe and reliable systems [6]. A well-know researcher wrote about the importance of formal methods for the future of the software system as: "It is clear to all the best minds in the field that a more mathematical approach is needed for software to progress much" [7].

Robotic systems are critical systems that are designed to perform several tasks intelligently on behalf of humans. They are intended to carry out several tasks intelligently to help us in our daily life; thus, reliability and safety of these systems are fundamental. They are expected to always behave correctly and accurately without causing any harm. Failure of these systems may endanger human life. Among the many uses of robots are in conflicts, at home, at work, and even as pets. That is, their ever-increasing interactions with humans demands reliability.

This issue is one of the motivating factors of our research. We would like to address reliability by constructing software systems using formal methods. Since navigation is a fundamental task of robots, we focus our study on the path planning issue. The choice is based on the fact that there is no previous research that was addressing the issue of enhancing reliability of the navigation algorithms. Thus, our focus is to investigate this type of algorithms, and specifically path planning, a fundamental and critical functionality in any autonomous system.

1.2 Literature review

Our research emphasizes the use of formal methods in software development, and specifically, the notion of refinement from a specification to an implementation. Thus, in the following sections, we will review the path planning algorithms specifically the A* algorithm, then we discuss some formal methods approaches, their successful applications, and formal refinement.

1.2.1 Path planning

Autonomous systems, such as robots, are designed with sophisticated capabilities. Navigation in dangerous environments cluttered with obstacles is a fundamental capability. Path finding is an essential navigation function in any autonomous system. Its objective is to aid robots find the shortest or the optimal path to move safely from one point to another [8]. The problem of finding the shortest path has received considerable attention, and it has been solved by different algorithms. Most of these algorithms are graph-based.

Some well-known path finding algorithms such as : A*, Dijkstra, and Bellman-Ford algorithms. Among these, the A* algorithm is used in many applications which are related to the path finding problem, such as games. This algorithm is designed to find an optimal and safe path between two points.

Dijkstra algorithm, another fundamental well-known algorithm, is capable of solving a single-source shortest path problem, in which we need to find the shortest path from a source node S to all the other nodes in the graph. This algorithm works for both directed and undirected graphs. However, it does not work for graphs with negative weights. The Bellman-Ford algorithm targets the same problem and is capable of handling negative weighted graphs [9].

1.2.2 A* algorithm

The A* algorithm is a well-known algorithm that emerged in 1968 as an extension of Dijkstra's algorithm. It succeeds in achieving faster time performance using heuristics. This algorithm is used to find the efficient shortest path from one point to another while avoiding obstacles [10].

However, this algorithm works by checking the neighbors of the current point, and using some metrics to determine which one of these neighbors is the next node to be evaluated. These metrics include the G value, the H value, and the F value; the G value is the distance from the START node to the current node, the H value is distance from the current node to the GOAL, and the F value is the sum of the G value and the H value. The algorithm keeps checking and evaluating the nodes until the algorithm found the target.

The main scenario of the A* algorithm is described below: [11]:

1. Assign a START and a GOAL node.
2. Define the OPEN set, which is a priority queue that holds the nodes to be evaluated, the nodes are ordered by the F value of each node from the lowest to the highest, while the node with the lowest F is called the “best node”. initially the OPEN set contains the START node.
3. Define the CLOSED set, that holds the nodes that have been evaluated already, initially it is empty.
4. Compare if the best node in the OPEN set is not the GOAL:
    - Make this node as the current node
    - Remove this node from the OPEN set
    - Add this node to the CLOSED set
    - Find the adjacent neighbors of the current node
        (a) Calculate for each neighbor the cost value, which is equivalent to the sum of the G value of the current node (the distance between the current node and the START node) and the distance between the neighbor and the current node as follows:

Cost = G(current) + distance-between(current, neighbor)

(b) If the neighbor in the OPEN set and the Cost is less than g(neighbor):

remove this neighbor from the OPEN set, because the new path is better

(c) If the neighbor in the CLOSED set: ignore it because this node has been evaluated already

(d) If the neighbor is neither in the OPEN set nor in the CLOSED set:

Set G(neighbor) to the cost

Add the neighbor to the OPEN set

Set priority queue rank to the F(neighbor)

Set the current node as a parent of this neighbor

5. If the best node in the OPEN set is the GOAL

   - Construct a path from the GOAL node to the START node by following the parent of each node

1.2.3 Formal methods

Formal methods are rigorous mathematical techniques that are used in specifying, verifying, and developing hardware and software systems [12]. These methods can contribute to developing reliable and robust systems [13]. These methods are used to formally specify the software functionally and to verify its correctness, or to build a software system from an abstract specification to its implementation [14]. There are a lot of formal methods that can be used to specify software systems such as: Z, VDM, Larch, Temporal logic, CSP, transition axioms, and they can be classified into two major types: model-oriented and property-oriented. The model-oriented method describes the behavior of the system by constructing a model of the system using mathematical structures such as: sets, functions and functions, while the property-oriented methods specifies the system's behavior by specifying a set of the system properties [15].

The cost effectiveness of the use of formal methods in industry from a CEO point of view was demonstrated by Martyn Thomas [16]. Anthony Hall [17] discusses seven myths about formal methods and suggests instead seven facts as follows : 1) formal methods are beneficial approaches to find errors in early stages; 2) they let you think deeply about the system that you plan to build; 3) they can be used for almost any kind of applications (critical or non-critical); 4) formal methods are based on mathematical techniques which are easier to understand than program syntax; 5) they can lower the development cost; 6) they help clients to be more aware about the product they buy; 7) they are being used in successful industrial projects.

In a technical report [18], Rushby discussed formal methods, certification of critical systems,the advantages of formal specification and verification, and the effective contribution of these methods to the assurance of the safety of critical systems.

In the transputer project, the formal development process of the floating-point unit was faster by three months compared to the informal development, while the delay in production for each month was estimated to cost one million American Dollar [19].

SACEM system was responsible to control the speed of the RER line A trains in Paris [20], 63% of the code was safety-critical and has been conducted by formal specification and verification [21, 22]. In 1990 Guiho and Hennebert [21] declared that the system is safer than before as a consequence of the use of the formal specification and verification.

One of the formal specification languages (TLA+) was used in Amazon Web Services (AWS) to solve complex design problems in critical systems, the authors of [23] mentioned that the use of the formal specification helps to find subtle bugs that would never be found by other techniques, and it optimizes the performance without loosing correctness.

Several articles mentioned many obstacles that prevented the acceptance of the formal methods, such as: lack of well-trained people, weakness in both the available tools and notations [20].

1.2.4 Z specification language

Z notation is a well known type of formal methods, which has a global acceptance as a formal specification language. It is useful in specifying software systems. Z notation is a descriptive technique. This type of notation from a specification language perspective is a model oriented specification language which is based on logic and set theory [14].

There are two languages in Z notation : mathematical and schema language. The former is used to describe objects and relationships, while the latter is used to construct and compose descriptions [14]. Z notation is used to describe the behavior of systems by defining the states, the operations, and the enquiries. This specification language uses a graphical format which is called a schema to present the formal description. A schema consists of two major parts, which are the declarations and predicates as shown below [24]:

```
┌─ SchemaName ─────────────────────────────
│ Declarations
├──────────
│ Predicates
└──────────────────────────────────────────
```

Similar to the structure of the method in any programming language, the declarations part introduces variable definitions and imported types (i.e., other schemas), while the predicates part describes the behavior of the system in terms of predicates about the states of the variables.

1.2.5 Formal refinement

Refinement in software development is a concept that dates back to the 1970's [25]. Its emphasis was on the stepwise refinement of programs through successive stages to ensure clarity and correctness. Formal refinement fits in that evolution. It is the process of generating the executable code from the formal description of the system. The major goal of this process is to improve the specification by resolving uncertainty and ambiguity. Formal specification is highly abstracted description of the system, and by using the refinement process we add more data and more details to reveal the correct functionality of the system [26, 27].

1.2.6 Formal specifications and verifications of autonomous robots

A case study using a home service robot (Samsung Home Robot SHR 1000) showed the effectiveness of the formal verification. Through the verification process they detected and solved a feature interaction problem (in which the robot does not stop when commanded by the user). They used the Esterel framework to develop an executable code and to verify the system safety. This case study illustrated that the reliability of the system was enhanced by using formal verification and validation (V&V) [28].

The authors of [29] introduced an approach to analyze and model the path planning algorithms. They used mCRL2 language and the model μ- calculus to describe the behavior of a multi-robot system, they followed this by a formal verification process using mCRL2 toolset to verify some properties. They mentioned that some properties of a simple path planning algorithm can be verified efficiently using their proposed approach.

The robot system can be divided into 3 major subsystems: the planning system, the perceptual system, and the acting system. In their research [30], they focused on describing the behavior of the perceptual system of the autonomous mobile robots that is composed of other subsystems, so in order to describe the overall behavior of the perceptual system they described the behavior of these subsystems and the communication between them. They succeeded in specifying the perceptual system formally by using hybrid process algebra, then they implemented the resulting model as a software simulation to emulate the behavior of their proposed design.

Hybrid process algebra was used in [31] to formally specify the behavior of the path planning system of mobile robots. The moves of autonomous mobile robots were divided into four major modes as follows: 1) move-to-goal mode: it is the process that uses the control laws to let the robot moves to the goal, 2) obstacle-avoidance mode: it is the process that assigns an intermediary-goal between the robot and the goal in the case of obstacle existence, 3)move-to-intermediary-goal mode: it is the process that lets the robot move to an intermediary-goal that has been specified by the obstacle-avoidance process 4) path-smoothing mode: it is a process that will turn the robot smoothly between the different modes. While each motion mode has been described as a process, the communication between these four processes will describe the overall behavior of the path planning system. In this paper, they developed the formal specification of the path planner system, then they implemented their model as a simulation software, and they provided some test cases that illustrate the behavior of their specification.

Timing and concurrency are very important properties in real-time systems. Thus, it is crucial to verify these properties by using formal methods. This research [32] proves that Z specification can be used to specify small-scale embedded hard real-time systems, by providing a case study that formally described the behavior of a wall-climbing robot using Z notation.

In their research, Lynch, Segala, and Vaandrager [33] developed the Hybrid Input/Output Automaton (HIOA) model, which is a fundamental mathematical framework for specifying and analyzing hybrid systems. In another study [34], the authors provided a case study that used the hybrid I/O automaton (HIOA) framework to specify the behavior of a simple Lego car under some constraints such as: the car will always move forward on a black tape and it will never get out of tape or move backward. Another case study [35] used HIOA to specify and validate a part of automated transportation system. The major goal of this case study was to prove the usefulness of the HIOA model and several computer-science based techniques to particularly specify and validate automated transportation systems and hybrid systems generally.

According to the previous discussion, the last four researches are the most relevant to our research. We found that both of these researches [30] & [31] were limited to build the formal specification, while the simulation they did was constructed in such way that is similar to their abstract model without following a formal refinement process to get the equivalent implementation, and also the specified systems were different,the first research [30] specified the mobile robots perceptual system and the second research[30] specified a different motion planning algorithm which is not the A* algorithm.

Another research [32] was limited to the formal specification using Z and there was no concrete implementation, and even the specified problem(the behavior of a wall climbing robot)is different compared to our problem.

The other researches [34] & [35], that used HIOA framework, were limited to build the formal specification, although the second research [35] has some kind of proofing but there is no concrete implementation that is equivalent to the abstract specification.

However, the contribution of our research is original as it is the first research in this area that formally specifies a modified version of the A* path planning algorithm, and formally refines the abstract specification to get the equivalent concrete implementation.

1.3 Potential contributions and limitations of the study

Our research aims to develop a reliable autonomous robot system that is capable to move safely from one point (start point) to another (target point) (see Figure 1.1), in which the robot has to reach the target safely by using the shortest path while avoiding the obstacles in the surrounding environment. Our major claim is to enhance reliability of the path planning issue in the autonomous robot system by following a formal process that starts by the formal specification and ends by the formal refinement process that results in an equivalent implementation that mirrors our specification.

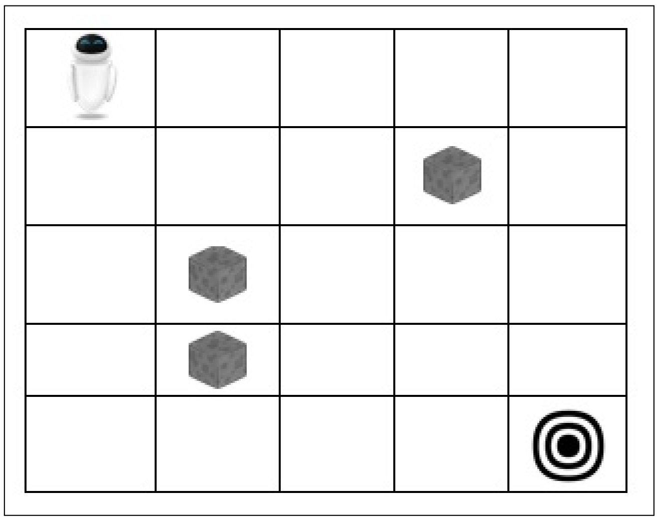


Figure 1.1: An example of path planning grid

Our study is limited to the navigation tasks in terms of path planning, as we think it is a fundamental functionality in any robot system.

We proves that out results are consistent and correct by using three different formalisms: (1) specification; (2) refinement; and (3) simulation. For the formal specification, we use Z notation to develop the needed schemas to describe the behavior of our system, which is a widely used specification language to specify critical or non-critical systems. For the refinement process, we develop refinement strategies that support the transformation from the specification into an implementation. The basic idea is to construct two types of semantic-preserving mappings. The first mapping refines data and the second one refines operations. A complete refinement may require several stages and thus can be viewed as composition of the intermediate mappings. By following the formal refinement, we derive the final implementation that mirrors our specification, then we develop a simulation that is consistent with the implementation to demonstrate the effectiveness of our software development approach.

The general objectives of this research are the following:

(1) To enhance reliability of the A* path planning algorithm and to become more convenient and safer for autonomous robots.

(2) To specify formally our new version of the A* path planning algorithm using Z notation.

(3) To derive an equivalent implementation that mirrors our formal model through a formal refinement process.

(4) To demonstrate the feasibility of our software development approach by building a simulation.

1.4 Outline of the thesis

This thesis is structured as follows: In chapter 2, we present our semi formal description of the problem, which is a mathematical description that we used as a base to build the formal specification. In chapter 3, we discuss our formal model (our Z document), which identifies the major schemas that describe the behavior of our system. In chapter 4, we introduce the formal refinement process that we follow to derive formally the equivalent implementation. In chapter 5, we show the implementation process and some test cases that prove the feasibility of our approach. In chapter 6, we analyze our findings by providing a general discussion and some suggestions for the possible future work, then we conclude our thesis.

Chapter 2: Semi-formal specification

In this chapter, we discuss the semi-formal description of our system, which is going to be used as a base to build our formal model. The semi-formal description is considered as a mixture of English language statements with mathematical statements that describe the overall behavior of our system. See Appendix A (Notations) that describes each notation that appears in this chapter.

2.1 Definitions

First of all, we need to introduce the set (pair) that includes all the pairs of natural numbers (x,y). We can generate these pairs by the Cartesian product of 2 sets of type natural numbers:

$$pair : \mathbb{N} \times \mathbb{N}$$

The map can be presented as a power set of the set pair, because we need to limit the size of the map by adding specific pairs to the map set (see Figure 2.1).

$$map : \mathbb{P}\, pair$$

We set the size of the map by giving a specific range to its elements; the range of the X value should be between 0 and maxX(any constant), and the range of the Y value should be between 0 and maxY (any constant) as follows:

$$maxX = const1$$

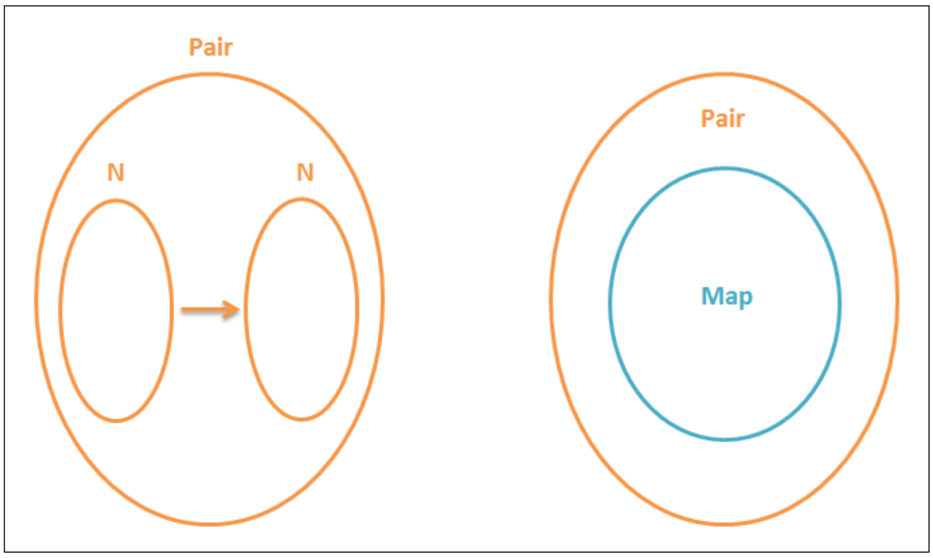


Figure 2.1: The pair and the map set

$$maxY = const2$$

$$\{\forall\, x_m : 0 \ldots maxX;\ \forall\, y_m : 0 \ldots maxY \bullet (x_m, y_m)\}$$

We need to have a border to surround the map, and it is going to be defined as a power set of the set Pair to hold the border coordinates.

$$border : \mathbb{P}\, Pair$$

The major need of having a border is to solve the problem of having undefined neighbors during the path finding process. For example, if the current position is located in the corner of the map, the system will not be able to identify the neighbors correctly as shown (in Figure 2.2); the system can recognize only 3 neighbors, and can't recognize the other neighbors.

The set border is initialized by taking the union of four sets that hold the border coordinates(see Figure 2.3).

$$border' = \{\{\forall\, x_p : 0 \ldots maxX \bullet (x_p, 0)\} \cup$$

$$\{\forall\, x_p : 0 \ldots maxX \bullet (x_p, maxY)\} \cup$$

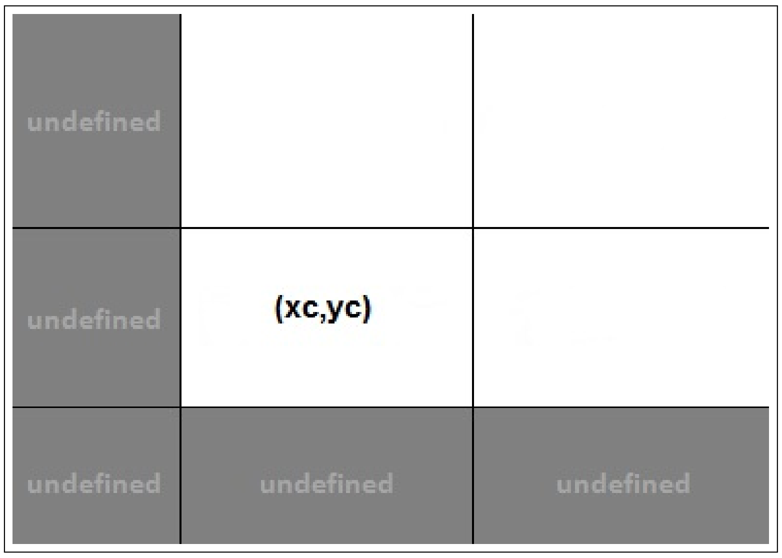


Figure 2.2: Example of undefined neighbors

$$\{\forall\, y_p : 0 \ldots maxY \bullet (0, y_p)\} \cup$$

$$\{\forall\, y_p : 0 \ldots maxY \bullet (maxX, y_p)\}$$

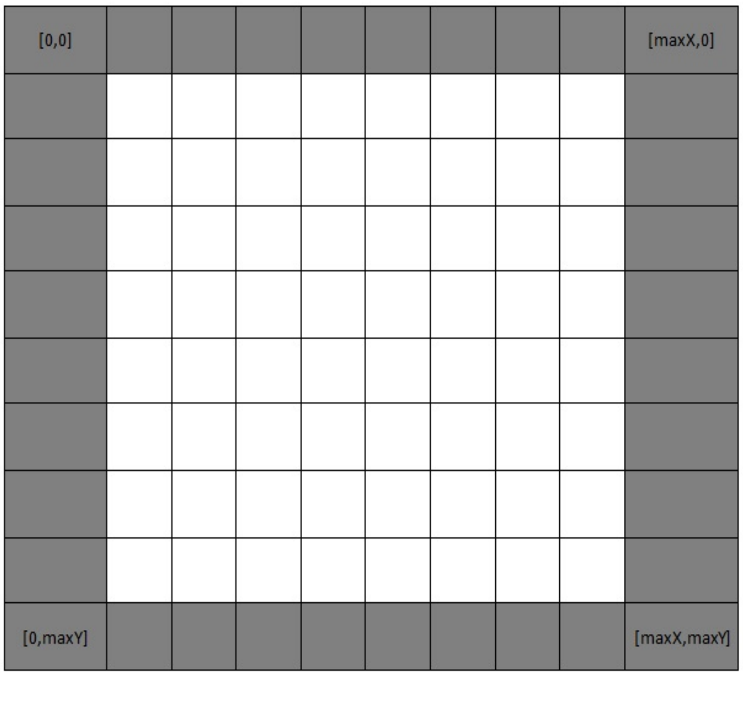


Figure 2.3: The range of the border coordinates

Then we start adding specific locations for the robot, the target and the obstacles; the pair (x_r, y_r) represents the robot location, the pair (x_t, y_t) represents the target location, and the obstacles will be represented as a set of pairs called (obstacleList).

Both the robot and the target locations should be initialized with an undefined value (any value that does not belong to the map set), and the obstacleList is initially empty.

$$undefined == \perp$$

$$(x'_r, y'_r) = (undefined, undefined)$$

$$(x'_t, y'_t) = (undefined, undefined)$$

$$obstacleList' = \varnothing$$

The several responses of the system will be defined as a set called RESPONSE as follows:

$$RESPONSE ::= FreeThePairFirst \mid NoPathFound \mid youCantFreeBorder \mid$$

$$ItsBorderPosition \mid APathIsFound$$

The different moves that can be done by the robot will be defines as a set called ROBOTMOVE as follows:

$$ROBOTMOVE ::= up \mid down \mid right \mid left \mid upRight \mid upLeft \mid downRight \mid downLeft$$

The set openF is declared as a bijective function that relates each pair to its F value, while the set openG is also declared as a bijective function to relate each pair to its G value.

$$openF : Pair \rightarrowtail\!\!\!\twoheadrightarrow \mathbb{N}$$

$$openG : Pair \rightarrowtail\!\!\!\twoheadrightarrow \mathbb{N}$$

The openF set will be initialized by adding the robot coordinates to the domain and its F value to the range, while the openG set will be initialized by adding the robot coordinates to the domain and its g value to the range.

$$openF' = \{(x_r \mapsto y_r) \mapsto f\}$$

$$openG' = \{(x_r \mapsto y_r) \mapsto g_c\}$$

We need to define some pairs to be used during the search process such as: the bestPair (x_b, y_b), which is the pair in the domain of the openF that is associated with the lowest F value; and the currentPair (x_c, y_c) is the pair that holds the coordinates of the current position during the search process. Both of these pairs are initialized with undefined values.

$$(x_b', y_b') = (undefined, undefined)$$

$$(x_c', y_c') = (undefined, undefined)$$

During the search process to find the path, we need to declare three power sets of type Pair (neighbors1,neighbors2 and neighbors3); the neighbors1 set is declared to hold the adjacent eight neighbors of the current position; the neighbors2 set is declared to hold only the safe neighbors after excluding the unsafe neighbors; and the neighbors3 set is declared to hold the free and the safe neighbors after excluding any neighbor that whether belongs to the border set or it is an obstacle, and also we need to declare a pair to hold the coordinates of any neighbor cell.

$$neighbors1 : \mathbb{P}\, Pair$$

$$neighbors2 : \mathbb{P}\, Pair$$

$$neighbors3 : \mathbb{P}\, Pair$$

$$(x_n, y_n) \;\; //neighbor\, pair$$

Initially these sets are empty while the neighbor pair is assigned to an undefined value.

$$neighbors1' = \varnothing$$

$$neighbors2' = \varnothing$$

$$neighbors3' = \varnothing$$

$$(x_n', y_n') = (undefined, undefined)$$

The pair (x_{cp}, y_{cp}) is defined to hold the current position of the robot and the pair (x_{np}, y_{np}) is defined to hold the next position of the robot. Both of these pairs are initialized with undefined values.

$$(x_{cp}', y_{cp}') = (undefined, undefined)$$

$$(x_{np}', y_{np}') = (undefined, undefined)$$

The set ParentChild is defined as a bijective function that relates each parent (of type Pair) to its child (of type Pair). Initially this set is empty.

$$parentChild : Pair \rightarrowtail\!\!\!\!\twoheadrightarrow Pair$$

$$parentChild' = \varnothing$$

The closedList is a set that holds the pairs that have been evaluated before. Initially this set is empty.

$$closedList : \mathbb{P}\, Pair$$

$$closedList' = \varnothing$$

The final path is defined as a sequence of pairs (the sequence is equivalent to a function that relates the natural numbers set to the pair set: $path : \mathbb{N} \nrightarrow Pair$), and the pair (x_{path}, y_{path}) is one of the path elements.

$$path : \mathrm{seq}\, Pair$$

$$(x_{path}, y_{path}) \;\; //path\, pair$$

The path set will be initialized by adding the target position, while the path pair is initialized by the target coordinates.

$$path' = path \frown \langle (x_t, y_t) \rangle$$

$$(x'_{path}, y'_{path}) = (x_t, y_t)$$

2.2 Operations

2.2.1 Set the target position

To set the target position, the input value should belong to the map set ,and the system should check if the target coordinates (x_t, y_t) are already specified before or not because it is not allowed to have more than one target location. If the target location is not specified before then reserve the entered pair as the target location.

$$(x?, y?) \in map$$

$$(x_t, y_t) = (undefined, undefined)$$

$$(x'_t, y'_t) = (x?, y?)$$

In case, the target location was assigned before, and the system needs to change the target position, then the new coordinates should belong to the map set, and the system should replace the old coordinates by the new coordinates.

$$(x?, y?) \in map$$

$$(x_t, y_t) \neq (undefined, undefined)$$

$$(x'_t, y'_t) = (x?, y?)$$

The operation of setting the target will be prohibited in 2 cases:

1. If the input value belongs to obstacleList set, the user should free the pair first then assign it as a target location.

$$rep : RESPONSE$$

$$(x?, y?) \in obstacleList$$

$$rep! = freePairFirst$$

2. If the input value belongs to the border set, the system responses by “it is a border position” , and nothing will be changed as it is not allowed to assign a target position on the border.

$$rep : RESPONSE$$

$$(x?, y?) \in border$$

$$rep! = ItsBorderPosition$$

2.2.2 Set the robot position

To set the robot position, the input value should belong to the map set, and the system should check if the robot coordinates (x_r, y_r) are already specified before or not because it is not allowed to have more than one robot location. So if the robot location is not specified before then reserve the entered pair as the robot location.

$$(x?, y?) \in map$$

$$(x_r, y_r) = (undefined, undefined)$$

$$(x_r', y_r') = (x?, y?)$$

In case, the robot location was assigned before, and the system needs to change the robot position, then the new coordinates should belong to the map set, and the system should replace the old coordinates by the new coordinates.

$$(x?, y?) \in map$$

$$(x_r, y_r) \neq (\mathit{undefined}, \mathit{undefined})$$

$$(x_r', y_r') = (x?, y?)$$

The operation of setting the robot will be prohibited in 2 cases:

1. If the input value belongs to obstacleList set, the user should free the pair first then assign it as a robot location.

$$rep : RESPONSE$$

$$(x?, y?) \in obstacleList$$

$$rep! = freePairFirst$$

2. If the input value belongs to the border set, the system responses by “it is a border position” , and nothing will be changed as it is not allowed to assign a robot position on the border.

$$rep : RESPONSE$$

$$(x?, y?) \in border$$

$$rep! = ItsBorderPosition$$

2.2.3 Set obstacle Position

To add an obstacle position to the obstacleList, the input value should belong to the map set and should not be included in the obstacleList set.

$$(x?, y?) \in map$$

$$(x?, y?) \notin \mathit{obstacleList}$$

$$\mathit{obstacleList}' = \mathit{obstacleList} \cup \{(x?, y?)\}$$

This operation will be prohibited in one case:

1. If the input value belongs to the border set, the system responses by "it is a border position", and nothing will be changed as it is not allowed to set an obstacle position on the border.

