Utrecht University

Master Thesis

Parallel Algorithms for Sliding Cubes

M.S. Wolters

supervised by
dr. Maarten Löffler
dr. Irene Parada
dr. Fabian Klute

second assessment
prof. dr. Marc van Kreveld

September 26, 2023

Abstract

In this thesis we formalise the *Parallel Sliding Cubes* model for modular robots. This model describes a theoretical framework for several prototypes of modular robots and the operations they can perform. The *Sliding Cubes* model was introduced 20 years ago and it is the best studied model for modular robots. The model formalised in this thesis expands it, to allow modules to move in parallel while keeping the configuration connected at all times. Reconfiguration is the main goal of modular robots. The reconfiguration problem asks for a schedule for the modules to transform from one shape to another. A parallel reconfiguration schedule tries to minimize the makespan, that is, the number of parallel moves. This thesis proposes two methods for parallelising moves during reconfiguration: a graph-based approach, applicable to a wide range of shapes; and a geometric approach which efficiently reconfigures between xy-monotone shapes. This algorithm was experimentally verified by comparing the number of moves it takes to reconfigure such shapes to a state-of-the-art reconfiguration algorithm. There are still many lines of open research and a number of suggestions are given at the end of this work to explore this field further.

Contents

1 Introduction **4**
1.1 A short overview of modular robots 5
1.2 Closely related work 8

2 Parallel Sliding Cubes Model **9**
2.1 Some preliminary definitions 9
2.2 Connectivity or the backbone property 10
2.3 Movement of a single block 11
2.3.1 Sliding over other blocks 11
2.3.2 Convex transition 12
2.4 Coordinated movement 12
2.4.1 Coordinated sliding move 13
2.4.2 Coordinated convex move 13
2.5 Avoiding collisions 14

3 Algorithm 1: Graph-Based Approach **16**
3.1 Important concepts 16
3.2 Defining the path graph 18
3.3 Finding the path 22

4 Algorithm 2: Geometric Approach for xy-Monotone Configurations **24**
4.1 Transforming one xy-monotone shape to another 25
4.2 Parallel reconfiguration of xy-monotone shapes 26

5 Results **31**
5.1 Experiments 31
5.2 Results 32

6 Discussion 35
6.1 Errors in the parallel $\boldsymbol{xy}$-monotone reconfiguration 35
6.2 Performance gains for parallel $\boldsymbol{xy}$-monotone reconfiguration . 37
6.3 Improving the graph-based approach 38

Chapter 1

Introduction

This thesis focuses on designing new, state-of-the-art algorithms for the reconfiguration of modular robots. Modular robots have the potential of being applied in many fields like space exploration and medical applications. Because the modular robots are small and can take many different shapes, they could be used to build infrastructure and for nano bot applications within a person's body. Another potential advantage due to their modularity and resilience due to redundancy is the ability to adapt to complex terrains, such as penetrating collapsed buildings or other confined spaces.

Contemporary research is, however, not at the point that these applications can be realised. Much theoretical work focuses on the reconfiguration problem, in order to change the shape of the configuration. More precisely, the (connected) reconfiguration problem is about connected sets of modular robots that make up a particular shape, attempting to change that shape, by moving modules over each other while keeping the shape connected. This problem has been studied for many models and prototypes and it is quite complex. In some cases, reconfiguration between two shapes is not even possible and in some pivoting models in which the modules rotate around each other the problem is PSPACE-complete [1]. For sliding squares, which is the model studied in this thesis, reconfiguration is always possible. However, minimising the number of moves in reconfiguration is NP-hard [2].

Looking at the reconfiguration problem from its geometric abstraction in the different theoretical models allows to tackle it using tools from computational geometry and combinatorics. By approaching the problem by analysing the shapes and underlying graph structure, one can get guarantees for the soundness of (reconfiguration) algorithms. These guarantees, however, might come at a performance cost in practice. That is why this

thesis attempts to approach the problem from two different angles.

Firstly, we will discuss the more heuristic approach of mapping a configuration of modular robots to a graph. Using this graph we search for paths that correspond to sets of modules that can be moved in parallel. This approach has the potential for significant performance gains over sequential algorithms or in general approaches that make use of canonical shapes. However, as we will discuss, this approach might get stuck as it does not guarantee sound reconfiguration.

Secondly, we will discuss the parallelisation of one step used in different reconfiguration algorithms like the sequential algorithm by Akitaya et al. [2] and the parallel algorithm by Fekete et al. [11]. These algorithms do not use a canonical shape in their reconfiguration, but a set of them, namely xy-monotone shapes. The definition for this class of shapes can be found in Chapter 4. This is an important class of shapes, as it is natural for reconfiguration algorithms make use of it. Therefore, any performance gains that can be realised in reconfiguring this class of shapes could be applied to many different approaches to reconfiguration.

While physical prototypes for many of the models described in the next section exist, it is important to mention that the modular robots we will be working with are theoretical. The goal is to design algorithms that could be applied by mechanical engineers, or even to inspire engineers to work within this model of modular robots.

After defining two algorithms for reconfiguration, we experimentally verify the results of the geometric approach. Finally, we discuss some directions for future research.

1.1 A short overview of modular robots

Modular robots are defined as a collection of independent robotic modules that each have the ability to connect to and disconnect from each other, and to move over the surface of adjacent modules. A collection of modular robots have historically been called a metamorphic robotic system [4]. This is to differentiate the modular robots we are discussing from industrial robots that have a modular aspect, such as removable tools. In this thesis we will be using the term modular robots rather than metamorphic systems, as that is the norm today. Modular robots are a subset of the more general *self-reconfigurable* or *self-organising* robotic systems. Modular robots have the additional properties that each module is (largely) identical in structure, motion and computation power. A modular robotic system can be viewed as

a swarm of interconnected robotic modules that collectively act as a single entity [4, 5]. The modular robots studied in this thesis are lattice-based and thus the modules can be viewed as regular polygons or polyhedra that form either two- or three-dimensional lattices [25].

There are many different prototypes and models for modular robots in research. The modules in each model can differ in many ways: size and shape, but also in movement type (e.g. *sliding* or *pivoting*), in the communication between modules (e.g. *centralised* or *distributed*), in the connectivity requirements (e.g. connected between moves or not), and in whether multiple modules can move at once or not (*parallel* vs *sequential* reconfiguration). In the following paragraphs we touch upon each of these differences.

The most common shapes for modules are: *Square* [2], *Hexagonal* [25], *Spherical* [21, 20], and *Expanding* [3]. These types of shape have different mechanisms, which are useful for different goals and applications.

There are two main abstractions for the movement of modules in modular robotics: *sliding* [2, 22, 19] and *pivoting* [1, 13]. Sliding means that the module slides over other modules and pivoting means that modules pivot around their own axis to achieve locomotion. See Figure 1.1.

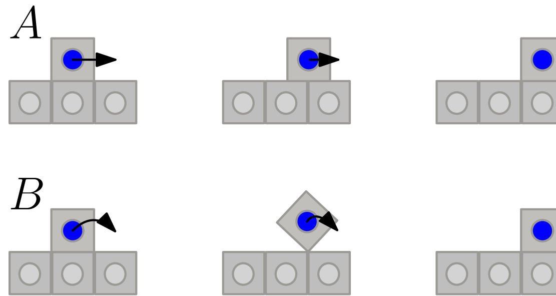


Figure 1.1: An example of a sliding (A) versus a pivoting model (B).

The last distinction to be made is whether the configuration is *strictly* connected or not. This has implications during reconfiguration, which is explained in the next section. Strictly connected means that when modules are moving, the rest of the configuration must be fully connected. This makes the configuration more robust during movement, but also restricts the potential moves that modules can make. It allows for articulation points to become real problems during the reconfiguration. These articulation points are modules that, if removed, disconnect the configuration. This is akin to the concept of cut nodes in a graph [24], see Figure 1.2. As such, the blocks that coincide with these articulation points are known as *cut-blocks*.

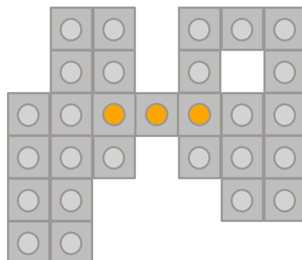


Figure 1.2: An example configuration with its articulation points marked in orange.

There is also a distinction to be made between *centralised* and *distributed* reconfiguration algorithms. Reconfiguration, in short, is the act of changing the shape of a configuration. Centralised algorithms refer to algorithms that define an all-knowing entity that keeps track of the configuration (this could be one of the modules or a separate computer). On the other hand, a distributed algorithm uses the distributed computing power of all the modules to construct the reconfiguration schedule. In this scenario, modules communicate with their neighbours to construct the schedule for the reconfiguration [19]. *Programmable matter* is a different distributed model in which the modules have very limited capabilities [15, 6, 10].

Finally, there is the difference between *parallel* and *sequential* algorithms. For some models of modular robots the only provably-correct algorithms for reconfiguration are sequential. Sequential algorithms for reconfiguration make schedules that move each module sequentially, i.e. one by one [2]. Parallel algorithms, on the other hand, make schedules such that more than one block can move at a time. It should be clear that this increases the complexity considerably, as one has to deal with collisions, crossing paths, and disconnecting the configuration. In previous work [11, 12] the connectivity requirements are weaker than the ones that we consider in this thesis. This allows them to use moves and strategies that do not translate to the sliding cubes model.

