BACHELOR'S THESIS

Degree in Mathematics

Title: Distributed locomotion of 2D lattice-based modular robotic systems

Author: Xavier Salvador Nomen

Advisor: Vera Sacristán Adinolfi

Department: Mathematics

Academic year: 2018-2019

Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Degree in Mathematics
Bachelor’s Degree Thesis

Distributed locomotion of 2D lattice-based modular robotic systems

Xavier Salvador Nomen

Supervised by Vera Sacristán Adinolfi

June, 2019

Acknowledgments

I wish to thank my supervisor Vera Sacristán for all the help and time spent during the project.
I also thank my family and partner for all the endless support.

Abstract

In this thesis we have solved the two-dimensional problem of locomotion of modular robots on square lattices in the presence of obstacles of arbitrary forms, for the class of modular robots that follow the model of movement known as the *sliding-cube* model. The algorithm we propose is deterministic and distributed. All robot modules apply the same set of local rules. That is, they make their own decisions based on local information from their immediate surroundings. The result is a scalable algorithm that solves the problem efficiently. Our results close a problem that was open for more than ten years.

Keywords

Computer science, robotics and automation, locomotion, modular robots, distributed algorithms, geometric rules.

Contents

1. Introduction 1

2. Model 5

3. Locomotion without obstacles 8

4. Locomotion over pyramids 11

5. Preventing collisions and deadlocks at bottlenecks 19

6. Correctness 34
   - 6.1. Locomotion without obstacles ………………………………………………….. 34
   - 6.2. Locomotion over pyramids ………………………………………………………... 37
   - 6.3. Preventing collisions and deadlocks at bottlenecks ……………………………….. 41

7. Complexity 42

8. Simulation 44
   - 8.1. AgentSystem ………………………………………………………………………. 44
   - 8.2. Orientation and other difficulties ………………………………………………….. 44
   - 8.3. Test Bank ………………………………………………………………………….. 44
   - 8.4. Results ……………………………………………………………………………... 49
   - 8.5. Downloading the rules …………………………………………………………….. 56

9. Conclusions 57

References 59

1. Introduction

Modular robots are robots designed with parts that can be reconfigured to assume different shapes and functions. In many cases, such robots are able to reconfigure their own shape autonomously.

They are usually composed of multiple building blocks of a relatively small repertoire, with uniform docking interfaces that allow transfer of mechanical forces and moments, electrical power, and communication throughout the robot. Figure 1 illustrates the concept of modular robotics.

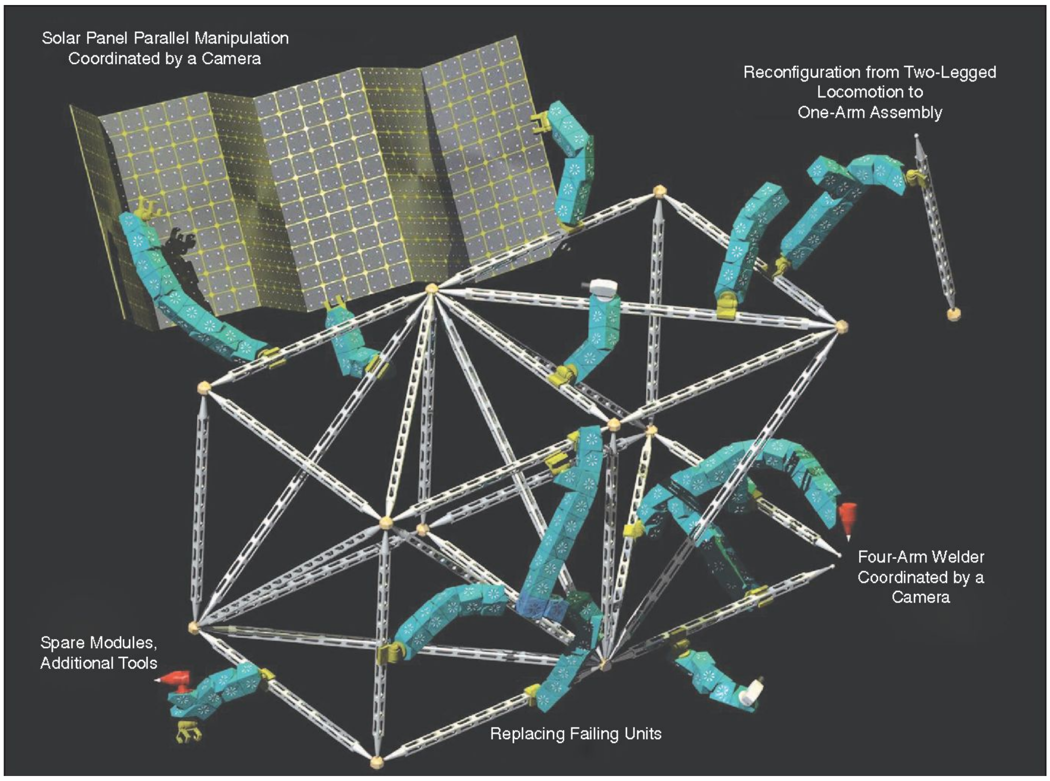
Solar Panel Parallel Manipulation Coordinated by a Camera
Reconfiguration from Two-Legged Locomotion to One-Arm Assembly
Four-Arm Welder Coordinated by a Camera
Spare Modules, Additional Tools
Replacing Failing Units

Figure 1: *Autonomous modular robotics working in space.* Source: [1].

Over the last three decades, the field of modular robotics has advanced from proof-of-concept systems to elaborate physical implementations and simulations. However, they have not been fully developed.

We find this type of robots interesting because they have some advantages that others do not have. Mainly, we can describe three key motivations for designing modular robotic systems:

- *Versatility*: Modular robotic systems are potentially more adaptive than conventional systems. The ability to reconfigure allows a robot or a group of robots to disassemble and reassemble to form new morphologies that are better suited for new tasks, such as changing from a legged robot to a snake robot and then to a rolling robot.

- *Robustness*: Since robot parts are interchangeable, modular robots can also replace faulty parts autonomously, leading to self-repair.

- *Low Cost*: Modular robotic systems can potentially lower overall robot cost by making many copies of one type of modules so economies of scale and mass production come into play. Also, a range of complex machines can be made from one set of modules, saving costs through reuse and generality of the system.

Our project focuses on the study of modular robotic systems on square lattices. A practical 3D example is the Miche system, made in 2006, shown in Figure 2.

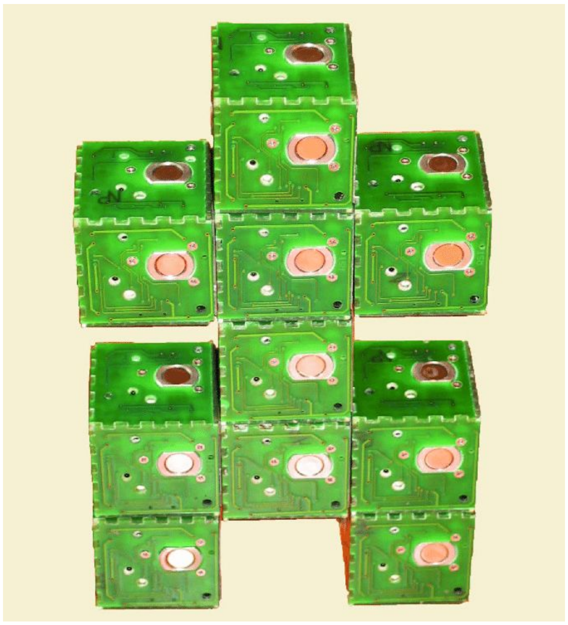

Figure 2: *Miche modular robotic system.* Source [1].

It is a modular lattice system capable of arbitrary shape formation. This system achieves self-assembly by disassembly and has demonstrated robust operation over hundreds of experiments. Each module is an autonomous robot cube capable of connecting to and communicating with its immediate neighbors. The connection mechanism is provided by switchable magnets.

The modules use face-to-face communication implemented with an infrared system to detect the presence of neighbors. When assembled into a structure, the modules form a system that can be virtually sculpted using a computer interface and a distributed process. The group of modules collectively decides who is and is not on the final shape using algorithms that minimize the information transmission and storage.

Finally, the modules not in the structure let go and fall off under the control of an external force, in this case gravity. All the algorithms controlling these processes are distributed and are very efficient in their space and communication consumption. In [1] many of the other instantiated modular robot systems are presented.

The problem now is how robots move in space. This problem is called *robot locomotion*. This is the collective name for the various methods that modular robots can use to transport themselves from a place to another.

This problem has been under study for years and continues to be so. Locomotion has many different solutions depending on the type of modular robots in question. For example, for *chain* types, locomotion modes as diverse as *rolling*, *flowing like a snake*, and various *gaits* have been used, as described in [2].

In the case of modular robots on lattices, which is the object of this project, it is important to find scalable locomotion methods, since the idea of this type of robot is that in the future they will be miniaturized and will move in large quantities.

That is where use distributed algorithms come into play. They are algorithms designed to run on computer hardware constructed from interconnected processors, so that some part of the algorithm is run on one processor, other part in another processor, and so forth. Typically data is partitioned in advance so that each processor doesn't need to wait for the output of another processor.

There is already a history of distributed algorithms for locomotion of modular robots on lattices. Probably the first proposal in this sense is [3]. In this article, however, the distribution model used simulates the asynchrony of the movements by making them sequential and, therefore, does not take into account the possible collisions between moving modules, which may take place in reality. In addition, the authors study the locomotion of a particular shape - a rectangle - in the presence of obstacles of very limited forms.

Subsequently, the same authors extended the types of obstacles they dealt with in [4] but not to the point of admitting any type of obstacle. In any case, their execution model for the rules is still sequential.

Probably, the best-known work about distributed locomotion algorithms is [5]. In this paper, the authors propose a non-deterministic but quite effective locomotion strategy for large modular robots.

Our project focuses on distributed locomotion of 2D square lattice-based modular robotic systems. In Chapter 2 we describe the model proposed to define the locomotion and the distributed algorithm.

Then, in chapters 3, 4 and 5, we present, in an incremental manner, sets of rules to solve the locomotion problem: in the absence of obstacles, over pyramids and preventing collisions and deadlocks at bottlenecks, respectively. In Chapter 6 proofs are given of the correctness of the proposed rules. Chapter 7 offers an analysis of the complexity of the algorithm obtained.

In Chapter 8 we describe the simulator with which we have implemented our algorithm and discuss our experimental results.

Finally, in Chapter 9, we present the project conclusions.

The report concludes with the enumeration of the references used along this work.

2. Model

A module is any robotic unit located in a *2*-dimensional cell in a square lattice. We represent modules by squares occupying one grid cell, although their actual shape need not be a square. A cell can be empty, or occupied by an obstacle or a module.

A modular robot is a connected configuration of homogeneous modules like the ones described above. By "connected" we mean that the edge-adjacency graph of the robot configuration is connected. Figure 3 shows the definition of connected configuration.

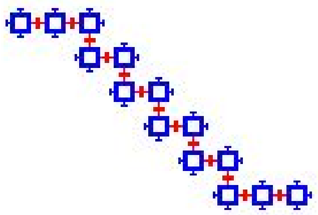
Figure 3: *A connected configuration and its adjacency graph.*

We call *first neighborhood* of a module the four cell positions edge-adjacent to it. We describe such positions as top $(0,1)$, right $(1,0)$, bottom $(0,-1)$ and left $(-1,0)$. Each module can detect whether or not any of its neighboring positions are empty, occupied by another module or an obstacle.

Modules also have a state - a short text string -.

Actions that modules can do are attach, detach, change position or change their states. Attaching and detaching is done with reference to a neighboring module, and does not involve changing position.

There are two ways a module can change position: by sliding or by a convex transition.

A module *m* can slide towards its right if positions $(0,-1)$ and $(1,-1)$ are occupied by other modules and position $(1,0)$ is empty. Sliding means that *m* moves from its current position $(0,0)$ to position $(1,0)$, as we can see in Figure 4. Sliding up, down or to the left are defined analogously.

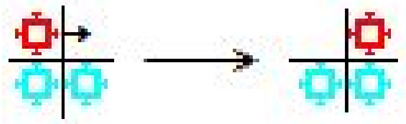
Figure 4: *Definition of sliding to the right.*

A module m can perform a convex transition from position $(0,0)$ to position $(1,-1)$ provided that position $(0,-1)$ is occupied by another module and positions $(1,0)$ and $(1,-1)$ are empty. We illustrate this move in Figure 5. Analogously, the module can perform a convex transition to positions $(1,1)$, $(-1,-1)$ and $(-1,1)$.

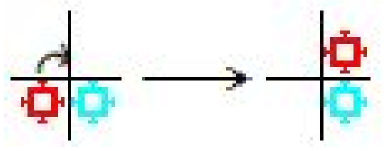

Figure 5: *Definition of convex transition to the lower right position.*

States we consider in our algorithm are "static", "cork", "opener" and "active". At the beginning, all modules are static. Only active modules can move. States cork and opener are two specific kinds of static modules that adopt a specific behavior intended to unlock locked situations during the locomotion. Chapter 5 describes these states in detail.

Notice that the two moves - slide and convex transition - can be described in terms of the relative position of the static module with reference to the active module moving relative to it. If we always assign the static module $(0,-1)$ as relative coordinates with respect to the active module, then the active module will always slide to its right $(1,0)$ or make a convex transition to the position $(1,-1)$.

The goal of our work is to solve the following problem for 2-dimensional lattice-based modular robotic systems: given a initial connected configuration with n modules - the strip configuration -, and a set of obstacles, we want to pass all obstacles from left to right. Along the locomotion, the robotic system must stay connected at all times, and no collisions can happen.

Within this framework, our algorithm is completely distributed. It consists of a set of rules, each one having a priority, a precondition, and an action or postcondition.