$$\mathit{rep} : \mathit{RESPONSE}$$

$$(x?, y?) \in \mathit{border}$$

$$\mathit{rep}! = \mathit{ItsBorderPosition}$$

2.2.4 Free robot Position

To free the robot position, the system should check whether the input value equals to the robot position; if yes, then the robot position will be assigned to an undefined value.

$$(x?, y?) = (x_r, y_r)$$

$$(x_r', y_r') = (\mathit{undefined}, \mathit{undefined})$$

2.2.5 Free target Position

To free the target position, the system should check whether the input value equals to the target position; if yes, then the target position will be assigned to an undefined value.

$$(x?, y?) = (x_r, y_r)$$

$$(x'_t, y'_t) = (\mathit{undefined}, \mathit{undefined})$$

2.2.6 Free obstacle Position

To free an obstacle position, the system should check if the input value belongs to the obstacleList, then this position will be deleted from the obstacleList.

$$(x?, y?) \in \mathit{obstacleList}$$

$$\mathit{obstacleList}' = \mathit{obstacleList} \setminus \{(x?, y?)\}$$

2.2.7 Free border pair

This operation will be prohibited, because it is not allowed to free any of the border coordinates.

$$\mathit{rep} : \mathit{RESPONSE}$$

$$(x?, y?) \in \mathit{border}$$

$$\mathit{rep}! = \mathit{youCantFreeBorder}$$

2.2.8 Evaluate the best pair

This operation is responsible to retrieve the best pair (x_b, y_b); the best pair is the domain of the openF that is associated with the lowest range of the openF.

$$(x'_b, y'_b) = \mathit{dom}(\mathit{openF} \rhd \mathit{min}(\mathit{ran}\ \mathit{openF}))$$

2.2.9 Search pairs no path

If the best pair is not equal to the target and the openF is empty so there is no path found.

$$rep : RESPONSE$$

$$(x_b, y_b) \neq (x_t, y_t)$$

$$openF = \varnothing$$

$$rep! = NoPathFound$$

2.2.10 Construct path

If the bestPair equals to the target, the system should construct a path by following the parent of each pair.

$$rep : RESPONSE$$

$$(x_b, y_b) = (x_t, y_t)$$

$$rep! = APathIsFound$$

$$(x_{path}, y_{path}) \neq (undefined, undefined)$$

$$(x_{path}, y_{path}) \neq (x_r, y_r)$$

$$(x'_{path}, y'_{path}) = dom(parentChild \rhd (x_{path}, y_{path}))$$

$$path' = path \cup \{(x_{path}, y_{path})\}$$

2.2.11 Search pairs

While the openF set is not empty, if the best pair is not equal to the target

$$(x_b, y_b) \neq (x_t, y_t)$$

The system should empty the neighbors1, neighbors2 and neighbors3 sets first to make sure that the system will not reevaluate the old neighbors again, and only the neighbors of the current location will be evaluated.

$$neighbors1' = \varnothing$$

$$neighbors2' = \varnothing$$

$$neighbors3' = \varnothing$$

Then the system will start the searching process, by assigning the best pair to the current pair, adding the best pair to the closedList, deleting the best pair from both openF and openG, and adding the neighbors of the current pair to the neighbors1 set.

$$(x_c, y_c) = (x_b, y_b)$$

$$closedList' = closedList \cup \{(x_b, y_b)\}$$

$$openF' = \{(x_b, y_b)\} \lhd openF$$

$$openG' = \{(x_b, y_b)\} \lhd openG$$

$$neighbors1' = \{\{(x_c, y_c + 1)\} \cup \{(x_c, y_c - 1)\} \cup \{(x_c + 1, y_c)\} \cup \{(x_c - 1, y_c)\}$$
$$\cup \{(x_c - 1, y_c + 1)\} \cup \{(x_c - 1, y_c - 1)\} \cup \{(x_c + 1, y_c - 1)\} \cup \{(x_c + 1, y_c + 1)\}\}$$

2.2.12 Refine diagonal neighbors

After adding the 8 possible neighbors to the neighbors1 set,These neighbors should be evaluated to determine which neighbors we should keep and which we should discard. This operation is responsible to discard some of the unwanted diagonal neighbors from the neighbors1 set by adding these neighbors to a temporary placeholder(the neighbor2 set). These neighbors will be considered unwanted when they match any of the following 8 cases:

Case 1: If there is an obstacle behind the robot and at the right side of the robot. The robot will not be able to move diagonally to the down right direction (see Figure 3.2).

$$(x_c, y_c + 1) \in neighbors1 \land ((x_c, y_c + 1) \in obstacleList$$

$$(x_c + 1, y_c) \in neighbors1 \land (x_c + 1, y_c) \in obstacleList$$

$$neighbors2' = neighbors2 \cup \{(x_c + 1, y_c + 1)\}$$

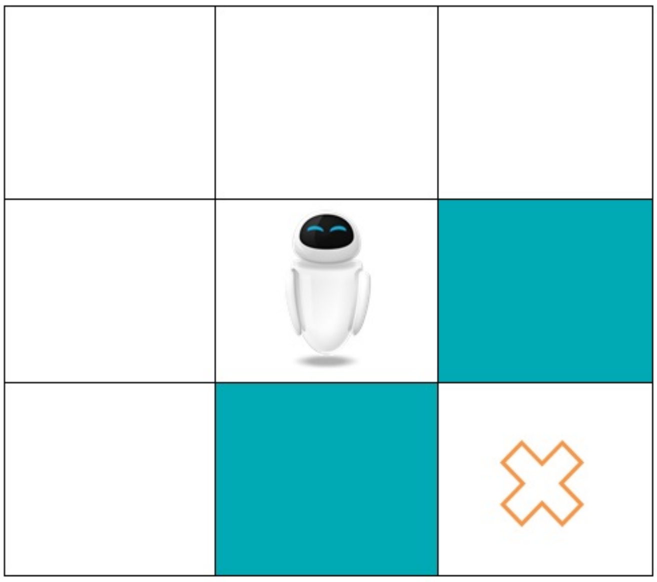


Figure 2.4: Case1: The down-right neighbor will be excluded

Case 2: If there is an obstacle behind the robot and at the left side of the robot. The robot will not be able to move diagonally to the down left direction (see Figure 2.5).

$$(x_c, y_c+1) \in neighbors1 \wedge (x_c, y_c+1) \in obstacleList$$

$$(x_c-1, y_c) \in neighbors1 \wedge (x_c-1, y_c) \in obstacleList$$

$$neighbors2' = neighbors2 \cup \{(x_c-1, y_c+1)\}$$

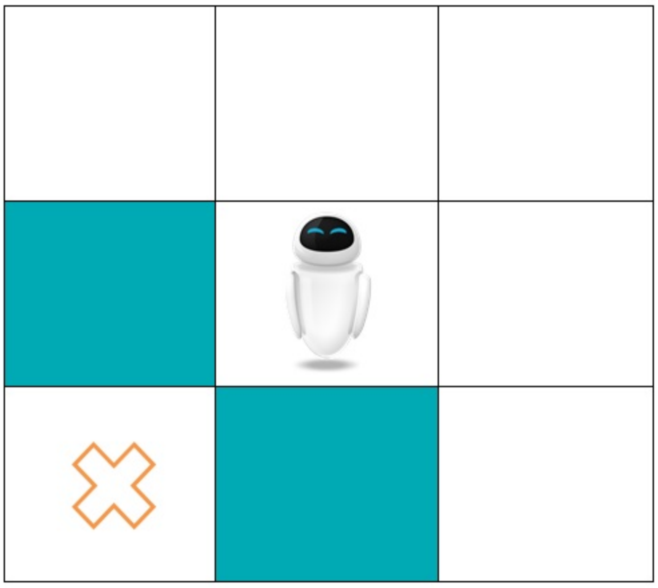


Figure 2.5: Case2: The down-left neighbor will be excluded

Case 3: If there is an obstacle in front of the robot and at the right side of the robot. The robot will not be able to move diagonally to the up right direction(see Figure 2.6).

$$(x_c, y_c-1) \in neighbors1 \wedge (x_c, y_c-1) \in obstacleList$$

$$(x_c+1, y_c) \in neighbors1 \wedge (x_c+1, y_c) \in obstacleList$$

$$neighbors2' = neighbors2 \cup \{(x_c+1, y_c-1)\}$$

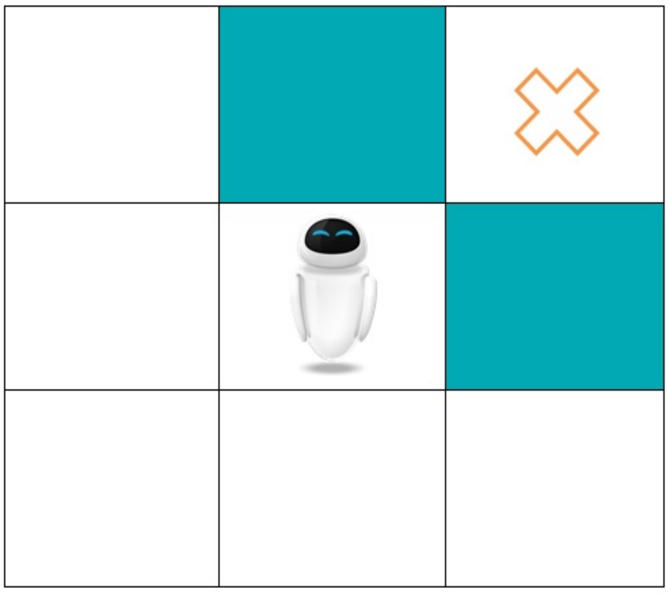


Figure 2.6: Case3: The up-right neighbor will be excluded

Case 4: If there is an obstacle in front of the robot and at the left side of the robot. The robot will not be able to move diagonally to the up left direction (see Figure 2.7) .

$$(x_c, y_c - 1) \in neighbors1 \wedge (x_c, y_c - 1) \in obstacleList$$

$$(x_c - 1, y_c) \in neighbors1 \wedge (x_c - 1, y_c) \in obstacleList$$

$$neighbors2' = neighbors2 \cup \{(x_c - 1, y_c - 1)\}$$

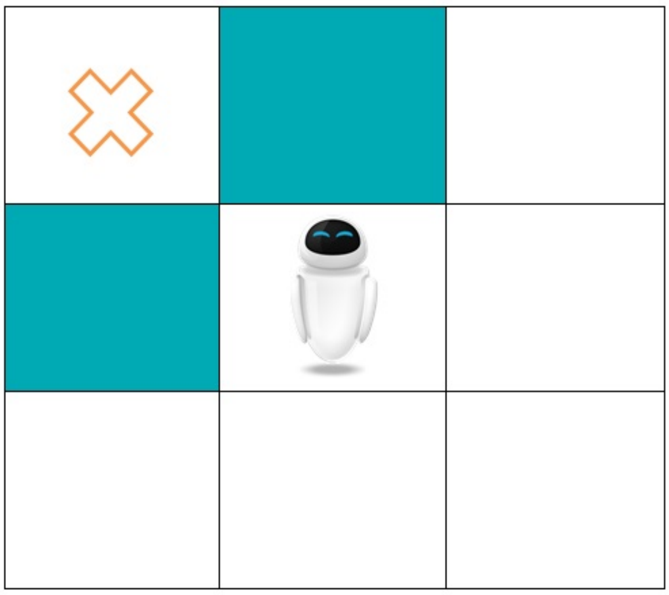


Figure 2.7: Case4: The up-left neighbor will be excluded

Case 5: If there is an obstacle behind the robot. The robot will not be able to move diagonally to the down right or to the down left direction (see Figure 2.8).

$$(x_c, y_c+1) \in neighbors1 \land (x_c, y_c+1) \in obstacleList$$

$$neighbors2' = neighbors2 \cup \{(x_c+1, y_c+1)\} \cup \{(x_c-1, y_c+1)\}$$

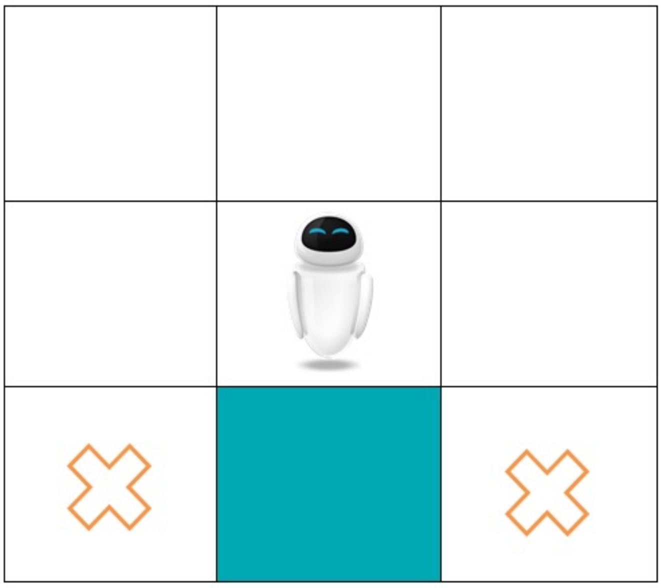


Figure 2.8: Case5: The down-left and the down-right neighbor will be excluded

Case 6: If there is an obstacle in front of the robot. The robot will not be able to move diagonally to the up right or to the up left direction (see Figure 2.9).

$$(x_c, y_c - 1) \in neighbors1 \wedge (x_c, y_c - 1) \in obstacleList$$

$$neighbors2' = neighbors2 \cup \{(x_c + 1, y_c - 1)\} \cup \{(x_c - 1, y_c - 1)\}$$

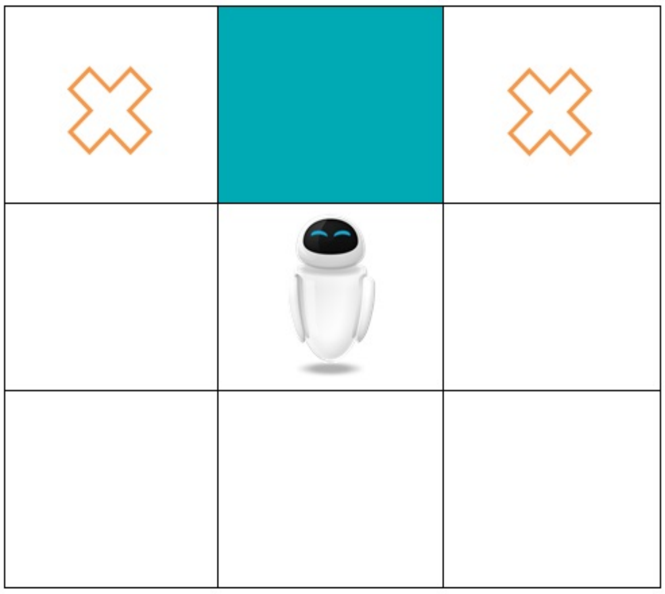


Figure 2.9: Case6: The up-left and the up-right neighbor will be excluded

Case 7: If there is an obstacle at the right side of the robot. The robot will not be able to move diagonally to the up right or to the down right direction (see Figure 2.10).

$$(x_c + 1, y_c) \in neighbors1 \wedge (x_c + 1, y_c) \in obstacleList$$

$$neighbors2' = neighbors2 \cup \{(x_c + 1, y_c + 1)\} \cup \{(x_c + 1, y_c - 1)\}$$

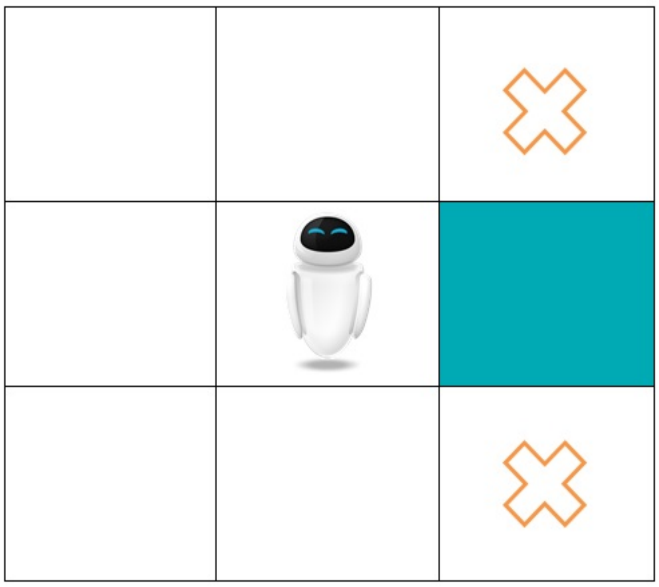


Figure 2.10: Case7: The up-right and the down-right neighbor will be excluded

Case 8: If there is an obstacle at the left side of the robot. The robot will not be able to move diagonally to the up left or to the down left direction (see Figure 2.11).

$$(x_c - 1, y_c) \in neighbors1 \wedge (x_c - 1, y_c) \in obstacleList$$

$$neighbors2' = neighbors2 \cup \{(x_c - 1, y_c + 1)\} \cup \{(x_c - 1, y_c - 1)\}$$

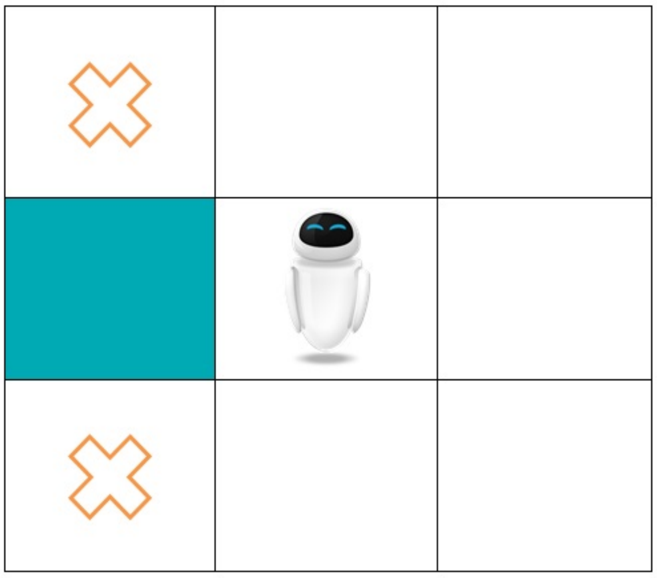


Figure 2.11: Case8: The up-left and the down-left neighbor will be excluded

2.2.13 Delete diagonal neighbors

After adding the unwanted neighbors to the neighbors2 set, the system should delete the elements of neighbors2 set from neighbors1.

$$neighbors1' = neighbors1 \backslash neighbors2$$

2.2.14 Refine neighbors from obstacles and border

The system should check if there is any neighbor that belongs to the obstaclList set or to the border set, and then add it temporary to neighbors3

$$((x_n, y_n) \in neighbors1$$

$$(x_n, y_n) \in obstacleList$$

$$neighbors3' = neighbors3 \cup (x_n, y_n))$$

$$\vee$$

$$((x_n, y_n) \in neighbors1$$

$$(x_n, y_n) \in border$$

$$neighbors3' = neighbors3 \cup (x_n, y_n))$$

2.2.15 Delete neighbors from obstacles and border

The system should delete the elements of neighbors1 that belong to the set neighbors3 , to make sure that the pairs that belong to neighbors1 are all free and ready to be evaluated.

$$neighbors1' = neighbors1 \setminus neighbors3$$

2.2.16 Calculate H Value

This operation is designed to calculate the H value of any neighbor, which is equivalent to the Manhattan distance between the neighbor and the target.

$$h : \mathbb{N}$$

$$h' = \mid y_n - y_t \mid + \mid x_n - x_t \mid$$

2.2.17 Calculate G Current

This operation is designed to calculate the G value of the current pair, which is equivalent to the Manhattan distance between the current pair and the robot pair.

$$g_c : \mathbb{N}$$

$$g_c' = \mid y_c - y_r \mid + \mid x_c - x_r \mid$$

2.2.18 Calculate G neighbor

This operation is designed to calculate the G value of any neighbor pair, which is equivalent to the Manhattan distance between the neighbor pair and the robot pair.

$$g_n : \mathbb{N}$$

$$g_n' = \mid y_n - y_r \mid + \mid x_n - x_r \mid$$

2.2.19 Calculate F value

This operation is designed to calculate the F value of any neighbor pair, while the f value is the sum of the G value and the H value (F = G+H).

$$f : \mathbb{N}$$

$$f' = g_c + h$$

2.2.20 Calculate Cost Value

This operation is designed to calculate the Cost value of any neighbor pair, which is equivalent to the sum of G value of the current pair and the distance between the current pair and the neighbor.

$$cost : \mathbb{N}$$

$$cost' = gc + (\mid y_c - y_n \mid + \mid x_c - x_n \mid)$$

2.2.21 Evaluate neighbors

The system should evaluate each neighbor based on its existence in the openF as the following cases:

Case 1: If the neighbor belongs already to the openF, and the cost was less than the old G vlaue of the neighbor, then delete this neighbor from both the openF and the openG sets because the new path is better.

$$oldGn : \mathbb{N}$$

$$(x_n, y_n) \in openF$$

$$oldGn' = openG(x_n, y_n)$$

$$cost < oldGn$$

$$openF' = \{(x_n, y_n)\} \lhd openF$$

$$openG' = \{(x_n, y_n)\} \lhd openG$$

Case 2: If the neighbor does belong to the openF or the closedList, then assign the g value of the neighbor to the cost, add this neighbor to both of the openG and the openF, and assign the current node as the parent of this neighbor.

$$newGn : \mathbb{N}$$

$$(x_n, y_n) \notin openF$$

$$(x_n, y_n) \notin closedList$$

$$newGn' = cost$$

$$openF' = openF \cup \{(x_n \mapsto y_n) \mapsto (newGn + h)\}$$

$$openG' = openG \cup \{(x_n \mapsto y_n) \mapsto newGn\}$$

$$parentChild' = parentChild \cup \{(x_c, y_c) \mapsto (x_n, y_n)\}$$

2.2.22 Follow path

To let the robot follow the path, the system should take the head of the path sequence and store it as a current position, then remove it from the sequence then make the new head of the path sequence as the next position.

$$path \neq \langle\rangle$$

$$(x'_{cp}, y'_{cp}) = head\, path$$

$$path' = tail\; path$$

$$(x'_{np}, y'_{np}) = head\; path$$

After that, the system should evaluate the next position to determine the upcoming robot movement, while the basic moves of the robot are listed in ROBOT-MOVE set as follows: up, down,right, left, upRight, upLeft, downRight, downLeft (see Figure 2.12). The next move of the robot will be determined by comparing the coordinates of the current position with the coordinates of the next position (see Figure 2.13).

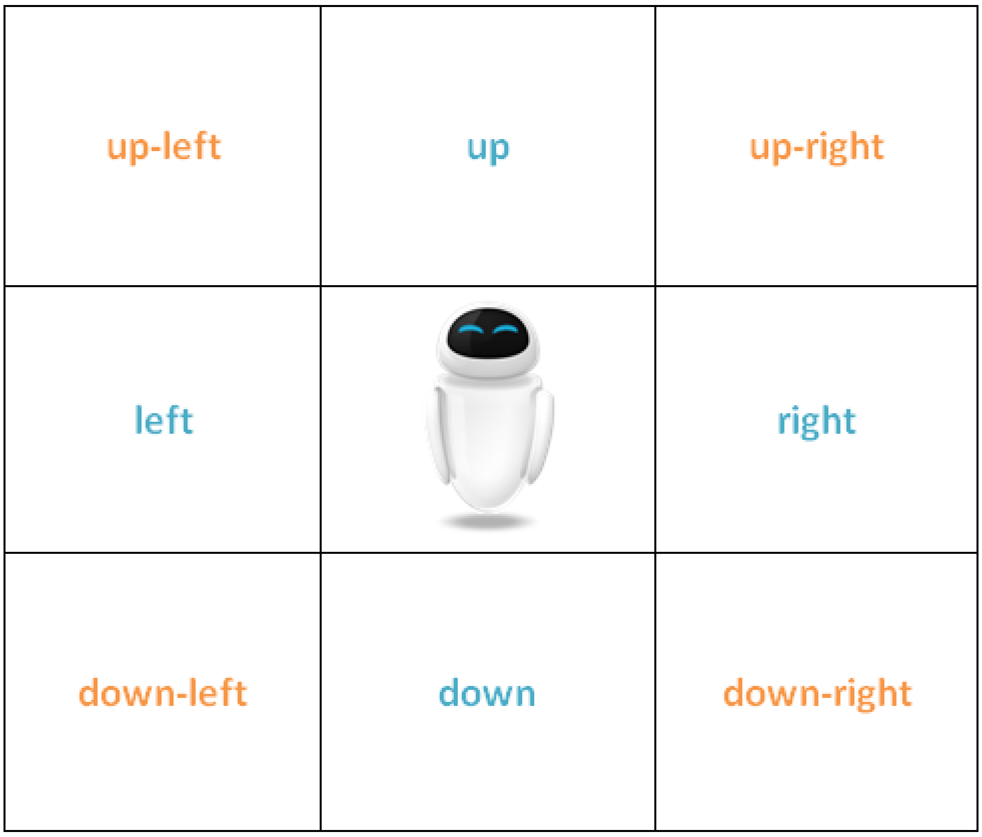


Figure 2.12: The directions of the robot moves

After the robot movement the current robot position should be replaced by the next position to prepare the robot for the next movement.