For the interested reader we suggest the survey on this topic by Thalamy et al. [23]. They give a comprehensive overview of the current state of modular robotics.

It is important to mention most of the models mentioned in this section have related physical prototypes. In this thesis we worked with a theoretical abstraction of one such a model rather than with physical prototypes.

1.2 Closely related work

A number of works are closely related to this thesis. The first paper to design a provably-correct algorithm for the sliding cubes model was by Dumitrescu & Pach [8]. They presented a reconfiguration algorithm for the sliding cubes model that achieved reconfiguration in $\mathcal{O}(n^2)$ moves. After the work by Dumitrescu & Pach, Akitaya et al. [2] showed a different sequential reconfiguration algorithm for the sliding cubes model, which achieves reconfiguration in $\mathcal{O}(Pn)$ moves, where P is the maximum perimeter of the two bounding boxes of the start and target configuration and n is the number of modules in the modules in the configuration. Furthermore, they proved that it is NP-hard to minimise the number of moves for a given pair of edge-connected configurations. The paper by Fekete et al. [11] discusses the problem of connected coordinated motion planning of a configuration. Their work focuses on a parallel model similar to the sliding cubes model but does not require the backbone property, defined in Chapter 2. This paper proves that it is NP-hard to decide whether a configuration can be reached in a particular number of parallel moves. Another recent paper presents a parameterised version of coordinated motion planning [9].

Chapter 2

Parallel Sliding Cubes Model

This chapter goes in to explicit detail on the movement model used in this research. It is based on the model used by Akitaya et al. [2], with some additions to allow for parallel movements. This model is known as the Sliding Cubes model, which was proposed by Fitch et al. [14]. In Chapter 1 we discussed different types of modular robotic models. Here, we only discuss the details of our model based on the work by Akitaya et al. [2] and Fitch et al. [14] and the extensions we have made to this previous work.

2.1 Some preliminary definitions

As mentioned in the previous chapter, modular robots are a collection of robots that move on a lattice. A connected collection and the particular shape that this collection makes up is called the *configuration* of the system. *Reconfiguration*, then, is the process of transforming one configuration into another. From this it follows that reconfiguration algorithms are algorithms designed to create a way for the modules to reconfigure. Usually, these algorithms create a *schedule* of moves for the modules.

As originally proposed by Fitch et al. [14], a single module is a cube, with connectors on each of the faces. They have some simple movement rules. Cubes can only make lateral sliding moves and convex transitions around other modules, which is explained inSection 2.3. It is relevant to mention we worked with this model in two dimensions, as working in three dimensions adds a layer of complexity which requires further research.

Single modules in a configuration are generally referred to as blocks throughout this work (due to the cube-like nature of the modules). A cell refers to a cell on the grid in the xy-plane, which can either be empty or

occupied by a block. Finally, a note on the types of connection that blocks can have between each other: a block is vertex-connected to its neighbours if they share a corner. Thus, blocks that are not orthogonal neighbours are *vertex-connected*. Blocks that are orthogonal neighbours of a block are called *edge-connected*. This is because these blocks share an edge in the grid on the xy-plane. Furthermore, we define a configuration to be *connected* if and only if a path can be traced from an arbitrary module to any other module through only their edge-connected neighbours. In other words, a connected configuration is a single set of modules and each module is edge-connected to at least one other module, that is also connected to at least one other module.

Figure 2.1: Left: The two blue modules are vertex-connected. Right: Edge-connected modules.

A *directional vector* points from the centre of the moving block to the centre of the target cell. There are eight distinct directional vectors possible, that correspond to the directions on a compass. These directional vectors can only point to cells that are either edge- or vertex-connected to a block.

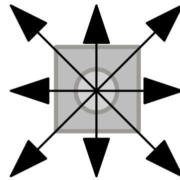


Figure 2.2: All possible directional vectors for a single block.

2.2 Connectivity or the backbone property

One of the fundamental properties of this model is that the stationary blocks in the configuration must stay connected at all times. More formally we can define this as follows: if the moving blocks are removed from the configuration then the remaining configuration (the *stationary* blocks) must be

connected. This is known as the *backbone property.* Moreover, when parallel moves are allowed, moving blocks cannot move over other moving blocks. This implies that the speed at which blocks move remains constant in the parallel model. It is important to note that all the moves defined in this chapter, and algorithms defined in the next chapters, take this into account.

2.3 Movement of a single block

As mentioned in the introduction to this chapter, there are two types of moves that blocks can make. The sliding move and the convex transition. This section discusses the details of each of these moves for a single block.

2.3.1 Sliding over other blocks

The sliding move is a lateral move where a block slides over its neighbours to reach a new position. In this section we are discussing the case of a single block moving. Thus, a block can only move to an unoccupied location. The one exception to this rule occurs when multiple blocks are moving. The so-called *chain-move* allows blocks to form a chain. More details can be found in Section 2.4. For now, the focus lies on a single moving block.

It should be noted that the moving block should have sufficient neighbours to support its move. A move is defined to be *supported* if the neighbours of the block allow it to make that particular move. Consequently, the *supporting neighbours* of a block are the neighbours that support a particular move. Supporting neighbours can only be direct neighbours of a block, that is, any neighbour to which a directional vector can point.

A block can make a sliding move to and from a cell if and only if the block is edge-connected to a (stationary) block in the cell it is leaving and the cell it is going to. Additionally, these neighbours must be parallel to the directional vector of the moving block.

In the example below (Figure 2.3) the block b with the blue center is moving east. The blocks with a grey center show the rest of the configuration. The movement direction is denoted by the vector v originating from the centre of the block. It should be clear that the grey blocks under the vector v are the supporting neighbours.

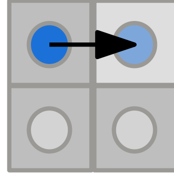


Figure 2.3: A sliding move.

2.3.2 Convex transition

Blocks can move around corners if and oly if both of the cells that the block moves through are empty, not only the target cell but also the intermediate cell. This move is know as a convex transition. Analogous to the sliding move, the convex transition requires sufficient neighbours to complete the move. The moving block must be edge-connected to the block it is moving around. We can see this in action in Figure 2.4 in the example below. The dark blue block is attempting to make a north-east convex transition. As in Figure 2.3, the directional vector is denoted by the black arrow. The blue arrow signifies the physical movement to be made by the block and, therefore, also shows the cells that need to be empty for the convex transition to take place.

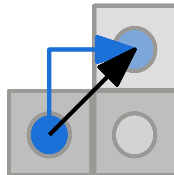


Figure 2.4: A convex transition.

In contrast to the sliding move, the convex transition does not need to transfer connection from the old neighbour to the new neighbour. This means that, while the distance of a convex transition is twice as long, the full timestep can be used to move the block. This allows the convex transition to take as long as a sliding move, and allows for synchronised movement in the parallel case. All moves take a single and full timestep.

2.4 Coordinated movement

The intention of this thesis is to do exploratory research into designing a parallel algorithm for the sliding cubes model. As such, it would be beneficial if

we can define rules for coordinated movement. The following section defines some rules to allow coordinated movement in the sliding cubes model.

By the constraints set in the original model a set of blocks can make a coordinated movement if and only if the whole set of blocks can move over a stationary set of blocks. A set of blocks can only make a collectively non-convex (sliding) move if the move is fully supported by the set of stationary blocks. Each individual block has a vector v_i that represents the direction of movement. A block may only be part of a coordinated set $\mathbf{C}$ if and only if the dot product of v_i and v_j is positive, where $j \in \mathbf{C}$ and $j \neq i$. This means that each block must be moving in the same direction as all the other blocks in the same chain. The full movement of a coordinated set of blocks is called a *chain move*.[1]

2.4.1 Coordinated sliding move

For an example of a coordinated sliding move one can look at Figure 2.5. It shows the supporting set of blocks (grey) that are necessary for a chain move (blue) to occur. This means that in the example below, the set of blue blocks can make the coordinated sliding move in configuration A but not in configuration B as the leading block in configuration B is no longer supported by stationary blocks.

Figure 2.5: An example of a coordinated sliding move.

2.4.2 Coordinated convex move

A convex coordinated move is defined as a coordinated move that happens around some other (stationary) block. For a simple example see Figure 2.6 below.

[1]We remark that the term *chain move* was used in previous work [2] to refer to a particular type or coordinated but sequential set of moves in the sliding cubes model.

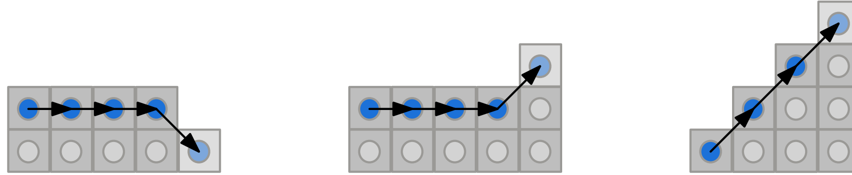


Figure 2.6: Some example of coordinated convex transitions.