Priorities, represented as small integers, are used by the modules to decide which of possibly several rules that apply to their situation to execute.

A precondition is any constant-size boolean combination of the following: compare priorities and check neighboring positions (whether they are empty, obstacles, static or active modules).

A postcondition can be a movement, a state change, or both.

Rules are identical for all modules, and are simultaneously executed by all of them. Each module applies the same set of rules. Each module can only make one movement or one state change per round. Only modules with “cork” or “opener” state can be activated and move in the same round.

The algorithm runs synchronously. This means that all modules apply one rule per round at the same time. Moreover, at most one rule is applied per module and round.

Our locomotion strategy consists in producing locomotion in a caterpillar way, i.e., a strip of modules will advance estwards by moving the leftmost module and sending it to the rightmost position of the strip. This kind of locomotion is shown in Figure 6.

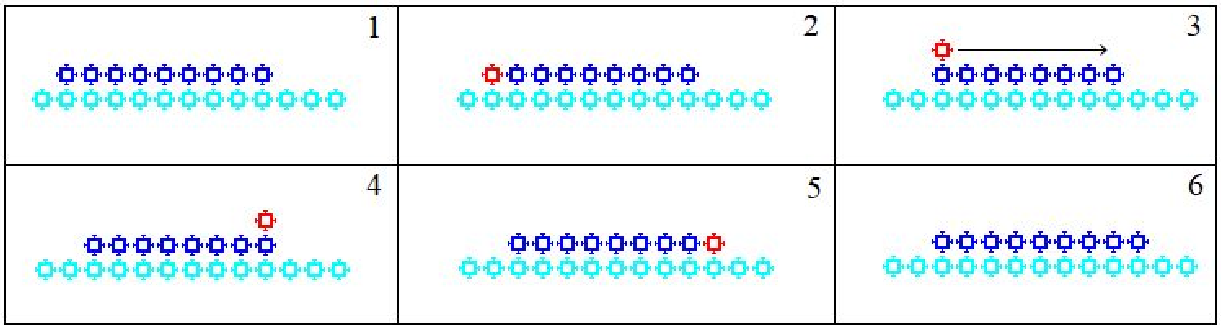

Figure 6: *Locomotion in a caterpillar way.*

We will present the algorithm incrementally. First, we will discuss the rules - ie, the algorithm - that produce this locomotion in the absence of obstacles in Chapter 3. Then, we will progressively complicate the difficulty of the obstacles to overcome and, with it, the set of rules to use in chapters 4 and 5.

3. Locomotion without obstacles

The first step to create a distributed algorithm to move the robot from left to right is to produce locomotion in the absence of obstacles. The surface supporting the modules is totally horizontally, and the modules are on it. In Figure 7 we can see an example of a robot with twenty modules.

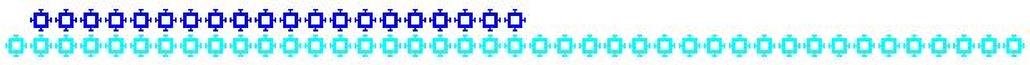

Figure 7: *Initial robot over a horizontal surface.*

The goal is to move the leftmost module over the rest of the modules until it reaches the first empty position next to the rightmost module.

Suppose that the horizontal surface is line $y = -1$. So, our robot is located on line $y = 0$. This means that the robot modules are located at $(x,0)$, where x is between 0 and n, the number of modules.

We distinguish two cases:

1. If the algorithm was sequential, after m steps we would like the robot to have advanced exactly one position to the right. Therefore, the modules would be at positions $(x,0)$ such that x is between 1 and $n + 1$.

2. Since the algorithm runs in parallel, we must take into account the collisions and be sure not disconnect the robot at any time. However, from the point of view of a module, the actions to perform are the same.

The algorithm for this locomotion consists of a total of five rules. They are the following:

```
Rule 1: Activation
Priority: 10
Preconditions:
   -  the state of the current module is static
   -  the top position (0,1) is empty
   -  the right position (1,0) is occupied by a static module
   -  the left position (-1,0) is empty
   -  the upper right position (1,1) is empty
Postcondition: change state to active.
```

```
Rule 2: Initial convex transition NE
Priority: 10
Preconditions:
   -  the state of the current module is active
   -  the top position (0,1) is empty
   -  the right position (1,0) is occupied by a static module
   -  the upper right position (1,1) is empty
Postcondition: convex transition to the upper right position (1,1).
```

```
Rule 3: Slide E
Priority: 10
Preconditions:
   -  the state of the current module is active
   -  the right position (1,0) is empty
   -  the bottom position (0,-1) is occupied by a static module
   -  the lower right position (1,-1) is occupied by a static module
Postcondition: slide to the right (1,0).
```

```
Rule 4: Final convex transition SE
Priority: 10
Preconditions:
   -  the state of the current module is active
   -  the right position (1,0) is empty
   -  the bottom position (0,-1) is occupied by a static module
   -  the lower right position (1,-1) is empty
Postcondition: convex transition to the lower right position (1,-1).
```

To deactivate a module, we define a rule with lower priority than all previous ones. Thus, if a module can not apply another rule, it applies this last rule and deactivates.

```
Rule 5: Deactivation
Priority: 1
Preconditions:
   -  the state of the current module is active.
Postcondition: change state to static.
```

In Chapter 6 we prove the correctness of this algorithm.

All the rules that we have defined and will define in the following chapters have their counterparts by rotations of *90*º. This means that if we rotate *90*º counterclockwise about the origin (*0*,*0*), the rules will produce an analogous result, now with respect to a vertical surface. Indeed, locally the environment of each module is the same, although from our point of view we see it differently. The modules move upward along a vertical surface, following the right-hand rule. Now, the right of a module changes from east to north. If we again rotate counterclockwise *90*º, the surface would be a ceiling, and the modules would go west. Finally, rotate again *90*° counterclockwise, the robot will move to the south along a vertical surface located to its left.

We can therefore think of our rules as performing the advance of a modules strip following the right-hand rule over any horizontal or vertical surface without obstacles.

4. Locomotion over pyramids

After locomotion without obstacles our next step is to move the strip configuration over pyramids.

A pyramid is an obstacle that is an orthogonal polygon whose boundary is monotonously increasing starting at *y* = *0* and, once its maximum height has been reach, it becomes monotonous decreasing until reaching back the line *y* = *0*, as we can illustrate in Figure 8.

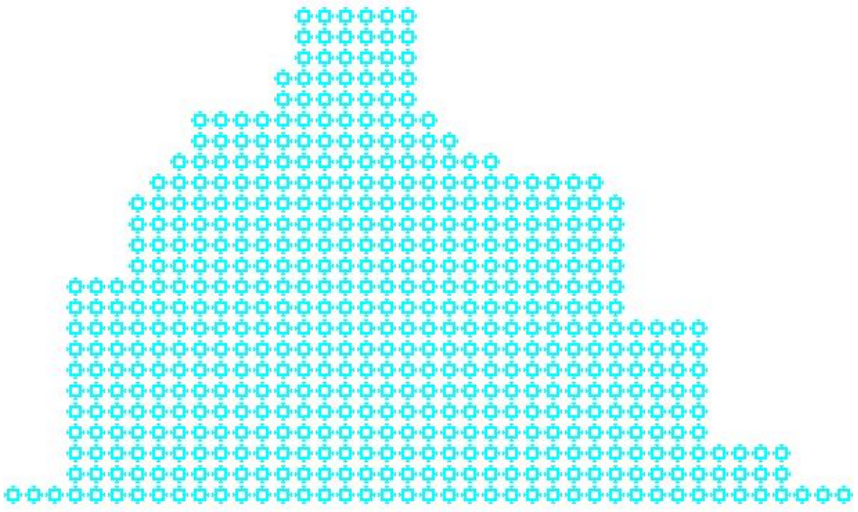
Figure 8: *A pyramid.*

We want to extend our set of rules for locomotion so that they work in the presence of pyramidal obstacles. The goal is to make the robot crowl over the obstacle from left to right keeping the robotic system connected at all times along the locomotion, and so that no collisions happen.

From the previous chapter, our rules can deal with horizontal and vertical surfaces. Therefore, we only need to cope with their intersections, that is, to be able to turn corners. We can distinguish two kinds of such intersections: when the surface is increasing and when it is decreasing. In each case there are two different corners depending on whether they turn from horizontal to vertical or the other way around. Figure 9 shows the four types of intersections.

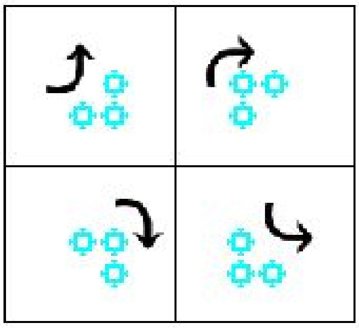

Figure 9: *Above, monotonously increasing concave and convex corners, respectively. Below, monotonously decreasing convex and concave corners, respectively.*

When the surface is monotonously increasing and the intersection is formed by a horizontal line followed by a vertical line, we have a concave corner. When the intersection has a vertical line first, then we have a convex corner.

When the surface is monotonously decreasing and the intersection is formed by a horizontal line followed by a vertical line, we have a convex corner. When the intersection has a vertical line first, then we have a concave corner.

We need the rules to make the robot turn such corners.

Since a pyramid has two kinds of vertical edges and one kind of horizontal edges, the robot will perform three kinds of slide motions: slide right, slide up and slide down, and four type of convex transition moves depending on the surface and on whether the robot is in the monotonously increasing part of the pyramid or in its monotonously decreasing chain. We can see all the slide moves illustrated in Figure 10 and all convex transitions in Figure 11.

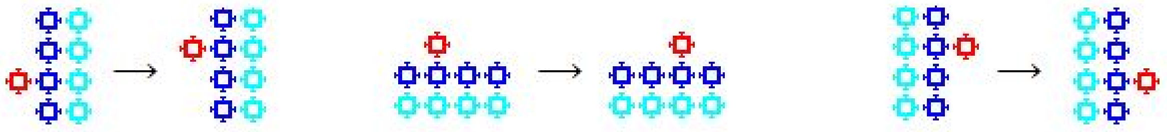

Figure 10: *From left to right: slide up, right and down.*

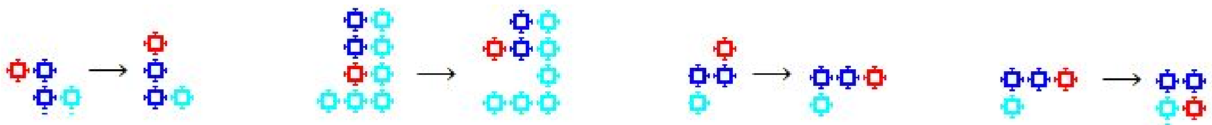

Figure 11: *From left to right: convex transition to upper right, upper left, lower right and lower left.*

The concave corners can be dealt with without the need of any specific move.

Thus, the new set of rules contains seven movement rules, five activation rules, and one deactivation rule. There are several activation rules because now the last module of the strip can be in different positions, and its environment may be different from that without obstacles. The increase in the movement rules is due to the different orientations.

Therefore, the algorithm for this locomotion has a total of thirteen rules, that we describe next.

We have added two preconditions to *`Rule 1: Activation`* from Chapter 3. The bottom position precondition intended to deal with monotonously increasing concave corners. Also, we add the possibility that the left position be an obstacle, in order to deal with monotonically decreasing concave corners. All remaining activation rules in convex transitions are rotations of first one.

```
Rule 1: Activation convex transition NE
Priority: 10
Preconditions:
   -  the state of the current module is static
   -  the top position (0,1) is empty
   -  the right position (1,0) is occupied by a static module
   -  the bottom position (0,-1) is empty or is an obstacle
   -  the left position (-1,0) is empty or is an obstacle
   -  the upper right position (1,1) is empty
Postcondition: change state to active.
```

```
Rule 2: Activation convex transition SE
Priority: 10
Preconditions:
   -  the state of the current module is static
   -  the top position (0,1) is empty or is an obstacle
   -  the right position (1,0) is empty
   -  the bottom position (0,-1) is occupied by a static module
   -  the left position (-1,0) is empty or is an obstacle
   -  the lower right position (1,-1) is empty
Postcondition: change state to active.
```

```
Rule 3: Activation convex transition NW
Priority: 10
Preconditions:
   - the state of the current module is static
   - the top position (0,1) is occupied by a static module
   - the bottom position (0,-1) is empty or is an obstacle
   - the left position (-1,0) is empty
   - the upper left position (-1,1) is empty
Postcondition: change state to active.
```

In order for the robot to be able to leave concave corners, we also need activation rules in the sliding case.

```
Rule 4: Activation slide N
Priority: 10
Preconditions:
   - the state of the current module is static
   - the top position (0,1) is empty
   - the right position (1,0) is occupied by a static module
   - the bottom position (0,-1) is empty or is an obstacle
   - the left position (-1,0) is empty
   - the upper right position (1,1) is occupied by a static module
Postcondition: change state to active.
```

The second slide activation rule is a rotation of the first one. The same thing happens with all movement rules - convex transitions and slides.

```
Rule 5: Activation slide E
Priority: 10
Preconditions:
   - the state of the current module is static
   - the top position (0,1) is empty
   - the right position (1,0) is empty
   - the bottom position (0,-1) is occupied by a static module
   - the left position (-1,0) is empty or is an obstacle
   - the lower right position (1,-1) is occupied by a static module
Postcondition: change state to active.
```

The next rule has the same preconditions as `Rule 2: Initial convex transition NE` from Chapter 3. Also, we add three convex transition rules in order to deal with all intersections.

```
Rule 6: Convex transition NE
Priority: 10
Preconditions:
   - the state of the current module is active
   - the top position (0,1) is empty
   - the right position (1,0) is occupied by a static module
   - the upper right position (1,1) is empty
Postcondition: convex transition to the upper right position (1,1).
```