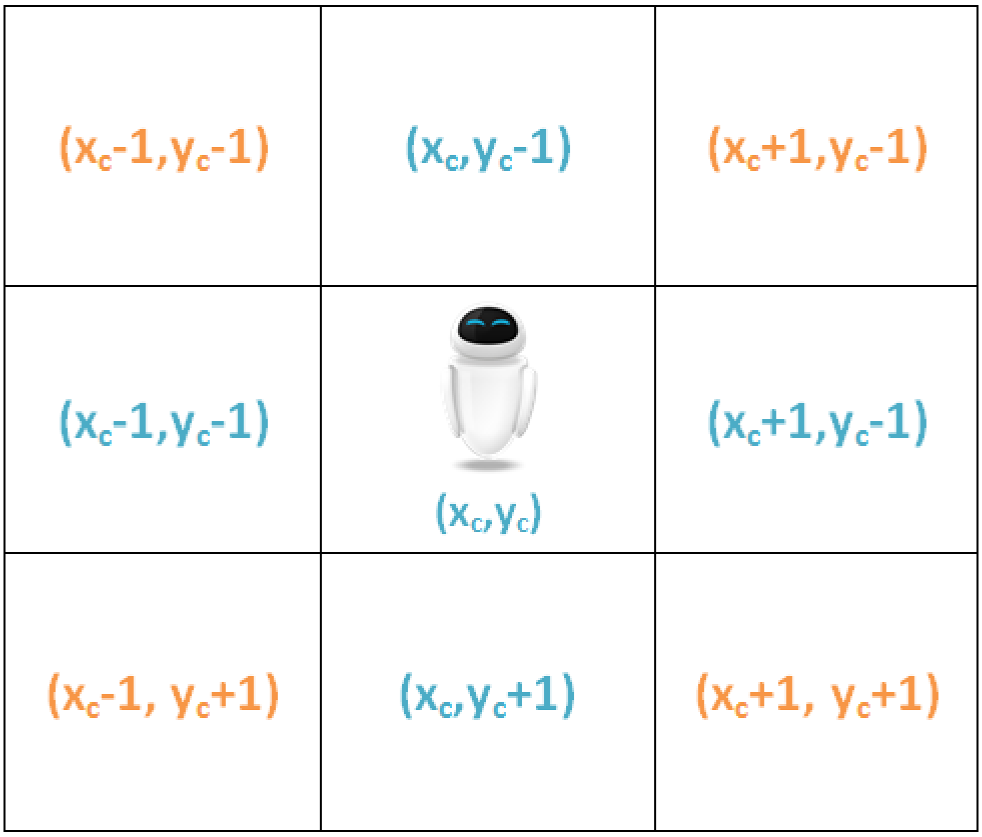


Figure 2.13: The way of determining the next robot movement

2.2.23 Move down

The robot will move down if the next position is equivalent to $(x_{cp}, y_{cp} + 1)$.

$$rm : ROBOTMOVE$$

$$(x_{np}, y_{np}) = (x_{cp}, y_{cp} + 1)$$

$$rm' = down$$

$$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$$

2.2.24 Move up

The robot will move up if the next position is equivalent to $(x_{cp}, y_{cp} - 1)$.

$$rm : ROBOTMOVE$$

$$(x_{np}, y_{np}) = (x_{cp}, y_{cp} - 1)$$

$$rm' = up$$

$$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$$

2.2.25 Move right

The robot will move right if the next position is equivalent to $(x_{cp} + 1, y_{cp})$.

$$rm : ROBOTMOVE$$

$$(x_{np}, y_{np}) = (x_{cp} + 1, y_{cp})$$

$$rm' = right$$

$$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$$

2.2.26 Move left

The robot will move left if the next position is equivalent to $(x_{cp} - 1, y_{cp})$.

$$rm : ROBOTMOVE$$

$$(x_{np}, y_{np}) = (x_{cp} - 1, y_{cp})$$

$$rm' = left$$

$$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$$

2.2.27 Move up right

The robot will move diagonally to the up right direction if the next position is equivalent to $(x_{cp} + 1, y_{cp} - 1)$.

$$rm : ROBOTMOVE$$

$$(x_{np}, y_{np}) = (x_{cp} + 1, y_{cp} - 1)$$

$$rm' = upRight$$

$$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$$

2.2.28 Move up left

The robot will move diagonally to the up left direction if the next position is equivalent to $(x_{cp} - 1, y_{cp} - 1)$.

$$rm : ROBOTMOVE$$

$$(x_{np}, y_{np}) = (x_{cp} - 1, y_{cp} - 1)$$

$$rm' = upLeft$$

$$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$$

2.2.29 Move down right

The robot will move diagonally to the down right direction if the next position is equivalent to $(x_{cp} + 1, y_{cp} + 1)$.

$$rm : ROBOTMOVE$$

$$(x_{np}, y_{np}) = (x_{cp} + 1, y_{cp} + 1)$$

$$rm' = downRight$$

$$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$$

2.2.30 Move down left

The robot will move diagonally to the down left direction if the next position is equivalent to $(x_{cp} + 1, y_{cp} + 1)$.

$$rm : ROBOTMOVE$$

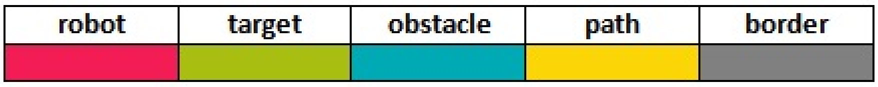
$$(x_{np}, y_{np}) = (x_{cp} - 1, y_{cp} + 1)$$

$$rm' = downLeft$$

$$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$$

2.3 Example of finding the shortest path

In this example the robot coordinates are (2,6) and the color of the robot cell is fuchsia , the target coordinates are (4,6) and this cell is colored by green, the border cells are colored by grey, the obstacles cells are colored by blue, the final path will be colored by orange, and the free cells are all colored by white (see Figure 2.14). The map is shown in Figure 2.15



robot	target	obstacle	path	border
🟥	🟩	🟦	🟨	⬛

Figure 2.14: The color palette of the different locations in the map

1. The system will initialize the openF set by adding the robot coordinates (x_r, y_r) to its domain and calculating the F value then add it to the range of the function

dom openF	ran openF
(2,6)	2

2. The system will initialize the openG set by adding the robot coordinates (x_r, y_r) to its domain and the calculating the G value then add it to the range of the function

dom openG	ran openG
(2,6)	0

3. As long as the robot position is the only element in the openF set so it is going to be considered as the bestPair.

4. The system should check if the bestPair is equal to the target position or not

$$(x_r, y_r) \neq (x_t, y_t)$$

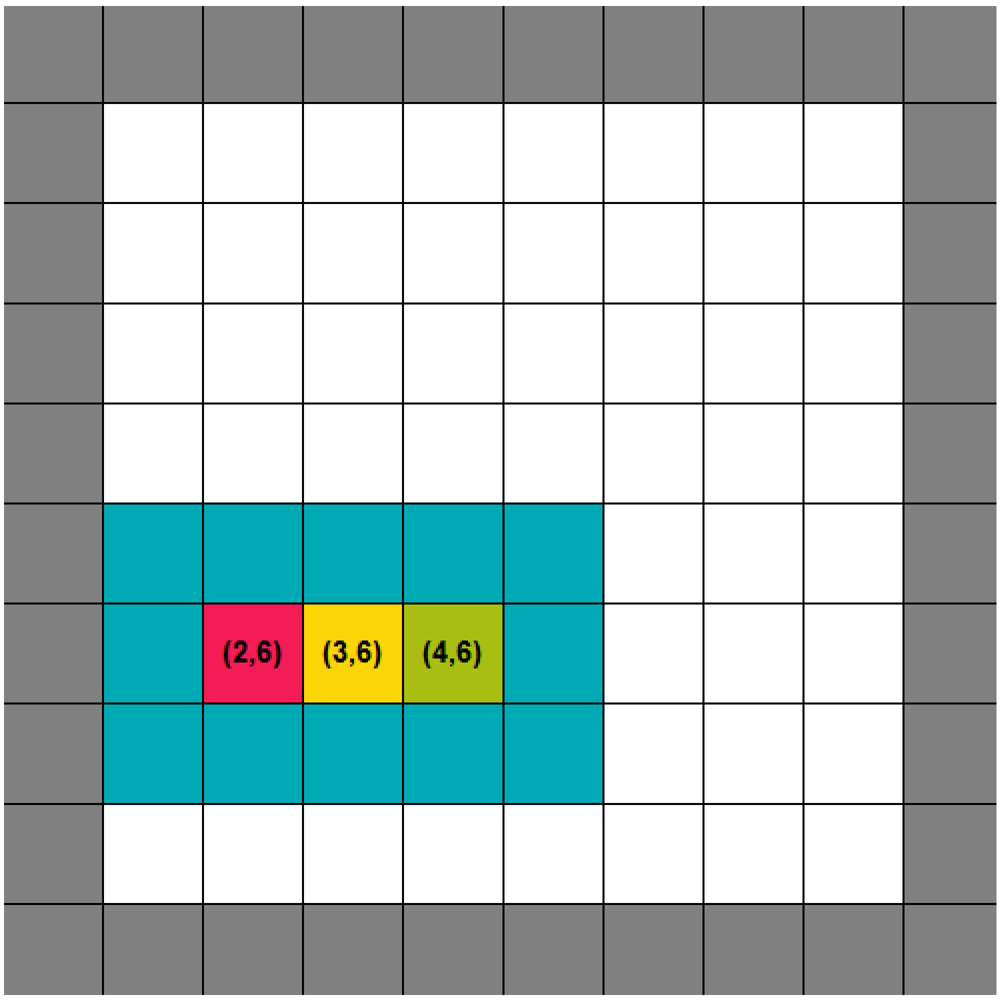


Figure 2.15: Example of finding the shortest path

$$(2,6) \neq (4,6)$$

5. If the bestPair is not equal to the target, make the robot position as the currentPair

$$(x_c, y_c) = (2,6)$$

6. Then remove the bestPair from both the openF and the openG, and Add it to the closedList.

closedList
(2,6)

7. Before Finding the neighbors of the currentPair, the system should empty neighbors1, neighbors2, neighbors3 sets:

$$neighbors1 = \varnothing$$

$$neighbors2 = \varnothing$$

$$neighbors3 = \varnothing$$

8. The eight neighbors of the currentPair will be added to neighbors1 set

$$neighbors1 = \{(3,6),(1,6),(2,7),(2,5),(3,7),(1,7),(3,5),(1,5)\}$$

9. The system will refine the diagonal neighbors, by adding the unwanted neighbors temporary to the set neighbors2, in case there is a neighbor matches any of the eight cases that were discussed before.

   $(1,6) and (2,7)$ will be added to neighbors2 because they match case 2

   $(2,5) and (1,6)$ will be added to neighbors2 because they match case 4

   $(1,7) and (3,7)$ will be added to neighbors2 because they match case 5

   $(3,5) and (1,5)$ will be added to neighbors2 because they match case 6

$(2,5) and (2,7)$ will be added to neighbors2 because they match case 8

10. Then the system will delete the elements that belong to the neighbors2 set from neighbors1 set, so the neighbors1 is going to be equivalent to:

$$neighbors1' = neighbors1 \backslash neighbors2$$

$$neighbors1 = \{(3,6)\}$$

11. One more time, the system will refine the neighbors1 set , by adding any neighbor that belong to the border set or to obstacleList set temporary to the neighbors 3 set and then delete them later on. but in this round, there is not any neighbor that belongs to border or obstacleList sets.

12. The system should evaluate each neighbor based on its existence in the openF set:

$$neighbors1 = \{(3,6)\}$$

The neighbor (3,6) neither belong to the openF nor belong to the closedList, so we should calculate the cost of each neighbor, assign its G value to the cost, add the neighbor to both of the openF and the openG, then make the currentPair as the parent of this neighbor.

dom openG	ran openG
(3,6)	1

dom openF	ran openF
(3,6)	2

dom parentChild	dom parentChild
(2,6)	(3,6)

One more round, the bestPair is the pair in the domain of the openF that is associated with the lowest range(the minimum F value):

(3,6) is the bestPair

1. If the bestPair is not equal to the target, make the bestPair as the currentPair

$$(x_c, y_c) = (3,6)$$

2. Then remove the bestPair from both the openF and the openG, and Add it to the closedList.

dom openG	ran openG
-	-

dom openF	ran openF
-	-

closedList
(3,6)
(2,6)

3. Before Finding the neighbors of the currentPair, the system should empty neighbors1, neighbors2, neighbors3 sets:

$$neighbors1 = \varnothing$$

$$neighbors2 = \varnothing$$

$$neighbors3 = \varnothing$$

4. The eight neighbors of the currentPair will be added to neighbors1 set

$$neighbors1 = \{(4,6),(2,6),(3,7),(3,5),(4,7),(2,7),(4,5),(2,5)\}$$

5. The system will refine the diagonal neighbors, by adding the unwanted neighbors temporary to the set neighbors2, in case there is any neighbor matches any of the eight cases that were discussed before.

$(2,5) and (4,5)$ will be added to neighbors2 because it matches case 6

$(2,7) and (4,7)$ will be added to neighbors2 because they match case 5

6. Then the system will delete the elements that belong to the neighbors2 from neighbors1 set, so the neighbors1 is going to be equivalent to:

$$neighbors1' = neighbors1 \backslash neighbors2$$

$$neighbors1 = \{(4,6),(2,6),(3,7),(3,5)\}$$

7. One more time, the system will refine the neighbors1 set , by adding any neighbor that belong to the border set or to obstacleList set temporary to the neighbors 3 set.

   $(3,7) and (3,5)$ will be added to neighbors3 because these pairs belong to the obstacleList set.

8. The system will delete any element belong to the neighbors3 set from the neighbors1 set, so the neighbors1 set is equivalent to:

$$neighbors1' = neighbors1 \backslash neighbors3$$

$$neighbors1 = \{(4,6),(2,6)\}$$

9. Then the system should evaluate each neighbor based on its existence in the openF set:

$$neighbors1 = \{(4,6),(2,6)\}$$

   The pair (2,6) is already in the closedList so the system will ignore it.

   The pair (4,6) neither belong to the openF nor belong to the closedList, so we should calculate the cost of each neighbor, assign its G value to the cost, add the neighbor to both of the openF and the openG, then make the currentPair as the parent of this neighbor.

dom openG	ran openG
(4,6)	2

dom openF	ran openF
(4,6)	2

dom parentChild	dom parentChild
(3,6)	(4,6)
(2,6)	(3,6)

One more round, the bestPair is the pair in the domain of the openF that is associated with the lowest range(the minimum F value):

(4,6) is the bestPair

1. If the bestPair is not equal to the target, make the bestPair as the currentPair. In this round the bestPair equals the target.

$$(4,6) = (x_t, y_t)$$

So the system found a path; the path will be constructed by following the parent of each pair:

(4,6) is the bestPair

The parent of the target pair is (3,6), and the parent of the pair(3,6) is (2,6), so the final path is: $(4,6) -> (3,6) -> (2,6)$

The required robot moves to reach the target are:

$$rm : ROBOTMOVE$$

$$(x_{cp}, y_{cp}) = (2,6)$$

$$(x_{np}, y_{np}) = (3,6)$$

dom parentChild	dom parentChild
(3,6)	(4,6)
(2,6)	(3,6)

As long as: $(x_{np}, y_{np}) = (x_{cp} + 1, y_{cp})$

Then $rm' = right$

After the robot movement the current position will be updated by the next position, and the next position will be updated by a new the next value from the path sequence.

$(x_{cp}, y_{cp}) = (3,6)$

$(x_{np}, y_{np}) = (4,6)$

As long as: $(x_{np}, y_{np}) = (x_{cp} + 1, y_{cp})$

Then $rm' = right$

So finally, the required robot moves to reach the target are:

$$right- > right$$

Chapter 3: Formal specification

The previous chapter forms a basis for the elaboration of the formal model. Here, we re-express our system formally using Z notation, by generating the equivalent schemas that are used to describe the different states and operations of our system. This chapter is organized as follows: The following section (3.1) describes the needed definitions and types, while section (3.2) mentions the system status. Section (3.3) specifies some of the major operations, while section (3.4) discusses how we formally prove some properties like the robot safety.

3.1 Definitions

The type Pair is defined as the Cartesian product of $(\mathbb{N} \times \mathbb{N})$, which identifies the set of coordinates x and y. We introduce two constants maxX and maxY, which present the maximum X value and the maximum Y value of any coordinate that belongs to the map, and they are initialized by const1 (any constant) and const2 (any constant) accordingly. We introduce an undefined constant as a global variable that holds an undefined value ($\perp$). The several responses of the system are declared as a free type named RESPONSE; also, the robot movements are declared as a free type named ROBOTMOVE.

$$maxX = const1$$

-- The maximum X value of any coordinate that belongs to the map.

$maxY = const2$

-- The maximum Y value of any coordinate that belongs to the map.

$Pair = \mathbb{N} \times \mathbb{N}$

-- Identifying the type Pair

$undefined == \perp$

-- Identifying the constant with the undefined value

$RESPONSE ::= FreeThePairFirst \mid NoPathFound \mid youCantFreeBorder \mid ItsBorderPosition \mid APathIsFound$

-- The several responses of the system

$ROBOTMOVE ::= up \mid down \mid right \mid left \mid upRight \mid upLeft \mid downRight \mid downLeft$

-- The several movements of the robot.

3.2 System status

In the following schema, a set of type Pair called map is declared. which presents the set of the map coordinates. The state of the map is defined as:

Map

$map : \mathbb{P}\, Pair$

The map set is initialized, by giving a range to the pairs that belong to the map. For any pair (x_m, y_m) that belongs to map, its X value should be between 0 ... maxX and its Y value should be between 0 ... maxY. The initial state of the map is defined as:

$$\begin{array}{|l}
\text{\textit{MapInit}} \\
\Delta Map \\
\hline
map' = \{\forall\, x_m : 0 \ldots maxX;\ \forall\, y_m : 0 \ldots maxY \bullet (x_m, y_m)\} \\
\hline
\end{array}$$

In the following schema, a set of type Pair called border is declared, which holds the border coordinates (see Figure 3.1) .

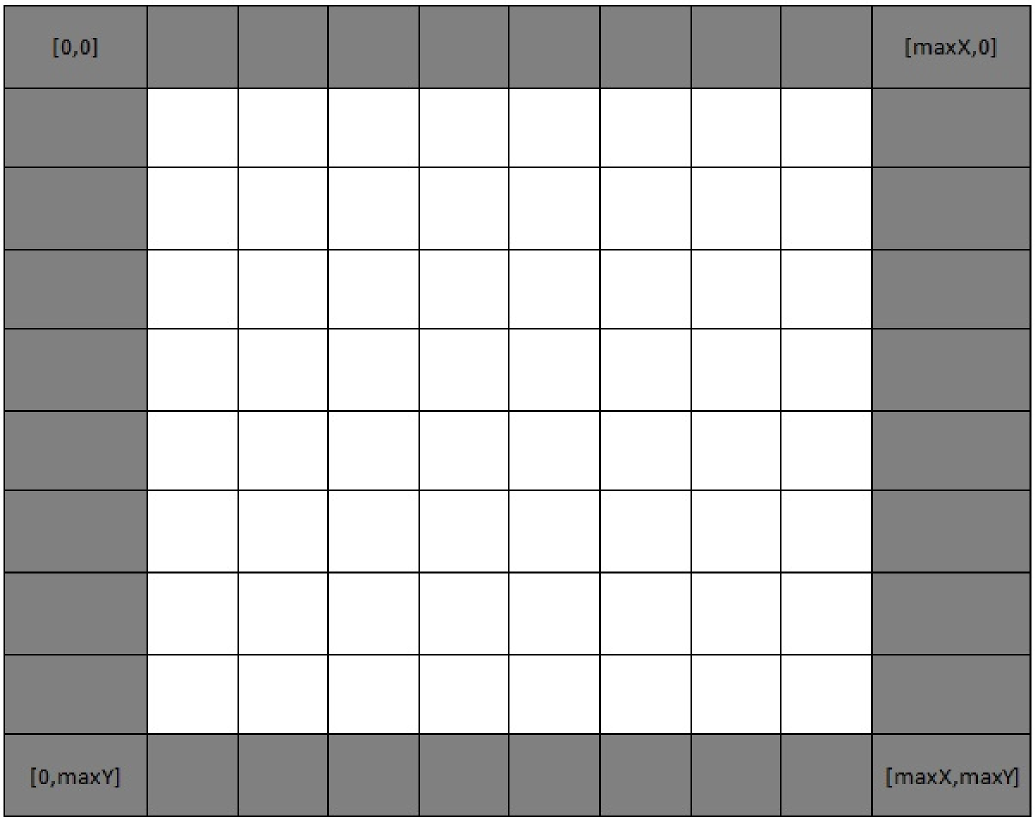


Figure 3.1: The border coordinates

The state of the border is defined as:

Border

$border : \mathbb{P}\, Pair$

The set border is initialized by uniting 4 sets that hold the border coordinates. Its initial state is defined as:

BorderInit

$\Delta Border$

$border' = \{\{\forall\, x_p : 0 \ldots maxX \bullet (x_p, 0)\} \cup$

$\{\forall\, x_p : 0 \ldots maxX \bullet (x_p, maxY)\} \cup$

$\{\forall\, y_p : 0 \ldots maxY \bullet (0, y_p)\} \cup$

$\{\forall\, y_p : 0 \ldots maxY \bullet (maxX, y_p)\}$

The following schema declares two variables x_t and y_t of type natural numbers that will hold the target coordinates.

Target

$x_t : \mathbb{N}$

$y_t : \mathbb{N}$

The target coordinates are initialized by an undefined value.

TargetInit

$\Delta Target$

$x_t' = undefined$

$y_t' = undefined$

The following schema declares two variables x_r and y_r of type natural numbers that will hold the robot coordinates.

Robot

$x_r : \mathbb{N}$

$y_r : \mathbb{N}$

The robot position is initialized by an undefined value.

RobotInit

$\Delta Robot$

$x_r' = undefined$

$y_r' = undefined$

A set of type Pair is defined in the following schema named “obstacleList” to hold the obstacles coordinates

$$\textit{Obstacle}$$

$$obstacleList : \mathbb{P}\, Pair$$

Initially the obstacleList is an empty set.

$$\textit{ObstacleInit}$$

$$\Delta Obstacle$$

$$obstacleList' = \emptyset$$

Two variables are defined to hold the best pair.

$$\textit{BestPair}$$

$$x_b : \mathbb{N}$$

$$y_b : \mathbb{N}$$

These variables will be initialized by an undefined value.

$$\textit{InitBestPair}$$

$$\Delta BestPair$$

$$x_b' = undefined$$

$$y_b' = undefined$$

The pair (x_c, y_c) is defined to hold the current position during the search process.

CurrentPair

$x_c : \mathbb{N}$

$y_c : \mathbb{N}$

The current position is initialized by an undefined value.

InitCurrentPair

$\Delta Current$

$x'_c = undefined$

$y'_c = undefined$

The following schema defines three sets that hold the neighbors of the current position, and defines two variables to hold the neighbor position.

Neighbor

$neighbors1 : \mathbb{P}\,Pair$

$neighbors2 : \mathbb{P}\,Pair$

$neighbors3 : \mathbb{P}\,Pair$

$x_n : \mathbb{N}$

$y_n : \mathbb{N}$

Initially, the three sets: neighbors1, neighbors2, and neighbors3 are empty and the two variables are undefined.

$$
\begin{array}{|l}
\hline \textit{InitNeighbor} \\
\Delta \textit{Neighbor} \\
\hline
\textit{neighbors}1' = \varnothing \\
\textit{neighbors}2' = \varnothing \\
\textit{neighbors}3' = \varnothing \\
x_n' = \textit{undefined} \\
y_n' = \textit{undefined} \\
\hline
\end{array}
$$

The pair (x_{cp}, y_{cp}) holds the current position of the robot.

$$
\begin{array}{|l}
\hline \textit{currentPosition} \\
x_{cp} : \mathbb{N} \\
y_{cp} : \mathbb{N} \\
\hline
\end{array}
$$

Initially, this pair is set to an undefined value.

InitCurrentPosition

$\Delta currentPosition$

$x'_{cp} = undefined$

$y'_{cp} = undefined$

The pair (x_{np}, y_{np}) holds the next position of the robot.

nextPosition

$x_{np} : \mathbb{N}$

$y_{np} : \mathbb{N}$

This pair is initialized by an undefined value.

InitNextPosition

$\Delta nextPosition$

$x'_{np} = undefined$

$y'_{np} = undefined$

The following schema defines the set openF as a bijective function that relates each pair to its F value.

$OpenF$

$openF : Pair \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathbb{N}$

We initialize the OpenF set by adding the robot location to the domain, and adding the F value of the robot position to the range of the openF function.

$InitOpenF$

$\Delta OpenF$

$\Xi Robot$

$\Delta CalculateFValue$

$openF' = \{(x_r \mapsto y_r) \mapsto f\}$

The following schema defines the set openG as a bijective function that relates each pair to its G value.

$OpenG$

$openG : Pair \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathbb{N}$

Then we initialize the openG set by adding the robot location to the domain, and adding the G value of the robot position to the range of the openG function.

$InitOpenG$

$\Delta OpenG$

$\Xi Robot$

$\Delta CalculateGCurrent$

$openG' = \{(x_r \mapsto y_r) \mapsto g_c\}$

The following schema defines the set ParentChild as a bijective function that relates each parent (of type Pair) to its child (of type Pair).

$ParentChild$

$parentChild : Pair \rightarrowtail\!\!\!\!\twoheadrightarrow Pair$

Initially the parentchild set is empty.

$InitParentChild$

$\Delta ParentChild$

$parentChild' = \varnothing$

The closedList is a set of type Pair.

$$\begin{array}{|l}\hline \textit{Closed} \\ \\ closedList : \mathbb{P}\, Pair \\ \hline \\ \hline\end{array}$$

Initially, the closedList is an empty set.

$$\begin{array}{|l}\hline \textit{InitClosed} \\ \\ \Delta Closed \\ \hline closedList' = \varnothing \\ \hline\end{array}$$

The final path is defined a sequence of points of type Pair, and the point (x_{path}, y_{path}) is one of the path points.

$$\begin{array}{|l}\hline \textit{Path} \\ \\ path : \mathrm{seq}\, Pair \\ x_{path} : \mathbb{N} \\ y_{path} : \mathbb{N} \\ \hline \\ \hline\end{array}$$

The path will be initialized by adding the target position.

$InitPath$

$\Delta Path$

$\Xi\, Target$

$path' = \langle (xt, yt) \rangle$

$x'_{path} = x_t$

$y'_{path} = y_t$

3.3 Operations

The following schema shows the normal flow of assigning a robot position, in which to set the robot position, the system should check whether the input value (x?,y?) belongs to the map set, and the robot pair (x_r, y_r) should not be defined before because it is not allowed to add more than one location for the robot,and the system allows to replace the old position of the target by a new one.

$SetRobotPositionOK$

ΞMap

$\Delta Robot$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$((x?, y?) \in map$

$(x_r, y_r) = (undefined, undefined)$

$(x_r', y_r') = (x?, y?))$

$\vee$

$((x?, y?) \in map$

$(x_r, y_r) \neq (undefined, undefined)$

$(x_r', y_r') = (x?, y?))$

While the following schema shows the exceptional flow in which the operation of assign the robot position will be prohibited in 2 cases

1. If the input value belongs to obstacleList.

2. If the input value belongs to border.

SetRobotPositionNotOK

$\Xi Border$

$\Xi Obstacle$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$rep! : RESPONSE$

$((x?, y?) \in obstacleList$

$rep! = freePairFirst)$

$\vee$

$((x?, y?) \in border$

$rep! = ItsBorderPosition)$

Regarding the other operations that are related to setting the environment (such as: assigning the target and the obstacles positions .. etc) see Appendix B (Z document).

One of the major operations in our system is retrieving the best pair; the pair that is associated with the lowest F value, the following schema is describing this operation formally as follows:

$$\textit{EvaluateBestPair}$$

$\Delta BestPair$

$\Xi\, OpenF$

$(x_b', y_b') = dom(openF \rhd min(ran\ openF))$

After retrieving the best pair, The system should evaluate it as follows:

1. If the best pair is not equal to the target and the openF is empty so there is no path found(see SearchPairsNoPath schema).
2. While the openF set is not empty, if the best pair is not equal to the target, the system should keep searching, by assigning the current pair to the best pair, adding the best pair to the closedList, deleting the best pair from both openF and openG, and adding the neighbors of the current pair to the neighbors1 set (see SearchPairs schema).
3. If the best pair equals to the target, the system should construct a path by following the parent of each pair (see ConstructPath schema).

$SearchPairsNoPath$

$\Xi BestPair$

$\Xi Target$

$\Xi OpenF$

$rep! : RESPONSE$

$(x_b, y_b) \neq (x_t, y_t)$

$openF = \varnothing$

$rep! = NoPathFound$

$SearchPairs$

$\Xi BestPair$

$\Xi Target$

$\Delta CurrentPair$

$\Delta Neighbor$

$\Delta Closed$

$\Delta OpenF$

$\Delta OpenG$

$(x_b, y_b) \neq (x_t, y_t)$

$openF \neq \varnothing$

$(x_c, y_c) = (x_b, y_b)$

$closedList' = closedList \cup \{(x_b, y_b)\}$

$openF' = \{(x_b, y_b)\} \lhd openF$

$openG' = \{(x_b, y_b)\} \lhd openG$

$neighbors1' = \{\{(x_c, y_c + 1)\} \cup \{(x_c, y_c - 1)\} \cup \{(x_c + 1, y_c)\} \cup \{(x_c - 1, y_c)\}$

$\cup \{(x_c - 1, y_c + 1)\} \cup \{(x_c - 1, y_c - 1)\} \cup \{(x_c + 1, y_c - 1)\} \cup \{(x_c + 1, y_c + 1)\}\}$

$ConstructPath$

$\Delta Path$

$\Xi Robot$

$\Xi BestPair$

$\Xi Target$

$rep! : RESPONSE$

$(x_b, y_b) = (x_t, y_t)$

$rep! = APathIsFound$

$(x_{path}, y_{path}) \neq (undefined, undefined)$

$(x_{path}, y_{path}) \neq (x_r, y_r)$

$(x'_{path}, y'_{path}) = dom(parentChild \rhd (x_{path}, y_{path}))$

$path' = \langle (x_{path}, y_{path}) \rangle \frown path$

The system should discard some of the unwanted (unsafe) diagonal neighbors in the neighbors1 set by adding these neighbors to a temporary placeholder(the neighbor2 set). These neighbors will be considered unwanted if they match one the eight cases that were described before in the previous chapter:

The following schema shows how we formally specifies the first case of discarding an unwanted (unsafe) diagonal neighbor (see Figure 3.2). The rest of the eight cases were fully described in Appendix B (Z document).

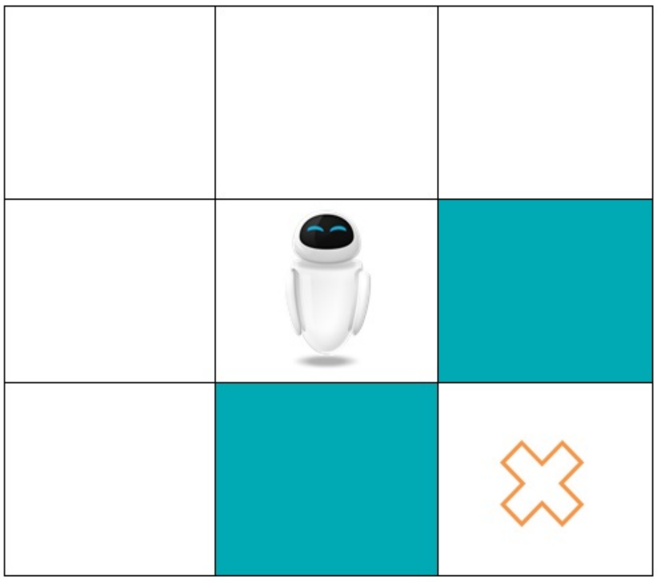


Figure 3.2: Case1: The down-right neighbor will be excluded

$RefineDiagonalNeighborsCase1$

$\Delta Neighbor$

$\Xi CurrentPair$

$\Xi Obstacle$

$(x_c, y_c + 1) \in neighbors1 \wedge ((x_c, y_c + 1) \in obstacleList$

$(x_c + 1, y_c) \in neighbors1 \wedge (x_c + 1, y_c) \in obstacleList$

$neighbors2' = neighbors2 \cup \{(x_c + 1, y_c + 1)\}$

After adding the unwanted neighbors to the neighbors2 set, the system should delete the elements of neighbors2 set from neighbors1.The following schema presents the operation of deleting the (unsafe) unwanted neighbors.