2.5 Avoiding collisions

To avoid collisions between blocks we use the definitions that have been provided in the previous sections in this chapter. These definitions should not allow for any collisions to occur if they are respected. Specifically, the definition of moving blocks within a single timestep ensures that there is no coordination necessary between blocks that might be half way through a convex transition and a sliding move. That, coupled with the rule that a block only being part of a chain if and only if the dot product of the directional vectors is positive, means that collisions are avoided. A block can only intend to move to an occupied cell if they are part of the same chain move. As long as these rules are respected, collisions are avoided.

The main type of collision we would like to avoid is one where a block tries to occupy a cell that is not (fully) empty at the time of movement. An example where it is possible for a block to occupy a cell that is not empty at the time of moving is the coordinated sliding move. Say, a block b_1 wants to occupy a cell currently occupied by block b_2. As long as block b_2 starts its movement at the same time and in the same direction as block b_1 then no collision occurs. If block b_2 moves orthogonal to or in the opposite direction to block b_1 then it should be obvious that a collision will occur during the movement. Figure 2.7 clarifies the different possible cases.

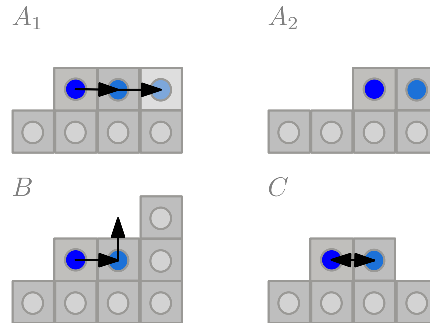


Figure 2.7: Two cases of block collision (B and C) and a successful transition of occupied cells (A_1 and A_2). Block b_1 is dark blue and block b_2 is light blue.

It should be clear that configuration A_1 can transform into configuration A_2 using a coordinated sliding move. Because the blocks belong to the same move iteration they start moving simultaneously and both move in the same direction. It should also be clear that no movement can take place in configuration C as both blocks are trying to move into each other. Finally in configuration B it should be clear that block b_1 (dark blue) can only start to move once block b_2 (light blue) has vacated the square it occupies at the start of its movement. In essence, this is what the dot product of the directional vectors captures. It is the alignment of the moves that allow or disallows the moves to be made.

Chapter 3

Algorithm 1: Graph-Based Approach

In this chapter we present a novel graph-based approach to reconfiguring a set of sliding cubes, using parallel moves, from a starting configuration to a target configuration. More precisely, we discuss a heuristic approach to parallel reconfiguration, using a graph representation and path-planning. This approach can also be integrated into existing sequential algorithms to parallelise moves and thus reduce the makespan of the plan.

3.1 Important concepts

This algorithm requires there to be at least an overlap in the perimeter of the start and target configuration. Whilst it is possible that there is no overlap at all between the start and target configurations in space, in practice this will not often be the case. One can also define intermediate target configurations to achieve overlap between the desired start and target configurations, if they do not have any overlap. It is also common to define the reconfiguration problem between shapes without a specific position in space. In such case, one can just define a position that produces an overlap.

The target locations for reconfiguration are found if one looks at the difference between the start configuration and the target configuration. *Target cells* are defined to be cells that contain a block in the target configuration but not in the start configuration. *Source blocks*, then, are cells or blocks that are present in the start configuration but not in the target configuration. Any other block is present in both the start and target configuration and these are called *shared blocks*. The cells that contain the shared blocks are

called *shared positions*. Figure 3.1 shows an example. In A the shared and source blocks are shown. B shows the same configuration with the target locations marked. Finally, C shows the perimeter locations marked. Note that the grid in the xy-plane has been included in this example for clarity.

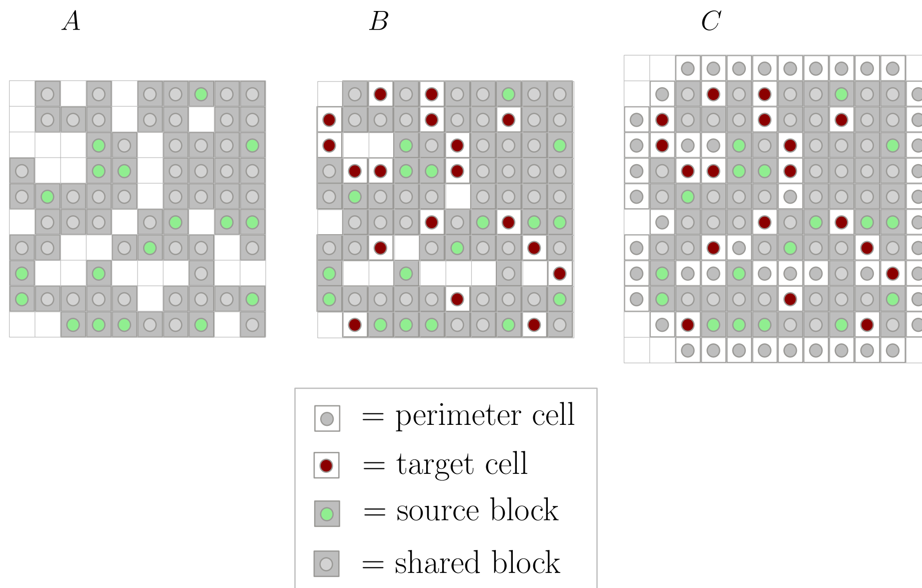


Figure 3.1: A: Source blocks (grey blocks/green centre). B: Target cells (empty blocks/red centre). C: Perimeter cells (empty blocks/grey centre).

From this definition it follows that all cells that contain a source block must be emptied and all cells that are a target cell must be filled by a block.

Another important definition is the perimeter of the configuration. The perimeter of the configuration is defined as the set of cells that do not contain a block but are edge-connected to any block in the configuration. This set of cells signify the set of locations that a block could move to using a sequential path planning approach.

We can now give the basic premise of our algorithm. If we can find a path through the configuration and/or around the perimeter, from any source blocks to any target cell, we are able to move blocks towards target cells and thus solve the reconfiguration problem. Issues could still arise if we **cannot** find a path from a source block to a target cell, which, for example, happens when the source block is a cut-block. Remember from the previous chapter that a cut-block is an articulation point in the configuration.

Using this basic premise we must define a way to find paths through and

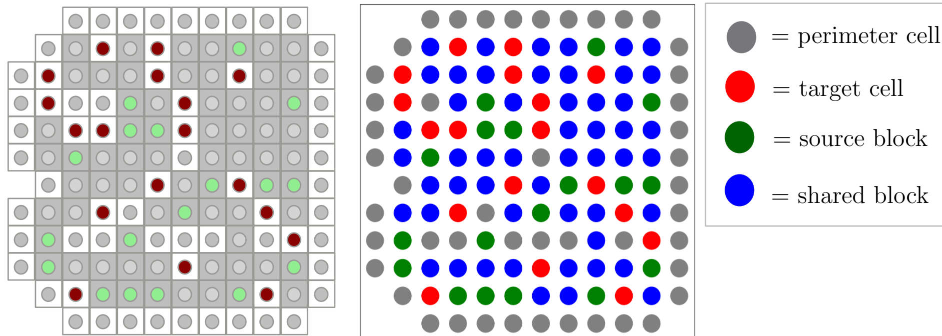


Figure 3.2: Relating the different nodes to the different types of blocks and cells.

around the configuration. We have opted for capturing the configuration in a graph because there are well defined path finding algorithms for graphs. Furthermore, it is an intuitive mapping as graphs preserve the topology of the configuration and we can use edges to define what moves could possibly be made. One of the difficulties with defining such a graph is trying to capture the possible chain moves that can be made. This is due to the fact that the connectivity of the graph is dependent on the current state of the graph and chain moves are dependent on the future state of the graph.

First, we define the graph and then we can discuss the potential problems that could arise and possible solutions for these problems.

3.2 Defining the path graph

Given a start configuration C_s and a target configuration C_t, the goal is to reconfigure C_s into C_t while not causing any collisions or disconnecting the graph at any point during the reconfiguration. A configuration can be represented by a graph $G = (V, E)$, where V is the set of vertices that represent the block, perimeter, and target locations on the xy-plane, and where E is the set of edges that contain all the possible moves that could take place. The set E is to be defined more precisely below.