The next rule has the same preconditions as `Rule 4: Final convex transition SE` from Chapter 3.

```
Rule 7: Convex transition SE
Priority: 10
Preconditions:
   - the state of the current module is active
   - the right position (1,0) is empty
   - the bottom position (0,-1) is occupied by a static module
   - the lower right position (1,-1) is empty
Postcondition: convex transition to the lower right position (1,-1).
```

We need the following rule in order to leave monotonously decreasing convex corners.

```
Rule 8: Convex transition SW
Priority: 10
Preconditions:
   - the state of the current module is active
   - the bottom position (0,-1) is empty
   - the left position (-1,0) is occupied by a static module
   - the lower left position (-1,-1) is empty
Postcondition: convex transition to the lower left position (-1,-1).
```

We also need the next rule in order to leave monotonously increasing concave corners.

```
Rule 9: Convex transition NW
Priority: 10
Preconditions:
   - the state of the current module is active
   - the top position (0,1) is occupied by a static module
   - the left position (-1,0) is empty
   - the upper left position (-1,1) is empty
Postcondition: convex transition to the upper left position (-1,1).
```

The following rule is needed to go up on vertical surfaces. The following rules are rotations to be able to slide on horizontal surfaces and go down on vertical surfaces.

```
Rule 10: Slide N
Priority: 10
Preconditions:
   -  the state of the current module is active
   -  the top position (0,1) is empty
   -  the right position (1,0) is occupied by a static module
   -  the upper right position (1,1) is occupied by a static module
Postcondition: slide to the top (0,1).

Rule 11: Slide E
Priority: 10
Preconditions:
   -  the state of the current module is active
   -  the right position (1,0) is empty
   -  the bottom position (0,-1) is occupied by a static module
   -  the lower right position (1,-1) is occupied by a static module
Postcondition: slide to the right (1,0).

Rule 12: Slide S
Priority: 10
Preconditions:
   -  the state of the current module is active
   -  the bottom position (0,-1) is empty
   -  the left position (-1,0) is occupied by a static module
   -  the lower right position (1,-1) is occupied by a static module
Postcondition: slide to the bottom (0,-1).
```

As in Chapter 3, we add the deactivation rule with lower priority than all other rules.

```
Rule 13: Deactivation
Priority: 1
Preconditions:
   -  the state of the current module is active
Postcondition: change state to static.
```

In Chapter 6 we prove the correctness of this algorithm.

Applying rotations by multiples of *90º*, we can get the analogous surface and set of rules for each direction. We can therefore think of our rules as performing the advance of a strip robot following the right-hand rule over any pyramid in any direction.

All together, the moving rules 6, 7, 8, 9, 10, 11 and 12 can be synthetized in just two rules using the right-hand rule:

```
1 - Slide
Priority: 10
Preconditions:
   -  state is active
   -  right position is empty
   -  bottom position is occupied by a static module
   -  lower right position is occupied by a static module
Postcondition: slide right.

2 - Convex transition
Priority: 10
Preconditions:
   -  the state is active
   -  right position is empty
   -  bottom position is occupied by a static module
   -  lower right position is empty
Postcondition: convex transition to lower right position.
```

Furthermore, the activation rules 1, 2, 3, 4 and 5 can be synthetized in one:

```
3 - Activation
Priority: 10
Preconditions:
   -  state is static
   -  right position is empty
   -  bottom position is occupied by a static module
   -  left position is empty or is an obstacle
   -  ⎧ lower right position is occupied by a static module
      ⎪ and top position is empty
      ⎨
      ⎪     or
      ⎪
      ⎩ lower right position is empty and top position is
        empty or is an obstacle
Postcondition: change state to active.
```

Finally, the deactivation rule:

```
4 - Deactivation
Priority: 1
Preconditions:
   -  state is active
Postcondition: change state to static.
```

Therefore, we obtain four rules to the advance of a strip robot following the right-hand rule over any pyramid in any direction.

5. Preventing collisions and deadlocks at bottlenecks

In this chapter we want to describe how we have achieved our distributed algorithm to move the robot from left to right in a locomotion preventing collisions and deadlocks at bottlenecks in the presence of obstacles of arbitrary shape.

A bottleneck is a narrow section of the surface or a junction that prevents the robot modules from naturally flowing. At bottlenecks, modules accumulate and cannot follow the flow from left to right in a natural way. By "natural way" we mean the following: given n modules labelled from right to left, module $i + 1$ is activated before module i, and reaches the rightmost position before module i.

In our locomotion strategy all the obstacle is covered with static modules. Then, if the active modules that pass over them have a narrow corridor to move around, they may collide or form a deadlock.

In this chapter we describe the final set of rules that are able to deal with these two kinds of troubles.

To prevent collisions we apply new preconditions to detect potential collisions prior to a move (Figure 12 shows an example of how two active modules could collide). We also define criteria to decide which module is going to move and which is going to wait in order to prevent the collision.

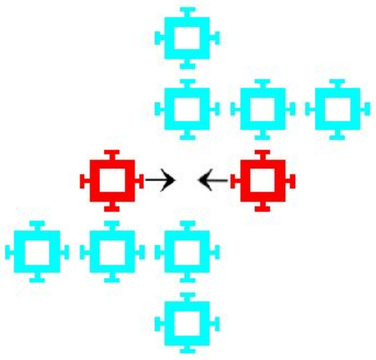

Figure 12: *Two active modules about to collide.*

A deadlock is a state in which each member of a group of modules is waiting for another member, including possibly itself, to take action. In this case, each module is waiting for another to activate or move. An example can be found in Figure 13.

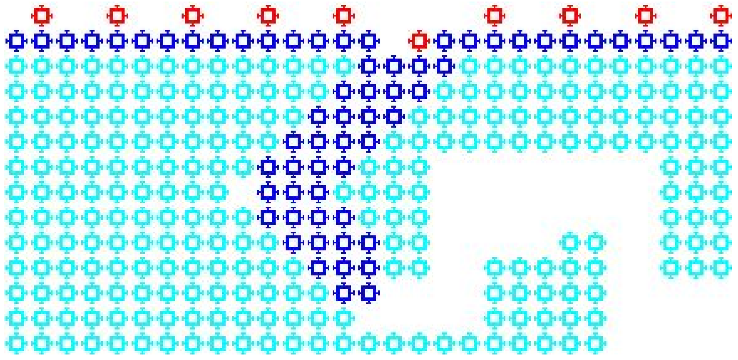
Figure 13: *Obstacle covered with static modules and active modules passing over them forming a deadlock.*

In our locomotion strategy, we have found three kinds of deadlocks:

- Dense deadlocks: a set of modules accumulates. The accumulated modules do not allow other modules to move, and there is no empty space between them. See Figure 14, left for an example.

- Cycles enclosing empty holes: the modules form a closed connected path surrounding empty cells. Figure 14, center shows an example.

- Cycles enclosing holes with active modules: the modules form a closed connected path surrounding empty cells, and there are active modules that move in that empty space. See Figure 14, right.

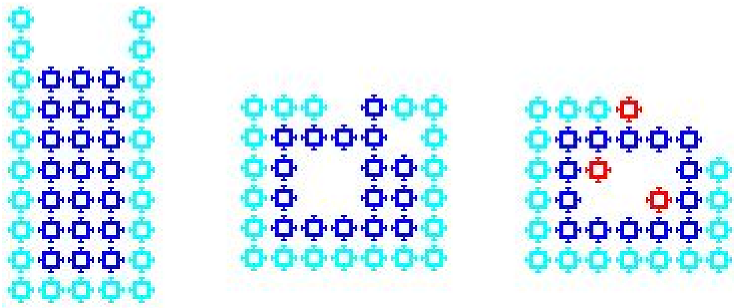
Figure 14: *Examples of a dense deadlock (left), a cycle enclosing an empty hole (center) and a cycle enclosing a hole containing active modules.*

In order to solve deadlocks we have defined two new states: "cork" and "opener". The cork state acts in dense deadlocks. The opener state solves both hole deadlocks.

Consequently, we have defined more rules. Some modules change their state from static to one of the new states. Then, if they can move and no collision is going to happen, they activate and move in the same round. Finally, they can deactivate. We show examples of modules in cork and opener states in Figures 15 and 16, respectively.

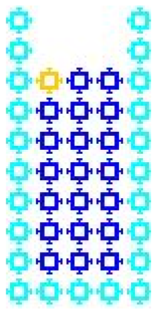
Figure 15: *Example of a module in cork state (yellow).*

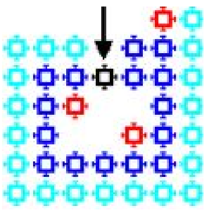
Figure 16: *Example of a module in opener state (black).*

Following our rules, some static or active modules are conditioned by the existence of these. Some modules activate when these new states appear in a neighboring module, and then follow it. Others simply slide or turn a conver corner over them.

Naturally, although all this could generate collisions, also in these new rules with new states it is necessary to introduce criteria to prevent them.

Collision prevention requires considering the second environment: that is, neighbors of neighbors. This is inevitable, as shown in Figure 13. In all movement rules we apply some common preconditions to prevent collisions.

- For a slide to the (relative) right, like the ones illustrated in Figure 17, we require that in the second right position there is no active module that is going to make a convex transition to the upper right position and this active module is inside a hole, and there is no active module in the first neighborhood. Also, we require that in the second right position there is no active module that is going to slide to the left.

- For a convex transition to the (relative) lower right position, like the one illustrated in Figure 18, we require that in position $(-2,-2)$ there is no active module that is going to make a convex transition to the upper left position and this active module is not inside a hole, and there is no active module in the first neighborhood.

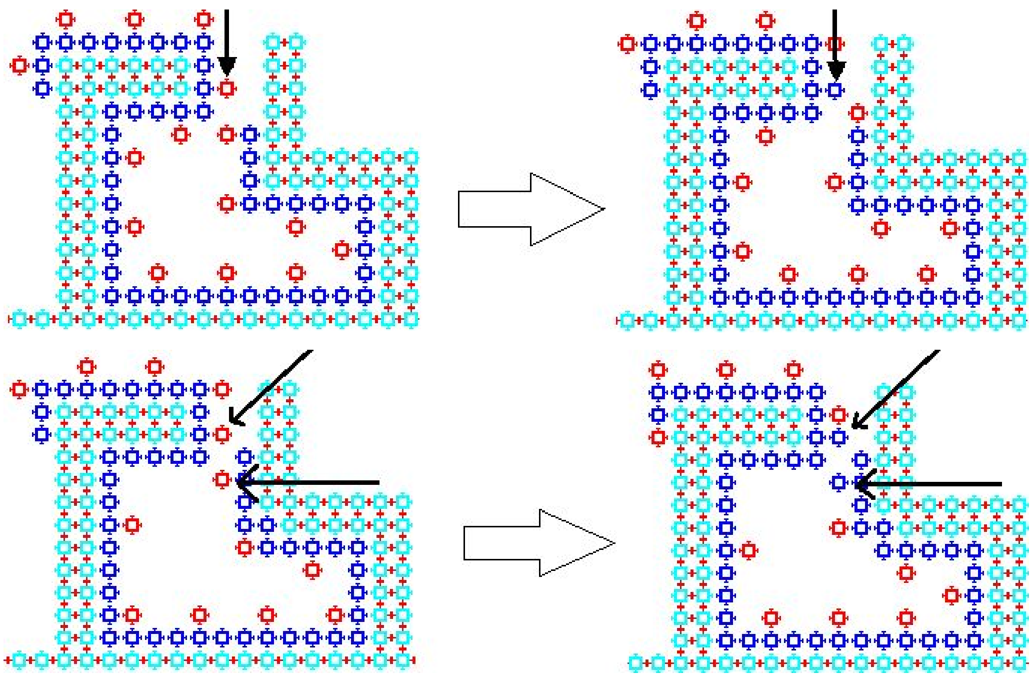
Figure 17: *Two examples of an active module that does not slide in order to prevent a collision.*

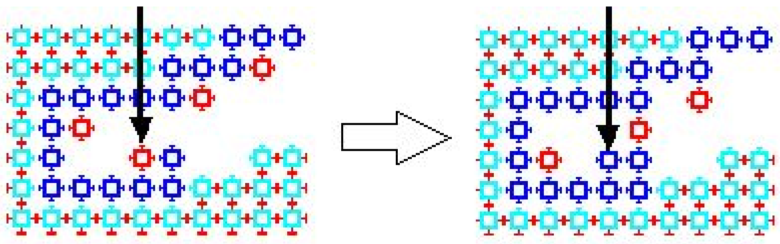
Figure 18: *Example of an active module that does not make a convex transition in order to prevent a collision.*

In addition, the second environment is also used to maintain order in the movement and choose who should activate and move and who should not according to the situation. The modules that have more empty cells around them are the first to activate, while those that are surrounded by obstacles or static modules tend not to move. This is determined by the priorities of the corresponding rule.

In more detail: there are the standard rules for the two initial states: static and active. The modules activate. Once active, they move. Finally, when they cannot move, they deactivate again.

These standard rules are the ones already defined in Chapter 4, to which more preconditions are added to prevent collisions.

There are also new activation rules that consist of following a cork or a opener. That is, when a cork or opener module activates and moves, the first static module that activates and does not trigger another deadlock will use one of these rules.

Finally, there are some added rules for the active modules to walk (with slide or convex transition) over corks and openers in case they have order preference.

At the same level than the standard rules are the rules that solve deadlocks: stopper rules. That is, those related to the two new states. These rules consider the four possible states and the two moves. We have already defined which three kinds of deadlocks are possible and which states act in each case.

The previous definitions provide us the scheme of rules shown in Figure 19.