$DeleteDiagonalNeighbors$

$\Delta Neighbor$

$neighbors1' = neighbors1 \setminus neighbors2$

After deleting the unsafe diagonal neighbors, the system should check if there is any neighbor that belongs to the obstaclList set or to the border set, and then add it temporary to the neighbors3 set.The following schema shows how we formally specifies this operation.

$RefineNeighborsFromObstaclesAndBorder$

$\Xi Obstacle$

$\Xi Border$

$\Delta Neighbor$

$((x_n, y_n) \in neighbors1$

$(x_n, y_n) \in obstacleList$

$neighbors3' = neighbors3 \cup (x_n, y_n))$

$\vee$

$((x_n, y_n) \in neighbors1$

$(x_n, y_n) \in border$

$neighbors3' = neighbors3 \cup (x_n, y_n))$

After adding the neighbors that may belong to the obstacleList or to the border temporary to the neighbors3 set, The system should delete these neighbors from neighbors1 set, to make sure that the neighbors that belong to neighbors1 are all safe and free and ready to be evaluated.

```
DeleteNeighborsFromObstaclesAndBorder
  $\Delta Neighbor$
  ----------
  $neighbors1' = neighbors1 \setminus neighbors3$
```

After finding the safe path that the robot can follow, the system should take the head of the path sequence and store it as a current position, then remove it from the sequence then make the new head of the path sequence as the next position.

```
FollowPath
  $\Delta Path$
  $\Delta currentPosition$
  $\Delta nextPosition$
  ----------
  $path \neq \langle\rangle$
  $(x'_{cp}, y'_{cp}) = head\, path$
  $path' = tail\, path$
  $(x'_{np}, y'_{np}) = head\, path$
```

Now, the system should evaluate the next position to determine the upcoming robot movement. The following schema shows if the next position is equivalent to $(x_{cp}, y_{cp} - 1)$ then the robot should move " up" . After the robot movement the current robot position should be replaced by the next position to prepare the robot for the next movement.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{MoveUp}} \\
\Delta CurrentPosition \\
\Xi NextPosition \\
rm : ROBOTMOVE \\
\hline
(x_{np}, y_{np}) = (x_{cp}, y_{cp} - 1) \\
rm' = up \\
(x'_{cp}, y'_{cp}) = (x_{np}, y_{np}) \\
\hline
\end{array}
$$

The rest of the possible robot movements schemas (down, right, left .. etc) and the rest of the system operations schemas are fully described in Appendix B (Z document).

3.4 Proof of some properties

In this section, we try to prove some of the system properties. The robot safety is one of the essential properties in our system. In this section we try to focus on this property and how we formally assert the robot safety in our formal specification.

- Property 1: the robot safety from the obstacles

Based on our previous formal model, the robot will never collide with any obstacle in the environment. So for any robot position (x_r, y_r), until the robot reaches the target, should never be equivalent to any obstacle position.

$$(x_r, y_r) \notin obstacleList$$

- Proof:

  1. If we assume that (xr_i, yr_i) is the initial position of the robot. We stated formally by (SetRobotPositionNotOk Schema) that the initial robot position will never belong to the obstacleList set, and by this we proved that the initial position of the robot will never be an obstacle position:

  $$(xr_i, yr_i) \notin obstacleList$$

  2. The next position of the robot, until the robot reaches the target, will never be an obstacle position. We stated this formally by (DeleteNeighborsFromObstaclesAndBorder schema), in which we discarded any neighbor that belongs to the obstacleList set from the evaluation process, so the final path, that the robot will follow, will never include any obstacle position so:

  $$(xr_{i+1}, yr_{i+1}) \notin obstacleList$$

- Property 2: the robot safety from the border

Based on our previous formal model, the robot will never collide with the border. So for any robot position (x_r, y_r), until the robot reaches the target, should never be equivalent to any of border points.

$$(x_r, y_r) \notin border$$

- Proof:

  1. If we assume that (xr_i, yr_i) is the initial position of the robot. We stated formally by (SetRobotPositionNotOk Schema) that the initial robot position will never belong to the border set, and by this we proved that the initial position of the robot will never be a border point:

  $$(xr_i, yr_i) \notin border$$

  2. The next position of the robot, until the robot reaches the target, will never be a border point. We stated this formally by (DeleteNeighborsFromObstaclesAndBorder schema), in which we discarded any neighbor that belongs to the border set from the evaluation process, so the final path, that the robot will follow, will never include any border point so:

  $$(xr_{i+1}, yr_{i+1}) \notin border$$

Chapter 4: Formal refinement

Developing the abstract model (Z specification) is an important step towards correctness, as it provides a formal expression of the overall behavior of the system. However, to ensure the equivalence between this model and its implementation, a formal refinement process is required in order to transform the abstract model into its equivalent implementable code. Consequently, it will be possible to test and evaluate the behavior of the formal model that we built.

The basic idea underlying refinement is the transformation of an abstract model into a concrete one. Formal refinement in Z as defined in [26, 27] consists of data refinement and operation refinement. In both cases, the refinement process consists of reducing non-determinism and selecting data structures.

The formal model is described as a highly abstracted and uncertain model, through the refinement process we try to reduce non-determinism in order to reveal the actual behavior of the system. One of the data refinement approaches that we follow is called forwards simulation in which we convert the abstract data types into a concrete implementable data structures by identifying a relation to allow this transformation.

This chapter will be organized as follows: section 4.1 and 4.2 discusses the data refinement and the forwards simulation accordingly, our discussion in these two sections follow the reference [26]. The data refinement of the formal specification presents in section 4.3, and the operations refinement of the formal specification presents in section 4.4, while the last section 4.5 provides a summery about the chapter.

4.1 Data refinement

The idea of the data refinement is to analyze the data types that construct the software system (in the abstract model), in order to transform the abstract data types into implementable data structures. A data type consists of a set of values (states) and a series of indexed operations. If we assume that X is a data type, then any use of X in a global state G should start by an initialization step and end by a finalization step and there are series of indexed operations in between. Based on that we can present the X as: $(X, x_i, x_f, i : I \bullet xo_i)$ where:

- $x_i \in G \leftrightarrow X$ (initialization)
- $x_f \in X \leftrightarrow G$ (finalization)
- $\{i : I \bullet xo_i\}$ (a series of indexed operations)

When we look at the structure of the computer program, we find that it consists of a set of operations upon a data type, thus any program P that uses the data type X can be presented as:

$$P(x) = x_i \fatsemi xo_a \fatsemi xo_b \fatsemi xo_c \fatsemi ... \fatsemi x_f$$

For our purposes, one of the proper definitions of refining the abstract data types can be described as: if the data type A and C have the same set of indexed operations, A is refined by C if and only if:

- $P(\dot{C}) \subseteq P(\dot{A})$ where:
- $P(\dot{C}) = \dot{c}i \fatsemi \dot{c}o_{S1} \fatsemi \dot{c}o_{S2} \fatsemi ... \fatsemi \dot{c}o_{Sn} \fatsemi \dot{c}_f$

- $P(\dot{A}) = \dot{a}i \mathbin{⨾} \dot{a}o_{S1} \mathbin{⨾} \dot{a}o_{S2} \mathbin{⨾} ... \mathbin{⨾} \dot{a}o_{Sn} \mathbin{⨾} \dot{a}_f$

The dotted symbols signifies the totalization of the relation, that is, whenever a relation is partial is extended to a total relation.

4.2 Forwards simulation

If the data types A and C have the same set of indexed operations, and r is a relation that relates the set A to the set C ($A \leftrightarrow C$) then r is a forwards simulation if it obeys the following conditions:

- $\dot{c}_i \subseteq \dot{a}_i \mathbin{⨾} \mathring{\mathrm{r}}$
- $\mathring{\mathrm{r}} \mathbin{⨾} \dot{c}f \subseteq \dot{a}f$
- $\mathring{\mathrm{r}} \mathbin{⨾} \dot{c}o_i \subseteq \dot{a}o_i \mathbin{⨾} \mathring{\mathrm{r}}$

The first condition means that the effect of the c_i is equivalent to a_i followed by r; the second one means that the effect of r followed by cf is equivalent to af. The last one shows that moving from r then to co_i is matched by moving from ao_i then to r (see Figure 4.1 in reference [26]).

Based on that, to transform any abstract data type, we should identify a relation (r) that maps each element from the (abstract) set to its equivalent element in the (concrete) set.

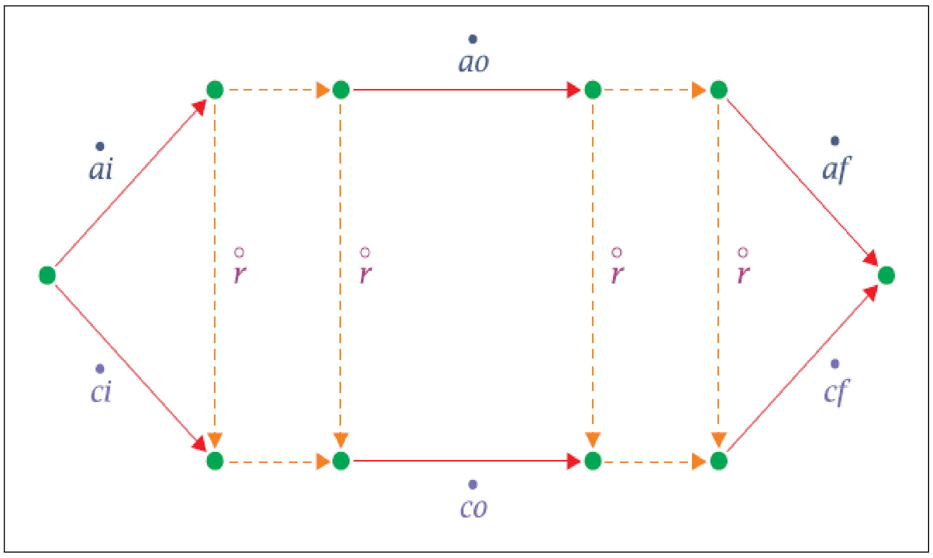


Figure 4.1: Forwards simulation

4.3 Data refinement of the formal specification

In this section we analyze each data type that was previously mentioned in the abstract model, and then we try to transform it to a more concrete one.

The major data type in our system is the Pair set ($Pair = \mathbb{N} \times \mathbb{N}$) which relates each X value of type natural numbers to a Y value of type natural numbers.

The set of natural numbers $\mathbb{N}$ can be transformed into the set of integers (Int), while (r) is the identical relation that maps each element from the set of natural numbers(abstract) to its equivalent number in the(concrete) set of integers.

The Pair data type can be implemented as a class called Pair that has two attributes of type integers which hold the X value and the Y value as follows:

```
class Pair {

int x;

int y;
```

}

The map set was defined previously (in the Z document) as the power set of the set Pair as follows:

$$\begin{array}{|l}
\hline
\textit{AMap} \\
\textit{map} : \mathbb{P}\,\textit{Pair} \\
\hline
\\
\hline
\end{array}$$

And it was initialized by adding specific elements; their X values will be between 0..maxX and their Y values will be between 0..maxY.

$$\begin{array}{|l}
\hline
\textit{AMapInit} \\
\Delta \textit{AMap} \\
\hline
\textit{map}' = \{\forall\, x_m : 0 \ldots \textit{maxX};\ \forall\, y_m : 0 \ldots \textit{maxY} \bullet (x_m, y_m)\} \\
\hline
\end{array}$$

A more concrete specification can transform the map set into an injective sequence iseq (duplicate-free sequence) to make sure that there are no duplicate elements belong to the map set. The following diagram (Figure 4.2) shows how the relation (r) relates each element in the set (abstract)to the same element in the range of the iseq (concrete) and it shows that both of the sets are equivalent but the iseq gives an index for each element which makes the set an ordered set with no duplicate.

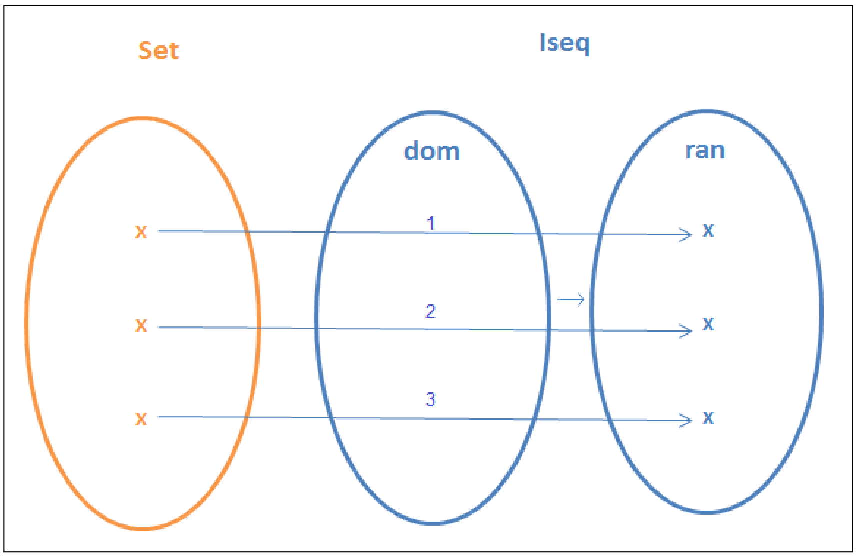


Figure 4.2: Example of data refinement

CMap

$map : \text{iseq}\, Pair$

The map iseq will be initialized by the same set of elements.

CMapInit

$\Delta CMap$

$map' = < \forall x_m : 0 \ldots maxX; \forall y_m : 0 \ldots maxY \bullet (x_m, y_m) >$

Regarding the previous refinement, the first specification is more abstract while the second specification adds more restrictions that prevent the repetition of the elements in the map set and give a particular arrangement of the elements. Based on that the map set can be implemented as an array of elements of type Pair as follows:

```
Pair [maxX][maxY] map;
```

The border set was defined previously as the power set of the set Pair as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, ABorder\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
border : \mathbb{P}\, Pair \\
\hline
\\
\hline
\end{array}$$

And it was initialized by adding specific elements

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, ABorderInit\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta ABorder \\
\hline
border' = \{\{\forall\, x_p : 0 \ldots maxX \bullet (x_p, 0)\} \cup \\
\{\forall\, x_p : 0 \ldots maxX \bullet (x_p, maxY)\} \cup \\
\{\forall\, y_p : 0 \ldots maxY \bullet (0, y_p)\} \cup \\
\{\forall\, y_p : 0 \ldots maxY \bullet (maxX, y_p)\} \\
\hline
\end{array}$$

The border set can be transformed into an iseq to prevent duplicate elements

$CBorder$

$border : \mathrm{iseq}\, Pair$

And the border will be initialized as follows:

$CBorderInit$

$\Delta CBorder$

$border' =< \{\forall\, x_p : 0 \ldots maxX \bullet (x_p, 0) > ^\frown$

$< \forall\, x_p : 0 \ldots maxX \bullet (x_p, maxY) > ^\frown$

$< \forall\, y_p : 0 \ldots maxY \bullet (0, y_p) > ^\frown$

$< \forall\, y_p : 0 \ldots maxY \bullet (maxX, y_p) >$

Based on the previous refinement, the border set can be implemented as an ArrayList to store the border coordinates as follows:

```
ArrayList <Pair> border;
```

The final path was defined in the abstract model as a sequence of points of type Pair, and the point (x_{path}, y_{path}) was declared as one of the path points.

$APath$

$path : \mathrm{seq}\, Pair$

$x_{path} : \mathbb{N}$

$y_{path} : \mathbb{N}$

The path was initialized by adding the target position, while the path point was initialized by adding the target coordinates.

$AInitPath$

$\Delta APath$

$\Xi\, Target$

$path' = \langle (xt, yt) \rangle$

$x'_{path} = x_t$

$y'_{path} = y_t$

We can change the seq into iseq to make sure that the whole path elements are all unique, because the final path points should not contain duplicates.

$CPath$

$path : \text{iseq}\, Pair$

$x_{path} : \mathbb{N}$

$y_{path} : \mathbb{N}$

And will be initialized as follows:

$CInitPath$

$\Delta CPath$

$\Xi\, Target$

$path' = \langle (xt, yt) \rangle$

$x'_{path} = x_t$

$y'_{path} = y_t$

We implement the path sequence as a stack because the order of the path elements is fundamental while the path point can be implemented as two integers:

```
Stack <Pair> path;
int xpath;
int ypath;
```

The other sets that were declared as freetype in Z such as : (RESPONSE, and ROBOTMOVE) can be implemented as (ArrayLists) of type String, because the

RESPONSE set contains the several system responses (textual statements), and the ROBOTMOVE is a set that contains the several robot moves (textual statements) to reach the target. note that the relation r is the identical relation that maps each element in the (abstract) set to its equivalent element in the (concrete) data structure.

```
ArrayList <String> RESPONSE;
ArrayList <String> ROBOTMOVE;
```

The following table shows the result of the data refinement process, in which how each abstract data type was transformed to its equivalent implementable data structure (see Table 4.1).

Abstract data type	**Implementable data structure**

$Pair = \mathbb{N} \times \mathbb{N}$	class Pair {
	int x;
	int y;
	}

maxX = const1
maxY = const2
	int maxX
	int maxY

$undefined == \perp$	int undefined
$RESPONSE ::= FreeThePairFirst \mid$ $NoPathFound \mid youCantFreeBorder \mid$ $ItsBorderPosition$ $\mid APathIsFound$	$ArrayList < String > RESPONSE$
$ROBOTMOVE ::= up \mid down \mid right \mid left$ $\mid upRight \mid upLeft \mid downRight \mid downLeft$	$ArrayList < String > ROBOTMOVE$
$map : \mathbb{P}\, Pair$	Pair[maxX][maxY] map
$border : \mathbb{P}\, Pair$	$ArrayList < Pair > border$
$x_t : \mathbb{N}$ $y_t : \mathbb{N}$	$int\ x_t$ $int\ y_t$
$x_r : \mathbb{N}$ $y_r : \mathbb{N}$	$int\ x_r$ $int\ y_r$
$obstacleList : \mathbb{P}\, Pair$	$ArrayList < Pair > obstacleList$
$x_c : \mathbb{N}$ $y_c : \mathbb{N}$	$int\ x_c$ $int\ y_c$
$neighbors1 : \mathbb{P}\, Pair$ $neighbors2 : \mathbb{P}\, Pair$ $neighbors3 : \mathbb{P}\, Pair$	$ArrayList < Pair > neighbors1$ $ArrayList < Pair > neighbors2$ $ArrayList < Pair > neighbors3$
$x_n : \mathbb{N}$ $y_n : \mathbb{N}$	$int\ x_n$ $int\ y_n$
$x_{cp} : \mathbb{N}$ $y_{cp} : \mathbb{N}$	$int\ x_{cp}$ $int\ y_{cp}$
$x_b : \mathbb{N}$ $y_b : \mathbb{N}$	$int\ x_b$ $int\ y_b$
$x_{np} : \mathbb{N}$ $y_{np} : \mathbb{N}$	$int\ x_{np}$ $int\ y_{np}$
$openF : Pair \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}$	$Hashtable < Pair, Integer > openF$
$openG : Pair \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}$	$Hashtable < Pair, Integer > openG$
$parentChild : Pair \rightarrowtail\!\!\!\!\rightarrow Pair$	$Hashtable < Pair, Pair > parentChild$
$closedList : \mathbb{P}\, Pair$	$ArrayList < Pair > closedList$
$path : \mathrm{seq}\, Pair$	$Stack < Pair > path$
$x_{path} : \mathbb{N}$ $y_{path} : \mathbb{N}$	$int\ x_{path}$ $int\ y_{path}$

Table 4.1: The result of the data refinement process

4.4 Operations refinement of the formal specification

After we succeeded in transforming the abstract data types into its equivalent implementation. In this section, we discuss how we add more details to the operational schemas until we get the final equivalent implantation that mirrors the formal description.

We start the implementation process by defining the major type in our specification which is the class Pair that has two attributes to hold the X and the Y value of each pair:

```
class Pair {
int x;
int y;
}
```

We define two supportive methods to retrieve the X and the Y value of any pair as follows:

```
 public int getX() { return x; }
 public int getY() { return y; }
```

Then we define all the needed (data structures) ArrayLists, arrays, hashtables, and variables that we discussed before in the previous section.

Afterwards, we transform each operational schema to its equivalent implementation; the first operational schema is to set the robot position, in which the system should check whether the input value (x?,y?) belongs to the map set, and the robot pair

(x_r, y_r) should not be defined before because it is not allowed to add more than one location for the robot, and it is allowed to replace the robot position by a new one. The following is the concrete version of the (SetRobotPositionOK) schema after applying the data refinement process.

$CSetRobotPositionOK$

$\Xi CMap$

$\Delta CRobot$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$((x?, y?) \in ran\ map$

$(x_r, y_r) = (undefined, undefined)$

$(x_r', y_r') = (x?, y?))$

$\vee$

$((x?, y?) \in ran\ map$

$(x_r, y_r) \neq (undefined, undefined)$

$(x_r', y_r') = (x?, y?))$

While the following is the concrete version of the (SetRobotPositionNotOK) schema after applying the data refinement process, that is showing the two cases in which this operation will be prohibited:

1. If the input value belongs to obstacleList.

2. If the input value belongs to border.

$CSetRobotPositionNotOK$

$\Xi CBorder$

$\Xi CObstacle$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$rep! : RESPONSE$

$((x?, y?) \in \mathrm{ran}\ obstacleList$

$rep! = freePairFirst)$

$\vee$

$((x?, y?) \in \mathrm{ran}\ border$

$rep! = ItsBorderPosition)$

We noticed that there is no need for extra details to be added to both of the previous schemas and they can both transformed directly to an equivalent implementation. Let us assume that the pair (x,y) presents the new robot position ;the equivalent implementation is going to be as follows :

```
int x ;
int y ;
for(int j=0 ; j<maxY ; j++) {
for(int i=0; i <maxX; i++) {
Pair mapPair = map[i][j];
```

```
    //If the new value belongs to the map
if (x == mapPair.getX() && y == mapPair.getY() ) {
          for(int i=0 ; i<obstacleList.size() ; i++) {
          Pair obstaclePair = obstacleList.get(i);
          xo = obstaclePair.getX();
          yo = obstaclePair.getY();
          //If the new value belongs to the obstacleList
          if(x == xo && y == yo ){
          System.out.println("Free The Pair First");
          }}
          for(int i=0 ; i<border.size() ; i++) {
          Pair borderPair = border.get(i);
          xbo = borderPair.getX();
          ybo = borderPair.getY();
          //If the new value belongs to the border
          if(x == xbo && y == ybo ){
          System.out.println("It's Border Position");
          }}
          if((x!=xo || y !=yo) && (x!=xbo || y!=ybo) ){
          if(x!=undefined || y!=undefined ){
           // Set new robot position
            xr=x;
            yr=y;
```

```
        }
        else{
      // Replace the old robot position by a new one
          xr=x;
          yr=y;
        }}}}}
```

To free the robot position, the system should check whether the input value equals to the robot position; if yes, then the robot position will be assigned to an undefined value, and this operation is described in the following schema:

CFreeRobotPosition

$\Delta CRobot$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$(x?, y?) = (x_r, y_r)$

$(x_r', y_r') = (\mathit{undefined}, \mathit{undefined})$

The equivalent implementation is going to be as follows :

```
int x ;
int y ;
if (x = = xr && y = = yr) {
        xr= undefined;
```

```
          yr= undefined;

  }
```

The following schema is showing the concrete version of one of the cases (Case 1) of discarding an unwanted diagonal neighbor

$CRefineDiagonalNeighborsCase1$

$\Delta CNeighbor$

$\Xi CurrentPair$

$\Xi CObstacle$

$(x_c, y_c + 1) \in ran\ neighbors1 \wedge (x_c, y_c + 1) \in ran\ obstacleList$

$(x_c + 1, y_c) \in ran\ neighbors1 \wedge (x_c + 1, y_c) \in ran\ obstacleList$

$neighbors2' = neighbors2 \frown \{(x_c + 1, y_c + 1)\}$

The previous schema needs more details to transform it to its equivalent implementation, we need to declare a counter to count the neighbors that belong to the obstacleList, if the counters counts two neighbors, then one of the diagonal neighbors will be added temporary to neighbors2 to be discarded later on. The equivalent implementation of the previous schema is going to be as follows:

```
public void refineDiagonalNeighborsCase1 (Pair
neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn = = xc && yn = = yc+1 &&
obstacleList.contains(neighbor)){
counter1++;
}
if (xn = = xc+1 && yn = = yc &&
obstacleList.contains(neighbor)){
counter1++;
}
if(counter1= =2)  {
Pair n = neighbors1.get(4);
neighbors2.add(n);
 }}
```

The following two schemas are showing the operation of calculating the G value of any neighbor pair, and calculating the G value of the current pair.

$CCalculateGCurrent$

$\Xi CRobot$

$\Xi CCurrent$

$g_c : \mathbb{N}$

$g'_c = \mid y_c - y_r \mid + \mid x_c - x_r \mid$

CalculateGNeighbor

$\Xi Robot$

$\Xi CNeighbor$

$g_n : \mathbb{N}$

$g'_n = \mid y_n - y_r \mid + \mid x_n - x_r \mid$

We noticed that both of the previous schemas are doing the exact role, so we chose to merge them in one method in the code; the equivalent implementation of the previous two schemas is going to be as follows:

```
public int calculateGValue(int x, int y) {
return (Math.abs(x - xr) + Math.abs(y - yr)); }
```

The rest of the operations refinement process and the final implementation of the modified version of the A* are listed in Appendix C (Equivalent implementation).

4.5 Summary

We started with a Z specification of the A* algorithm. In a stepwise fashion, we refined each Z schema into a concrete schema. The modularity of the specification and the refinement theory in Z allowed to decompose our refine steps and focus on one schema at a time. Subsequently, we transformed the concrete schemas into executable code, thus completing the refinement process. In the following chapter, we provide a complete description of the implementation process.

Chapter 5: Simulation

After getting the implementable code through the formal refinement process, we simulate the resulting code to evaluate and test the correctness of our approach. The simulation was built using the Java programming language. The GUI of the simulation was created by David Fontenot [36]. We did some modifications on the GUI then we implemented the modified version of the A* path planning algorithm.

This chapter will be organized as follows: section 5.1 shows some of execution scenarios, while the last section 5.2 provides a summery about the chapter.

The structure of the code is organized as follows:

1. AStar.java: this is the entry point of the program that is going to launch the GUI and add the listeners.
2. AStarAlgo.java: this class contains our own path planning algorithm.
3. Pair.java: this class defines the type (Pair) and its associated methods and attributes.
4. PairGraphPanel.java: this class is responsible to create the GUI that will appear to the user. When the user clicks (right-click) on a cell, the pop-up menu is made visible and the user can choose to set whether the robot or the target position.
5. PairPlacedListener.java: this class is responsible to listen to the mouse events on the PairGraphPanel, and then make the desired changes, it also controls the pop-up menu.

6. PathButtonState.java: this class controls the state of the button(which the user uses to run the path planning algorithm and to reset the GUI) . This is where the path planning algorithm is called from.
7. StartEndListener.java: this class controls the entry of the target and the robot location.

5.1 Execution Scenarios

In this section we develop some execution scenarios that show the consistency of our approach. These scenarios mimic the Z schemas and are viewed as testing the schemas. They are classified into three major categories:

- **Setting the environment**: These test cases are showing the correctness of setting the environment, such as: specifying the robot location, target location, free cells .. etc.
- **Path planning operations**: This category shows some test cases that are showing the correctness of our path planning algorithm.
- **The robot movements**: These test cases are showing how the robot will safely reach the target by making series of moves.

5.1.1 Setting the environment

- Set the robot position:

  To set the robot position, the user should right-click on the desired cell, which is going to display the pop-up menu that has two choices: assigning the robot position or the target position.The user should choose to set the robot position. (see Figure 5.1).

