INTERLOCKING STRUCTURE DESIGN AND ASSEMBLY

Dartmouth Computer Science Technical Report TR2019-874

by

Yinan Zhang

DARTMOUTH COLLEGE

Hanover, New Hampshire

September 2019

Abstract

Many objects in our life are not manufactured as whole rigid pieces. Instead, smaller components are made to be later assembled into larger structures. Chairs are assembled from wooden pieces, cabins are made of logs, and buildings are constructed from bricks. These components are commonly designed by many iterations of human thinking. In this report, we will look at a few problems related to interlocking components design and assembly.

Given an atomic object, how can we design a package that holds the object firmly without a gap in-between? How many pieces should the package be partitioned into? How can we assemble/extract each piece? We will attack this problem by first looking at the lower bound on the number of pieces, then at the upper bound. Afterwards, we will propose a practical algorithm for designing these packages.

We also explore a special kind of interlocking structure which has only one or a small number of movable pieces. For example, a burr puzzle. We will design a few blocks with joints whose combination can be assembled into almost any voxelized 3D model. Our blocks require very simple motions to be assembled, enabling robotic assembly. As proof of concept, we also develop a robot system to assemble the blocks.

In some extreme conditions where construction components are small, controlling each component individually is impossible. We will discuss an option using global controls. These global controls can be from gravity or magnetic fields. We show that in some special cases where the small units form a rectangular matrix, rearrangement can be done in a small space following a technique similar to bubble sort algorithm.

Acknowledgments

I would like to thank my advisor Prof. Devin Balkcom for guiding, encouraging and supporting me over the years. Thank you for selflessly providing your wisdom in many aspects. Thank you for your patience in educating an ignorant student. And thank you for generously giving your time and discussing problems in details.

I also acknowledge the discussions, ideas, and feedback from my committee members: Prof. Devin Balkcom, Prof. Jeff Trinkle, Prof. Alberto Quattrini Li, and Prof. Xia Zhou.

I am grateful to the big Dartmouth RLab family: Weifu Wang, Yu-Han Lyu, Evan Honnold, Monika Roznere, Sam Lensgraf, Amy Sniffen, Luyang Zhao, Qijia Shao, Zhao Tian, and Kizito Masaba. Thank you for helping me with the research presentations, paper writing, and many hours of discussions.

I am grateful to Haopeng Zhang and Geoffrey Hsuan-Chieh Huang, who helped build 3d models of blocks, built robot grippers and recorded videos. The authors also thank Jeremy Betz for useful insights on the geometry of joints. Thanks to Adam Arnold, Yotto Koga and Hui Li of Autodesk for their insights of 3D printing, interlocking structures, as well as their help in design of the physical robot assembly system, and to Autodesk for the use of the lab space at Pier 9, San Francisco, CA.

This project received seed funding from the National Science Foundation (NSF RI: Computational joinery, IIS-1813043), the Dubai Future Foundation (MBR Space Settlement Challenge) and the Neukom Institute (Neukom Comp-X grant: Computational design of deployable structures).

Contents

Chapter 1

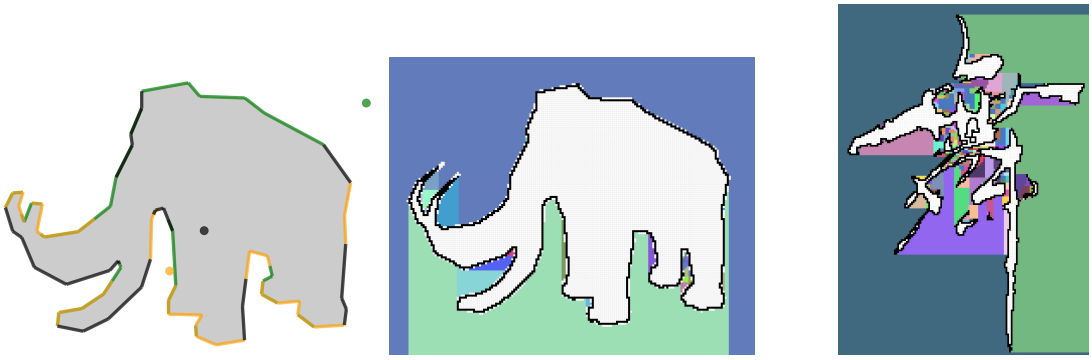
Introduction

This thesis will explore some problems related to assemblable structure design and planning.