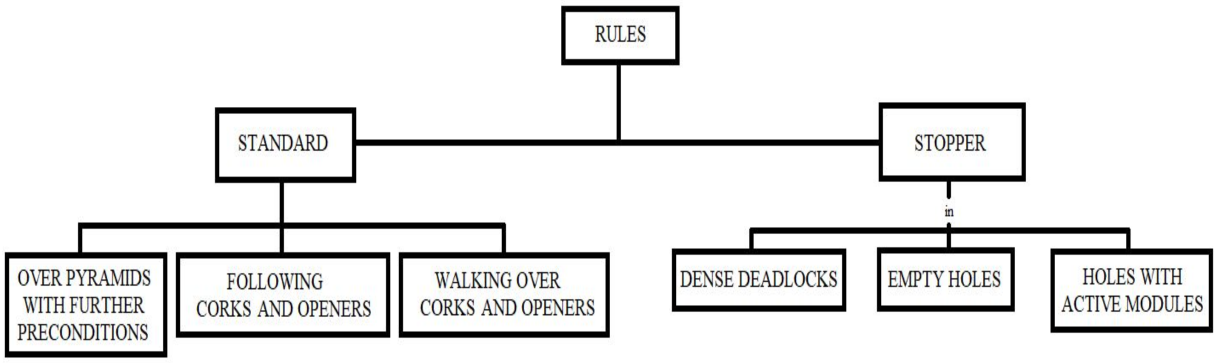
RULES
STANDARD
STOPPER
in
OVER PYRAMIDS WITH FURTHER PRECONDITIONS
FOLLOWING CORKS AND OPENERS
WALKING OVER CORKS AND OPENERS
DENSE DEADLOCKS
EMPTY HOLES
HOLES WITH ACTIVE MODULES

Figure 19: *Scheme of rules classes.*

We have already explained the standard rules, which apply to static or active modules. The stopper rules apply to cork, opener and static - in order to change state from or to cork or opener - modules and they are different according to the three mentioned situations of deadlocks.

Another important functionality in this set of rules is that of priorities. The maximum priority is *10* and the minimum is *1*. The rules with highest priority are those that tend to act more and those with the lowest priority are those that appear in extreme cases.

Consequently, when a module can apply two different rules due to their preconditions, the priorities define an order. The order that we define is that the first module to move is the one that is in the rightmost position following the right-hand rule, whenever possible.

We defined *18* rules. Applying rotations by multiples of *90º*, and synthesizing the second environment preconditions and the order criterion, we can get the analogous surface and set of rules for each direction. The actual implementation of the *18* rules on simulator can be found in Chapter 8. As in Chapter 4, we can therefore think of our rules as performing the advance of a strip robot following the right-hand rule over any obstacle in any direction. In the remaining of this chapter, we describe the different sets of rules.

STANDARD RULES: OVER PYRAMIDS WITH FURTHER PRECONDITIONS

1 - Activation
Priority: 10
Preconditions:

- state is static
- right position is empty
- bottom position is occupied by a static module
- left position is empty or is an obstacle
- lower right position is occupied by a static module and top position is empty

  or

  lower right position is empty and top position is empty or is an obstacle
- already described constraints to prevent collisions

Postcondition: change state to active.

We added top and left preconditions to the next rule in order to keep the robotic system connected at all times along the locomotion and in order.

2 - Slide
Priority: 10
Preconditions:

- state is active
- top position is not occupied by a static module
- right position is empty
- bottom position is occupied by a static module
- left position is not occupied by an active module
- lower right position is occupied by a static module
- already described constraints to prevent collisions

Postcondition: slide to right.

The same happens with the following rule and the top position.

```
3 - Convex transition
Priority: 10
Preconditions:
   -  state is active
   -  top position is not occupied by a static module
   -  right position is empty
   -  bottom position is occupied by a static module
   -  lower right position is empty
   -  already described constraints to prevent collisions
Postcondition: convex transition to lower right position.
```

Finally, the deactivation rule stays the same as in Chapter 4.

```
4 - Deactivation
Priority: 1
Preconditions:
   -  state is active
Postcondition: change state to static.
```

STANDARD RULES: FOLLOWING CORKS AND OPENERS

The "following corks and openers" standard rules only need activation. Once activated, the modules use one of the other standard rules. We set the priority to 9 because it is not as used as frequently as Rule 1 but it is still very important because of the order. The same observation applies to the next two rules.

```
5 - Activation
Priority: 9
Preconditions:
   -  state is static
   -  right position is empty
   -  bottom position is occupied by a static module
   -  left position is occupied by a static module
   -  lower left position is occupied by a static module
   -  ⎧ lower right position is occupied by a static module
      ⎪ and top position is empty
      ⎨
      ⎪                            or
      ⎪
      ⎩ lower right position is empty and top position is
        empty or is an obstacle
   -  already described constraints to prevent collisions
Postcondition: change state to active.
```

STANDARD RULES: WALKING OVER CORKS AND OPENERS

The “walking over corks and openers” standard rules only need movement. That is, slide and convex transition. The modules activate or deactivate using one of the other standard rules.

In the following rule, the active module can walk over cork and opener modules when its current position has at its bottom one of them or its next position - that is, its right position - has below it one of them.

```
6 - Slide
Priority: 9
Preconditions:
   -  state is active
   -  top position is not occupied by a static module
   -  right position is empty
   -  left position is not occupied by an active module
      ⎧ bottom position is occupied by a cork or opener module
      ⎪ and lower right position is occupied by a static
      ⎪ module
   -  ⎨      or
      ⎪
      ⎪ bottom position is occupied by a static module and
      ⎪ lower right position is occupied by a cork or opener
      ⎩ module
   -  already described constraints to prevent collisions
Postcondition: slide to right.
```

```
7 - Convex transition
Priority: 9
Preconditions:
   -  state is active
   -  top position is not occupied by a static module
   -  right position is empty
   -  bottom position is occupied by a cork or opener module
   -  lower right position is empty
   -  already described constraints to prevent collisions
Postcondition: convex transition to lower right position.
```

STOPPER RULES: IN DENSE DEADLOCKS

The "in dense deadlocks" stopper rules have priorities of 5 to 9 depending on whether they are used a lot or a little. Moreover, they apply constraints to prevent collisions, keep the robot connected at all times and by order criterion. These constraints make their preconditions very particular according to the situation of the environment. Synthesizing, we obtain the following rules:

```
8 - Change state
Priority: from 5 to 9
Preconditions:
   -  state is static
   -  right position is empty
   -  bottom position is occupied by a static module
   -  left position is occupied by a static module
   -  lower right position is occupied by a static module
   -  lower left position is occupied by a static module

          ┌
          │    lower right position is occupied by a static module
          │
   -    ──┤         or
          │
          │    lower right position is empty and top position is
          │    empty, is an obstacle or is occupied by a static
          └    module

   -  already described constraints to prevent collisions
Postcondition: change state to cork.
```

The difference between change state and activation slide or convex transition is the number of other constraints: the next two rules contain less preconditions than the previous one. This is due to the need for the current module to be in cork state.

```
9 - Activation and slide
Priority: from 6 to 9
Preconditions:
   - state is cork
   - right position is empty
   - bottom position is occupied by a static module
   - left position is occupied by a static module
   - lower right position is occupied by a static module
   - already described constraints to prevent collisions
Postcondition: change state to active and slide to right.
```

```
10 - Activation and convex transition
Priority: 10
Preconditions:
   - state is cork
   - right position is empty
   - bottom position is occupied by a static module
   - left position is occupied by a static module
   - upper right position is not occupied by an active module
   - lower right position is empty
   - lower left position is occupied by a static module
   - already described constraints to prevent collisions
Postcondition: change state to active and convex transition to lower
right position.
```

The following rule goal is not to maintain the cork state if it is not necessary. Therefore, in the situation that a module is surrounded by other static modules, it deactivates, even if its lower right position is empty.

```
11 - Deactivation
Priority: 1
Preconditions:
   - state is cork
   - top, right, bottom, left, upper right, lower left and upper
     left positions are occupied by static modules
   - lower right position is empty or is occupied by a static module
Postcondition: change state to static.
```

STOPPER RULES: IN EMPTY HOLES

Commonly the “in empty holes” stopper rules need their right and left positions empty. But there is a corner case that requires it to be static. Due to this case we have added some of the needed preconditions to make this corner case unique. The same observation applies to the next two rules.

```
12 - Change state
Priority: from 6 to 9
Preconditions:
   - state is static
   - top position is occupied by a static module
   - right position is empty
   - bottom position is occupied by a static module
   - { left position is empty
       or
       left position is occupied by a static module, upper
       right position is not empty, position (-1,2) is
       occupied by a static module and position (2,1) is not
       occupied by an active module }
   - { lower right position is occupied by a static module
       or
       lower right position is empty, lower left position is
       empty, is an obstacle or is occupied by a static
       module and upper left position is occupied by a static
       module }
   - already described constraints to prevent collisions
Postcondition: change state to opener.
```

```
13 - Activation and slide
Priority: from 7 to 9
Preconditions:
   - state is opener
   - top position is occupied by a static module
   - right position is empty
   - bottom position is occupied by a static module
   - lower right position is occupied by a static module
   - { left position is empty and second top position is
       empty or is an obstacle

             or

       left position is occupied by a static module and
       second top position is occupied by a static module }
   - already described constraints to prevent collisions
Postcondition: change state to active and slide to right.
```

We prioritize the activation and convex transition of the next rule to follow the order criterion.

```
14 - Activation and convex transition
Priority: 9
Preconditions:
   - state is opener
   - top position is occupied by a static module
   - right position is empty
   - bottom position is occupied by a static module
   - left position is empty
   - lower right position is empty
   - upper left position is occupied by a static module
   - already described constraints to prevent collisions
Postcondition: change state to active and convex transition to lower
right position.
```

As in the previous deactivation stopper rule, there is a last rule in this set to deactivate modules.

15 - Deactivation
Priority: 1
Preconditions:

- state is opener
- top, right, bottom, left, upper right, lower left and upper left positions are occupied by static modules
- lower right position is empty

Postcondition: change state to static.

STOPPER RULES: IN HOLES WITH ACTIVE MODULES

The following set of rules requires an active module inside a hole. We have also a corner case. In our implementation, the most frequent is that the top position is occupied by a static module and the left position is occupied by an active module.

16 - Change state
Priority: from 7 to 9
Preconditions:

- state is static
- right position is empty
- bottom position is occupied by a static module
- { top position is occupied by a static module and left position is occupied by an active module

  or

  top position is an obstacle and left position is occupied by a static module }

```
   -  ⎧ upper right position is an obstacle, lower right
      ⎪ position is occupied by a static module and upper left
      ⎪ position is empty
      ⎨
      ⎪     or
      ⎪
      ⎩ lower right position is empty
   -  already described constraints to prevent collisions
Postcondition: change state to opener.
```

17 - Activation and slide

```
Priority: from 7 to 9
Preconditions:
   -  state is opener
   -  top position is occupied by a static module
   -  right position is empty
   -  bottom position is occupied by a static module
   -  upper right position is an obstacle
   -  lower right position is occupied by a static module
   -  already described constraints to prevent collisions
Postcondition: change state to active and slide to right.
```

18 - Activation and convex transition

```
Priority: from 7 to 9
Preconditions:
   -  state of the current module is opener
   -  top position is an obstacle or is occupied by a static module
   -  right position is empty
   -  bottom position is occupied by a static module
   -  lower right position is empty
   -  upper left position is not occupied by a static module
   -  already described constraints to prevent collisions
Postcondition: change state to active and convex transition to lower
right position.
```

Finally, the deactivation rule is the same as in the “in empty hole cycles” stopper rules.

In conclusion, our algorithm uses *18* rules divided into two big sets of rules - each with three subsets - to the advance of a strip robot following the right-hand rule over any obstacle in any direction. The rules use four states and require exploring the second environment in order to prevent collisions and deadlocks at bottlenecks.

6. Correctness

6.1. Locomotion without obstacles

In this section, we prove that the rules described in Chapter 3 produce any connected strip of modules to advance eastwards in the absence of obstacles.

Proposition 1*: Let* R *be a connected modular robot with* n *modules forming a horizontal strip on a horizontal surface in the absence of obstacles. The set of rules for locomotion without obstacles from Chapter 3 makes the leftmost module and only this module to activate.*

Proof: All modules are static initially. There is just one rule to activate a module: `Rule 1: Activation`. This rule requires the left position to be empty. The only module who has its left position empty is the leftmost module. Therefore, it is the only module that can activate. This module is static, its right position is a static module, and its top and upper right positions are empty. Therefore, it applies Rule 1 and activates.

Proposition 2*: Let* R *be a connected modular robot with* n *modules forming a horizontal strip on a horizontal surface in the absence of obstacles. Under the rules for locomotion without obstacles from Chapter 3, the lattice cells where modules can move are exactly at distance from the initial shape.*

Proof: There are just three rules involving movement, namely rules 2, 3 and 4. All of them need the module to be activated. By Proposition 1, only the leftmost module is activated.

The active module has its right position occupied by a static module, and his top and upper right positions empty. Then, by `Rule 2: Initial convex transition NE`, the module moves to its upper right position. As a result, the module is at a distance one from the initial shape, just above the second leftmost module.

Then, the module applies `Rule 3: Slide E` and slides to its right. Since its bottom and lower right positions are occupied by static modules, and its right position is empty. As it is still in contact with an initial static module, the moving module stays at distance one from the initial robot configuration. This process repeats until the moving module ends up located lust above the rightmost static module in the strip. Then, it applies `Rule 4: Final convex transition SE` since its right and lower right positions are empty, and its bottom position is occupied by a static module. Therefore, the module moves to its lower right position. Again, the module stays at a distance one from the initial configuration, next to the initially rightmost module in the strip.

We have proved that the lattice cells occupied by moving modules are all at distance one from the initial robot configuration.