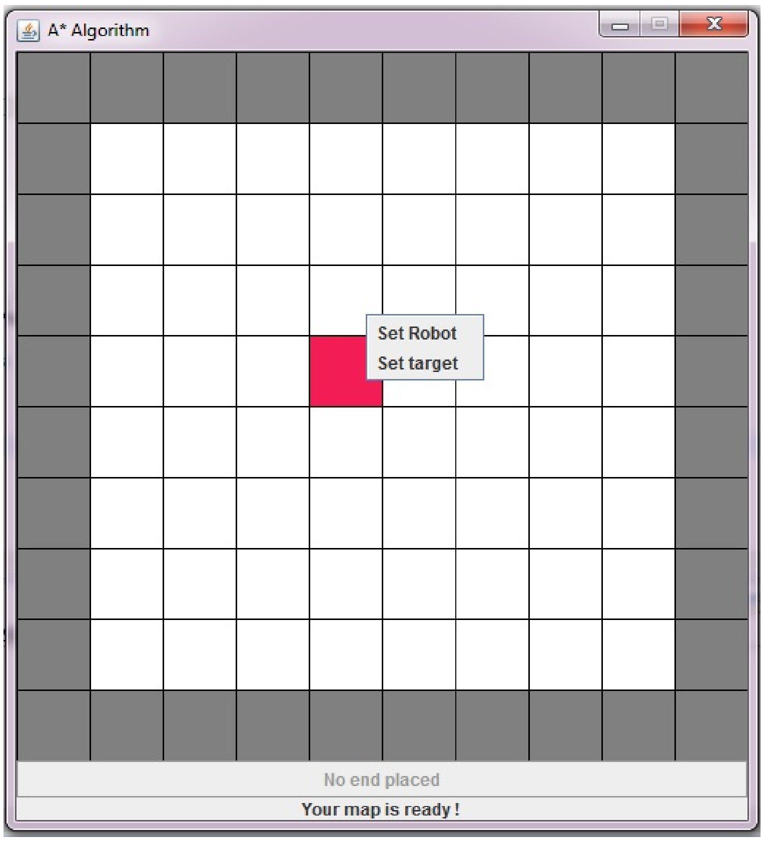
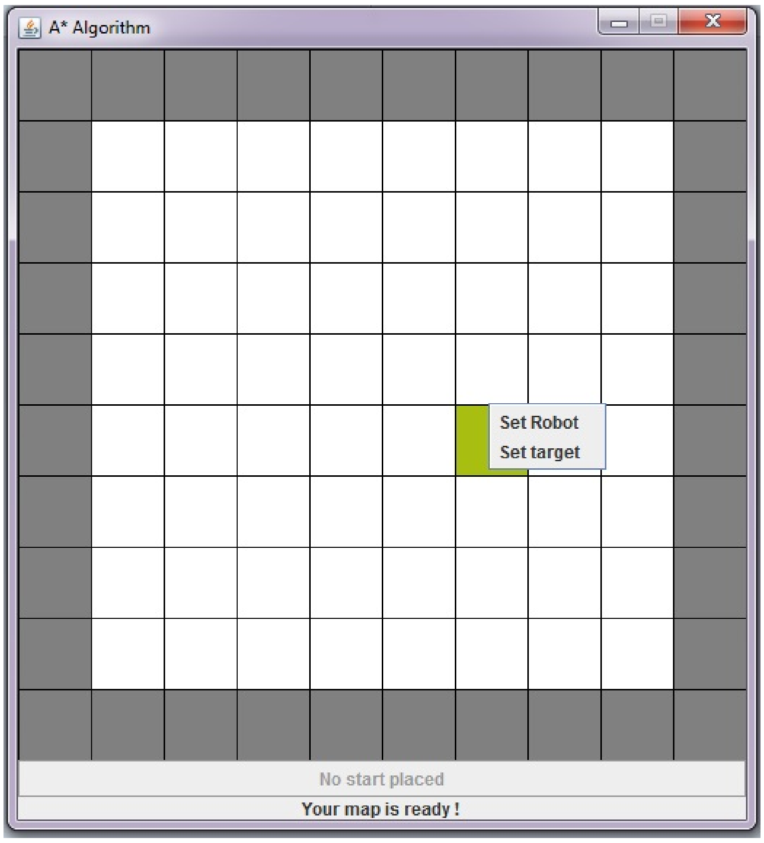


Figure 5.1: Setting the robot position

- Set the target position:

To set the target position, the user should right-click on the desired cell, the pop-up menu will be displayed, then the user should choose to set the target position. (see Figure 5.2).



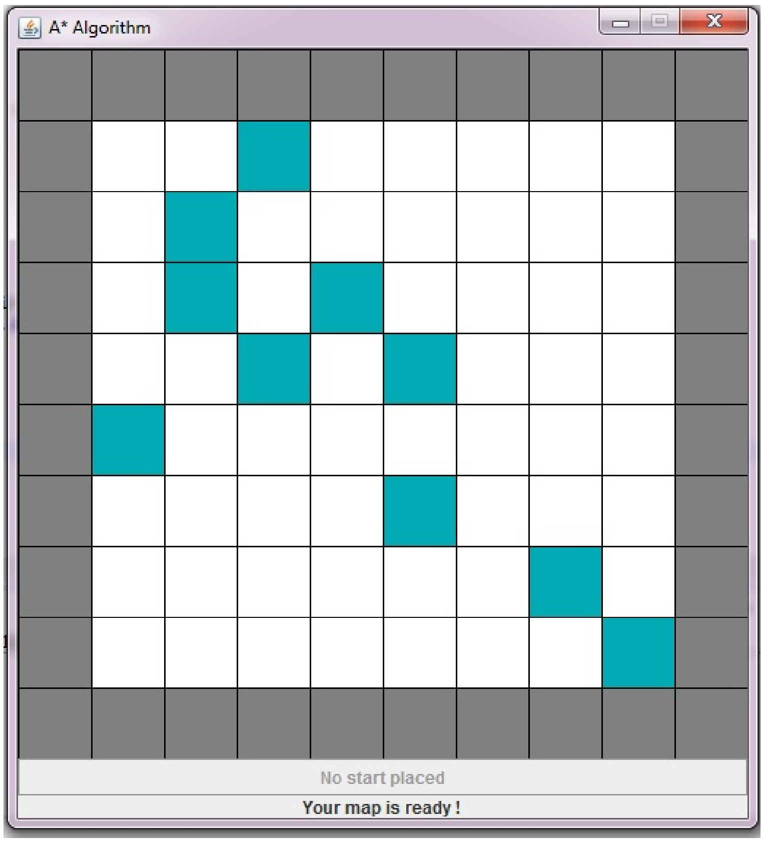
No start placed

Your map is ready !

Figure 5.2: Setting the target position

- Set obstacle position:

  To set an obstacle position, the user should left-click on the desired cell to assign it as an obstacle (see Figure 5.3).



No start placed

Your map is ready !

Figure 5.3: Setting the obstacle position

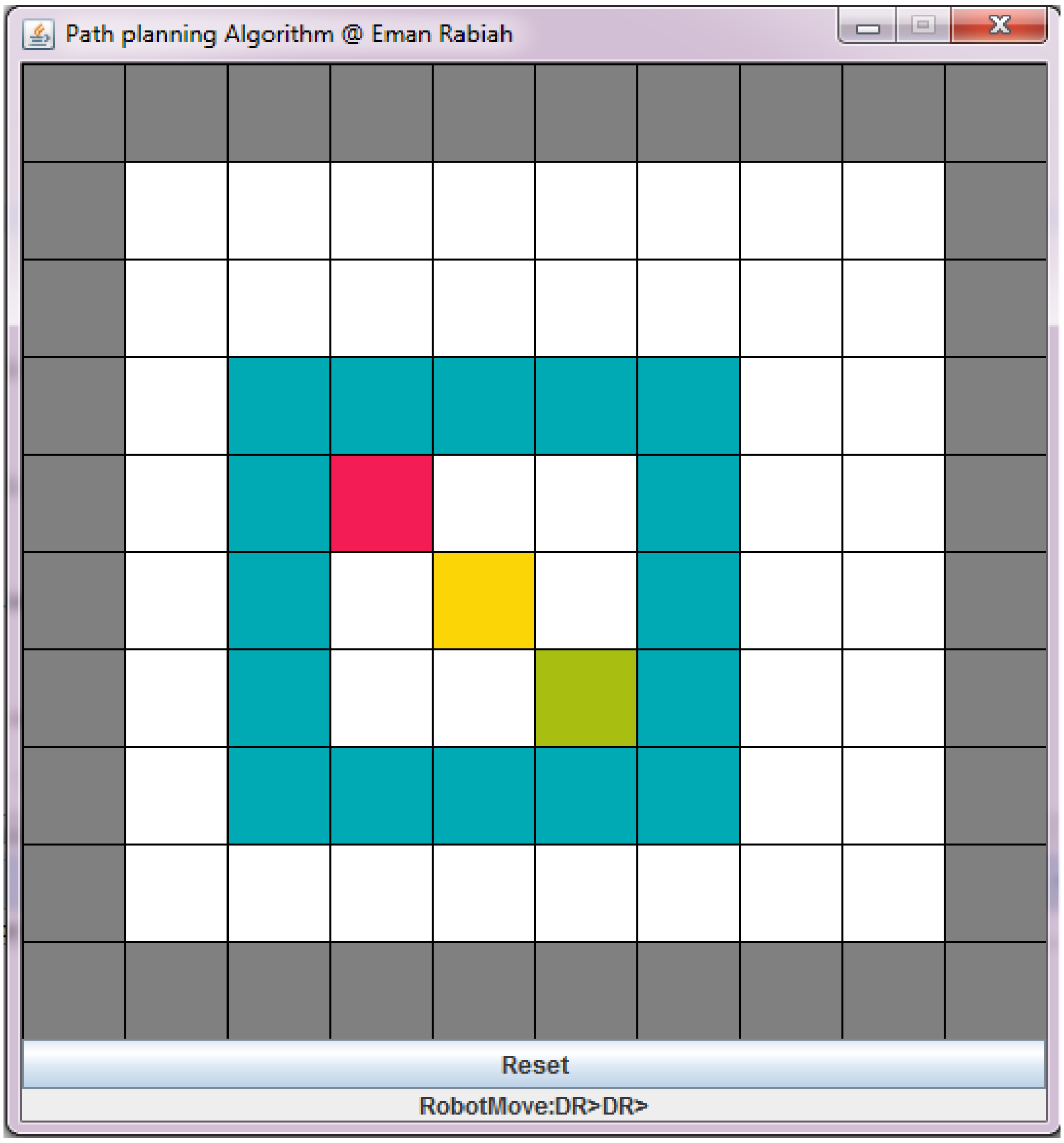
- Free a desired position:

  To free the robot, or the target or an obstacle position the user can left-click on the desired cell to make it free, while the system will not allowed to free any of the border cells.

5.1.2 Path planning operations

- Searching with a path is found:

  First of all, the user should assign the robot and the target position then press on the button to start searching to find the shortest path. The figure below (Figure 5.4) shows an example of finding the shortest path between the robot position and the target position.



RobotMove:DR>DR>

Figure 5.4: An example of finding the shortest path

- Searching with no path is found:

In this test case, we surround the robot with obstacles from all directions, then we start the searching process that resulted without finding any possible path between the robot and the target (see Figure 5.5)

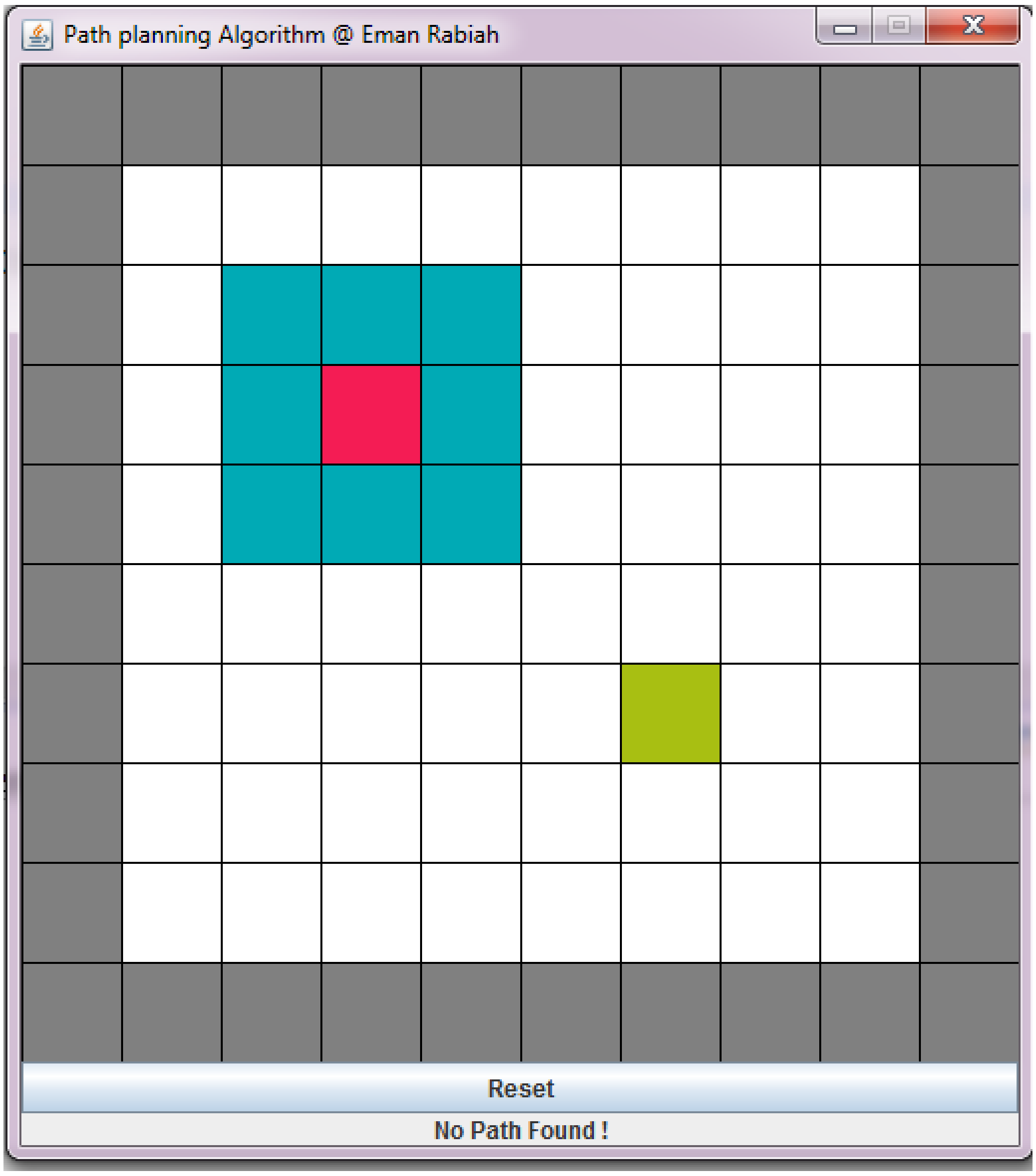


Figure 5.5: An example of not finding any possible path

- Refine diagonal neighbors (Case 1):

  In this test case, we filled all the cells that surround the robot position by obstacles except the (down-right) cell. The system responses by “No path found” because the robot can’t move to the target through this cell, which proves that case 1 of discarding the diagonal neighbor (down-right) is working correctly (see

Figure 5.6).

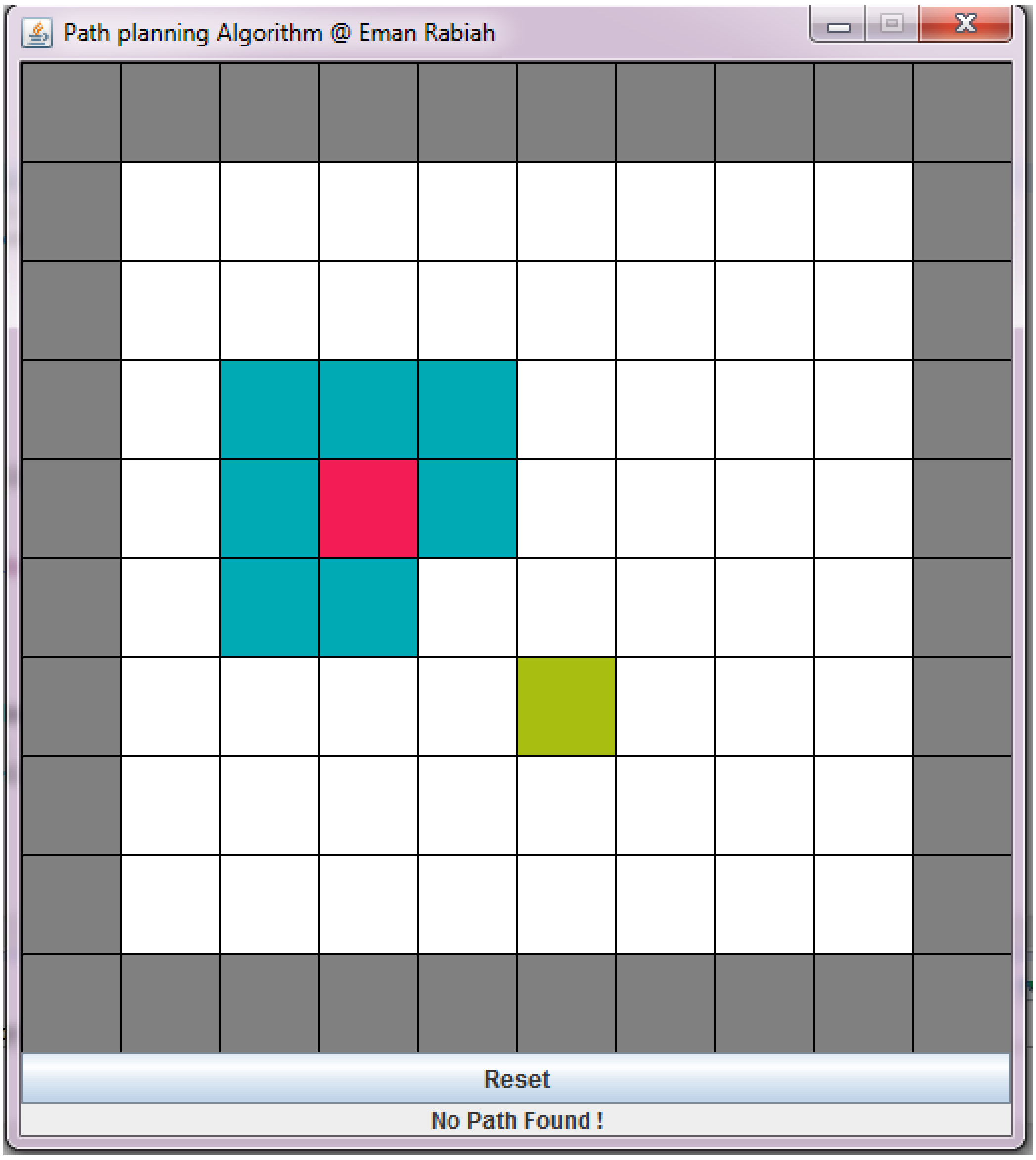


Figure 5.6: An example of case 1: discarding the down-right neighbor

- Refine diagonal neighbors (Case 2):

In this test case, we fill all the cells that surround the robot position by obstacles except the (down-left) cell. The system responses by “No path found” because the robot cannot move to the target through this cell, which proves that case 2 of discarding the diagonal neighbor (down-left) is working correctly (see Figure 5.7).

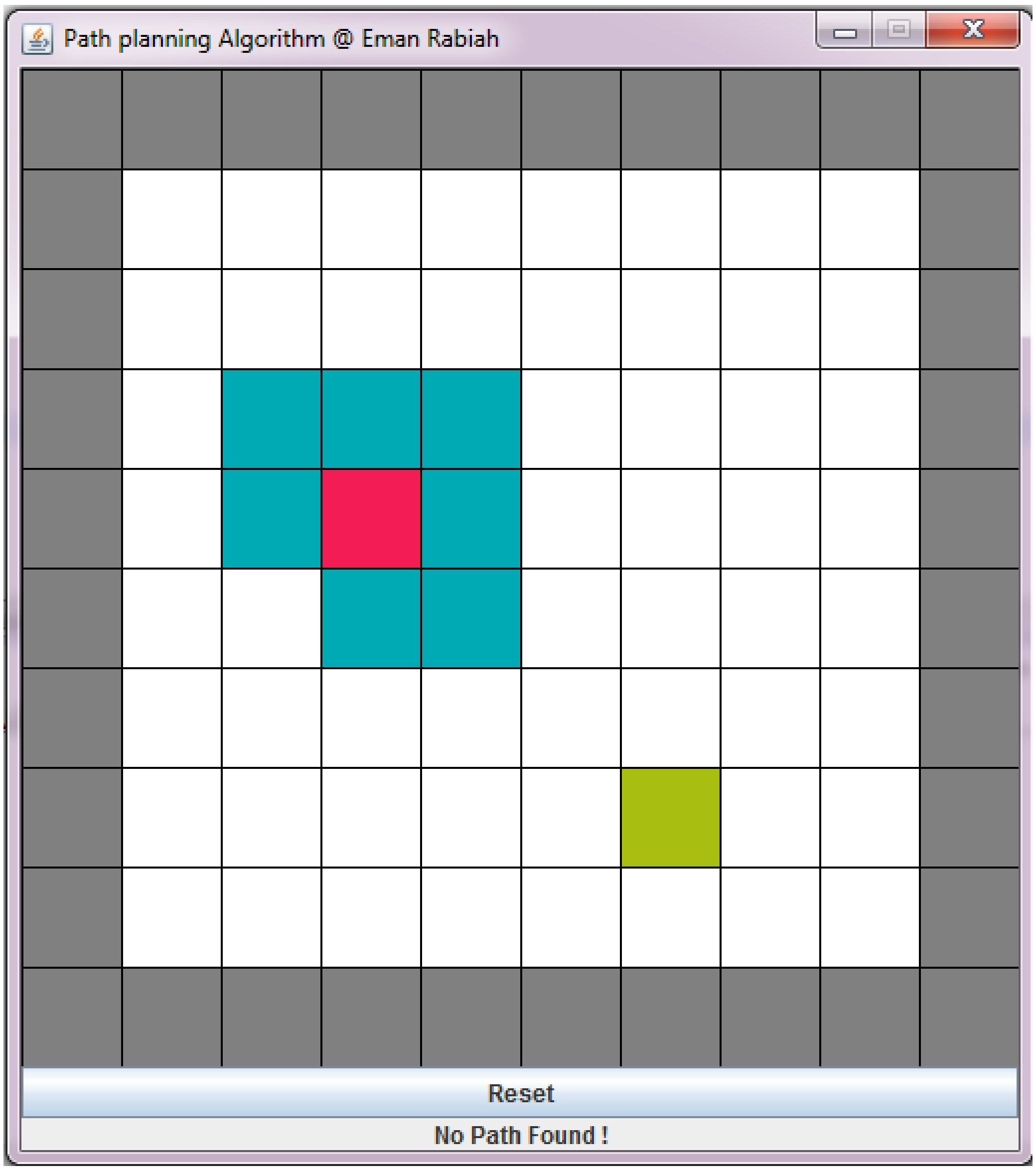


Figure 5.7: An example of case 2: discarding the down-left neighbor

- Refine diagonal neighbors (Case 3):

  In this test case, we fill all the cells that surround the robot position by obstacles except the (up-right) cell. The system responses by "No path found" because the robot can't move to the target through this cell, which proves that case 3 of discarding the diagonal neighbor (up-right) is working correctly (see Figure 5.8).

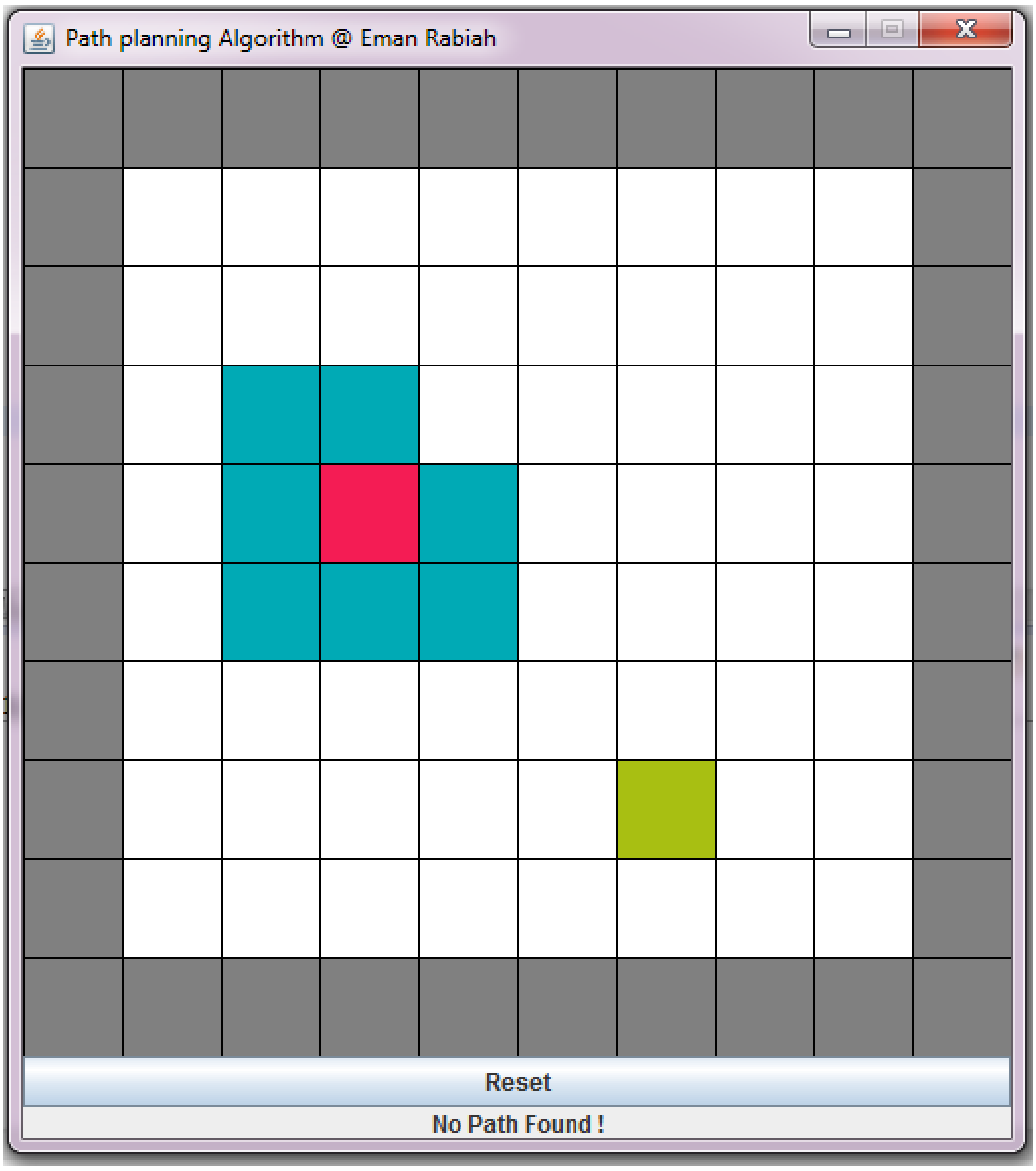


Figure 5.8: An example of case 3: discarding the up-right neighbor

- Refine diagonal neighbors (Case 4):

  In this test case, we fill all the cells that surround the robot position by obstacles except the (up-left) cell. The system responses by "No path found" because the robot cannot move to the target through this cell, which proves that case 4 of discarding the diagonal neighbor (up-left) is working correctly (see Figure 5.9).

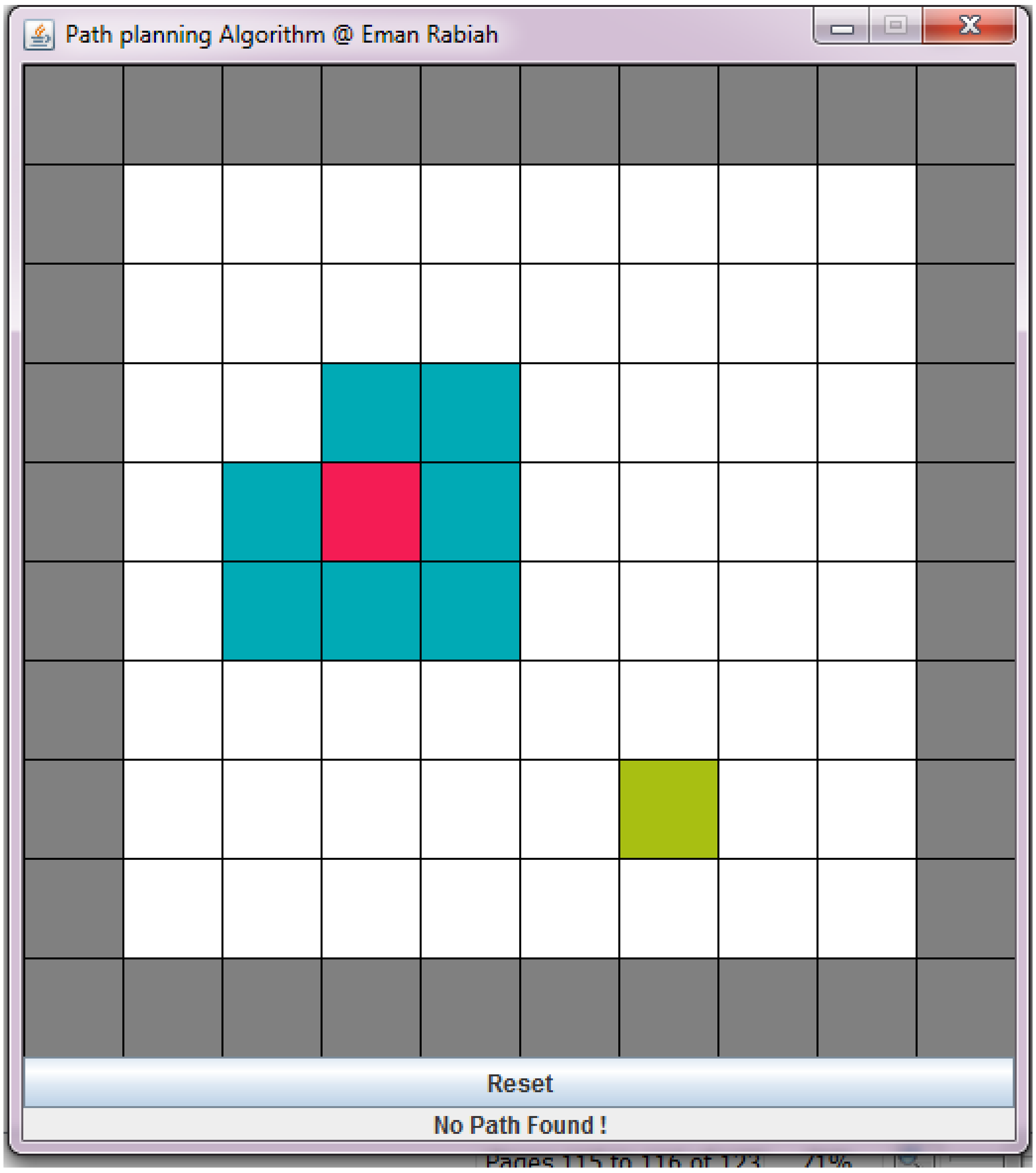


Figure 5.9: An example of case 4: discarding the up-left neighbor

- Refine diagonal neighbors (Case 5):

  In this test case, we fill all the cells that surround the robot position by obstacles except the (down-left) and the (down-right) cells. The system responses by "No path found" because the robot cannot move to the target through any of these cells, which proves that case 5 of discarding the diagonal neighbors (down-left and down-right) is working correctly (see Figure 5.10).