The set V contains 2 classes of nodes, with each class having two distinct types. The first class is the *block* class. This class represents the set of blocks present in the configuration. Its distinct types are *shared* blocks and *source*

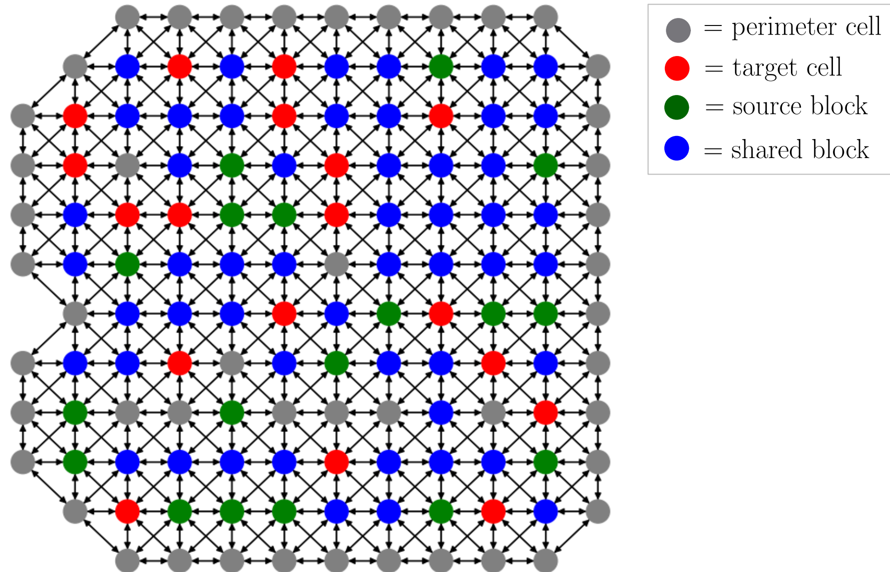


Figure 3.3: The adjacency graph

blocks. As defined above, the shared block's locations must be filled in the target configuration. However, during reconfiguration it might be necessary to empty these locations temporarily to allow a source block to move. The other type in the block class is the source block. Remember, this type of block *must* move in order to reconfigure C_s to C_t. The next class of nodes is the *perimeter* nodes. These nodes represent empty cells in the grid. They are either target cells or perimeter cells, that are edge-adjacent to occupied or target cells in the grid. The target cells are considered perimeter nodes because at the time of generating the graph these locations are empty. Since we are trying to find paths within this graph, perimeter nodes are considered to be *ghost* blocks. This means that when defining legal moves through these cells, they must adhere to all the rules a normal block would adhere to. This is because, at some point, they might be filled with a block and the path should not try to make moves in the future that would not be legal if that location was actually a block.

To define the set of edges E we start with the adjacency graph of the configuration. It should be noted that the edges of the graph we are defining are analogous to the directional vectors, mentioned in the previous chapter,

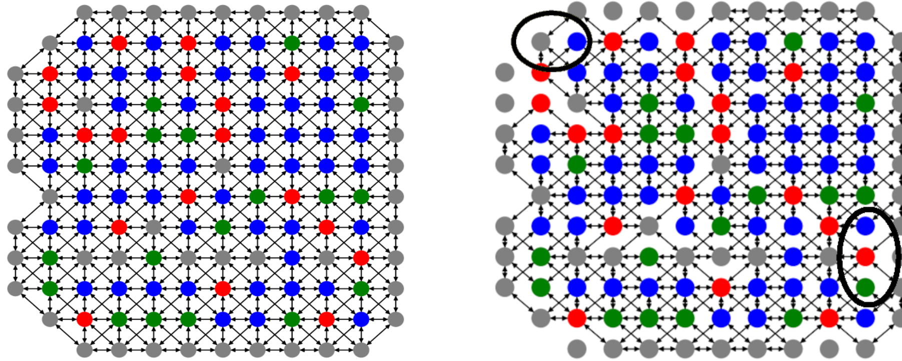


Figure 3.4: Transforming the adjacency graph with the first rule. Some interesting positions have been marked.

denoting the movement of a block. The edges of the adjacency graph are defined as follows; *a*) there exists a directed edge in the adjacency graph between each cell that contains either (*i*) a block, (*ii*) a target cell, or (*iii*) is part of the perimeter and *b*) this cell is either vertex or edge connected to another cell containing *i*, *ii*, or *iii*. Another way to define this is by looking at the set of nodes we obtain from the definition in the previous paragraph. If any two nodes are adjacent in the xy-plane then there is a directed edge between them. From this adjacency graph we can identify and remove edges that would either break movement rules or connectivity rules. Each of the edges that does not break any of these rules is able to be used for the path finding.

We now discuss each of the rules that affect the set of edges. The first rule that we want keep in mind is the movement rules for a single block. For a single block to move, it needs to have the correct neighbours to make this move possible. Therefore, we remove any edges from the adjacency graph that do not support the move. This rule holds for all types of nodes. This is checked in the following way. For each outgoing edge of a node, check if that edge has the supporting neighbours to allow the move. For the sliding moves, if one is evaluating, for example, a move east, we need to check that the original vertex has a neighbour to the north or south and then a neighbour to the east of that neighbour. This is analogous to all the other sliding moves (north, south and west). For convex transitions the process is similar. We only check if the convex transition has the correct neighbour to move around. So, for a north-west move, there needs to be either a block

to the west or to the north. It is important to note that in this step we are not taking blocked locations into account. If we did, we would be creating a graph that works in the sequential model, but would not allow us to find most of the chain moves we are interested in.

We will now discuss how to deal with blocked positions. Because these moves could be of potential interest for forming chain moves, we want to remove only the edges that strictly do not allow a chain move to be performed. One such a move that can never occur as a chain move is what we call a *captured* convex-transition. This is defined as a block that has both neighbours on either side of the convex transition and also has a neighbour in the target of the convex transition. While this move could be legal when looking at the movement vectors of each of the blocks, it is impossible in practice to remove all of the blocks in time for the convex transition to take place.

Any other blocked moves, could potentially form a chain move, thus we will not remove the edges from the graph.

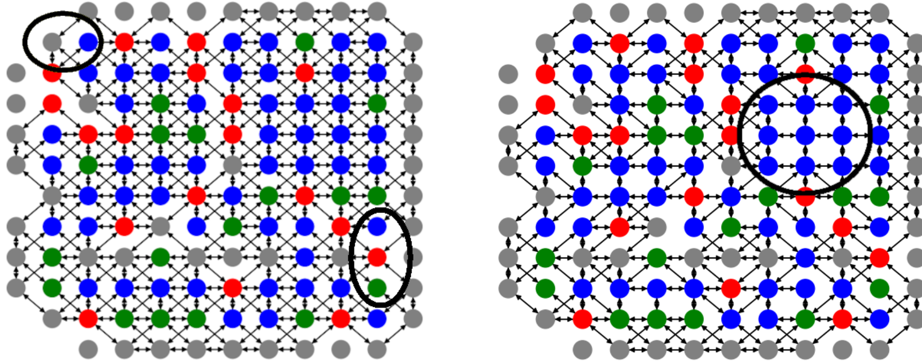


Figure 3.5: Transforming the adjacency graph with the second rule.

Finally, if a node is a cut-block, then it should have no outgoing edges, because in the current time step it cannot move. Additionally, *split-pairs*, which are pairs of nodes that disconnect the graph if they are both removed, should not share an edge. The problem with cut-blocks (or cut-nodes, for that matter) is that these nodes change based on the moves that are being made. So ideally, while searching the graph for a path, we would dynamically update the graph to reflect these changes. This approach is outside the scope of this thesis. It will be discussed in Chapter 6 Section 6.3.

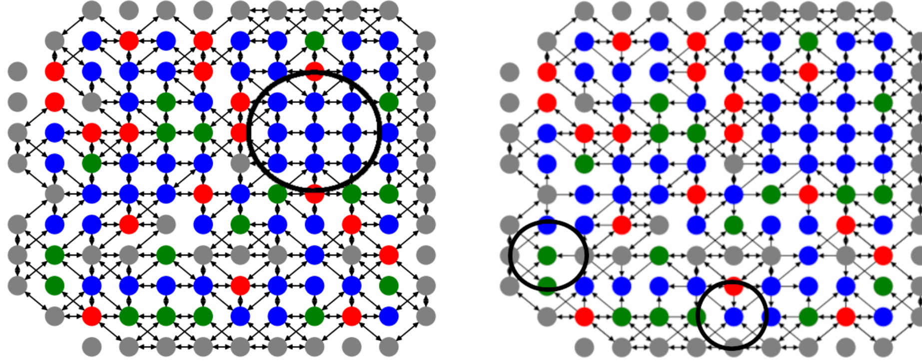


Figure 3.6: Transforming the adjacency graph with the third rule.

Figure 3.7 shows an example of the dynamic cut-block issue. It depicts a configuration and a chain move in that configuration. It should be clear that during the execution of the chain move, once the head of the chain starts moving, that one of the blocks along in the chain becomes a cut-block (marked with orange).

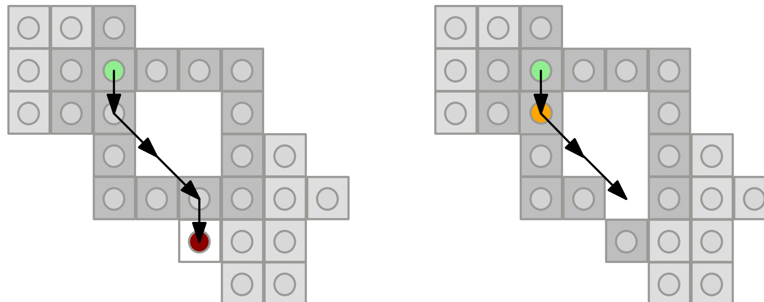


Figure 3.7: An example of the dynamic cut-block issue.