***Proposition 3**: Let* R *be a connected modular robot with* n *modules forming a horizontal strip on a horizontal surface in the absence of obstacles. The set of rules for locomotion without obstacles from Chapter 3 keeps the robotic system connected at all times along the locomotion, and no collisions happen.*

Proof: Let us first prove that the robotic system stays connected at all times throughout the reconfiguration. By Proposition 1, only the leftmost module is activated. So, at the beginning, all other modules stay static and connected.

By Proposition 2, the moving modules stay at a distance one from the static modules at all times. Therefore, the moving modules are connected to the static ones, and the entire robot stays connected.

When a module reaches the rightmost position, it can not apply any rule except the last one, *Rule 5: Deactivation*, and it is deactivated. Since it is sitting to the right of a static module, it is connected to it.

Therefore, the robotic system stays always connected.

Let us now prove that no collisions happen. By Proposition 1, only the leftmost module can be activated. Until it does not move, no other module can be activated.

The other rules are deactivation or movement. Deactivation does not involve a move and, therefore, cannot produce any collision. All movement follow the right-hand rule (upper right, right, and lower right). No collisions between active modules can happen, because the advancing rules require the goal position to always be empty. So, if there is a collision, it has to be between the active module in motion and a static module. By Proposition 2, the movement region is exactly at distance one from the initial robot. Then, an active module never collisions with a static module.

Therefore, no collisions happen.

***Theorem 1**: Let* R *be a connected modular robot with* n *modules forming a horizontal strip. The set of rules for locomotion without obstacles from Chapter 3 produces the endless advance of* R *from left to right on a horizontal surface. Along the locomotion, the robotic system stays connected at all times, and no collisions happen. When synchronously run, at most one rule is applied per module and round.*

Proof: It is only left to prove that the set of rules for locomotion without obstacles produces the endless advance of R from left to right on a horizontal surface, and that, when synchronously run, at most one rule is applied per module and round.

Let us start proving the first statement. For that, we start proving that the leftmost module always reaches the first empty position next to the rightmost module.

By Proposition 1, at the beginning, the first module is the only one activated. In the next step only *Rule 2: Initial convex transition NE* can be used. Then, the module is moved to the top of the second leftmost module.

Now, the active module can only apply *Rule 3: Slide E*. No other rule can be used.

When the module arrives to the top of the initially rightmost module, *Rule 3: Slide E* is no longer used. Then, *Rule 4: Final convex transition SE* moves the module to the first empty position next to the rightmost module. Then the module can only do one thing: deactivate itself by *Rule 5: Deactivation*. The module will remain static until it again happens to sit in the leftmost position of the robot configuration.

Therefore, the leftmost module always arrives at the first empty position to the right of the rightmost module.

Furthermore, the module uses no more than one rule at the same time. In other words, the module applies at most one rule per round.

Finally, let us prove that, when run in parallel, each module has the same behavior than the first module. By induction, it is enough to prove that, at the beginning, the leftmost and second leftmost modules do not interfere each other, since Proposition 2 guarantees that no collisions may happen.

When the first module is moved to the top of the second module, this one cannot activate because its top position is not empty. In the next step, the first module slides to the right, but the second one cannot activate either because its upper right position is not empty yet. It is after two movements - and three rounds - that the second module activates. At this point, it has a clear path, in the next step, to start moving, since the first module keeps sliding to the right.

When run synchronously, the distance between the first and the second modules is three. Therefore, they do not collide along the strip, and at most one rule is applied per module and round.

6.2. Locomotion over pyramids

In this section we prove that the rules described in Chapter 4 produce any connected strip of modules to advance from left to right in the presence of an obstacle with pyramidal shape.

Proposition 4: *Let* R *be a connected modular robot with* n *modules forming a horizontal strip. The set of rules for locomotion over pyramids from Chapter 4 produces the advance of* R *following the right-hand rule over horizontal and vertical surfaces.*

Proof: We are going to prove that the new set of rules works as the set of rules without obstacles. At the beginning, all modules are static. We distinguish two surfaces: horizontal and vertical.

On horizontal surfaces, `Rule 1: Activation convex transition NE` activates the leftmost module. Now preconditions are more permissive than in the previous set of rules, but for the horizontal surface this is enough. No other module will be activated because it is necessary that the left position of each module be empty or an obstacle in all activation rules, and that the module is static. No other rule can be applied by the leftmost module. Indeed `Rule 5: Activation slide E` and `Rule 2: Activation convex transition SE` require the bottom position to be occupied by a static module and the right position be empty, `Rule 4: Activation slide N` requires the upper right position to be occupied by a static module, and `Rule 3: Activation convex transition NW` requires the top position to be occupied by a static module. None of these conditions hold for the leftmost module. No other module will be activated due to the same argument.

Then, the leftmost module can move. `Rule 6: Convex transition NE` moves the module to the upper right position and no other rule can be applied: notice that rules 4, 6, 8 and 10 have the same preconditions as rules 3, 5, 7 and 9. `Rule 12: Slide S` and `Rule 12: Convex transition NW` require the left position to be occupied by a static module. Therefore, the leftmost module makes a convex transition to the upper right position. No collision happen because this position was empty.

Now, as long as the active module has its top and right positions empty, and its lower right position occupied by a static module, it can only slide to the right position using `Rule 11: Slide E`. No other module can activate because the leftmost module requires its top position to be empty. Therefore, the behavior is the same as with the previous set of rules.

From this point on, the modules progress as in the previous section. We only need to see what happens when the initially leftmost module reaches the end of the strip.

Once the initially leftmost module is above the initially rightmost module, as its left, top, right and lower right positions are empty, it moves to the lower right position using `Rule 7: Convex transition SE`.

Finally, the current rightmost module deactivates using `Rule 13: Deactivation` because it can not use any other rule: its left, bottom and bottom left positions are not empty.

Therefore, the set of rules for locomotion over pyramids produces the advance of R from left to right on horizontal surfaces.

On vertical surfaces, we distinguish two situations depending on whether the surface is located to the right or to the left of the strip of modules. If it is located to the right, the strip will advance up, while it will advance down if it is located to the left. In other words, the strip always advances following the right-hand rule over the surface. Due to the symmetry of the rules, the previous proof for horizontal surfaces is also valid for vertical surfaces.

Let us now discuss what happens in the presence of corners.

Proposition 5: *Let* R *be a connected modular robot with* n *modules forming a horizontal strip. The set of rules for locomotion over pyramids from Chapter 4 produces the advance of* R *following the right-hand rule at the intersections of horizontal and vertical surfaces (corners).*

Proof: We study the four intersections defined in Chapter 4. First we explain how a module reaches one intersection. By Proposition 4, when the corner position is empty, there is a module that will eventually occupy this cell.

We describe how the modules go over each of the four types of corners:

1. Monotonously increasing concave corner. In this case, following the right-hand rule, we have first a horizontal surface, and then, a vertical one. So, the robot modules arrive to the corner using rules from the set of rules for horizontal surfaces.

   The rightmost module has at its right an obstacle. So, when an active module reaches above it, it deactivates by `Rule 13: Deactivation`, since its right and upper right positions have obstacles and its top and left positions are empty. That is, no other rule can be applied.

   After that, when the next active module reaches the left position of the last deactivated module, it applies `Rule 6: Convex transition NE` and deactivates by `Rule 13: Deactivation`, due to its environment.

The following active modules just turn the corner by *`Rule 6: Convex transition NE`* and change their advance mode from horizontal to vertical.

2. Monotonously increasing convex corner. In this case, we have first a vertical surface, and then, a horizontal one. So, the robot modules arrive to the corner using rules of the set of rules for vertical surfaces.

   The topmost module has an obstacle to its right. There is an empty cell above this obstacle cell.

   The next active module advances as on a vertical surface. It advances using *`Rule 10: Slide N`* and reaches the topmost position with *`Rule 6: Convex transition NE`*. At this point it deactivates due to its environment

   When the next active module reaches the top position of the last deactivated module, it uses *`Rule 7: Convex transition SE`* to place itself to the right position of the previous module and deactivates too.

   The following modules just turn the corner by *`Rule 6: Convex transition NE`* and change their advance mode from vertical to horizontal.

3. Monotonously decreasing concave corners. By the rotational symmetry of the rules, this case is analogous to case 1.

4. Monotonously decreasing convex corner. By the rotational symmetry of the rules, this case is analogous to case 2.

Finally, we prove that the last module always leaves an intersection. This is because of the new activation rules. More in detail:

1. In case 1, the last static module has obstacle cells at its right and bottom positions. Its top cell is occupied by a static module and its left and upper left positions are empty. The module activates by *`Rule 3: Activation convex transition NW`*, and moves by *`Rule 9: Convex transition NW`*.

2. In case 2, the last static module has a static module to its right, and its left, top and upper right positions are empty. The module activates by *`Rule 1: Activation convex transition NE`*, and moves by *`Rule 6: Convex transition NE`*.

3. In case 3, the last static module has a static module to its bottom, and its top, right and lower right position are empty. The module activates by *`Rule 2: Activation convex transition SE`*, and moves by *`Rule 7: Convex transition SE`*.

4. In case 4, the last static module has obstacle cells at its bottom and left positions. Its right cell is occupied by a static module and its top and upper right positions are empty. The module activates by *`Rule 1: Activation convex transition NE`*, and moves by *`Rule 6: Convex transition NE`*.

Therefore, the set of rules for locomotion over pyramids produces the advance of *R* following the right-hand rule at the intersections of horizontal and vertical surfaces.

***Theorem 2**: Let* R *be a connected modular robot with* n *modules forming a horizontal strip. The set of rules for locomotion over pyramids produces the endless advance of* R *from left to right over a pyramid surface. Along the locomotion, the robotic system stays connected at all times, and no collisions happen. When synchronously run, at most one rule is applied per module and round.*

Proof: By Proposition 4, the set of rules for locomotion over pyramids produces the endless advance of *R* following the right-hand rule on horizontal and vertical surfaces, while the robot stays connected and no collisions happen. By Proposition 5, the same thing happens at all intersections.

Therefore, the theorem is almost proved. The only missing detail is to prove that at most one rule is applied per module and round. But this is guaranteed by the structure of the proofs of propositions 4 and 5.

The leftmost module in terms of the right-hand rule uses no more than one rule at the same time. In other words, the module applies at most one rule per round. By induction, it is enough to prove that second leftmost module in terms of the right-hand rule applies at most one rule per round.

After the leftmost module's first moving, the second module cannot activate because its target position is not empty. In the next step, the first module moves leaving the target position of the second one. Therefore, this position is empty and the second module is activated. It is after two movements - and three rounds - that the second module activates. At this point, it has a clear path, in the next step, to start moving, since the first module keeps advancing to the rightmost position.

When run synchronously, the distance between the first and the second modules is three. Therefore, at most one rule is applied per module and round.

6.3. Preventing collisions and deadlocks at bottlenecks

Due to the time constraints of the project, we have not been able to develop further correctness proofs. In order to verify the correctness of in practice of the set of rules for preventing collisions and deadlocks at bottlenecks, we have developed and run a large set of tests for each set of rules implemented. The description of the tests and the results of the experiments run can be found in Chapter 8.

7. Complexity

In this chapter we analyze the complexity of the locomotion described in chapters 3, 4 and 5.

Complexity, in this case, is established in terms of moves in the first place, since moving a module in a slide or a convex transition is the most time-consuming operation of all.

In the second place, we have communication. In our model, we have replaced communication protocols between modules by the capability of the modules to check the lattice positions of their first and second neighborhoods to know whether they are empty osr occupied. In the last case, they can also check whether the occupant is an obstacle or a module and, in the last case, which is its state. Therefore, in our model communication costs per module are a linear function of the number of rounds executed by the algorithm.

Finally, complexity also includes the classical computation costs in terms of running time and memory space required by each model when evaluating preconditions of rules. Again, the running time is obviously linear in the number of rounds performed by the algorithm. As for memory space requirements, it is easy to see that each module uses $O(1)$ memory space per round.

Therefore, we concentrate on counting the number of moves and the number of rounds executed by our algorithm.

We start computing the number of round of the algorithm and the number of moves per module for locomotion in the absence of obstacles.

The strip advances one position after the leftmost module activates, moves n positions to its right and deactivates. When the module advances three positions, the next leftmost module activates and, after four rounds, it reproduces the same movements.

If a_k is the number of rounds (parallel steps) that the strip needs in order to advance k positions, we can define the following recurrence:

$$a_0 = 0$$
$$a_1 = n + 2$$
$$a_{i+1} = a_i + 4, \; \forall i \geq 1.$$

This recurrence results in

$$a_k = n + 4k - 2, \; \forall k \geq 2.$$

Therefore, for locomotion in the absence of obstacles, the strip advances *k* positions after $O(n+k)$ rounds or parallel steps.

Regarding the number of moves per module, each module moves exactly *n* times in order to traverse the entire strip. Therefore, the strip advances *k* positions after $O(n)$ moves of *k* modules.

The same thing happens when it comes to locomotion over pyramids since the behavior is the very similar. The leftmost module moves over the rest of the modules without deactivating until it reaches the rightmost in terms of the right-hand rule. Therefore, the complexity is the same.

The problem appears when crawling over obstacles with bottlenecks. Then, it is possible to have a module deactivated before it has reached the rightmost position of the strip. This means that the progress made by a module is not constant and depends upon the obstacles and the number of deadlocks that can be formed.

However, we can still complete the complexity of the algorithm. In the best cases no deadlocks happen. In this case, we obtain the same complexity as before.

The problems arise when there is at least one deadlock.