The first problem we want to solve is related to package design. We want to design a package that encloses an object tightly such that there is no gap between the two parts. The package has to be partitioned into no less than two pieces, and the problem is to derive the shape of each piece and also generate the packing/extraction motions for each peice. Figure 1.1b is a 2D example package that encloses a mammoth. Each piece is extractable without colliding the held object. We will attack the problem by looking at the lower-bound (Figure 1.1a) and the upper-bound (Figure 1.1b) on the number of pieces. Then provide a practical algorithm for designing such a package. Figure 1.1c is a package designed to hold a character.

Our second task is to design a few simple components that can be assembled into many kinds of rigid structures. We designed two kinds of blocks with joints. See Figure 1.2a and 1.2b. Using these blocks, we can assemble almost any voxelized model such that there are only a few movable pieces. The ultimate goal is to enable robotic assembly. As a proof of concept, we also implemented a robotic construction example using a system of two robot arms. Figure 1.2c is the result of such assembly example.

Finally, we will consider a planning problem in an environment with limited controls.

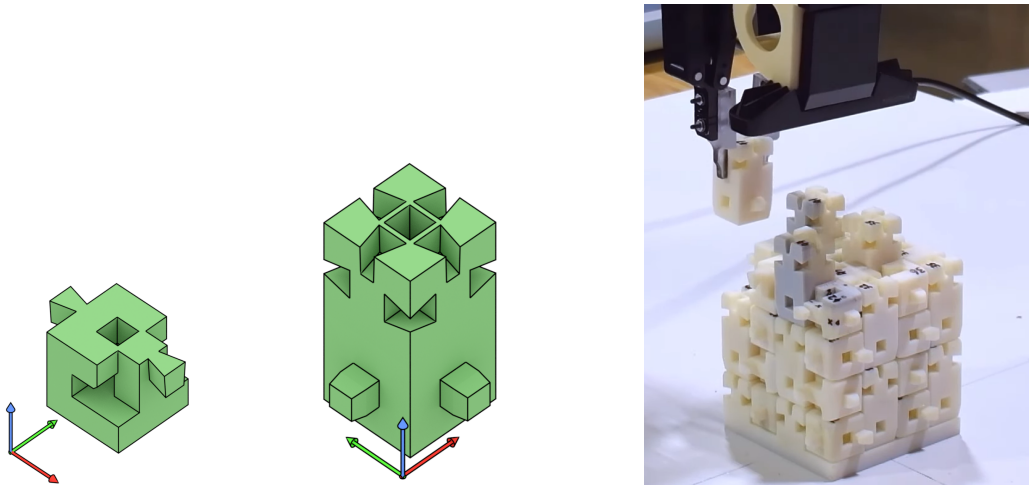


(a) Minimum 4 pieces are needed for the package.

(b) One package design for holding a 2D mammoth.

(c) Packaging a 2D Chinese character.

Figure 1.1: Designing 2D packages to hold an untouchable object (gray) inside.



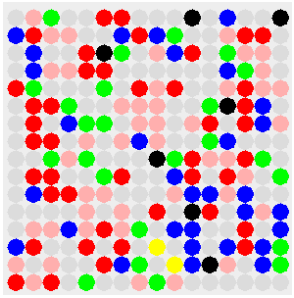
(a) A short block with joints.

(b) A tall block with joints.

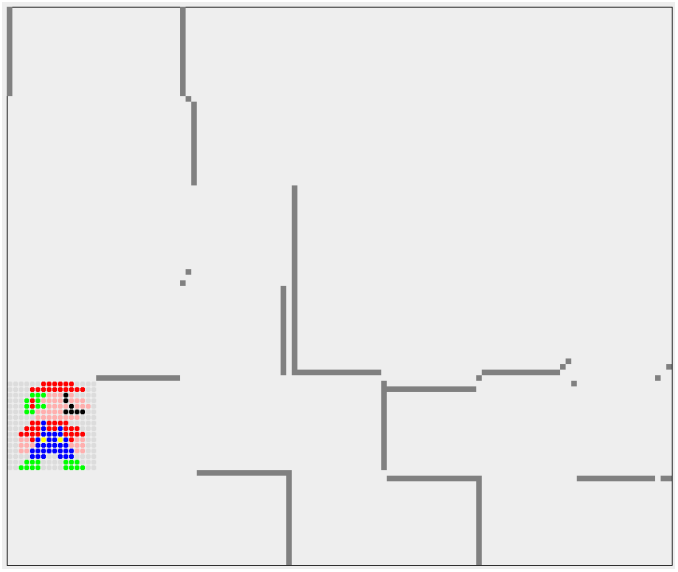
(c) A cube-like structure assembled by a robot arm.

Figure 1.2: Interlocking blocks and a structure assembled by robots.