3.3 Finding the path

Now that the path graph has been sufficiently well defined, we can find paths within it. While this problem appears easy on the surface, it has some peculiarities that make it more difficult than one initially suspects. The foundation of this algorithm uses Dijkstra's algorithm [7] to find shortest paths between nodes. In our case we aim to find the shortest path that connects a source block to a target block. Because of the dynamic nature of cut-blocks,

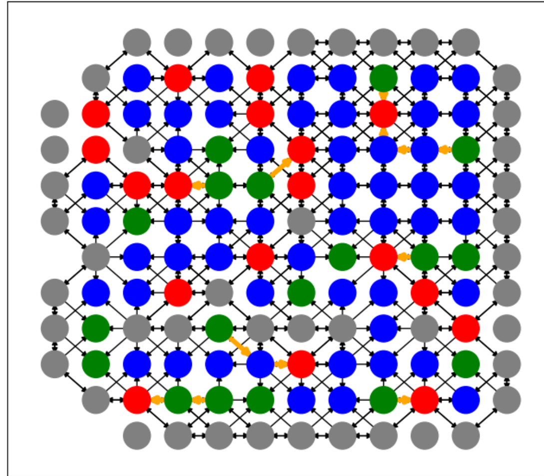


Figure 3.8: An example of possible paths (orange edges) found by the algorithm.

it is still entirely possible that we find a path that could disconnect the graph during execution. Thus, for each path we check if that is the case. If so, we split the path on the cut-block and turn the chain move into two or more chain moves. This guarantees that the path will never disconnect the graph.

This approach can solve many general reconfiguration instances. There are, however, two issues with this approach. Firstly, the final source block could be a cut-block. This means, due to the definition of the set of edges that no path from the source block to the final target cell can be found. Possible solutions to this issue exist, such as moving any block into the target cell until the source block is no longer a cut-block. Secondly, the algorithm could get stuck in a loop, such that the solution cannot be found. One solution for this problem could be similar to that of the previous issue, namely to move shared blocks into target location, rather than source blocks.

When taking these issues into account it should be clear that the reconfiguration problem is non-trivial. Taking a heuristic approach might lead to performance gains, but it does not guarantee solutions.

Chapter 4

Algorithm 2: Geometric Approach for *xy*-Monotone Configurations

This chapter discusses another approach to reconfiguration. As mentioned in previous chapters, the most common theoretical approach to reconfiguration is based on geometry and combinatorics. This makes sense as, when using this approach, we are more concerned with the actual properties of the shapes that the configuration takes. We try to use the properties of the shapes to give us guarantees that we do not have when using the heuristic graph-based approach. The downside of this approach is that to achieve such guarantees we might sacrifice performance. That, however, might not be of concern if the guarantees are more important. As such, we have tried to parallelise one of the steps of the sequential reconfiguration process, to increase its performance. This ensures that one can make use of the guarantees provided by the geometric approach, but can utilise (some) of the performance benefits of the parallel steps.

For this section we chose to parallelise the reconfiguration of one xy-monotone shape into another. First, let us define what an xy-monotone shape is: an xy-monotone shape is defined as a shape that for each column i, with $0 \leq i < k$, 0 being the first column and k being the final column, each next column j, with $i < j$, may be no higher than the column i. This creates tapered shapes such as the examples in Figure 4.1. The transforming of one xy-monotone shape into another is an important step in *Gather & Compact* [2], but also in the paper of Fekete et al. [11]. As mentioned in Chapter 1, the authors of *Gather & Compact* proved that one can transform

any configuration with an equal number of blocks into an xy-monotone shape and, as long as the algorithm is reversible, that it is possible to transform any xy-monotone shape into any other configuration with an equal number of blocks. The way they achieve this is by transforming the start configuration into an xy-monotone shape, then virtually transforming the target into an xy-monotone shape. Because of the reversibility of the algorithm one can then transform the xy-monotone shape achieved from the start configuration into the one achieved by transforming the target configuration and, finally, reverse the reconfiguration such that the target configuration is reached.

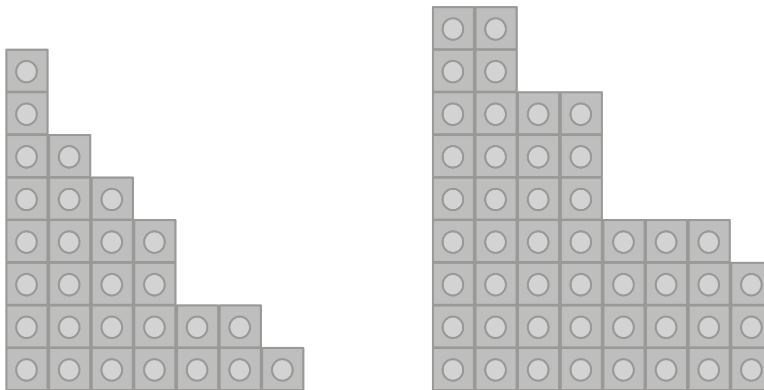


Figure 4.1: Some examples of xy-monotone shapes.

4.1 Transforming one xy-monotone shape to another

The advantage of using xy-monotone shapes is that they give us particular guarantees about where (source) blocks and target cells can or cannot be located. For example, if there is a source block at the end of a row of blocks, this guarantees that there cannot be a target cell in the next column. Because there is a source block in the preceding column, it means that that location is empty in the target configuration, due to the definition of an xy-monotone shape given in the previous paragraph.

In the paper by Akitaya et al. [2], transforming an xy-monotone shape into another is achieved by iteratively moving blocks from source cells to target cells. The order in which this is done is found by assigning a potential to each source block and target cell, based on their location in the xy-plane. The potential function being $x + y$, then the bottommost square

with maximum potential is matched (and moved) to the topmost target cell with minimum potential.

Using this idea as a basis we decided to use a similar tactic to transform xy-monotone shapes using parallel moves. The parallel move we decided to use was inspired by a paper by Fekete et al. [11]. In this paper they also transform xy-monotone shapes to other xy-monotone shapes during reconfiguration. The way they do that is with so-called *L-shaped moves*. These L-shaped moves consist of moving a linked row and column through the middle of the shape. However, in contrast to our model, the model used by Fekete et al. does not require the backbone property. As such, they can make any L-shaped move, without regard for the connectivity of the configuration. We need to add some additional constraints to make sure we respect the backbone property.

In principle we want to achieve the same movement as Fekete et al. but we need to make sure the shape does not disconnect. It should be clear that if a full row and column would make an L-shaped move, all the mass (i.e., the blocks) on the inside of the L-shaped move would be disconnected from the mass on the outside of the L-shaped move. Furthermore, using our definition of collision, the corner block would make a collision with either the row or the column that follows it (remember configuration B in Figure 2.7). To stop this from happening we can split the L-shaped move into two distinct moves. One for the column and one for the row. This ensures that the configuration is never disconnected. Using this definition we present the algorithm we defined.

4.2 Parallel reconfiguration of xy-monotone shapes

This section goes into detail on the algorithm for parallel reconfiguration of xy-monotone shapes. The algorithm consists of three steps to achieve reconfiguration. The first step is the filling of the boundary of the target shape. This is done sequentially and such that the xy-monotonicity is preserved. The next step is the parallel L-shaped moves to reconfigure the shape as close to the target shape as possible. This step makes a matching between source blocks and target cells, to ensure completion. Note, that in this step the boundary of the start configuration is left untouched. The final step of the algorithm is to sequentially empty the boundary of the shape to reach the complete target configuration. The reason for taking these steps is that we can guarantee connectivity and the appropriate neighbours for all L-shaped moves. In the next sections we provide more detail on these considerations.

1. Filling the boundary

Due to the connectivity constraints of our model it is preferable to have a completely filled boundary. The *boundary* of an xy-monotone shape is defined as the first row and column of the shape. Due to the definition of an xy-monotone shape we know that these sets of blocks (the first row and column) are at least as long as every other column or row in the shape. Therefore, if the boundary of the target configuration is (over) filled, we have guaranteed that each L-shaped move has sufficient neighbours for reconfiguration. In Figure 4.3 we can see an example of this in action. The orange box shows the maximal boundary column and the blue box shows the maximal boundary row. After this step in the algorithm the boundary of the target has at least been filled. Note that the boundary of the configuration does not yet need to be emptied.

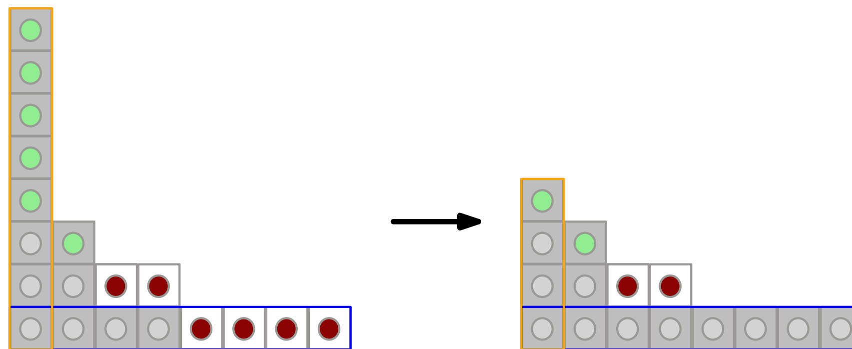


Figure 4.2: Maximal boundary of a configuration, given a start and target.

To fill the boundary such that the xy-monotonicity of the shape is preserved we must deal with three situations. Either the column boundary or the row boundary need to be filled, or they both need to be filled. First, we define *excess blocks*. Excess blocks are defined as blocks that make column (row) 1 higher than the next column (row).