In this case, the number of moves per module could increase due to some movements needed to solve deadlocks or some modules that move cyclically within a deadlock. But at least one module will make use of some rule, otherwise the strip would stop moving forward and our algorithm would be incorrect. Therefore, in the worst case, we have:

$$a_0 = 0$$
$$a_1 = n+2$$
$$a_{i+1} = a_i + n + 2, \ \forall i \geq 1.$$

Therefore:

$$a_k = nk + 2k, \ \forall k \geq 0.$$

Then, the strip advances *k* positions after $O(nk)$ rounds. In practice, though, deadlock situations do not increase the number of rounds substantially, as we will see in Chapter 8.

All together, the number of parallel steps in order to advance *k* positions a strip of length *n* is between $O(n + k)$ and $O(kn)$. The number of moves per module is $O(n + k)$.

8. Simulation

8.1. *AgentSystem*

We implemented our algorithm with *AgentSystem*, a practical Java tool for simulating synchronized distributed algorithms on sets of 2 (and 3) dimensional square (and cubic) lattice-based agents. It assumes that each module is capable to change position in the lattice and that neighboring agents can attach and detach from each other. In addition, it assumes that each module has some constant size memory and computation capability, and can send or receive constant size messages to or from its neighbors. The system allows the user to define sets of agents - obstacles and modules - and sets of rules and apply one to the other.

The system has its own language to implement rules. It simulates the synchronized execution of the set of rules by all the modules, and can keep track of all actions made by the modules at each step, supporting consistency warnings and error checking. For more details see [6].

8.2. Orientation and other difficulties

We described in chapters 3, 4 and 5 all the *18* rules we use to complete our distributed algorithm.

Their actual implementation was done into *323* rules. This was due to two reasons. On one hand, the simulator requires all *90º* rotations of the rules to be implemented. On the other, the many preconditions to prevent collisions and deadlocks in bottlenecks recommended to decompose the rules into smaller ones.

Finally, the fact that the simulator only accepts the logical operators *AND* and *NOT* make the preconditions longer to express when they contain an *OR*, and we decided to break them when convenient.

8.3. Test Bank

In this section we describe the obstacles used to test the algorithm.

For the set of rules in absence of obstacles, there is only one possible test in each direction.

For the set of rules for locomotion over pyramids, we created a pyramids with several step sizes. This size is defined by the width and the height of the steps. We illustrate a symmetric pyramid with several step sizes steps in Figure 20 to test when the robot is going up and when it is going down. Then, we did the same in the other directions.

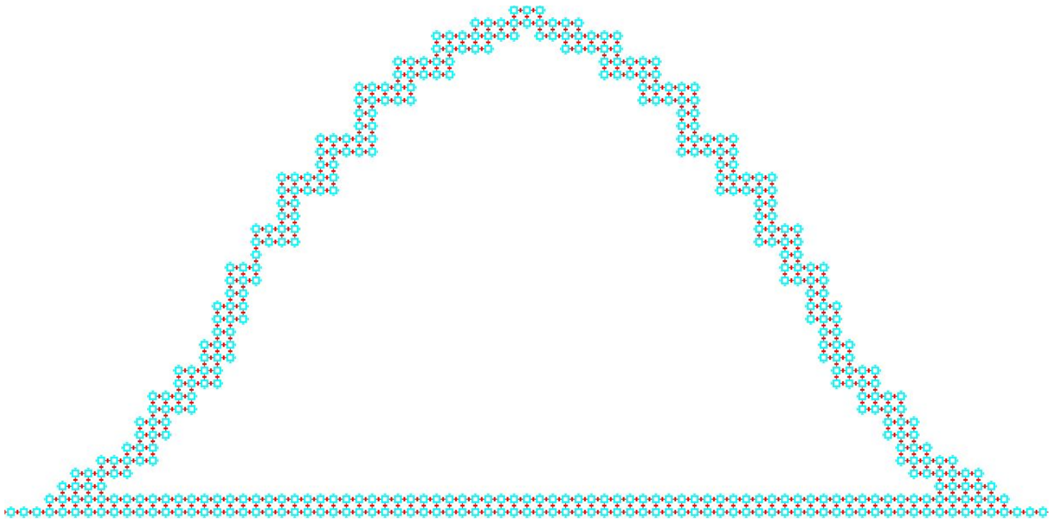
Figure 20: *Symmetric pyramid with several step sizes.*

For the set of rules for locomotion over any obstacle, we needed a wide variety of tests. So we defined their obstacles systematically.

We started expanding the pyramids. We made histograms with columns separated by a distance one at maximum. Each pair of columns could be either of the same height, or of different heights (the first lower and the second higher, or *vice versa*). See Figure 21 for an example. We defines this kind of histograms in the four directions.

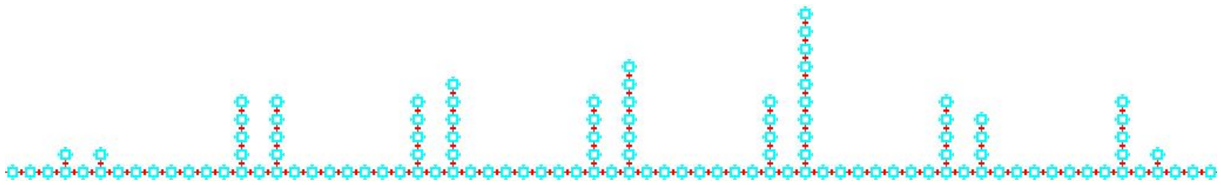
Figure 21: *Example of histogram with columns separated by distance one at maximum.*

Then, we considered distance two. That is, we had histograms with columns separated by distance two at maximum. In this case, we needed to do more experiments not only with side columns that had different heights. Also that the base of the holes of width two were aligned or with different heights. We illustrate an example in Figure 22. We did the same in all directions.

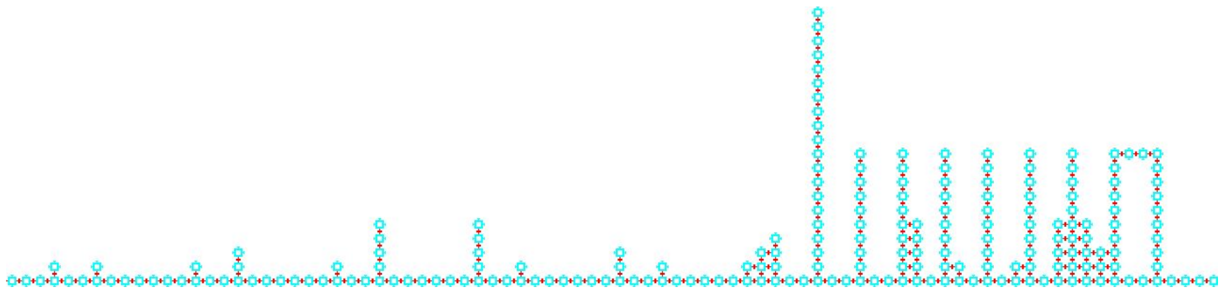
Figure 22: *Example of histogram with columns separated by distance two at maximum.*

We repeated the same process with columns at a distance three at maximum, and we created the obstacles in each direction as well. See the Figure 23 for an example.

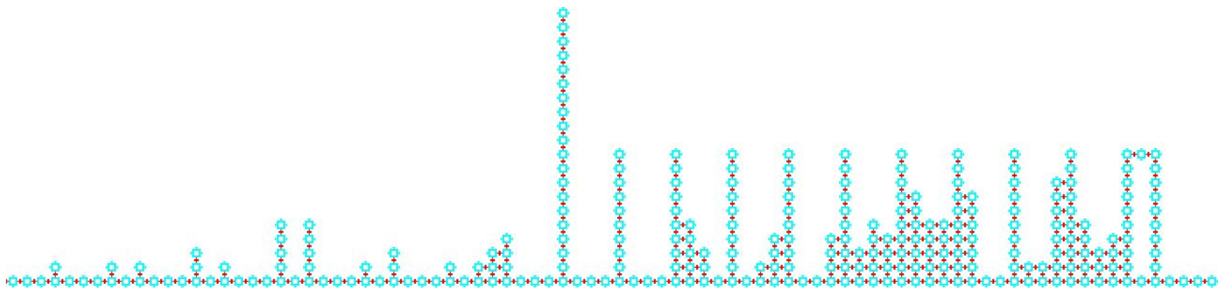
Figure 23: *Example of histogram with columns separated by distance three at maximum.*

Next, we prepared a set of obstacles that were not histograms. These contained diagonal cavities in all directions. We show three examples of this set in Figures 24, 25 and 26.

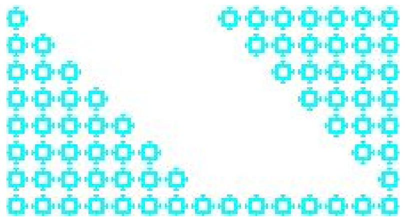
Figure 24: *Example of an obstacle with a simple diagonal cavity.*

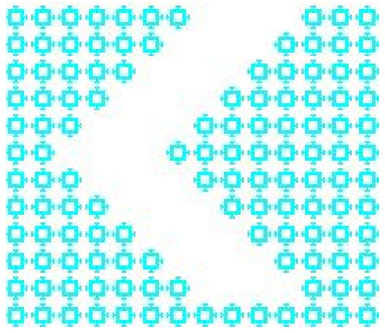
Figure 25: *Example of an obstacle with a double diagonal cavity.*

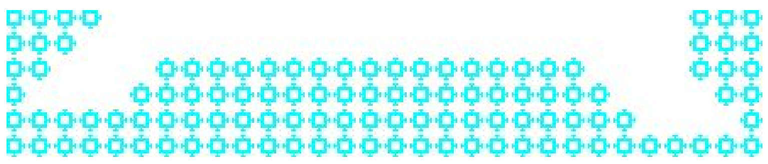
Figure 26: *Example of an obstacle with two diagonal cavities.*

The following set of obstacles contained obstacles with wide cavities and narrow entrances. We illustrate three examples in Figures 27, 28 and 29.

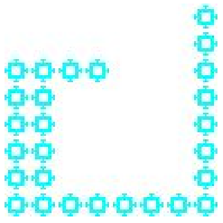
Figure 27: *Example of an obstacle with a wide cavity and a narrow entrance on the right.*

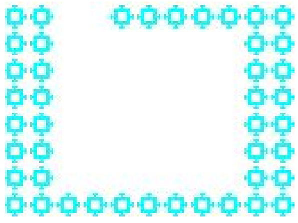
Figure 28: *Example of an obstacle with a wide cavity and a narrow entrance on the left.*

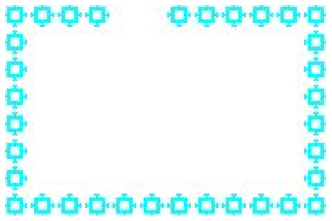
Figure 29: *Example of an obstacle with a wide cavity and a narrow entrance in the middle.*

Then, we combined previous sets to create obstacles with histograms, wide diagonal cavities and narrow entrances. See three examples in Figures 30, 31 and 32.

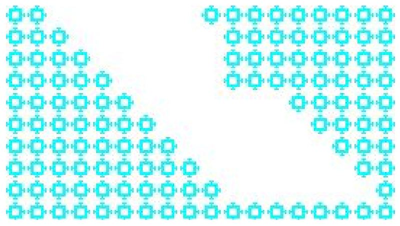
Figure 30: *Example of an obstacle with one wide diagonal cavity and a narrow entrance.*

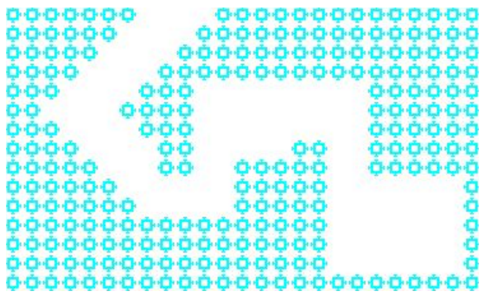
Figure 31: *Example of an obstacle with two wide diagonal cavities and narrow entrances.*

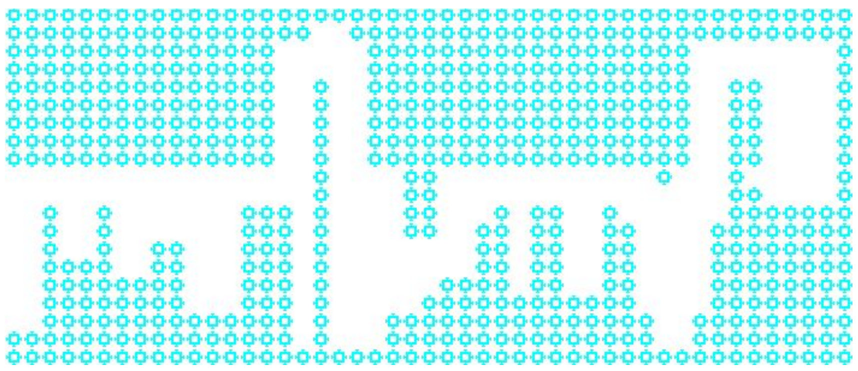

Figure 32: *Example of an obstacle with histograms and narrow entrances.*

In this point we defined a set of obstacles with some other more irregular cavities and bottlenecks. This set contained several obstacles with narrow entrances and holes of different irregular shapes. We can see three examples in Figures 33, 34 and 35.