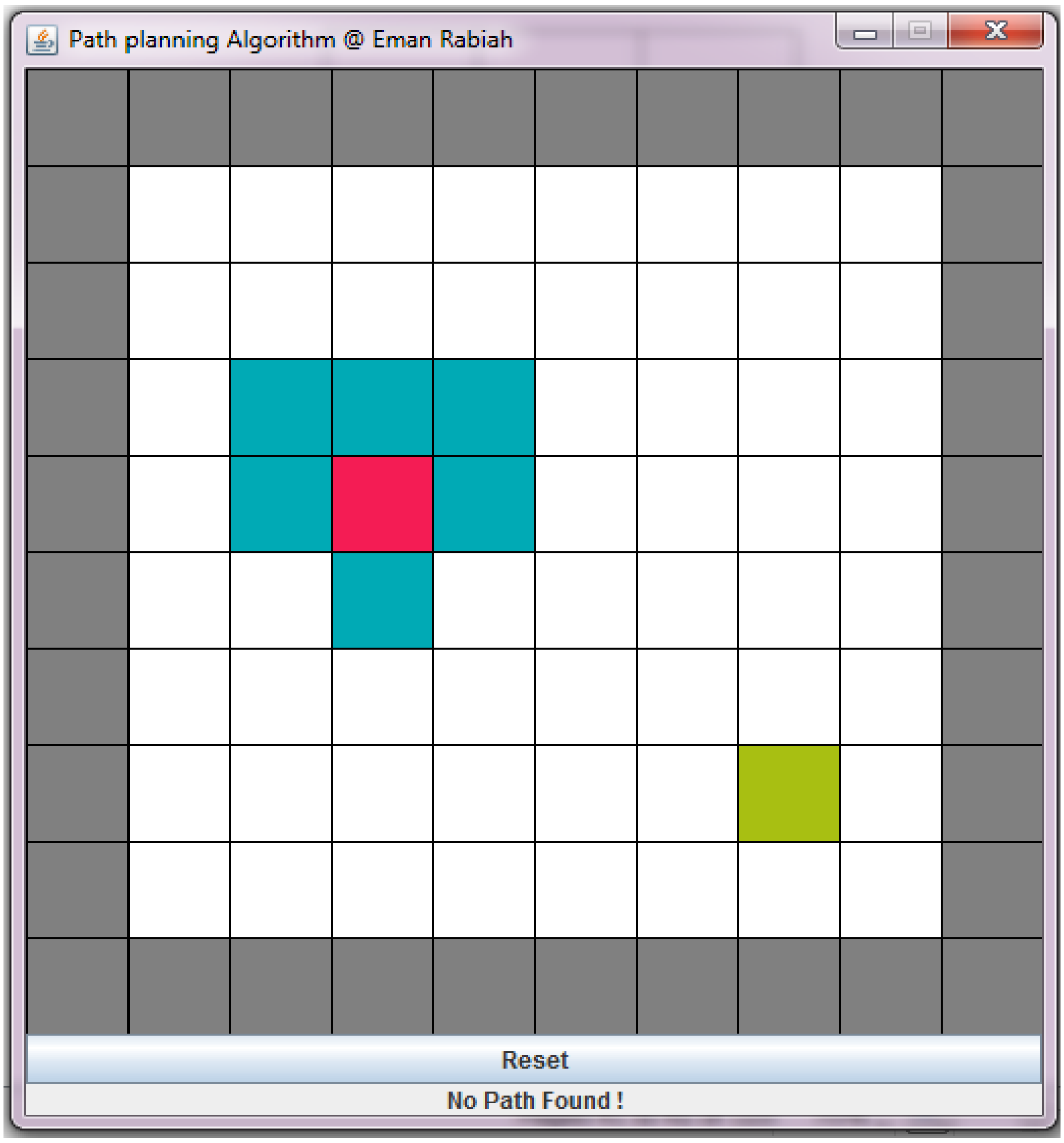


Figure 5.10: An example of case 5: discarding the down-left and down-right neighbors

- Refine diagonal neighbors (Case 6):

  In this test case, we fill all the cells that surround the robot position by obstacles except the (up-left) and the (up-right) cells. The system responses by "No path found" because the robot can not move to the target through any of these cells, which proves that case 6 of discarding the diagonal neighbors (up-left and up-right) is working correctly (see Figure 5.11).

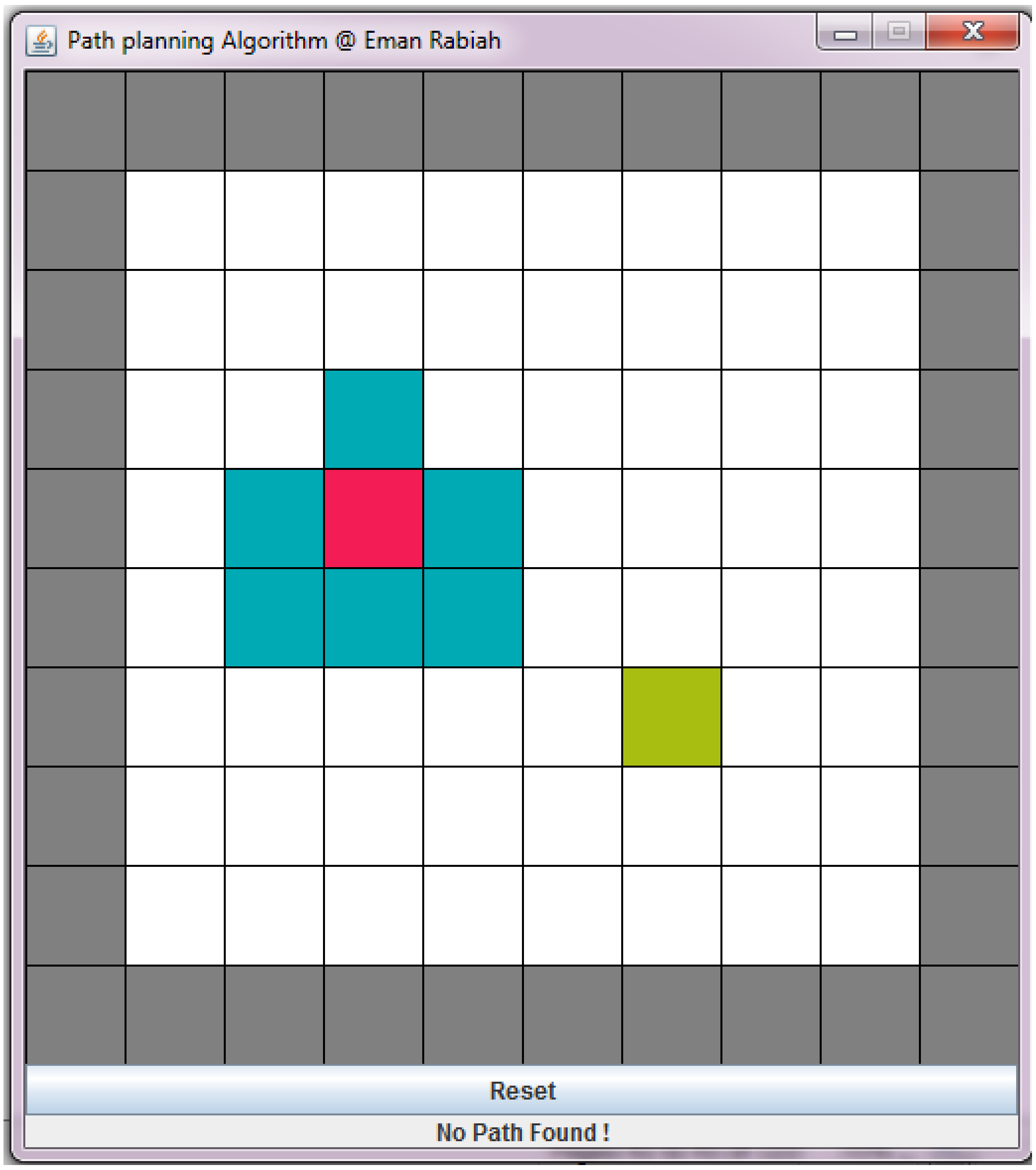


Figure 5.11: An example of case 6: discarding the up-left and up-right neighbors

- Refine diagonal neighbors (Case 7):

  In this test case, we fill all the cells that surround the robot position by obstacles except the (down-right) and the (up-right) cells. The system responses by "No path found" because the robot cannot move to the target through any of these cells, which proves that case 7 of discarding the diagonal neighbors (down-right and up-right) is working correctly (see Figure 5.12).

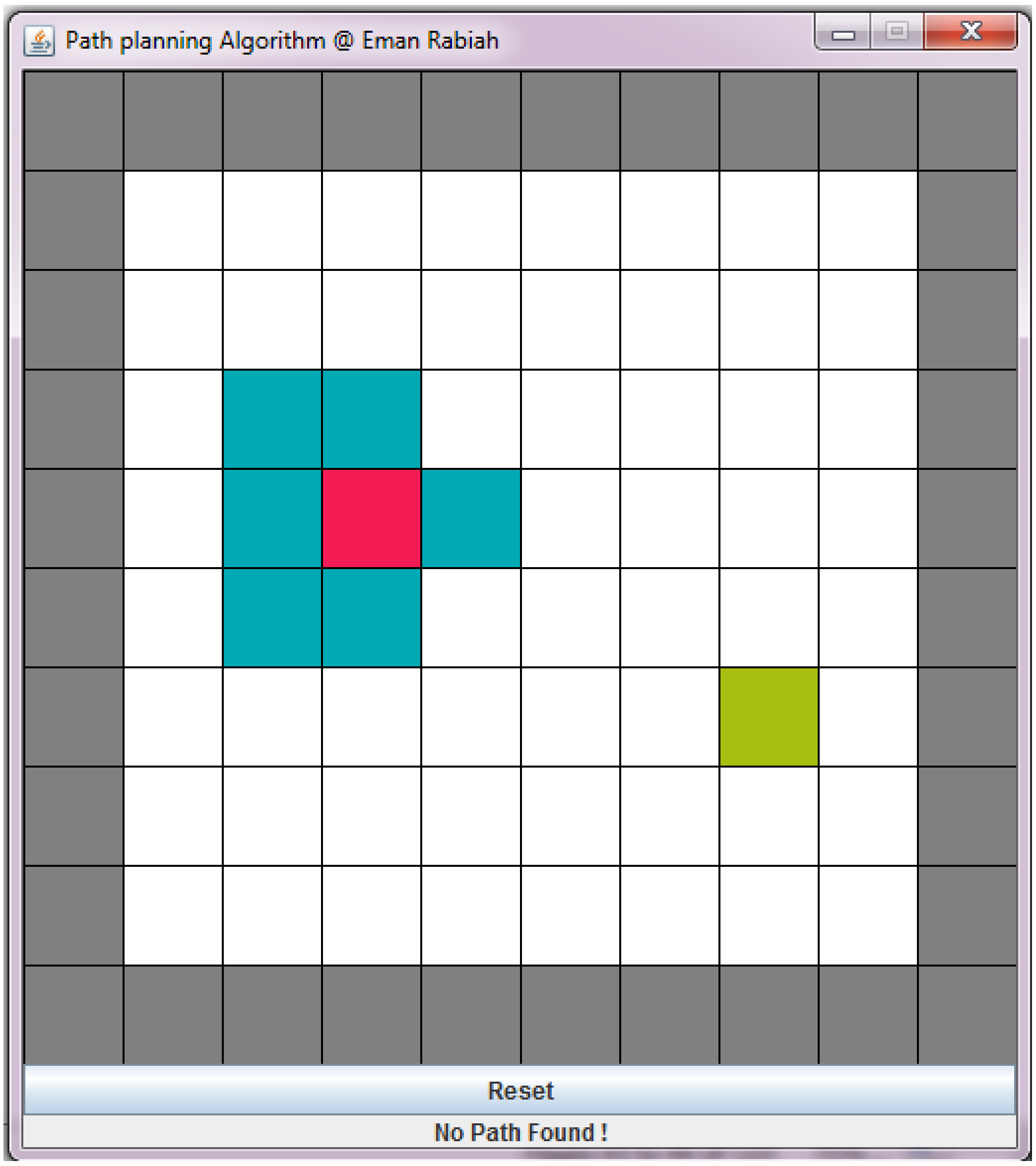


Figure 5.12: An example of case 7: discarding down-right and up-right neighbors

- Refine diagonal neighbors (Case 8):

  In this test case, we fill all the cells that surround the robot position by obstacles except the (down-left) and the (up-left) cells. The system responses by “No path found” because the robot cannot move to the target through any of these cells, which proves that case 8 of discarding the diagonal neighbors (down-left and up-left) is working correctly (see Figure 5.13).

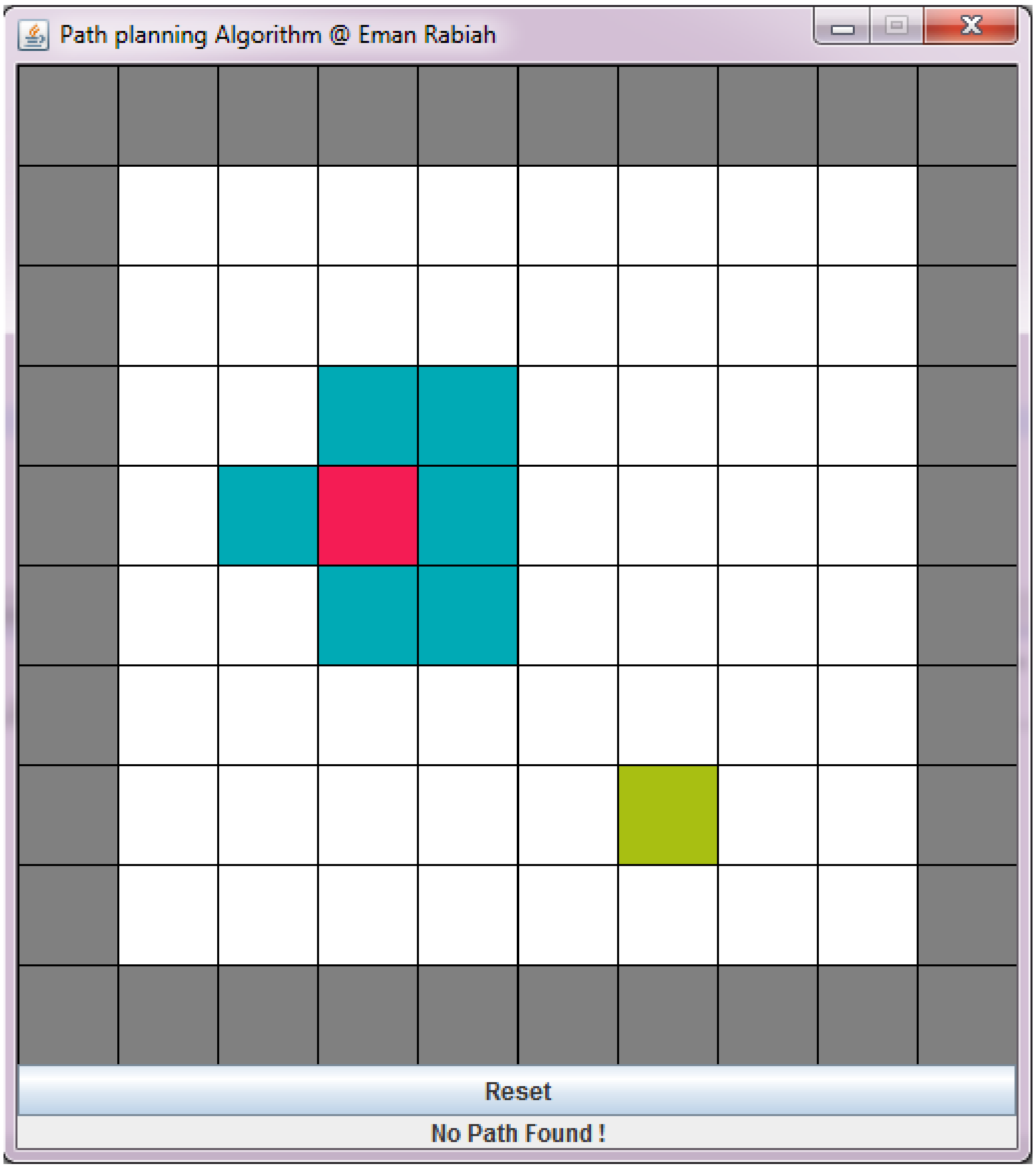


Figure 5.13: An example of case 8: discarding down-left and up-left neighbors

5.1.3 The robot movements

In this category, we illustrate some test cases that show the required robot moves to safely reach the target. The following figure (Figure 5.14) shows the basic moves of the robot, which are : up, down, right, left, up-right, up-left, down-left, and down-up. We assign an abbreviation for each move to make it easier to display it in the simulator (see Figure 5.13).

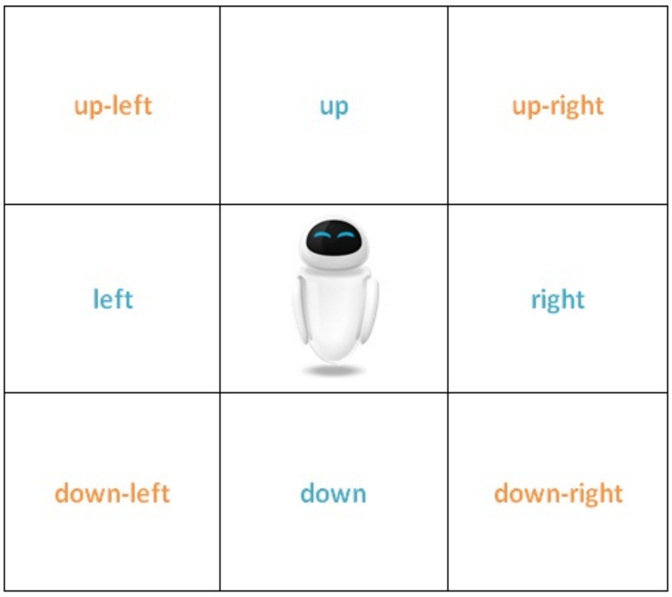
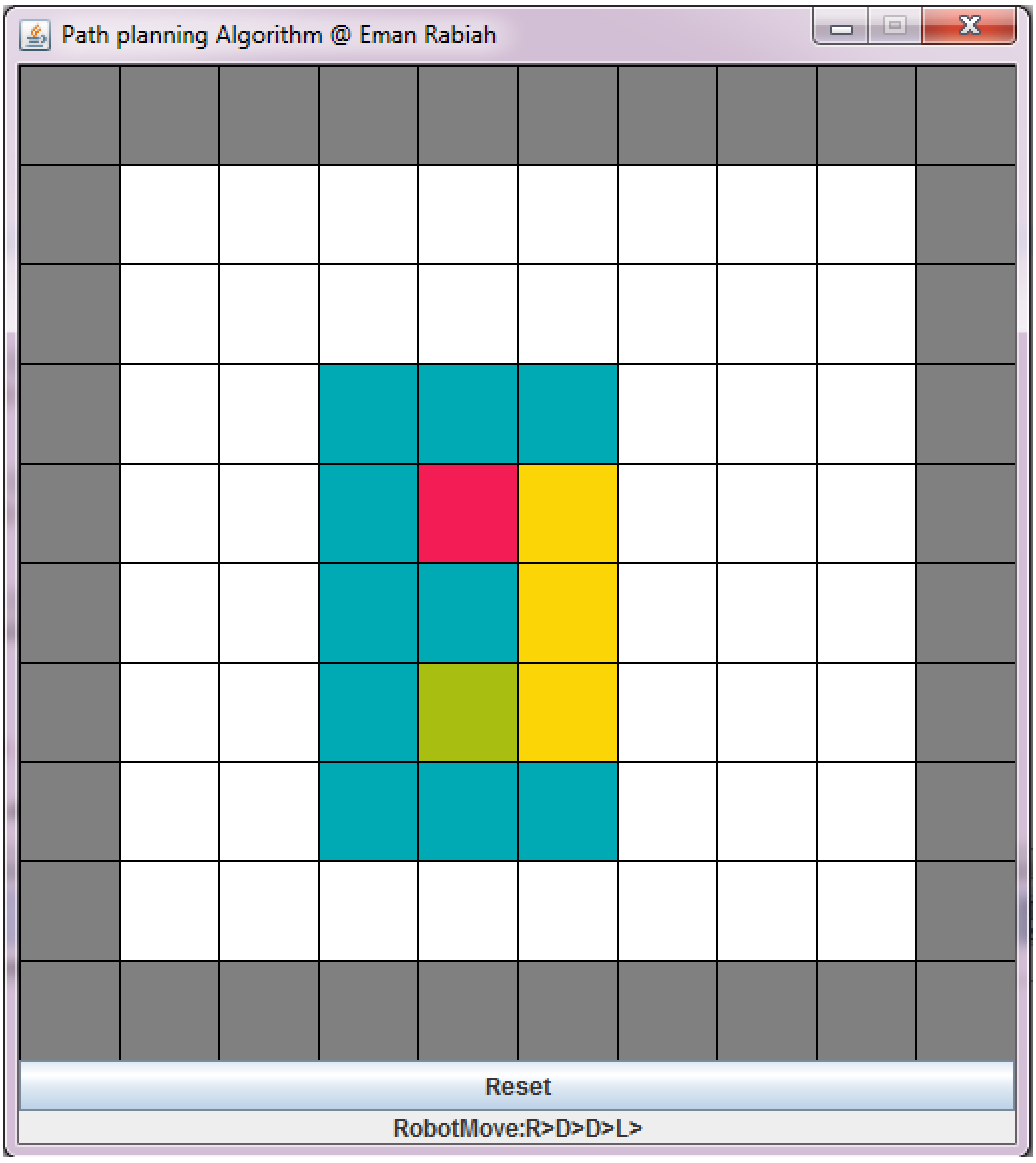


Figure 5.14: The basic moves of the Robot

up	down	right	left	up-right	up-left	down-right	down-left
U	D	R	L	UR	UL	DR	DL

Table 5.1: The abbreviation of each robot move that will appear in the simulator

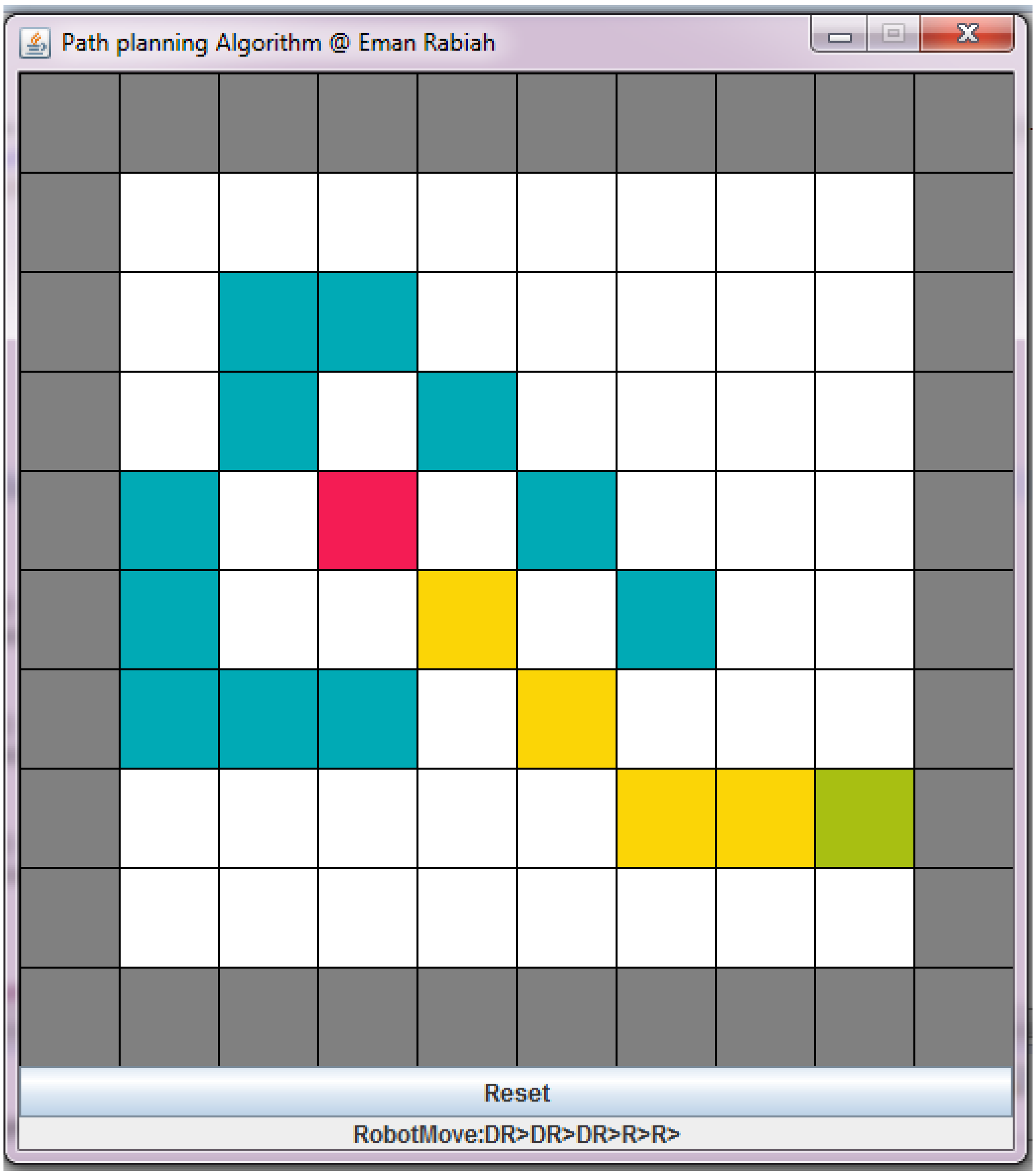
In the following test case (see Figure 5.15), we assign the robot the target and the obstacle location, then we trigger the path planning algorithm; the required robot moves are displayed on the GUI as : RobotMove: $R > D > D > L >$, which means the robot should move from its position to the right, then two times, then to the left to reach the target.



RobotMove:R>D>D>L>

Figure 5.15: Test case 1: The robot moves to reach the target

Another test case is illustrated in the below figure(see Figure 5.16) that shows the required robot moves on the GUI as : RobotMove: $DR > DR > DR > R > R >$, which means the robot should move from its position three times to the down-right, then two times to the right to reach the target.



RobotMove:DR>DR>DR>R>R>

Figure 5.16: Test case 2: The robot moves to reach the target

5.2 Summary

We developed an implementation that is consistent with the refined concrete specification. We use various scenarios to demonstrate that this implementation is consistent with Z specification. In fact, the scenarios simulated directly the behavior of each of our Z schemas.

Chapter 6: Discussion and Conclusion

Our overview of the state of the art in formal software development shows that research and case studies confirm the importance of using formal methods and the their effectiveness to derive reliable and robust software systems. These successful experiences formed the basis for our motivation to identify a critical problem, to specify it, and to refine it formally to get a reliable system.