In all cases of filling the approach is similar. To fill the column (row) boundary, we want to find the closest blocks that can be moved without breaking the xy-monotonicity of the shape. Start at column (row) 0, the boundary column (row). Then check the next column (row) 1 for excess blocks. Remember from the definition of xy-monotone shapes that preceding columns (rows) may not be lower than the current one. If the next column

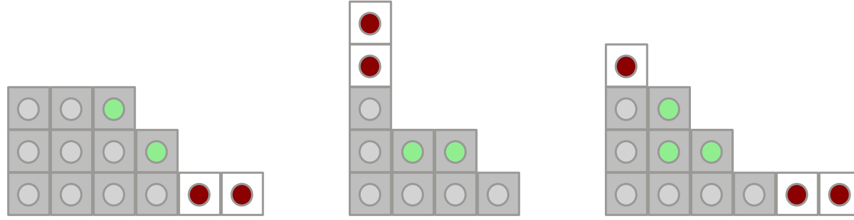


Figure 4.3: The three different situations when filling the boundary.

does not contain any excess blocks, move to the column after that. We can always fill the boundary column (row) because we can always remove the last column (row) of a shape to preserve the monotonicity. Furthermore, we can use the result by Akitaya et al. [2] to guarantee that this reconfiguration is always possible, seeing as we never break monotonicity. This method solves the cases where only the boundary column or row need to be filled. The only addition necessary for the case in which both the boundary column and row need to be filled is that, during the selection of candidate blocks to fill either boundary, we need to make sure that no block is selected for both operations.

2. Matching source and target blocks

To make parallel L-shaped moves possible, we need to define a way to allow the L-shaped moves to take place such that the configuration is not disconnected during reconfiguration. Making a matching between source blocks and target cells is a way to ensure that each source block is moved and each target cell is filled. To make this matching we want to use the properties of the xy-monotone shapes.

As mentioned previously, the reason for maximally filling the boundary is to ensure that all the L-shaped moves have the appropriate neighbours. The matching is made by assigning a source block to a target cell. The matching also contains the order in which the L-shaped moves need to be completed in order to reconfigure the shape correctly.

Firstly, we try and identify so-called *islands*. These are sections of an xy-monotone shape that contain an equal number of source blocks and target cells that all need to be moved in the same direction. We can identify these islands as if we are matching brackets. We can traverse the shape top-left (minimal x and maximal y in the xy-plane) to bottom-right (maximal x and minimal y). The first target cell or source block we come across functions as our opening bracket. Thus, for every cell of the same type (source block

or target cell) we come across we can add to the stack, and every cell of the opposite type we come across, we can pop from the stack. Note that shared blocks are not considered. As soon as the stack is empty, we have found an island of activity. These islands function independently from one another, as long as each island does not share any source blocks or target cells in the same row or column. This means that (most of the time) we could make L-shaped moves in islands simultaneously. However, we did not implement this functionality in our algorithm and this is, therefore, left as future work.

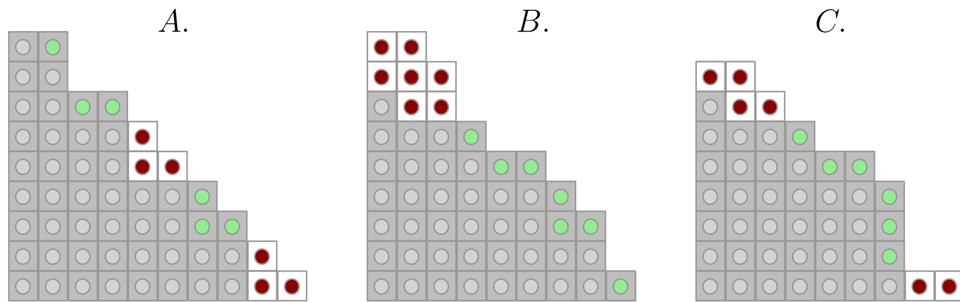


Figure 4.4: Different island types, showing left- and right-flow and mixed-flow configurations.

It is also important to note that there are two different kinds of islands. We have left flow islands and right flow islands. This denotes whether the source blocks need to move in the positive or negative x direction. Islands can only have one type of flow at any one time. This is because, for each opening bracket, there will always be a closing bracket until the stack is empty. Only then can the flow of blocks change direction.

Once we have found all of the islands, we can start matching. The matching works as follows: we differentiate between left- and right-flow islands. The matching for the left-flow islands works by assigning a potential $(x + y)$ to each of the source blocks. Then we sort the source blocks based on this potential. Furthermore, we subsort the source blocks on their negative row number (i.e. their y value). Then we take the target cells sorted by their row number first and their column number second. We match the source blocks and target cells based on this sorting. Thus, the first source block after sorting is matched to the first target cell after sorting. The right-flow islands are matched in a similar manner. The source blocks are sorted by their potential first and their positive row number. In this case the target cells are sorted first by their column and secondly by their row values. This gives us a matching such that we can reconfigure xy-monotone shapes using

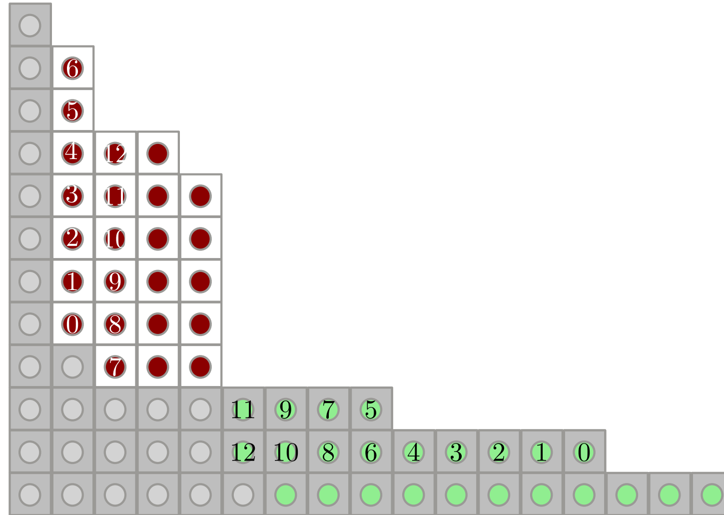


Figure 4.5: An example matching.

parallel moves.

3. Emptying the boundary

Emptying the boundary is the reverse process of filling the boundary. Even more so, it is the same process as the original sequential reconfiguration algorithm for xy-monotone shapes. The reason being that we never break the xy-monotonicity once a step in our algorithm has been completed. Thus, the source blocks can be moved to the target cells using the original algorithm by Akitaya et al. [2].

This concludes the description of the algorithm.

Chapter 5

Results

This chapter presents the results from the experiments run on the matching algorithm. The experiments were run on a laptop with an AMD Ryzen 7 4800H and 16 GB of RAM. The parallel sliding cubes model was implemented using Python (version 3.10), using NumPy [17], Matplotlib [18] and NetworkX [16] as additional packages for visualisation and additional features. The source code can be found on GitHub.[1]

Due to time constraints no experiments could be run for the graph-based approach. There is an implementation of the algorithm in the repository on GitHub, which is freely accessible. This chapter focuses on the experiments done to evaluate the geometric approach to transforming xy-monotone shapes.

5.1 Experiments

We compared our implementation of the algorithm described in Chapter 4 with the implementation of *Gather & Compact* [2]. To facilitate the experiments, a number of different xy-monotone shapes were randomly generated for different numbers of blocks. We tested 1000 different configurations for configurations with 10, 25, 50, and 100 blocks. Furthermore, we tested 100 different configurations with 200 blocks, 10 configurations with 500 blocks, and 1 configuration with 1000 blocks.

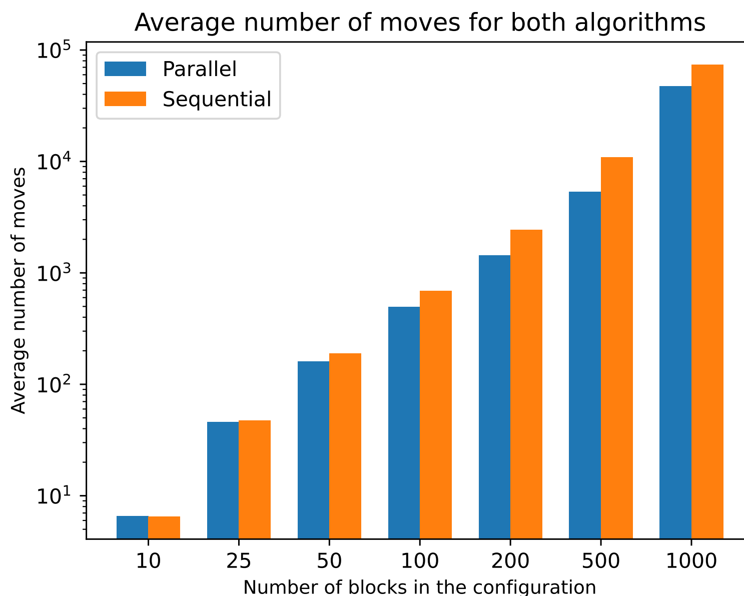


Figure 5.1: The average number of moves for the parallel and sequential algorithms.

5.2 Results

Figure 5.1 shows the main result. It plots the average number of moves in each set of different block numbers for each algorithm. The y-axis in this plot is logarithmic, because the number of moves increases so much that all detail is lost in the linear axis plot. It should be noted that the sequential model outperforms the parallel algorithm in the case with 10 blocks, the average for the sequential moves (in the 10 block experiment) is 6.51 and for the parallel algorithm it is 6.534. This is most likely the case because the parallel algorithm focuses on filling the boundary first, with less considerations for where the source blocks are. Whereas the sequential algorithm is only focusing on moving source blocks to target locations.