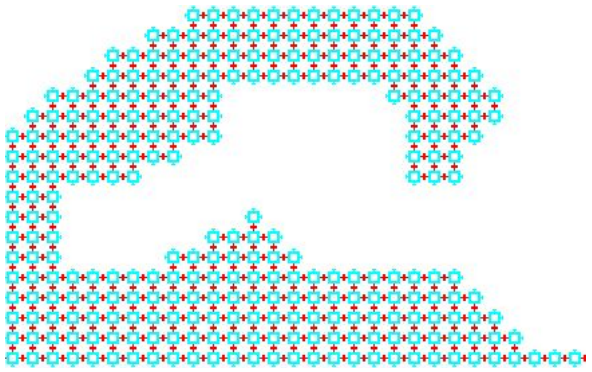

Figure 33: *Example of an obstacle with a narrow entrance on the right and a cavity with irregular shape.*

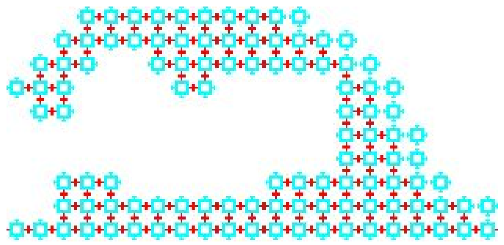

Figure 34: *Example of an obstacle with a narrow entrance on the left and a cavity with irregular shape.*

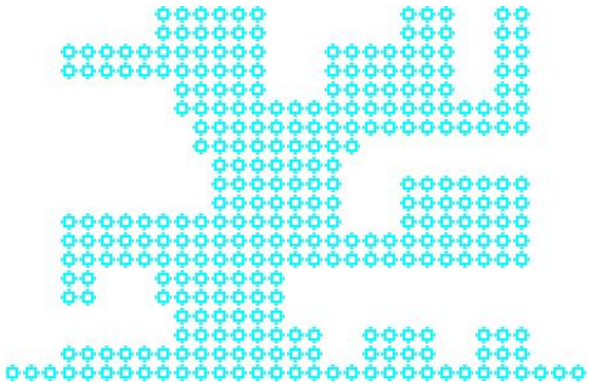

Figure 35: *Example of an obstacle with multiple narrow entrances and cavities with irregular shapes.*

Finally, we created a huge obstacle combining all sets of obstacles defined above. In addition, we included empty cells between obstacle cells, long corridors with no exit, nested bottleneck, and a long way to go to be able to make a more complete test and be able to perform a better analysis of the complexity of our algorithm. Figure 36 shows this large obstacle.

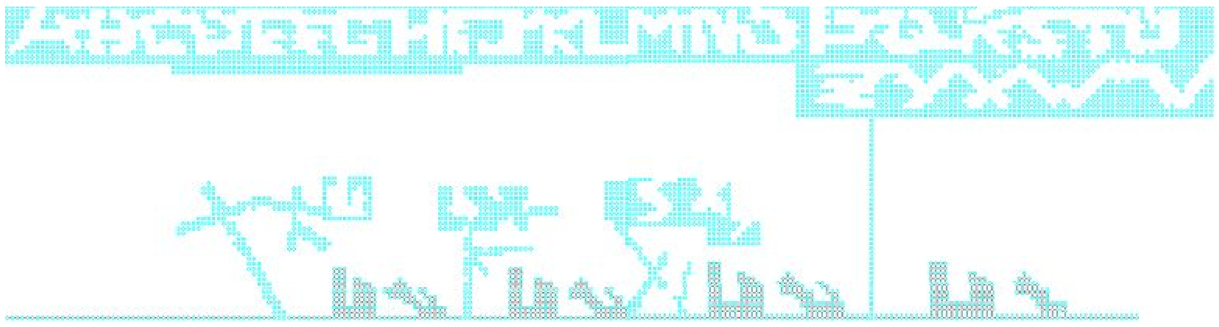

Figure 36: *Huge obstacle, with 3662 obstacle cells, multiple narrow entrances, cavities with irregular shapes, long corridors with no exit and a long way to go.*

In conclusion, we systematically defined several sets of obstacles with different characteristics. In total we obtained 61 scenarios.

8.4. Results

In this section we analyze the results of the tests carried out.

In Chapter 6 we proved the correctness of the set of rules for locomotion in absence of obstacles. Here, we will only determine the exact complexity in practice. In Chapter 7 we proved that the number of rounds was $O(n + k)$, where n is the number of modules of the strip and k is the number of positions that the strip advances. We want to find the exact constant that multiplies this value in our simulator.

We made several tests with different number of modules and number of positions advanced. For each test we defined a table with the number of parallel steps run by the algorithm, as a function of these number of modules (n) and these number of positions (k) advanced by the robot strip. In addition, we defined a graph with the same information. The trend lines join the number of parallel steps for each number of positions k advanced, and there is a line for each number of modules n. We also show the equation of each trend line. In these equations, $x = k$.

In Table 1 and Figure 37 we illustrate the table and graph of our test for locomotion in the absence of obstacles.

		Number of modules (n)				
		5	**10**	**20**	**50**	**100**
Number of positions advanced (k)	**1**	7	12	22	52	102
	2	11	16	26	56	106
	5	23	28	38	68	118
	10	43	48	58	88	138
	50	203	208	218	248	298

Table 1: *Number of parallel steps run by the algorithm, as a function of the number of modules (*n*) and the number of positions (*k*) advanced by the robot strip, in the absence of obstacles.*

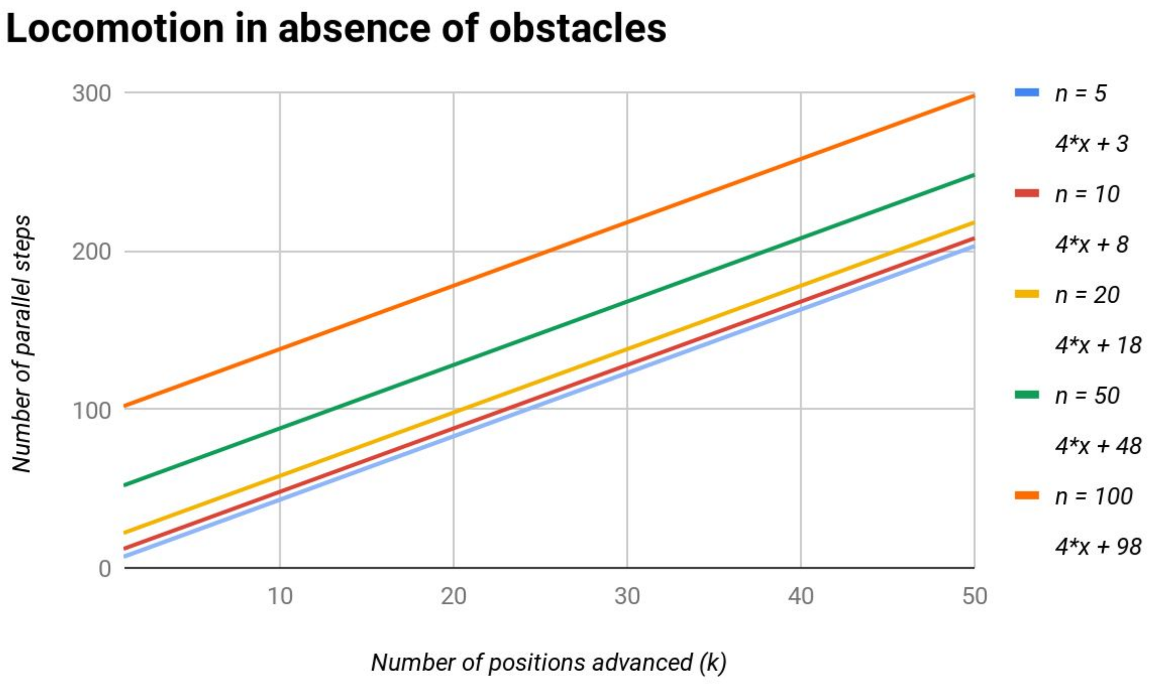

Figure 37: *Graphic of the number of parallel steps run by the algorithm, as a function of the number of modules (*n*) and the number of positions (*k*) advanced by the robot strip, in the absence of obstacles.*

The slope is constant, $\forall k$. Therefore, we obtain that the number of rounds (parallel steps) is exactly $n + 4k - 2$.

We also proved in Chapter 6 the correctness of the set of rules for locomotion over pyramids. We made similar tests with this algorithm to find its exact complexity on our simulator. Table

2 contains a new value k^*, which is the minimum value of k needed to overcome the symmetric pyramid of Figure 20. For this case, $k^* = 153 + n$.

		Number of modules (n)				
		5	**10**	**20**	**50**	**100**
Number of positions advanced (k)	**1**	6	11	21	51	101
	2	10	15	25	55	105
	5	20	25	35	65	115
	10	39	44	54	84	134
	50	188	193	203	233	283
	k^*	597	622	672	822	1072

Table 2: *Number of parallel steps run by the algorithm, as a function of the number of modules* (n) *and the number of positions* (k) *advanced by the robot strip, over a symmetric pyramid.*

Figure 38 illustrates the graph of the previous table.

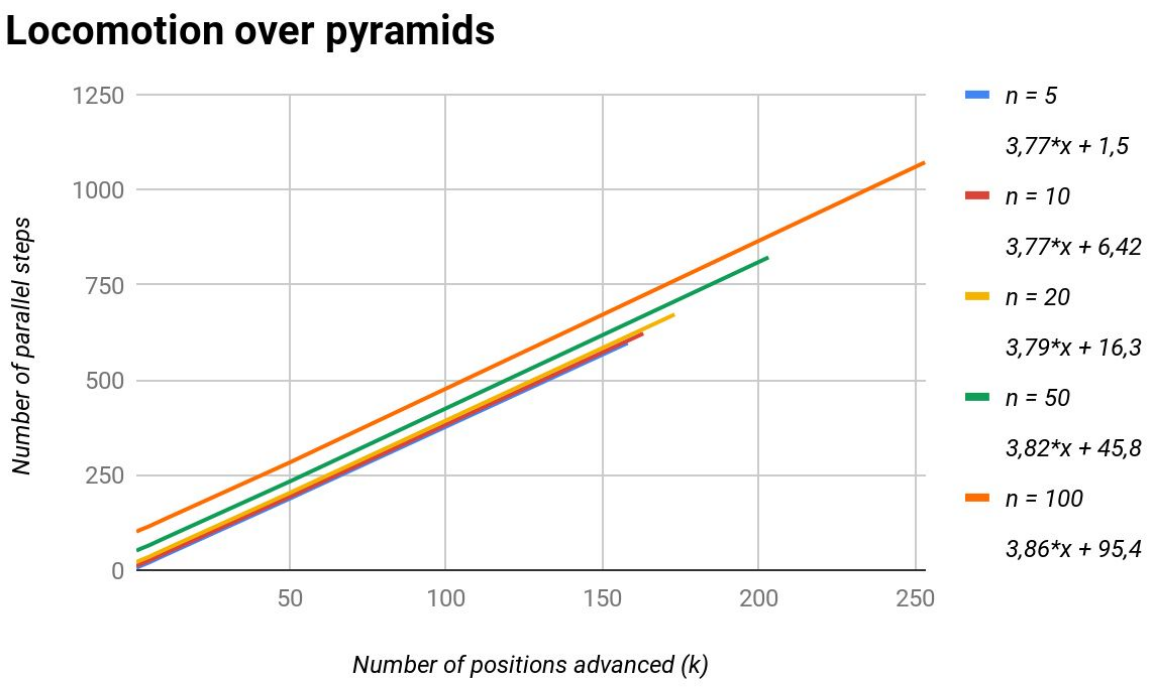

Figure 38: *Graphic of the number of parallel steps run by the algorithm, as a function of the number of modules* (n) *and the number of positions* (k) *advanced by the robot strip, over a symmetric pyramid.*

Here we observe that the slope almost does not increase when the number of modules is larger. Even so, the increase is very small. We conclude that the slope tends to 4 when $n \to \infty$. Therefore, we obtain that the number of rounds (parallel steps) is smaller than or equal to $n + 4k - 2$. That is, the more steps there are in a pyramid, then fewer rounds will have to pass for the strip to overcome the obstacle. Even so, this number is always greater than or equal to $n + 3k - 2$.

Therefore, we conclude that the algorithm for locomotion over pyramids is slightly faster than the one for locomotion in the absence of obstacles.

Finally, we run some tests with the set of rules for locomotion preventing collisions and deadlocks at bottlenecks. Our algorithm managed to overcome all the obstacles defined in the *Test Bank* section, including the ones illustrated in Figure 36.

Recall that for this case, In Chapter 6 we came up with an upper bound for the complexity that was clearly not tight. To obtain a better approximation to the complexity of this algorithm, we made some tests with three different obstacles. We set three small values of *n* that do not form a lot of deadlocks, and three other values of *n* large enough to obtain a great number of deadlocks. Also, we defined k^* to be the minimum number of steps needed to overcome the obstacle for each *n*. We did tests with $k = k^*$, $k^*/2$, $k^*/4$, $k^*/8$, $k^*/16$ and $k^*/32$. Table 3 and show the table and Figure 39 the graph of the results of the tests performed with the obstacle of Figure 29. In this case, $k^* = 32 + n$.