In this thesis, we chose to use one of the formal methods to specify a fundamental task of robotic systems, because these system are described as critical systems, and they are designed to carry out several tasks intelligently on behalf of human; thus, reliability and safety of these systems are fundamental. They are expected to always behave correctly and accurately without causing any danger. Failure of these systems may endanger human life.

There are several formal methods that we can use to specify software systems, such as: Z notation, process algebra, Alloy, VDM, etc. We chose Z notation, because of two major reasons: first, we found that Z notation is very descriptive language and its major component (which is the schema) is very close to the structure of the current object-oriented programming languages; the second reason is, the availability of the theory underlying the formal refinement process that allows us to transform each Z specification component into its corresponding concrete program.

Our focus in this research was on specifying formally the navigation tasks of the autonomous robots in terms of path planning; consequently, we set to describe formally the process of finding the shortest path that the robot can follow to safely reach the target. The modified A* algorithm was fully specified and refined.

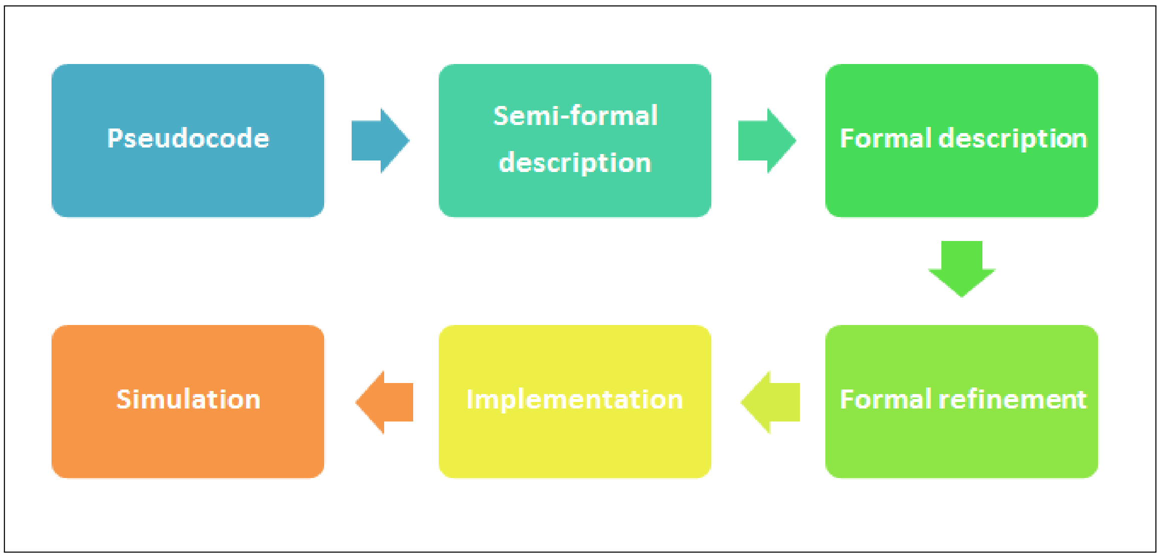


Figure 6.1: The stages of our research

We started our research by comparing between the several path planning algorithms such as: A* , D*, Dijkstra, etc. During the initial analysis of all of the path planning algorithms, we found that all of these algorithms are finding the shortest path for a (point) that does not have dimensions, while in our case we want to find the shortest path for a robot that has dimensions. We chose to focus on the A* algorithm because of its efficiency and versatility.

During the first stage of our research (see Figure 6.1), we analyzed the pseudocode of the A* algorithm, and we did a little modification to guarantee the robot safety, The major modification that we did in the A* algorithm is to exclude some impossible and unsafe cases, because of its high probability that a collision occurs while the robot is trying to reach the target. We called these cases (refinement of the diagonal neighbors) which means during the search process, there are some diagonal neighbors are going to be discarded because the robot can't move safely through them.

After we agreed upon the needed updates that ensure the robot safety, we started to describe the behavior of our system semi-formally, using basic mathematical structures and normal English language statements. We used The semi-formal description of our system as a base to build the formal model.

After describing the behavior of our system semi-fromally, we transformed that description into a formal description using one of the formal specification languages. We used Z notation to build the formal model (abstract) that describes the overall behavior of our system formally. This abstract model is also described as a mathematical model that captures the functionality of the system based on logic and discrete structures.

Our next step was the formal refinement process. The abstract model (mathematical model) was successively transformed into its corresponding implementable code. The refinement process involves removing uncertainty and elaborating the concrete design of our system.

After getting the code from the formal refinement process we implemented a simulation to show the consistency of the specification and the implementation. We did several execution scenarios to test the feasibility of our approach, and they showed correct results that proves feasibility of our approach.

To summarize, in this research we succeeded in generating a reliable and a safe path planning algorithm by applying a formal process. Our results demonstrate the benefits of the Z notation in the specification of critical systems. We showed also that formal methods are robust techniques that can contribute to enhance safety and reliability. The use formal refinement process shows that the transformation of an abstract model into its corresponding implementable code is a sound software engineering strategy that can contribute to the development of provable systems.

Z in its generality does not lend itself to automatic refinement. As a future extension of our work, we pose the question: what kind of constraints to impose on Z to be able to generate automatically the implementation of our system from the specification? Thus, we suggest to build a system that can aid the software engineers to transform the mathematical model (Z specification) into its equivalent implementation. This work can be initiated by defining a subset of Z and by identifying a set of general rules that can translate the abstract model into its equivalent code. After setting the general rules, the system can offer appropriate implementation suggestions for each mathematical construct. For example, for the power set in Z, the system can offer list of suggestions such as: transforming the power set into Array List or Array or stack, etc.

The use of formal methods in this research supports the many claims that formality in software development is highly effective [17, 20]. Formal methods provide robust techniques that software engineers can use to derive reliable software systems, especially when dealing with critical systems. Applying these methods from the specification stage to implementation will, not only enhance quality and productivity, but reduce errors, failures, and cost-overruns by ensuring that implementations are correct by construction.

Despite the many benefits of the formal methods, there are many reasons that may prevent software engineers to adopt them. One of these reasons is the complexity of applying these methods, for example: building the formal specification itself is a hard job that needs a lot of time of analysis and multiple tries until getting the final version of the formal model.

However, it is up to software engineers to decided whether to accept the difficulty of these methods to get a robust software system or to follow other techniques of software development that may lead to derive a low quality software. But we think that handling complexity at early stages is better than handling software failure later.

Bibliography

[1] Royce, W., "Current problems[in aerospace software engineering]," *IN: Aerospace software engineering- A collection of concepts(A 92-19376 06-61). Washington, DC, American Institute of Aeronautics and Astronautics, Inc., 1991,*, pp. 5–15, 1991.

[2] Naur, P. and Randell, B., "Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7-11 oct. 1968, brussels, scientific affairs division, nato," 1969.

[3] Ammann, P. and Offutt, J., *Introduction to software testing*. Cambridge University Press, 2008.

[4] Parmar, K. and Kathane, B., "The future prospects of software testing," 2014.

[5] "Triple modular redundancy." `http://en.wikipedia.org/wiki/Triple_modular_redundancy`. [Online; accessed 30-December-2014].

[6] "Formal methods." `http://en.wikipedia.org/wiki/Formal_methods`. [Online; accessed 02-August-2014].

[7] Meyer, B., "From process to product: Where is software headed," *IEEE Computer*, vol. 28, no. 8, p. 23, 1995.

[8] "Introduction to robotic # 4: Path-planning." `http://correll.cs.colorado.edu/?p=965`. [Online; accessed 28-January-2015].

[9] “Shortest path problem.” `http://en.wikipedia.org/wiki/Shortest_path_problem#Algorithms`. [Online; accessed 30-September-2014].

[10] “A* search algorithm.” `http://en.wikipedia.org/wiki/A*_search_algorithm`. [Online; accessed on 03-February-2015].

[11] “Implementation notes from amitâĂŹs thoughts on pathfinding.” `http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html`. [Online; accessed on 03-February-2015].

[12] “what is formal methods.” `http://shemesh.larc.nasa.gov/fm/fm-what.html`. [Online; accessed 30-January-2015].

[13] Holloway, C. M., “Why engineers should consider formal methods,” in *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*, vol. 1, pp. 1–3, IEEE, 1997.

[14] Janota, A., “Using z specification for railway interlocking safety,” *Transportation Engineering*, vol. 28, no. 1-2, pp. 39–53, 2001.

[15] Wing, J. M., “A specifier’s introduction to formal methods,” *Computer*, vol. 23, no. 9, pp. 8–22, 1990.

[16] Thomas, M., “The industrial use of formal methods,” *Microprocessors and Microsystems*, vol. 17, no. 1, pp. 31–36, 1993.

[17] Hall, A., “Seven myths of formal methods,” *Software, IEEE*, vol. 7, no. 5, pp. 11–19, 1990.

[18] Rushby, J., *Formal methods and the certification of critical systems.* SRI International. Computer Science Laboratory, 1993.

[19] Barrett, G., "Verifying the transputer," in *Proceedings of the first conference of the North American Transputer Users Group on Transputer research and applications 1*, pp. 17–24, IOS Press, 1990.

[20] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J., "Formal methods: Practice and experience," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 19, 2009.

[21] Guiho, G. and Hennebert, C., "Sacem software validation," in *Software Engineering, 1990. Proceedings., 12th International Conference on*, pp. 186–191, IEEE, 1990.

[22] Hennebert, C. and Guiho, G., "Sacem: A fault-tolerant system for train speed control," in *FTCS*, pp. 624–628, IEEE COMPUTER SOCIETY PRESS, 1993.

[23] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M., "Use of formal methods at amazon web services," 2014.

[24] Lightfoot, D., *Formal specification using Z.* Macmillan Press, 1991.

[25] Wirth, N., "Program development by stepwise refinement," *ACM*, vol. 14, no. 4, pp. 221–227, 1971.

[26] Woodcock, J. and Davies, J., *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., 1996.

[27] Derrick, J. and Boiten, E., *Refinement in Z and Object-Z.* Springer, 2014.

[28] Kim, M., Kang, K. C., and Lee, H., "Formal verification of robot movements - a case study on home service robot shr100," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pp. 4739–4744, April 2005.

[29] Saberi, A. K., Groote, J. F., and Keshishzadeh, S., "Analysis of path planning algorithms: a formal verification-based approach," in *Advances in Artificial Life, ECAL*, vol. 12, pp. 232–239, 2013.

[30] Yassine Belkhouche, M. and Belkhouche, B., "Formal specification and simulation of the robot perceptual system," in *Novel Algorithms and Techniques In Telecommunications, Automation and Industrial Electronics* (Sobh, T., Elleithy, K., Mahmood, A., and Karim, M., eds.), pp. 140–143, Springer Netherlands, 2008.

[31] Belkhouche, M. and Belkhouche, B., "Formal specification and simulation of the robot path planner," in *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, pp. 4484–4489, Oct 2009.

[32] Mohamad, R., Jawawi, D. N. A., Deris, S., and Mamat, R., "Formal specification of a wall-climbing robot using z–a case study of small-scale embedded hard real-time system," *Jurnal Teknologi*, vol. 34, no. 1, pp. 25–40, 2012.

[33] Lynch, N., Segala, R., and Vaandrager, F., "Hybrid i/o automata," *Information and computation*, vol. 185, no. 1, pp. 105–157, 2003.

[34] Fehnker, A., Vaandrager, F., and Zhang, M., "Modeling and verifying a lego car using hybrid i/o automata," in *Quality Software, 2003. Proceedings. Third*

International Conference on, pp. 280–289, IEEE, 2003.

[35] Weinberg, H. B., *Correctness of vehicle control systems: A case study*. PhD dissertation, Massachusetts Institute of Technology, 1996.

[36] Fontenot, D., "A java gui that visualizes the a* search algorithm." `https://github.com/davidfontenot/astar`. [Online; accessed 23-March-2015].

Appendix A: Notations

Symbol	Name
$\mathbb{P}$	Power set
$\mathbb{N}$	Set of natural numbers
$\forall$	For all
$\perp$	Undefined
$\cup$	Union
$\bullet$	Such that
$m \ldots n$	Set of values m to n inclusive
$\varnothing$	Empty set
$\rightarrowtail\!\!\!\twoheadrightarrow$	Bijective function
$\wedge$	And
$\vee$	Or
$\mapsto$	Maps to
seq	Sequence
iseq	Injective sequence (no duplicates)
$\langle\rangle$	Empty sequence
$\frown$	Concatenation operator
$\in$	Belongs to
$\notin$	Does not belong to
$\rhd$	Range restriction
⩤	Domain subtraction

Symbol	Name
$<$	Less than
$=$	Equals to
$=$	Not equals to
$\times$	Cartesian product
Δ	Delta (change of state)
Ξ	No change of state
$a?$	Input to an operation
$a!$	Output from an operation
a	State of a component before an operation
a'	State of a component after an operation

Appendix B: Z document

- Definitions $maxX = const1$

  $maxY = const2$

  $Pair = \mathbb{N} \times \mathbb{N}$

  $undefined == \perp$

  $RESPONSE ::= FreeThePairFirst \mid NoPathFound \mid youCantFreeBorder \mid$

  $ItsBorderPosition \mid APathIsFound$

  $ROBOTMOVE ::= up \mid down \mid right \mid left \mid upRight \mid upLeft \mid downRight \mid$

  $downLeft$

- System status

Map

$map : \mathbb{P}\, Pair$

$MapInit$

ΔMap

$map' = \{\forall\, x_m : 0 \ldots maxX;\ \forall\, y_m : 0 \ldots maxY \bullet (x_m, y_m)\}$

Border

$border : \mathbb{P}\, Pair$

BorderInit

$\Delta Border$

$border' = \{\{\forall\, x_p : 0 \ldots maxX \bullet (x_p, 0)\} \cup$

$\{\forall\, x_p : 0 \ldots maxX \bullet (x_p, maxY)\} \cup$

$\{\forall\, y_p : 0 \ldots maxY \bullet (0, y_p)\} \cup$

$\{\forall\, y_p : 0 \ldots maxY \bullet (maxX, y_p)\}$

Target

$x_t : \mathbb{N}$

$y_t : \mathbb{N}$

TargetInit

$\Delta Target$

$x_t' = undefined$

$y_t' = undefined$

$Robot$

$x_r : \mathbb{N}$

$y_r : \mathbb{N}$

$RobotInit$

$\Delta Robot$

$x_r' = undefined$

$y_r' = undefined$

$Obstacle$

$obstacleList : \mathbb{P}\, Pair$

$ObstacleInit$

$\Delta Obstacle$

$obstacleList' = \emptyset$

$BestPair$

$x_b : \mathbb{N}$

$y_b : \mathbb{N}$

$InitBestPair$

$\Delta BestPair$

$x'_b = undefined$

$y'_b = undefined$

$CurrentPair$

$x_c : \mathbb{N}$

$y_c : \mathbb{N}$

$InitCurrentPair$

$\Delta Current$

$x'_c = undefined$

$y'_c = undefined$

Neighbor

$neighbors1 : \mathbb{P}\, Pair$

$neighbors2 : \mathbb{P}\, Pair$

$neighbors3 : \mathbb{P}\, Pair$

$x_n : \mathbb{N}$

$y_n : \mathbb{N}$

InitNeighbor

$\Delta Neighbor$

$neighbors1' = \varnothing$

$neighbors2' = \varnothing$

$neighbors3' = \varnothing$

$x_n' = undefined$

$y_n' = undefined$

currentPosition

$x_{cp} : \mathbb{N}$

$y_{cp} : \mathbb{N}$

InitCurrentPosition

$\Delta currentPosition$

$x'_{cp} = undefined$

$y'_{cp} = undefined$

nextPosition

$x_{np} : \mathbb{N}$

$y_{np} : \mathbb{N}$

InitNextPosition

$\Delta nextPosition$

$x'_{np} = undefined$

$y'_{np} = undefined$

OpenF

$openF : Pair \rightarrowtail \mathbb{N}$

$InitOpenF$

$\Delta OpenF$

$\Xi Robot$

$\Delta CalculateFValue$

$openF' = \{(x_r \mapsto y_r) \mapsto f\}$

$OpenG$

$openG : Pair \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}$

$InitOpenG$

$\Delta OpenG$

$\Xi Robot$

$\Delta CalculateGCurrent$

$openG' = \{(x_r \mapsto y_r) \mapsto g_c\}$

$ParentChild$

$parentChild : Pair \rightarrowtail\!\!\!\!\rightarrow Pair$

InitParentChild

$\Delta ParentChild$

$parentChild' = \varnothing$

Closed

$closedList : \mathbb{P}\, Pair$

InitClosed

$\Delta Closed$

$closedList' = \varnothing$

Path

$path : \mathrm{seq}\, Pair$

$x_{path} : \mathbb{N}$

$y_{path} : \mathbb{N}$

$InitPath$

$\Delta Path$

$\Xi Target$

$path' = \langle (xt, yt) \rangle$

$x'_{path} = x_t$

$y'_{path} = y_t$

- Operations

$SetTargetPositionOK$

ΞMap

$\Delta Target$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$((x?, y?) \in map$

$(x_t, y_t) = (undefined, undefined)$

$(x'_t, y'_t) = (x?, y?))$

$\vee$

$((x?, y?) \in map$

$(x_t, y_t) \neq (undefined, undefined)$

$(x'_t, y'_t) = (x?, y?))$

$SetTargetPositionNotOK$

$\Xi Border$

$\Xi Obstacle$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$rep! : RESPONSE$

$((x?, y?) \in obstacleList$

$rep! = freePairFirst)$

$\vee$

$((x?, y?) \in border$

$rep! = ItsBorderPosition)$

SetRobotPositionOK

ΞMap

$\Delta Robot$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$((x?, y?) \in map$

$(x_r, y_r) = (undefined, undefined)$

$(x_r', y_r') = (x?, y?))$

$\vee$

$((x?, y?) \in map$

$(x_r, y_r) \neq (undefined, undefined)$

$(x_r', y_r') = (x?, y?))$

$SetRobotPositionNotOK$

$\Xi Border$

$\Xi Obstacle$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$rep! : RESPONSE$

$((x?, y?) \in obstacleList$

$rep! = freePairFirst)$

$\vee$

$((x?, y?) \in border$

$rep! = ItsBorderPosition)$

$SetObstaclePositionOK$

ΞMap

$\Delta Obstacle$

$x? : \mathbb{Z}$

$y? : \mathbb{Z}$

$(x?, y?) \in map$

$(x?, y?) \notin obstacleList$

$obstacleList' = obstacleList \cup \{(x?, y?)\}$

$SetObstaclePositionNotOK$

$\Xi Border$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$rep! : RESPONSE$

$((x?, y?) \in border$

$rep! = ItsBorderPosition)$

$FreeRobotPosition$

$\Delta Robot$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$(x?, y?) = (x_r, y_r)$

$(x_r', y_r') = (undefined, undefined)$

FreeObstaclePosition

$\Delta Obstacle$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$(x?, y?) \in obstacleList$

$obstacleList' = obstacleList \setminus \{(x?, y?)\}$

FreeTargetPosition

$\Delta Target$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$(x?, y?) = (x_t, y_t)$

$(x_t', y_t') = (undefined, undefined)$

FreeBorderPairNotOK

$\Xi Border$

$x? : \mathbb{N}$

$y? : \mathbb{N}$

$rep! : RESPONSE$

$(x?, y?) \in border$

$rep! = youCantFreeBorder$

EvaluateBestPair

$\Delta BestPair$

$\Xi OpenF$

$(x_b', y_b') = dom(openF \rhd min(ran\ openF))$

SearchPairsNoPath

$\Xi BestPair$

$\Xi Target$

$\Xi OpenF$

$rep! : RESPONSE$

$(x_b, y_b) \neq (x_t, y_t)$

$openF = \varnothing$

$rep! = NoPathFound$

EmptyNeighbors

$\Delta Neighbor$

$neighbors1' = \varnothing neighbors2' = \varnothing neighbors3' = \varnothing$

SearchPairs

$\Xi BestPair$

$\Xi Target$

$\Delta CurrentPair$

$\Delta Neighbor$

$\Delta Closed$

$\Delta OpenF$

$\Delta OpenG$

$(x_b, y_b) \neq (x_t, y_t)$

$openF \neq \varnothing$

$(x_c, y_c) = (x_b, y_b)$

$closedList' = closedList \cup \{(x_b, y_b)\}$

$openF' = \{(x_b, y_b)\} \lhd openF$

$openG' = \{(x_b, y_b)\} \lhd openG$

$neighbors1' = \{\{(x_c, y_c + 1)\} \cup \{(x_c, y_c - 1)\} \cup \{(x_c + 1, y_c)\} \cup \{(x_c - 1, y_c)\}$

$\cup \{(x_c - 1, y_c + 1)\} \cup \{(x_c - 1, y_c - 1)\} \cup \{(x_c + 1, y_c - 1)\} \cup \{(x_c + 1, y_c + 1)\}\}$

$ConstructPath$

$\Delta Path$

$\Xi Robot$

$\Xi BestPair$

$\Xi Target$

$rep! : RESPONSE$

$(x_b, y_b) = (x_t, y_t)$

$rep! = APathIsFound$

$(x_{path}, y_{path}) \neq (undefined, undefined)$

$(x_{path}, y_{path}) \neq (x_r, y_r)$

$(x'_{path}, y'_{path}) = dom(parentChild \rhd (x_{path}, y_{path}))$

$path' = \langle (x_{path}, y_{path}) \rangle \frown path$

$RefineDiagonalNeighborsCase1$

$\Delta Neighbor$

$\Xi CurrentPair$

$\Xi Obstacle$

$(x_c, y_c+1) \in neighbors1 \wedge ((x_c, y_c+1) \in obstacleList$

$(x_c+1, y_c) \in neighbors1 \wedge (x_c+1, y_c) \in obstacleList$

$neighbors2' = neighbors2 \cup \{(x_c+1, y_c+1)\}$

$RefineDiagonalNeighborsCase2$

$\Delta Neighbor$

$\Xi CurrentPair$

$\Xi Obstacle$

$(x_c, y_c+1) \in neighbors1 \wedge (x_c, y_c+1) \in obstacleList$

$(x_c-1, y_c) \in neighbors1 \wedge (x_c-1, y_c) \in obstacleList$

$neighbors2' = neighbors2 \cup \{(x_c-1, y_c+1)\}$

$RefineDiagonalNeighborsCase3$

$\Delta Neighbor$

$\Xi CurrentPair$

$\Xi Obstacle$

$(x_c, y_c - 1) \in neighbors1 \wedge (x_c, y_c - 1) \in obstacleList$

$(x_c + 1, y_c) \in neighbors1 \wedge (x_c + 1, y_c) \in obstacleList$

$neighbors2' = neighbors2 \cup \{(x_c + 1, y_c - 1)\}$

$RefineDiagonalNeighborsCase4$

$\Delta Neighbor$

$\Xi CurrentPair$

$\Xi Obstacle$

$(x_c, y_c - 1) \in neighbors1 \wedge (x_c, y_c - 1) \in obstacleList$

$(x_c - 1, y_c) \in neighbors1 \wedge (x_c - 1, y_c) \in obstacleList$

$neighbors2' = neighbors2 \cup \{(x_c - 1, y_c - 1)\}$

$RefineDiagonalNeighborsCase5$

$\Delta Neighbor$

$\Xi CurrentPair$

$\Xi Obstacle$

$(x_c, y_c + 1) \in neighbors1 \wedge (x_c, y_c + 1) \in obstacleList$

$neighbors2' = neighbors2 \cup \{(x_c + 1, y_c + 1)\} \cup \{(x_c - 1, y_c + 1)\}$

$RefineDiagonalNeighborsCase6$

$\Delta Neighbor$

$\Xi CurrentPair$

$\Xi Obstacle$

$(x_c, y_c - 1) \in neighbors1 \wedge (x_c, y_c - 1) \in obstacleList$

$neighbors2' = neighbors2 \cup \{(x_c + 1, y_c - 1)\} \cup \{(x_c - 1, y_c - 1)\}$

$\mathit{RefineDiagonalNeighborsCase7}$

$\Delta \mathit{Neighbor}$

$\Xi \mathit{CurrentPair}$

$\Xi \mathit{Obstacle}$

$(x_c + 1, y_c) \in \mathit{neighbors1} \wedge (x_c + 1, y_c) \in \mathit{obstacleList}$

$\mathit{neighbors2}' = \mathit{neighbors2} \cup \{(x_c + 1, y_c + 1)\} \cup \{(x_c + 1, y_c - 1)\}$

$\mathit{RefineDiagonalNeighborsCase8}$

$\Delta \mathit{Neighbor}$

$\Xi \mathit{CurrentPair}$

$\Xi \mathit{Obstacle}$

$(x_c - 1, y_c) \in \mathit{neighbors1} \wedge (x_c - 1, y_c) \in \mathit{obstacleList}$

$\mathit{neighbors2}' = \mathit{neighbors2} \cup \{(x_c - 1, y_c + 1)\} \cup \{(x_c - 1, y_c - 1)\}$

$\mathit{DeleteDiagonalNeighbors}$

$\Delta \mathit{Neighbor}$

$\mathit{neighbors1}' = \mathit{neighbors1} \setminus \mathit{neighbors2}$

$RefineNeighborsFromObstaclesAndBorder$

$\Xi Obstacle$

$\Xi Border$

$\Delta Neighbor$

$((x_n, y_n) \in neighbors1$

$(x_n, y_n) \in obstacleList$

$neighbors3' = neighbors3 \cup (x_n, y_n))$

$\vee$

$((x_n, y_n) \in neighbors1$

$(x_n, y_n) \in border$

$neighbors3' = neighbors3 \cup (x_n, y_n))$

$DeleteNeighborsFromObstaclesAndBorder$

$\Delta Neighbor$

$neighbors1' = neighbors1 \setminus neighbors3$

$CalculateHValue$

$\Xi Target$

$\Xi Neighbor$

$h : \mathbb{N}$

$h' = \mid y_n - y_t \mid + \mid x_n - x_t \mid$

$CalculateGCurrent$

$\Xi Robot$

$\Xi Current$

$g_c : \mathbb{N}$

$g_c' = \mid y_c - y_r \mid + \mid x_c - x_r \mid$

$CalculateGNeighbor$

$\Xi Robot$

$\Xi Neighbor$

$g_n : \mathbb{N}$

$g_n' = \mid y_n - y_r \mid + \mid x_n - x_r \mid$

$CalculateFValue$

$\Xi Robot$

$\Xi Current$

$\Delta CalculateGCurrent$

$\Delta CalculateHValue$

$f : \mathbb{N}$

$f' = g_c + h$

$CalculateCostValue$

$\Delta CalculateGCurrent$

$\Xi Current$

$\Xi Neighbor$

$cost : \mathbb{N}$

$cost' = gc + (\mid y_c - y_n \mid + \mid x_c - x_n \mid)$

$EvaluateNeighborsCase1$

$\Delta CalculateGNeighbor$

$\Delta CalculateCostValue$

$\Xi Neighbor$

$\Delta OpenF$

$\Delta OpenG$

$oldGn : \mathbb{N}$

$(x_n, y_n) \in openF$

$oldGn' = openG(x_n, y_n)$

$cost < oldGn$

$openF' = \{(x_n, y_n)\} \lhd openF$

$openG' = \{(x_n, y_n)\} \lhd openG$

$EvaluateNeighborsCase2$

$\Delta CalculateCostValue$

$\Delta CalculateHValue$

$\Xi Neighbor$

$\Xi CurrentPair$

$\Xi Closed$

$\Delta OpenG$

$\Delta OpenF$

$\Delta ParentChild$

$newGn : \mathbb{N}$

$(x_n, y_n) \notin openF$

$(x_n, y_n) \notin closedList$

$newGn' = cost$

$openF' = openF \cup \{(x_n \mapsto y_n) \mapsto (newGn + h)\}$

$openG' = openG \cup \{(x_n \mapsto y_n) \mapsto newGn\}$

$parentChild' = parentChild \cup \{(x_c, y_c) \mapsto (x_n, y_n)\}$

FollowPath

$\Delta Path$

$\Delta currentPosition$

$\Delta nextPosition$

$path \neq \langle\rangle$

$(x'_{cp}, y'_{cp}) = head\, path$

$path' = tail\, path$

$(x'_{np}, y'_{np}) = head\, path$

MoveDown

$\Delta CurrentPosition$

$\Xi NextPosition$

$rm : ROBOTMOVE$

$(x_{np}, y_{np}) = (x_{cp}, y_{cp} + 1)$

$rm' = down$

$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$

MoveUp

$\Delta CurrentPosition$

$\Xi NextPosition$

$rm : ROBOTMOVE$

$(x_{np}, y_{np}) = (x_{cp}, y_{cp} - 1)$

$rm' = up$

$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$

MoveRight

$\Delta CurrentPosition$

$\Xi NextPosition$

$rm : ROBOTMOVE$

$(x_{np}, y_{np}) = (x_{cp} + 1, y_{cp})$

$rm' = right$

$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$

MoveLeft

$\Delta CurrentPosition$

$\Xi NextPosition$

$rm : ROBOTMOVE$

$(x_{np}, y_{np}) = (x_{cp} - 1, y_{cp})$

$rm' = left$

$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$

MoveUpRight

$\Delta CurrentPosition$

$\Xi NextPosition$

$rm : ROBOTMOVE$

$(x_{np}, y_{np}) = (x_{cp} + 1, y_{cp} - 1)$

$rm' = upRight$

$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$

$MoveUpLeft$

$\Delta CurrentPosition$

$\Xi NextPosition$

$rm : ROBOTMOVE$

$(x_{np}, y_{np}) = (x_{cp} - 1, y_{cp} - 1)$

$rm' = upLeft$

$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$

$MoveDownRight$

$\Delta CurrentPosition$

$\Xi NextPosition$

$rm : ROBOTMOVE$

$(x_{np}, y_{np}) = (x_{cp} + 1, y_{cp} + 1)$

$rm' = downRight$

$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$

MoveDownLeft

$\Delta CurrentPosition$

$\Xi NextPosition$

$rm : ROBOTMOVE$

$(x_{np}, y_{np}) = (x_{cp} - 1, y_{cp} + 1)$

$rm' = downLeft$

$(x'_{cp}, y'_{cp}) = (x_{np}, y_{np})$

Appendix C: Equivalent implementation

- The type Pair

```
class Pair {
int x;
int y;
}
```

- Supportive methods

```
public int getX() { return x; }
public int getY() { return y; }
```