The important trend, however, is to note the performance gains for larger shapes. This makes sense, because the perimeter of the shape makes up less of the volume of the shape the larger the shape gets. Consequently, the second stage of the parallel algorithm can more significantly decrease the number of total moves. Another important thing to note is that the parallel

[1] https://github.com/Saeden/parallel-squares

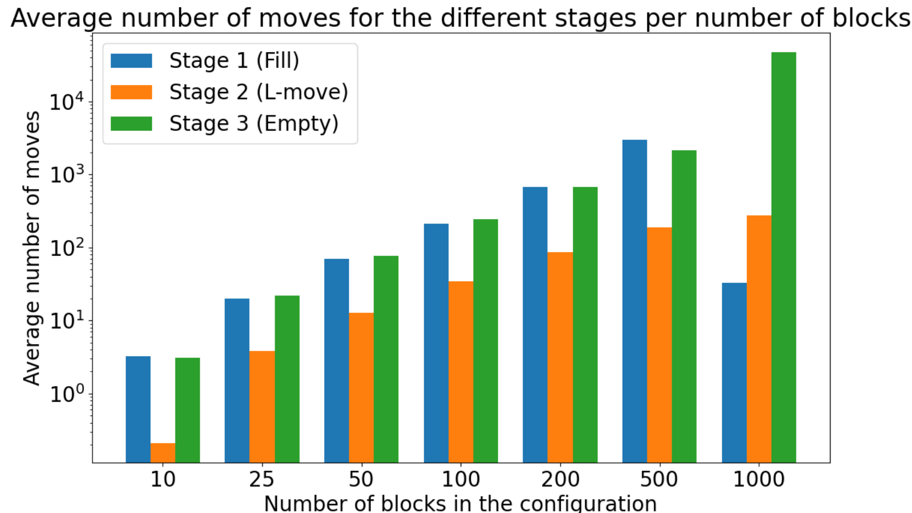


Figure 5.2: Different stages of the parallel algorithm

algorithm performs no better than the sequential algorithm in the worst case. The worst case is transforming a single column (or row) of blocks to a single row (column). In this case the parallel algorithm only ever makes sequential moves, because the shape is its own boundary, and only the first stage of the algorithm is performed. Using the proof for the sequential algorithm we can see that it proves that the sequential algorithm performs its reconfiguration in $\mathcal{O}(Pn)$ moves, with P the size of the perimeter of the shape. From this it follows that in the worst case we perform $\mathcal{O}(n^2)$ moves, as the size of the perimeter *is* the number of blocks.

Figure 5.2, shows the difference in number of moves between the different stages of the parallel algorithm. Once again the y-axis of this plot is logarithmic to preserve the details. This figure should make clear that the low number of L-shaped moves is the reason for the performance gain depicted in Figure 5.1. Furthermore, it shows where the algorithm is constrained in its performance. If we can find a way to parallelise the first and third stage (filling and emptying the boundary respectively), significant performance gains should be possible. Possible areas of research related to this are discussed in the next chapter.

Figure 5.3 shows the error rate for our algorithm. During the experiments we discovered that every once in a while the parallel algorithm would throw a collision error. The error rate stays under 1% for all different exper-

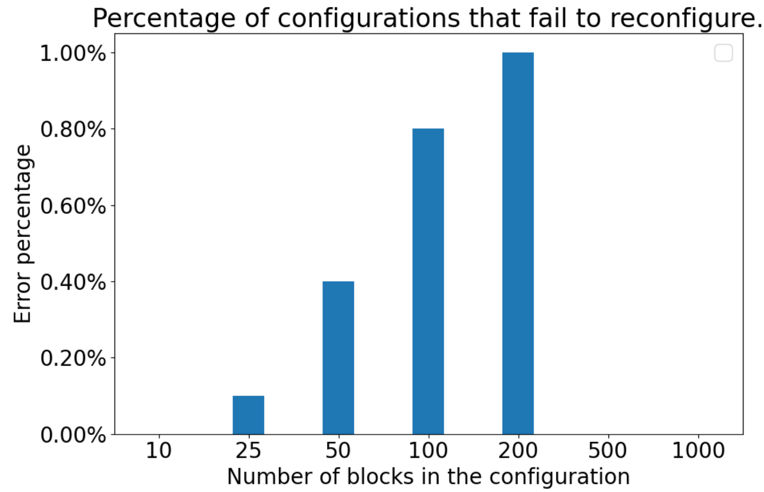


Figure 5.3: Percentage of configurations that threw an error

iments. The possible explanations for these errors will be discussed in the next chapter. It should be noted that due to the discrepancy of the amount of configurations tested per block amount there is some variance in the error rate. That is why, for the sizes larger than 200, there is no error rate. The sizes of these experiments were not sufficient to run into many configurations that gave errors. Even the error rate for the configurations with 200 blocks is influenced by the size of the experiment.

Chapter 6

Discussion

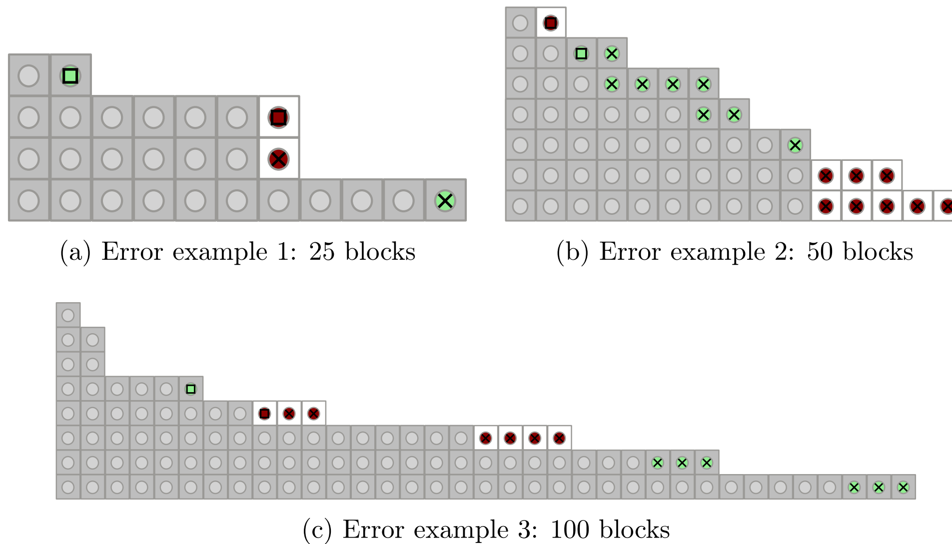
This chapter goes into more detail on a number of different aspects of this research. First, we discuss the error rates in the parallel xy-monotone reconfigurations. Then, we discuss possible performance gains for the parallel xy-monotone algorithm. Finally, we discuss potential areas of research for the graph-based approach to general reconfiguration.

6.1 Errors in the parallel xy-monotone reconfiguration

As was mentioned at the end of Chapter 5 it was discovered during the experimental phase that our reconfiguration algorithm does not correctly deal with some edge cases. There are particular cases that currently return either a collision error (which means our algorithm tried to move a block to an occupied location) or an invalid neighbours error (which means our algorithm tried to move a block but the neighbours were not valid to make the move).

Firstly, it should be mentioned that in a practical implementation setting this should not create issues, as we can always fall back to the sequential algorithm. As long as we make sure the shape is xy-monotone (by moving all blocks without a left neighbour to the left, except the blocks in the first column), we can reconfigure it using the sequential algorithm.

Below a number of examples of configurations that return an error can be found. Using these examples we attempt to explain what the causes of the errors are and what could be done to mitigate them in the future. Due to the previously mentioned time constraints we were unable to implement these changes. Hence, we are discussing them here.



(a) Error example 1: 25 blocks

(b) Error example 2: 50 blocks

(c) Error example 3: 100 blocks

Figure 6.1: Three examples of configurations that threw an error. Once again the source blocks are marked green and target cells marked red.

These examples depict the start configurations and include the target cells and source blocks for clarity.

The astute reader might notice a pattern among these examples. They all include more than one island. The islands have been marked with a cross or a box. This is the source of the errors, as it turns out that in particular cases islands are dependent upon each other for correct completion. The figure with 25 blocks (Figure 6.1a gives us the clearest example). Following the algorithm, we can see that filling the boundary is not necessary, so we find ourselves in the second stage of the algorithm. There are two islands, with one containing a boundary source block. This is the source of the issue. For the island on the left to complete its L-shaped move, the island containing a boundary block must first complete its reconfiguration. Our algorithm does not deal with this case properly, thus an error is thrown.

The second example has the opposite problem. Due to the island on the left completing its move first, there are not sufficient neighbours for the island on the right to complete all its L-shaped moves.

The final example throws a collision error, because it tries to move a block into an occupied location. The assumption is that due to the edge cases above not being considered during the design of the algorithm, there is an error in the islands retrieval and/or source block to target cell matching, that causes this error.

To make this algorithm sound and complete, one would have to take a look at islands that are dependent on each other. There should be some ordering of these islands such that the L-shaped moves can be completed without error. This is an intriguing area for future research.