		Number of modules (*n*)					
		5	**10**	**20**	**50**	**70**	**90**
Number of positions advanced (*k*)	$[k^*/32]$	12	17	27	66	90	110
	$[k^*/16]$	21	26	40	74	97	120
	$[k^*/8]$	29	34	47	92	115	146
	$[k^*/4]$	43	52	68	126	160	194
	$[k^*/2]$	77	86	114	200	260	320
	k^*	147	172	222	374	478	578

Table 3: *Number of parallel steps run by the algorithm, as a function of the number of modules (*n*) and the number of positions (*k*) advanced by the robot strip, over the obstacle of Figure 29.*

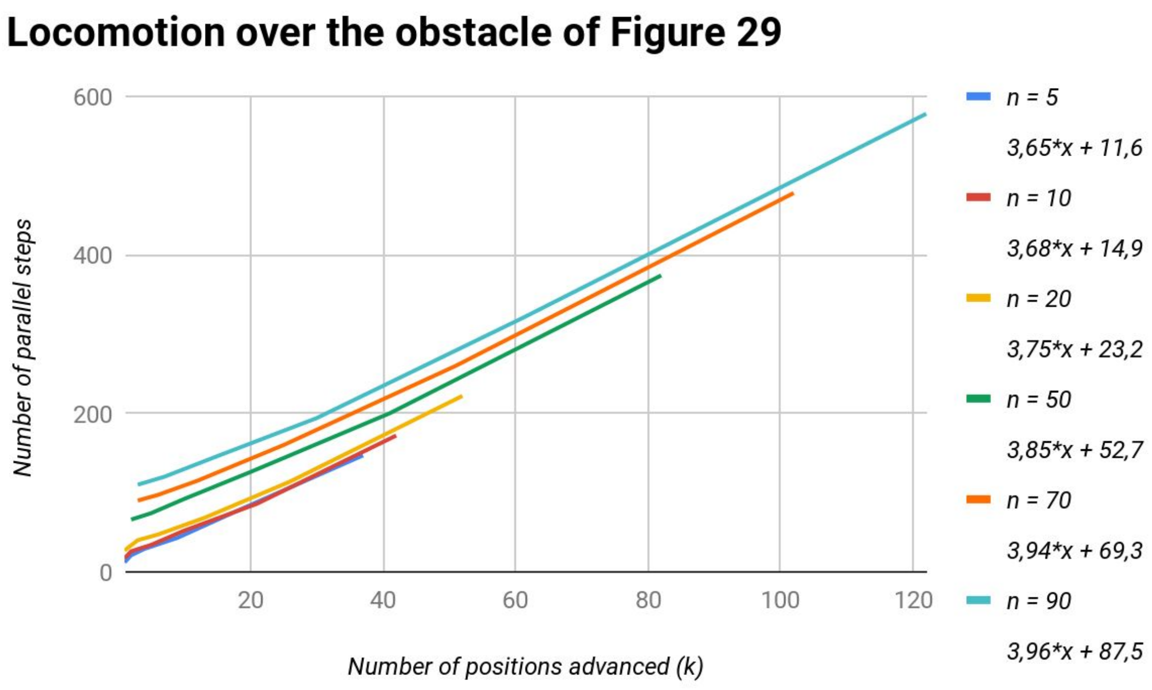

Figure 39: *Graphic of the number of parallel steps run by the algorithm, as a function of the number of modules (*n*) and the number of positions (*k*) advanced by the robot strip, over the obstacle of Figure 29.*

We can see that the difference between this test and the previous one is not decisive. In addition, the difference between the first three values of *n*, where no deadlock is formed, and the second, which do form deadlock, is not very broad. In fact, we obtain the same tendency as in locomotion over pyramids.

See the next test results in Table 4 and Figure 40. The obstacle used is the one illustrated in Figure 30. In this case, $k^* = 41 + n$.

		Number of modules (*n*)					
		5	**10**	**20**	**35**	**50**	**70**
Number of positions advanced (*k*)	[k^*/32]	12	17	27	42	57	84
	[k^*/16]	12	24	34	49	71	91
	[k^*/8]	26	31	48	70	92	119
	[k^*/4]	47	52	76	101	127	154
	[k^*/2]	86	94	109	161	208	268
	k^*	165	191	241	320	395	495

Table 4: *Number of parallel steps run by the algorithm, as a function of the number of modules* (n) *and the number of positions* (k) *advanced by the robot strip, over the obstacle of Figure 30.*

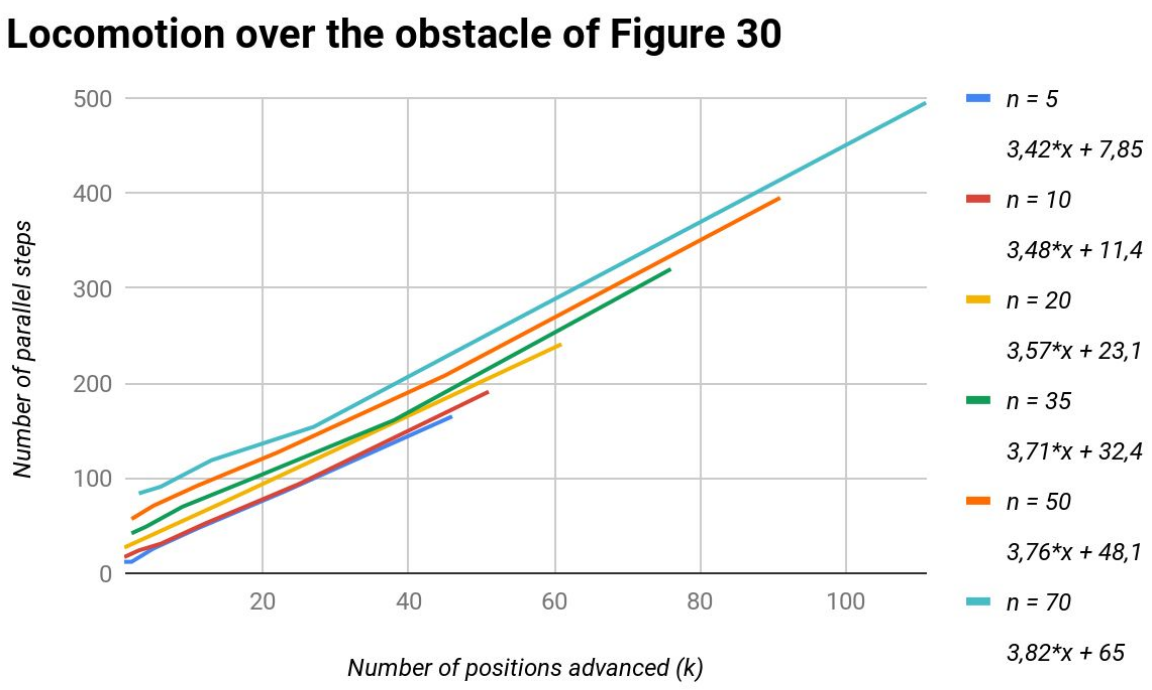

Figure 40: *Graphic of the number of parallel steps run by the algorithm, as a function of the number of modules* (n) *and the number of positions* (k) *advanced by the robot strip, over the obstacle of Figure 30.*

Locomotion over the obstacle of Figure 30 is faster than all the previous ones. The slope is between three and four, and increases as the number of modules increases. There is a small increase in the locomotion of the strip with few modules to when there are some more, but even so the algorithm quickly overcomes the obstacle.

See the last test, with the obstacle of Figure 33, in Table 5 and Figure 41. In this case, $k^* = 120 + n$.

		Number of modules (n)					
		5	**10**	**15**	**35**	**50**	**70**
Number of positions advance d (k)	$[k^*/32]$	9	18	23	43	62	82
	$[k^*/16]$	25	34	39	63	82	106
	$[k^*/8]$	57	69	74	105	127	151
	$[k^*/4]$	120	129	138	178	209	250
	$[k^*/2]$	252	263	276	334	378	428
	k^*	473	498	525	638	713	813

Table 5: *Number of parallel steps run by the algorithm, as a function of the number of modules* (n) *and the number of positions* (k) *advanced by the robot strip, over the obstacle of Figure 33.*

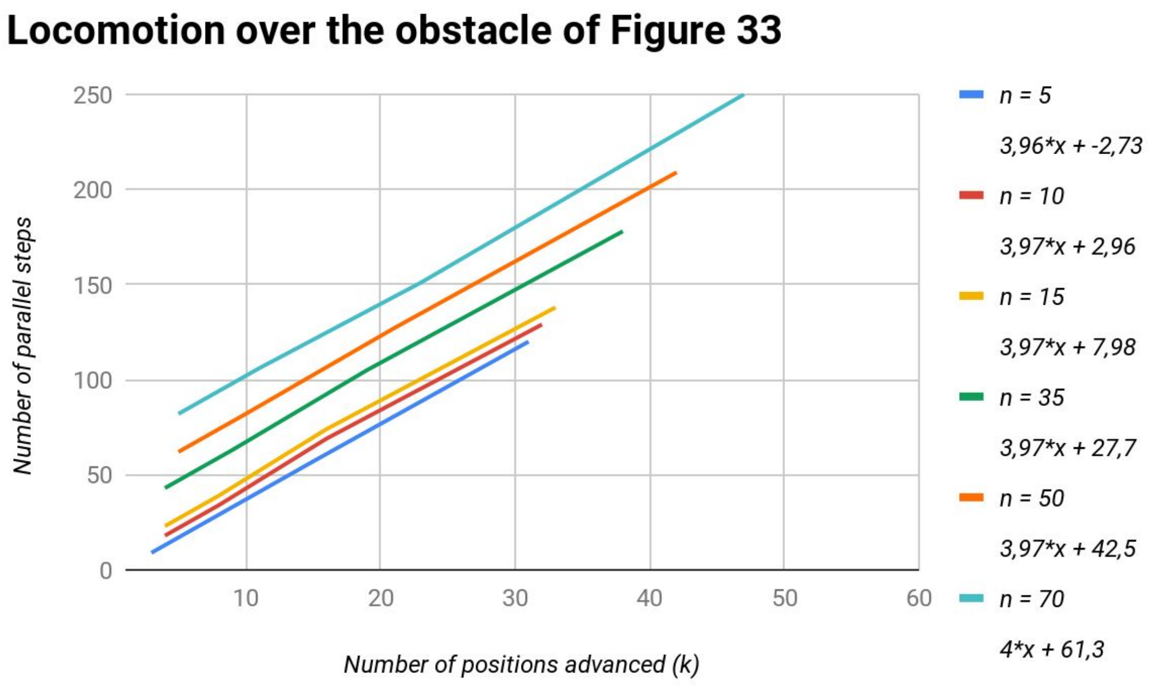

Figure 41: *Graphic of the number of parallel steps run by the algorithm, as a function of the number of modules* (n) *and the number of positions* (k) *advanced by the robot strip, over the obstacle of Figure 33.*

In the last test we observed that, regardless of whether deadlocks are formed, the number of rounds (parallel steps) increases when the number of modules n increases. In addition, the lines have a slope between three and four.

Therefore, we obtain that the complexity of our algorithm is $\Theta(n+k)$. Even more, we can limit the number of rounds between $n+3k-2$ and $n+4k-2$.

8.5. Downloading the rules

We stored our rules on one webpage. They can be downloaded in a zip file by clicking on the link [7]. The zip file contains the three sets of rules and all the obstacles with which we have tested them.

9. Conclusions

We have designed three sets of rules for a distributed algorithm to solve the locomotion problem of 2D square lattice sliding modular robots.

First, we obtained five rules that produce the endless advance of the robotic system from left to right on a horizontal surface, applying the right-hand rule. Then, we created a new set of rules for locomotion over pyramids. We proved the correctness of both sets of rules. Along the locomotion, the robotic system stays connected at all times, and no collisions happen.

The greatest challenge was to come up with a set of rules that achieved to solve locomotion preventing collisions and deadlocks at bottlenecks over arbitrary obstacles. We were able to perform many tests with the *AgentSystem* simulator and, in some way, prove that the distributed algorithm works. Moreover, in practice, the strip advances k positions after $\Theta(n+k)$ rounds or parallel steps.

The main further goal is to prove the correctness of the set of rules for locomotion preventing collisions and deadlocks at bottlenecks. In addition, the rules could be optimized. The preconditions to prevent collisions and establish a moving order can be cumbersome.

Moreover, a more exact complexity bound could be obtained, although we know it is linear in practice. This could facilitate a possible practical application of these robots with this distributed algorithm.

There are some extensions of our work that we would like to mention. An algorithm for hexagonal lattices could be obtained from our rules. This would be interesting since there exist several robot prototypes based on hexagonal lattices.

Another option would be to extend the model to 3D lattice-based modular robotic systems. Would the equivalent 3D sets of rules obtained, a practical application with cubic robots could be made. It could also be considered that the reticles had different fluids and the modules behavior was different in each of them.

In general, we believe that we have done a good job. We have achieved the goal of obtaining a distributed algorithm for locomotion over any obstacle. We have performed tests to prove how the rules work in practice. Even so, we have underestimated the time of designing the rules and, consequently, we have not been able to advance further in other directions.

We had difficulties in understanding exactly what the modular robotic systems are and all the vocabulary that surrounds it. In addition, the algorithm follows an order criterion that can sometimes slow down the strip advance. We believe that we could have thought about the strategy and the order criteria to further reduce the number of movements of the modules.

Finally, we think that the rules could be useful in real world application and we encourage the reader to think over it.

References

[1] Yim, M., Shen, W.-M., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., and Chirikjian, G. "Modular Self-Reconfigurable Robot Systems. Challenges and Opportunities for the Future." *IEEE Robotics & Automation Magazine*. March 2007: 2-11.

[2] Ahmadzadeh, H., and Masehian, E. "Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization." *Artificial Intelligence*. 223 (2015): 27-64.

[3] Butler, Z., Kotay, K., Rus, D., and Tomita, K. "Cellular automata for decentralized control of self-reconfigurable robots." *Workshop on Modular Robots at Robotics and Automation (ICRA), 2001 IEEE International Conference*. 2001: 21-26.

[4] Butler, Z., Kotay, K., Rus, D., and Tomita, K. "Generic Decentralized Control for Lattice-Based Self-Reconfigurable Robots." *The International Journal of Robotics Research*. 23.9 (2004): 919-937.

[5] Fitch, R., and Butler, Z. "Million Module March: Scalable Locomotion for Large Self-Reconfiguring Robots" *Workshop on Modular Robots at Robotics and Automation (ICRA), 2007 IEEE International Conference*. 2007: 2248-2253.

[6] Aichholzer, O., Hackl, T., Sacristán, V., Vogtenhuber, B., and Wallner, R. "Simulating distributed algorithms for lattice agents." Seville: *XV Spanish Meeting on Computational Geometry*, 2013.

[7] Salvador, Xavier. "Distributed locomotion of 2D lattice-based modular robotic systems." Gofile - File sharing platform, anonymous and free. Salvador, X., June 11, 2019. <https://gofile.io/?c=A9YgK1>.