- SetRobotPositionOK & SetRobotPositionNotOK

```
int x ;
int y ;
for(int j=0 ; j<maxY ; j++) {
for(int i = 0; i <maxX; i++) {
Pair mapPair = map[i][j];
if (x == mapPair.getX() && y == mapPair.getY() ) {
for(int i=0 ; i<obstacleList.size() ; i++) {
Pair obstaclePair = obstacleList.get(i);
xo = obstaclePair.getX();
yo = obstaclePair.getY();
if(x = = xo && y = = yo ){
System.out.println("Free The Pair First");   }}
for(int i=0 ; i<border.size() ; i++) {
Pair borderPair = border.get(i);
xbo = borderPair.getX();
ybo = borderPair.getY();
if(x = = xbo && y = = ybo ){
System.out.println("It's Border Position");}}
if((x!=xo || y !=yo) && (x!=xbo || y!=ybo) ){
            if(x!=undefined || y!=undefined ){
            // Set new robot position
            xr=x;
            yr=y;}
else{
            // Replace the old robot position by a new one
            xr=x;
            yr=y;          }}}}}
```

- SetTargetPositionOK & SetTargetPositionNotOK

```
int x ;
int y ;
for(int j=0 ; j<maxY ; j++) {
for(int i = 0; i <maxX; i++) {
Pair mapPair = map[i][j];
if (x = = mapPair.getX() && y = = mapPair.getY() ) {
for(int i=0 ; i<obstacleList.size() ; i++) {
Pair obstaclePair = obstacleList.get(i);
xo = obstaclePair.getX();
yo = obstaclePair.getY();
if(x = = xo && y = = yo ){
System.out.println("Free The Pair First ");
}}}}}
for(int i=0 ; i<border.size() ; i++) {
Pair borderPair = border.get(i);
xbo = borderPair.getX();
ybo = borderPair.getY();
if(x = = xbo && y = = ybo ){
System.out.println("It's Border Position " +
 xbo +"," +ybo );
break;}}
if(x!=xo && y !=yo || x!=xbo && y !=ybo ){
            if(x!=undefined || y!=undefined ){
            // Set new robot position
            xr=x;
            yr=y;
            }
            else{
            // Replace the old robot position by a new one
            xr=x;
            yr=y;
}}
```

• SetObstaclePositionOK & SetObstaclePositionNotOK

```
int x ;
int y ;
for(int j=0 ; j<maxY ; j++) {
for(int i = 0; i <maxX; i++) {
Pair mapPair = map[i][j];
if (x = = mapPair.getX() && y = = mapPair.getY() ) {
if(! obstacleList.contains(map[x][y]) ){
obstacleList.add(map[x][y]);
}
if( border.contains(map[x][y]) ){
System.out.println("It's Border Position");
}}}}
```

• FreeRobotPosition

```
int x ;
int y ;
if (x = = xr && y = = yr) {
```

```
xr= undefined;
yr= undefined;
}
```

- FreeObstaclePosition

```
int x ;
int y ;
if (obstacleList.contains(map[x][y])) {
obstacleList.remove(map[x][y]);
}
```

- FreeTargetPosition

```
int x ;
int y ;
if (x = = xt && y = = yt) {
xt= undefined;
yt= undefined;
}
```

- FreeBorderPairNotOK

```
int x ;
int y ;
if (border.contains(map[x][y])) {
System.out.println("you Cant Free Border");
}
```

- EvaluateBestPair

```
private Pair EvaluateBestPair() {
return Collections.min(openF.entrySet(),
new Comparator<Map.Entry<Pair,Integer>>() {
public int compare(Entry<Pair, Integer> o1,
Entry<Pair, Integer> o2) {
return o1.getValue().intValue() - o2.getValue().intValue();
}})
.getKey();
}
```

- SearchPairsNoPath

```
Pair bestPair = map [xb][yb] ;
Pair target = map [xt][yt] ;
if (openF.size() == 0 && bestPair != target){
System.out.println("No Path Found");
}
```

- EmptyNeighbors

```
neighbors1.clear();
neighbors2.clear();
neighbors3.clear();
```

- SearchPairs

```
while(openF.size() != 0 ) {
```

```
if (bestPair != target){
Pair currentPair = bestPair;
closedList.add(bestPair);
openF.remove(bestPair);
openG.remove(bestPair);
xc = currentPair.getX();
yc = currentPair.getY();
neighbors1.clear();
neighbors2.clear();
neighbors3.clear();
// add neighbors of the current Pair
Pair n1 = map [xc+1][yc];
Pair n2 = map [xc−1][yc];
Pair n3 = map [xc][yc+1];
Pair n4 = map [xc][yc−1];
Pair n5 = map [xc+1][yc+1];
Pair n6 = map [xc−1][yc+1];
Pair n7 = map [xc+1][yc−1];
Pair n8 = map [xc−1][yc−1];
neighbors1.add(n1);
neighbors1.add(n2);
neighbors1.add(n3);
neighbors1.add(n4);
neighbors1.add(n5);
neighbors1.add(n6);
neighbors1.add(n7);
neighbors1.add(n8);
}}
```

- ConstructPath

```
public void constructPath(){
if (bestPair == target){
System.out.println(R5);
Pair pathPair= map[xt][yt];
int xpath = pathPair.getX();
int ypath = pathPair.getY();
path.push(pathPair);
while(pathPair != map[xr][yr] && pathPair!=null ) {
pathPair = parentChild.get(pathPair);
path.push(pathPair);
}}}
```

- RefineDiagonalNeighborsCase1

```
public void refineDiagonalNeighborsCase1 (Pair
neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn = = xc && yn = = yc+1 &&
obstacleList.contains(neighbor)){
counter1++;
}
if (xn = = xc+1 && yn = = yc &&
obstacleList.contains(neighbor)){
counter1++;
}
if(counter1= =2)  {
Pair n = neighbors1.get(4);
neighbors2.add(n);
}}
```

- RefineDiagonalNeighborsCase2

```
public void refineDiagonalNeighborsCase2 (Pair
neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn = = xc && yn = = yc+1 &&
obstacleList.contains(neighbor)){
counter2++;
}
if (xn = = xc−1 && yn = = yc &&
obstacleList.contains(neighbor)){
counter2++;
}
if(counter2= =2) {
Pair n= neighbors1.get(5);
neighbors2.add(n);
}}
```

- RefineDiagonalNeighborsCase3

```
public void refineDiagonalNeighborsCase3 (Pair
neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn = = xc && yn = = yc−1 &&
obstacleList.contains(neighbor)){
counter3++;
}
if (xn = = xc+1 && yn = = yc &&
obstacleList.contains(neighbor)){
counter3++;
}
if(counter3= =2){
Pair n= neighbors1.get(6);
neighbors2.add(n);
}}
```

- RefineDiagonalNeighborsCase4

```
public void refineDiagonalNeighborsCase4 (Pair
neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn = = xc && yn = = yc−1 &&
obstacleList.contains(neighbor)){
counter4++;
}
if (xn = = xc−1 && yn = = yc &&
obstacleList.contains(neighbor)){
counter4++;
}
if(counter4= =2)  {
Pair n= neighbors1.get(7);
neighbors2.add(n);
}}
```

- RefineDiagonalNeighborsCase5

```
public void refineDiagonalNeighborsCase5 (Pair
neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc && yn == yc+1 &&
obstacleList.contains(neighbor)){
Pair n1= neighbors1.get(4);
Pair n2= neighbors1.get(5);
if(! neighbors2.contains(n1) ){
neighbors2.add(n1);
}
if(! neighbors2.contains(n2) ){
neighbors2.add(n2);
}}}
```

- RefineDiagonalNeighborsCase6

```
public void refineDiagonalNeighborsCase6 (Pair
neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc && yn == yc-1 &&
obstacleList.contains(neighbor)){
Pair n1= neighbors1.get(6);
Pair n2= neighbors1.get(7);
if(! neighbors2.contains(n1) ){
neighbors2.add(n1);
}
if(! neighbors2.contains(n2) ){
neighbors2.add(n2);
}}}
```

- RefineDiagonalNeighborsCase7

```
public void refineDiagonalNeighborsCase7 (Pair
neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc+1 && yn == yc &&
obstacleList.contains(neighbor)){
Pair n1= neighbors1.get(4);
Pair n2= neighbors1.get(6);
if(! neighbors2.contains(n1) ){
neighbors2.add(n1);
}
if(! neighbors2.contains(n2) ){
neighbors2.add(n2);
}}}
```

- RefineDiagonalNeighborsCase8

```
public void refineDiagonalNeighborsCase8 (Pair
neighbor,int xc, int yc){
```

```
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc-1 && yn == yc &&
obstacleList.contains(neighbor)){
Pair n1= neighbors1.get(5);
Pair n2= neighbors1.get(7);
if(! neighbors2.contains(n1) ){
neighbors2.add(n1);
}
if(! neighbors2.contains(n2) ){
neighbors2.add(n2);
}}}
```

- DeleteDiagonalNeighbors

```
public void deleteDiagonalNeighbors () {
neighbors1.removeAll(neighbors2);
}
```

- RefineNeighborsFromObstaclesAndBorder

```
public void refineNeighborsFromObstaclesAndBorder(Pair
neighbor,int xn, int yn){
if(obstacleList.contains(neighbor)){
neighbors3.add(neighbor);
}
if(border.contains(neighbor)){
neighbors3.add(neighbor);
}}
```

- deleteNeighborsFromObstaclesAndBorder

```
public void deleteNeighborsFromObstaclesAndBorder () {
neighbors1.removeAll(neighbors3);
}
```

- CalculateHValue

```
public int calculateHValue(int x, int y) {
return  (Math.abs(x − xt) + Math.abs(y − yt));
}
```

- CalculateGCurrent & CalculateGNeighbor

```
public int calculateGValue(int x, int y) {
return  (Math.abs(x − xr) + Math.abs(y − yr));
}
```

- CalculateFValue

```
public int calculateFValue(int x, int y) {
return  calculateHValue (x,y) + calculateGValue (x,y);
}
```

- CalculateCostValue

```
public int calculateCostValue(int xn, int yn,int xc,
 int yc) {
return calculateGValue(xc,yc) +(Math.abs(yc − yn) +
Math.abs(xc − xn));
}
```

• EvaluateNeighborsCase1 & EvaluateNeighborsCase2

```
public void evaluateNeighbors (Pair
 currentPair ,Pair
neighbor ,int xn, int yn){
int xc = currentPair.getX();
int yc = currentPair.getY();
int cost=calculateCostValue(xn, yn, xc, yc);
if(openF.containsKey(neighbor) ){
//return old G neighbor
int oldGn = openG.get(neighbor);
if(cost<oldGn){
openF.remove(neighbor);
openG.remove(neighbor);
}}
if(!closedList.contains(neighbor) &&
!openF.containsKey(neighbor) ){
int newGn= cost;
int newF;
newF =newGn+calculateHValue(xn, yn);
openF.put(neighbor,newF );
openG.put(neighbor, newGn);
parentChild.put(neighbor, currentPair);
}}
```

- FollowPath & MoveDown & MoveUp & MoveRight & MoveLeft & MoveUpRight & MoveUpLeft & MoveDownRight & MoveDownLeft

```
public void followPath(){
String S1="D>";
String S2="U>";
String S3="R>";
String S4="L>";
String S5="UR>";
String S6="UL>";
String S7="DR>";
String S8="DL>";
ROBOTMOVE.add(S1) ;
ROBOTMOVE.add(S2) ;
ROBOTMOVE.add(S3) ;
ROBOTMOVE.add(S4) ;
ROBOTMOVE.add(S6) ;
ROBOTMOVE.add(S7) ;
ROBOTMOVE.add(S8) ;
```

```
String c= "";
Pair currentPosition = path.pop();
xcp = currentPosition.getX();
ycp = currentPosition.getY();
System.out.println("currentPosition :"+ xcp +","+ ycp );
while(path.size() != 0) {
Pair nextPosition = path.pop() ;
xnp = nextPosition.getX();
ynp = nextPosition.getY();
System.out.println("nextPosition :"+ xnp+","+ ynp );
if(currentPosition.getX() == nextPosition.getX() &&
currentPosition.getY()+1 == nextPosition.getY()){
System.out.println("down");
currentPosition = nextPosition;
c = c.concat(S1);
}
if(currentPosition.getX() == nextPosition.getX() &&
currentPosition.getY()-1 == nextPosition.getY()){
System.out.println("up");
currentPosition = nextPosition;
c = c.concat(S2);
}
if(currentPosition.getX()+1 == nextPosition.getX()
&& currentPosition.getY() == nextPosition.getY()){
System.out.println("right");
currentPosition = nextPosition;
c = c.concat(S3);
}
if(currentPosition.getX()-1 ==  nextPosition.getX()
&& currentPosition.getY() == nextPosition.getY()){
System.out.println("left");
currentPosition = nextPosition;
c = c.concat(S4);
}
if(currentPosition.getX()+1 ==  nextPosition.getX()
&& currentPosition.getY()-1 == nextPosition.getY()){
System.out.println("upRight");
currentPosition = nextPosition;
c = c.concat(S5);
}
if(currentPosition.getX()-1 ==  nextPosition.getX()
&& currentPosition.getY()-1 == nextPosition.getY()){
System.out.println("upLeft");
currentPosition = nextPosition;
c = c.concat(S6);
}
```

```
if(currentPosition.getX()+1 ==  nextPosition.getX()
&& currentPosition.getY()+1 == nextPosition.getY()){
System.out.println("downRight");
currentPosition = nextPosition;
c = c.concat(S7);
}
if(currentPosition.getX()−1 ==  nextPosition.getX()
&& currentPosition.getY()+1 == nextPosition.getY()){
System.out.println("downLeft");
currentPosition = nextPosition;
c = c.concat(S8);
}}
System.out.println("RobotMove:" + c );
}
```

- The full implementation of our version of the A* algorithm

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Hashtable;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Stack;
import javax.swing.JLabel;
/*
*       Copyright 2015, Eman Rabiah. All rights reserved.
*/
public class AStarAlgo  {
private ArrayList<Pair> closedList;
private ArrayList<Pair> neighbors1;
private ArrayList<Pair> neighbors2;
private ArrayList<Pair> neighbors3;
Stack <Pair> path = new Stack <Pair >();
private ArrayList<Pair> obstacleList;
private ArrayList<String> ROBOTMOVE;
private ArrayList<String> RESPONSE;
private Pair[][] map;
int counter1;
int counter2;
int counter3;
int counter4;
public Pair robot;
public Pair target;
public Pair pathPair;
private int xr;
private int yr;
private int xt;
private int yt;
private int xb;
private int yb;
private int xc;
private int yc;
private int xcp;
private int ycp;
private int xnp;
private int ynp;
public  int maxX = 10;
public  int maxY = 10;
private int undefined = −1;
String R1 ="Free The Pair First";
String R2 ="No Path Found";
String R3 ="you Cant Free Border";
String R4 ="It's Border Position";
String R5 ="A Path Is Found";
public Pair bestPair;
Hashtable<Pair, Pair> parentChild = new
Hashtable<Pair, Pair>();
Hashtable<Pair, Integer> openG = new
Hashtable<Pair, Integer>();
```

```
Hashtable<Pair ,Integer> openF = new
Hashtable<Pair ,Integer >();
public AStarAlgo(Pair[][] map,int xr, int yr,
int xt, int yt, ArrayList<Pair> obstacleList) {
this.map = map;
this.xr = xr;
this.yr = yr;
this.xt = xt;
this.yt = yt;
xb=undefined;
xb=undefined;
xc=undefined;
xc=undefined;
xcp = undefined;
ycp = undefined;
xnp = undefined;
ynp = undefined;
this.obstacleList = obstacleList;
robot = map[xr][yr];
target = map[xt][yt];
closedList = new ArrayList<Pair >();
neighbors1 = new ArrayList<Pair >();
neighbors2 = new ArrayList<Pair >();
```

```
neighbors3 = new ArrayList<Pair>();
RESPONSE = new ArrayList<String>();
ROBOTMOVE = new ArrayList<String>();
RESPONSE.add(R1);
RESPONSE.add(R2);
RESPONSE.add(R3);
RESPONSE.add(R4);
RESPONSE.add(R5);
}
public void SearchPairs() {
openF.put(robot, calculateFValue(xr, yr));
openG.put(robot, calculateGValue(xr, yr));
while(openF.size() != 0 ) {
bestPair = EvaluateBestPair();
xb = bestPair.getX();
yb = bestPair.getY();
if (bestPair == target){
constructPath();
break;
}
if (bestPair != target){
Pair currentPair = bestPair;
closedList.add(bestPair);
```

```
openF.remove(bestPair);
openG.remove(bestPair);
xc = currentPair.getX();
yc = currentPair.getY();
neighbors1.clear();
neighbors2.clear();
neighbors3.clear();
counter1=0;
counter2=0;
counter3=0;
counter4=0;
// add neighbors of the current Pair
Pair n1 = map [xc+1][yc];
Pair n2 = map [xc-1][yc];
Pair n3 = map [xc][yc+1];
Pair n4 = map [xc][yc-1];
Pair n5 = map [xc+1][yc+1];
Pair n6 = map [xc-1][yc+1];
Pair n7 = map [xc+1][yc-1];
Pair n8 = map [xc-1][yc-1];
neighbors1.add(n1);
neighbors1.add(n2);
neighbors1.add(n3);
```

```
neighbors1.add(n4);
neighbors1.add(n5);
neighbors1.add(n6);
neighbors1.add(n7);
neighbors1.add(n8);
for(int i=0 ; i<neighbors1.size() ; i++) {
Pair neighbor = neighbors1.get(i);
refineDiagonalNeighborsCase1(neighbor,xc,yc);
refineDiagonalNeighborsCase2(neighbor,xc,yc);
refineDiagonalNeighborsCase3(neighbor,xc,yc);
refineDiagonalNeighborsCase4(neighbor,xc,yc);
refineDiagonalNeighborsCase5(neighbor,xc,yc);
refineDiagonalNeighborsCase6(neighbor,xc,yc);
refineDiagonalNeighborsCase7(neighbor,xc,yc);
refineDiagonalNeighborsCase8(neighbor,xc,yc);
}
deleteDiagonalNeighbors();
for(int i=0 ; i<neighbors1.size() ; i++) {
Pair neighbor = neighbors1.get(i);
int xn = undefined;
int yn = undefined;
xn = neighbor.getX();
yn = neighbor.getY();
refineNeighborsFromObstaclesAndBorder(neighbor ,xn ,yn );
}
deleteNeighborsFromObstaclesAndBorder ();
for(int i=0 ; i<neighbors1 . size () ; i++) {
Pair neighbor = neighbors1 . get(i );
int xn = undefined ;
int yn = undefined ;
xn = neighbor . getX ();
yn =neighbor . getY ();
evaluateNeighbors(currentPair , neighbor ,xn ,yn );
}}}
if (openF . size () == 0 && bestPair != target){
        System . out . println (R2);
}}
private Pair EvaluateBestPair () {
return Collections .min(openF . entrySet () , new
 Comparator<Map. Entry<Pair ,Integer >>() {
@Override
public int compare(Entry<Pair , Integer> o1 ,
Entry<Pair , Integer> o2) {
return o1 . getValue (). intValue () − o2 . getValue (). intValue ();
}})
. getKey ();
```

```
}
public void refineDiagonalNeighborsCase1 (Pair
 neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc && yn == yc+1 &&
 obstacleList.contains(neighbor)){
counter1++;
}
if (xn == xc+1 && yn == yc &&
 obstacleList.contains(neighbor)){
counter1++;
}
if(counter1==2)  {
Pair n= neighbors1.get(4);
neighbors2.add(n); }}
public void refineDiagonalNeighborsCase2 (Pair
neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc && yn == yc+1 &&
 obstacleList.contains(neighbor)){
counter2++;
```

```
}
if (xn == xc-1 && yn == yc &&
 obstacleList.contains(neighbor)){
counter2++;
}
if(counter2==2)  {
Pair n= neighbors1.get(5);
neighbors2.add(n);
}}
public void refineDiagonalNeighborsCase3 (Pair
 neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc && yn == yc-1 &&
 obstacleList.contains(neighbor)){
counter3++;
}
if (xn == xc+1 && yn == yc &&
 obstacleList.contains(neighbor)){
counter3++;
}
if(counter3==2)  {
Pair n= neighbors1.get(6);
```

```
neighbors2.add(n);
}}
public void refineDiagonalNeighborsCase4 (Pair
 neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc && yn == yc-1 &&
 obstacleList.contains(neighbor)){
counter4++;
}
if (xn == xc-1 && yn == yc &&
 obstacleList.contains(neighbor)){
counter4++;
}
if(counter4==2)  {
Pair n= neighbors1.get(7);
neighbors2.add(n);
}}
public void refineDiagonalNeighborsCase5 (Pair
 neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc && yn == yc+1 &&
 obstacleList.contains(neighbor)){
Pair n1= neighbors1.get(4);
Pair n2= neighbors1.get(5);
if(! neighbors2.contains(n1) ){
neighbors2.add(n1);
}
if(! neighbors2.contains(n2) ){
neighbors2.add(n2);
}}}
public void refineDiagonalNeighborsCase6 (Pair
 neighbor,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc && yn == yc-1 &&
 obstacleList.contains(neighbor)){
Pair n1= neighbors1.get(6);
Pair n2= neighbors1.get(7);
if(! neighbors2.contains(n1) ){
neighbors2.add(n1);
}
if(! neighbors2.contains(n2) ){
neighbors2.add(n2);
}}}
```

```
public void refineDiagonalNeighborsCase7 (Pair
 neighbor ,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc+1 && yn == yc &&
 obstacleList.contains(neighbor)){
Pair n1= neighbors1.get(4);
Pair n2= neighbors1.get(6);
if(! neighbors2.contains(n1) ){
neighbors2.add(n1);
}
if(! neighbors2.contains(n2) ){
neighbors2.add(n2);
}}}
public void refineDiagonalNeighborsCase8 (Pair
 neighbor ,int xc, int yc){
int xn = neighbor.getX();
int yn = neighbor.getY();
if (xn == xc−1 && yn == yc &&
 obstacleList.contains(neighbor)){
Pair n1= neighbors1.get(5);
Pair n2= neighbors1.get(7);
if(! neighbors2.contains(n1) ){
```

```
neighbors2.add(n1);
}
if(! neighbors2.contains(n2) ){
neighbors2.add(n2);
}}}
public void deleteDiagonalNeighbors () {
neighbors1.removeAll(neighbors2);
}
public void refineNeighborsFromObstaclesAndBorder(Pair
neighbor,int xn, int yn){
if(obstacleList.contains(neighbor)){
neighbors3.add(neighbor);
}
if(border.contains(neighbor)){
neighbors3.add(neighbor);
}}
public void deleteNeighborsFromObstaclesAndBorder () {
neighbors1.removeAll(neighbors3);
}
public void evaluateNeighbors (Pair currentPair,Pair
 neighbor,int xn, int yn){
int xc = currentPair.getX();
int yc = currentPair.getY();
int cost=calculateCostValue(xn, yn, xc, yc);
if(openF.containsKey(neighbor) ){
//return old G neighbor
int oldGn = openG.get(neighbor);
System.out.println("123" + oldGn);
if(cost<oldGn){
System.out.println(" The new path is better ");
openF.remove(neighbor);
openG.remove(neighbor);
}}
if(!closedList.contains(neighbor) &&
 !openF.containsKey(neighbor) ){
//int g= calculateGValue(xn, yn);
int newGn= cost;
int newF;
newF =newGn+calculateHValue(xn, yn);
openF.put(neighbor ,newF );
openG.put(neighbor , newGn);
//  a hash table to store parents
parentChild.put(neighbor , currentPair);
}}
public void constructPath(){
if (bestPair == target){
```

```
Pair pathPair= map[xt][yt];
int xpath = pathPair.getX();
int ypath = pathPair.getY();
System.out.println("The Path is :"+ xpath +","+ ypath );
path.push(pathPair);
while(pathPair != map[xr][yr] && pathPair!=null ) {
pathPair = parentChild.get(pathPair);
path.push(pathPair);
}
followPath();
}}
public void followPath(){
String S1="D>";
String S2="U>";
String S3="R>";
String S4="L>";
String S5="UR>";
String S6="UL>";
String S7="DR>";
String S8="DL>";
ROBOTMOVE.add(S1) ;
ROBOTMOVE.add(S2) ;
ROBOTMOVE.add(S3) ;
ROBOTMOVE.add(S4) ;
ROBOTMOVE.add(S6) ;
ROBOTMOVE.add(S7) ;
ROBOTMOVE.add(S8) ;
String c= "";
Pair currentPosition = path.pop();
xcp = currentPosition.getX();
ycp = currentPosition.getY();
System.out.println("currentPosition :"+ xcp +","+ ycp );
while(path.size() != 0) {
Pair nextPosition = path.pop() ;
xnp = nextPosition.getX();
ynp = nextPosition.getY();
System.out.println("nextPosition :"+  xnp+","+  ynp );
if(currentPosition.getX() ==  nextPosition.getX() &&
currentPosition.getY()+1 == nextPosition.getY()){
System.out.println("down");
currentPosition = nextPosition;
c = c.concat(S1);
}
if(currentPosition.getX() ==  nextPosition.getX() &&
 currentPosition.getY()−1 == nextPosition.getY()){
System.out.println("up");
```

```
currentPosition = nextPosition;
c = c.concat(S2);
}
if(currentPosition.getX()+1 == nextPosition.getX() &&
 currentPosition.getY() == nextPosition.getY()){
System.out.println("right");
currentPosition = nextPosition;
c = c.concat(S3);
}
if(currentPosition.getX()-1 == nextPosition.getX() &&
 currentPosition.getY() == nextPosition.getY()){
System.out.println("left" );
currentPosition = nextPosition;
c = c.concat(S4);
}
if(currentPosition.getX()+1 == nextPosition.getX() &&
 currentPosition.getY()-1 == nextPosition.getY()){
System.out.println("upRight" );
currentPosition = nextPosition;
c = c.concat(S5);
}
if(currentPosition.getX()-1 == nextPosition.getX() &&
 currentPosition.getY()-1 == nextPosition.getY()){
```

```
System.out.println("upLeft" );
currentPosition = nextPosition;
c = c.concat(S6);
}
if(currentPosition.getX()+1 ==  nextPosition.getX() &&
 currentPosition.getY()+1 == nextPosition.getY()){
System.out.println("downRight" );
currentPosition = nextPosition;
c = c.concat(S7);
}
if(currentPosition.getX()-1 ==  nextPosition.getX() &&
 currentPosition.getY()+1 == nextPosition.getY()){
System.out.println("downLeft" );
currentPosition = nextPosition;
c = c.concat(S8);
}}
System.out.println("RobotMove:" + c );
}
public int calculateGValue(int x, int y) {
return  (Math.abs(x - xr) +
Math.abs(y - yr));
}
public int calculateHValue(int x, int y) {
```

```
return  (Math.abs(x - xt) +
Math.abs(y - yt));
}
public int calculateFValue(int x, int y) {
return  calculateHValue (x,y) +
 calculateGValue (x,y);
}
public int calculateCostValue(int xn, int yn,
int xc, int yc) {
return calculateGValue(xc,yc)
+(Math.abs(yc - yn) + Math.abs(xc - xn));
}}
```