6.2 Performance gains for parallel xy-monotone reconfiguration

As one can see from Figure 5.2 in Chapter 5, the second stage of our parallel xy-monotone reconfiguration algorithm is where the real performance gains occur. As such, if one wanted to increase the performance of this algorithm, optimising those stages is one of the key places to look. This section discusses two ideas that could be applied to making the performance better.

Firstly, one could attempt a completely geometric approach to this problem. This should give the best performance gains, coupled with the best guarantees. The difficulty of such an approach is making the matching work properly, and dealing with multiple islands. One important thing to note is that an L-shaped move is possible on the boundary, using one extra move. This is necessary due to the collision and connection rules defined for this model. If one were to make an L-shaped move using one part of the boundary, then the corner block needs to move separately. This is because the configuration would disconnect if a whole row or column, including a boundary block, makes a chain move.

To make this approach work, one could look into making a matching for the first and third stages of the algorithm. This might be easier than the second approach, which is to lose the idea of the three stages and make a different matching altogether, that makes use of the boundary L-shaped moves.

Secondly, one could combine the graph-based approach with the current algorithm. Seeing as the graph-based approach makes use of parallel moves, there should be performance gains in the general case, if it is applied to the first and third stage. One of the drawbacks is the lack of guarantees, due to the heuristic approach of the graph-based approach, but one could always fall back to the sequential algorithm should one run into trouble. This should be the simpler approach of the two.

6.3 Improving the graph-based approach

There is also much potential for performance gains and better guarantees for the graph-based approach. In our algorithm, we focused on one aspect of parallelisation, which was making the longest chain possible. We felt that it was best to focus on one thing and later maybe expand to multiple chains per iteration. One could obviously take the opposite approach. Make the smallest chains possible, but move as many chains as possible during an iteration. This could be done by focusing on a BFS approach to finding legal moves. Rather than try and plot a path from a source block to a target cell, one could try and plot a path from a target cell to a source block. This approach might be beneficial as it focuses on filling the target cells, which moves them after each iteration.

Another approach could be to dynamically update the graph. Because the connectivity of the configuration is highly dependent on the local connections in the configuration, it is not easy to plot a path through the configuration, that adheres to our connectivity rules. One way to deal with this is to dynamically update the graph, as one is plotting a path. This should ensure that no illegal paths are ever found. One could even try to combine this idea with the idea mentioned in the previous paragraph.

Furthermore, additional research is necessary to define better heuristics for choosing a path. As we have noticed, making greedy choices works up until a point. But it is necessary to be able to deal with paths that inadvertently make the algorithm get stuck in a loop or a local optima.

We conjecture that it is possible to create a graph-based algorithm that is complete, if one can define a more detailed metric of completion and define a way to break out of local optima.

Bibliography

[1] Hugo A. Akitaya, Erik D. Demaine, Andrei Gonczi, Dylan H. Hendrickson, Adam Hesterberg, Matias Korman, Oliver Korten, Jayson Lynch, Irene Parada, and Vera Sacristán. Characterizing universal reconfigurability of modular pivoting robots. In *Proc. of the 37th International Symposium on Computational Geometry (SoCG'21)*, volume 189 of *LIPIcs*, pages 10:1–10:20. LZI, 2021. doi:10.4230/LIPIcs.SoCG.2021.10.

[2] Hugo A. Akitaya, Erik D. Demaine, Matias Korman, Irina Kostitsyna, Irene Parada, Willem Sonke, Bettina Speckmann, Ryuhei Uehara, and Jules Wulms. Compacting squares: Input-sensitive in-place reconfiguration of sliding squares. In *Proc. of the 18th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT'22)*, volume 227 of *LIPIcs*, pages 4:1–4:19. LZI, 2022. doi:10.4230/LIPIcs.SWAT.2022.4.

[3] Lillian Chin, Max Burns, Gregory Xie, and Daniela Rus. Flipper-style locomotion through strong expanding modular robots. *IEEE Robotics and Automation Letters*, 8(2):528–535, 2023. doi:10.1109/LRA.2022.3227872.

[4] Gregory S. Chirikjian. Kinematics of a metamorphic robotic system. In *Proc. of the 1994 International Conference on Robotics and Automation (ICRA'94)*, pages 449–455. IEEE, 1994. doi:10.1109/ROBOT.1994.351256.

[5] Gregory S. Chirikjian, Amit Pamecha, and Imme Ebert-Uphoff. Evaluating efficiency of self-reconfiguration in a class of modular robots. *Journal of Field Robotics*, 13(5):317–338, 1996. doi:10.1002/(SICI)1097-4563(199605)13:5<317::AID-ROB5>3.0.CO;2-T.

[6] Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. Amoebot-a new model

for programmable matter. In *Proc. of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'14)*, pages 220–222, 2014. doi:10.1145/2612669.2612712.

[7] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.

[8] Adrian Dumitrescu and János Pach. Pushing squares around. In *Proc. of the 20th ACM Symposium on Computational Geometry, (SoCG'04)*, pages 116–123. ACM, 2004. doi:10.1145/997817.997838.

[9] Eduard Eiben, Robert Ganian, and Iyad Kanj. The parameterized complexity of coordinated motion planning. In *Proc. of the 39th International Symposium on Computational Geometry (SoCG'23)*, volume 258 of *LIPIcs*, pages 28:1–28:16. LZI, 2023. doi:10.4230/LIPIcs.SoCG.2023.28.

[10] Sándor P Fekete, Robert Gmyr, Sabrina Hugo, Phillip Keldenich, Christian Scheffer, and Arne Schmidt. CADbots : Algorithmic Aspects of Manipulating Programmable Matter with Finite Automata. *Algorithmica*, 83(1):387–412, 2021. doi:10.1007/s00453-020-00761-z.

[11] Sándor P. Fekete, Phillip Keldenich, Ramin Kosfeld, Christian Rieck, and Christian Scheffer. Connected coordinated motion planning with bounded stretch. In *Proc. of the 32nd International Symposium on Algorithms and Computation (ISAAC'21)*, volume 212 of *LIPIcs*, pages 9:1–9:16. LZI, 2021. doi:10.4230/LIPIcs.ISAAC.2021.9.

[12] Sándor P. Fekete, Eike Niehs, Christian Scheffer, and Arne Schmidt. Connected reconfiguration of lattice-based cellular structures by finite-memory robots. In *Proc. of the 16th International Symposium on Algorithmics of Wireless Networks (ALGOSENSORS'20)*, volume 12503 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2020. doi:10.1007/978-3-030-62401-9_5.

[13] Daniel Feshbach and Cynthia Sung. Reconfiguring non-convex holes in pivoting modular cube robots. *IEEE Robotics and Automation Letters*, 6(4):6701–6708, 2021. doi:10.1109/LRA.2021.3095030.

[14] Robert Fitch, Zack J. Butler, and Daniela Rus. Reconfiguration planning for heterogeneous self-reconfiguring robots. In *Proc. of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*

(IROS'03), pages 2460–2467. IEEE, 2003. doi:10.1109/IROS.2003.1249239.

[15] Seth Copen Goldstein, Jason D Campbell, and Todd C Mowry. Programmable matter. *Computer*, 38(6):99–101, 2005. doi:10.1109/MC.2005.198.

[16] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In *Proc. of the 7th Python in Science Conference (SCIPY'08)*, pages 11 – 15, 2008. URL: https://networkx.org/documentation/stable/.

[17] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi:10.1038/s41586-020-2649-2.

[18] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. URL: https://matplotlib.org/stable/, doi:10.1109/MCSE.2007.55.

[19] Ferran Hurtado, Enrique Molina, Suneeta Ramaswami, and Vera Sacristán Adinolfi. Distributed reconfiguration of 2d lattice-based modular robotic systems. *Autonomous Robots*, 38(4):383–413, 2015. doi:10.1007/s10514-015-9421-8.

[20] Florian Pescher, Nils Napp, Benoît Piranda, and Julien Bourgeois. Gapcod: A generic assembly planner by constrained disassembly. In *Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'20)*, pages 1028–1036. AAMAS, 2020. doi:10.5555/3398761.3398881.

[21] Benoît Piranda and Julien Bourgeois. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Autonomous Robots*, 42(8):1619–1633, 2018. doi:10.1007/s10514-018-9710-0.

[22] Benoît Piranda, Guillaume J. Laurent, Julien Bourgeois, Cédric Clévy, Sebastian Möbes, and Nadine Le Fort-Piat. A new concept of planar self-reconfigurable modular robot for conveying microparts. *Mechatronics*, 23(7):906–915, 2013. doi:10.1016/j.mechatronics.2013.08.009.

[23] Pierre Thalamy, Benoît Piranda, and Julien Bourgeois. A survey of autonomous self-reconfiguration methods for robot-based programmable matter. *Robotics and Autonomous Systems*, 120, 2019. doi:10.1016/j.robot.2019.07.012.

[24] Liang Tian, Amir Bashan, Da Ning Shi, and Yang Yu Liu. Articulation points in complex networks. *Nature Communications*, 8:1–9, 2017. doi:10.1038/ncomms14223.

[25] Jennifer E. Walter, Elizabeth M. Tsai, and Nancy M. Amato. Algorithms for fast concurrent reconfiguration of hexagonal metamorphic robots. *IEEE Transactions on Robotics*, 21(4):621–631, 2005. doi:10.1109/TRO.2004.842325.