Given a set of small square objects sitting in a grid space, how can we reconfigure the position of each object using global controls? The objects can only move in one of four orthogonal directions parallel to x- or y-axis in a single step. Our controls are limited to four directions: *up, down, left* and *right*. Objects will stop moving when hitting an obstacle or another still object. We will design a space with obstacles that allows reconfiguring a rectangular matrix of objects from one arrangement to another. We will show that this process can be done in a small space. Figure 1.3 showns a result from this chapter. A board is designed for rearranging a matrix of random points to form a super mario image.



(a) The matrix was initially randomly arranged.



(b) A board capable of rearranging a matrix of small robots. The result is a Super Mario image.

Figure 1.3: Rearranging a matrix of small robots using global controls. The matrix was initial random, and the board transforms the robots into an image of Super Mario.

Section 1.1

Motivation

Many objects in our daily life are not manufactured as whole rigid pieces. Instead, smaller components are made to be later assembled into a larger structure. For example, our cell phones are assembled from chips, a screen, a shield and many other components. Figure 1.4 shows some components of an iPhone 6s. Similarly, chairs are assembled from wooden pieces, cabins are made of logs, and buildings are constructed from bricks. The components of

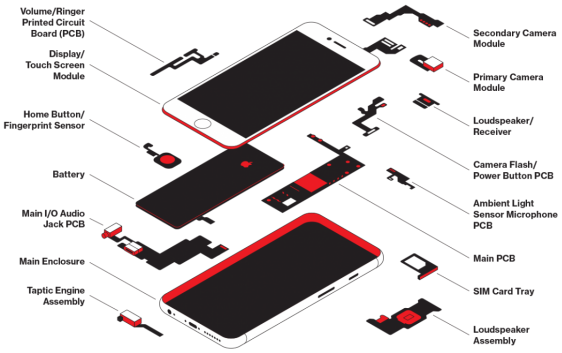


Figure 1.4: A cellphone is assembled from hundreds of small components. [38] How are each component designed? Can we simplify the shapes of components? How is each part assembled?

these structures are commonly designed to fit together by human and assembled by hand. The automation of the process of designing parts and assembly motions is a very interesting field and is constantly being explored by researchers.

One may have many questions when studying Figure 1.4. How is each part designed? Why are some components in various shapes? Are there any constraints to consider when designing parts? Can we reduce the number of components? How is each component assembled?

In this first chapter, we will raise some challenges in designing interlocking parts. Additionally, we will also address the scientific questions backed behind the challenges and why these questions are of interest from a robotics perspective. We then introduce our contributions and the structure of the thesis.

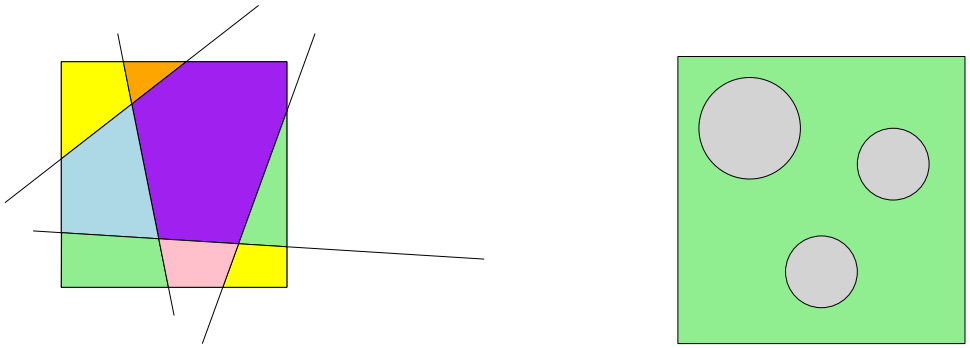
Section 1.2

Challenges

The idea of dividing a large complex structure into many small, individual components provides us many benefits. Smaller parts are easier to manufacture massively, reducing the cost of the target assembly. In an iPhone case, a camera module is about 1/3-inch size and costs around $20 to $25, while a regular 35mm image sensor could easily cost hundreds of dollars. Individual parts are made by different companies based on their strength. This allows for the overall improvement in quality of each component. Larger parts are more likely to be affected by small material defectives, degrading the whole object to be unusable.

1.2.1. Assembly piece design and planning

A very basic question is, given an object, how can we divide it into smaller assemblable components? How can we design pieces of an assembly? The problem seems trivial at first glance. One can arbitrary cut an object into arbitrarily many pieces using straight lines.



(a) A structure randomly cut into pieces using straight lines. Each piece can be disassembled.

(b) How can we design assembly components with untouchable (gray) zones?

Figure 1.5: Designing assembly components without constraints is easy. A challenging question is how can we solve the design problem under constraints?

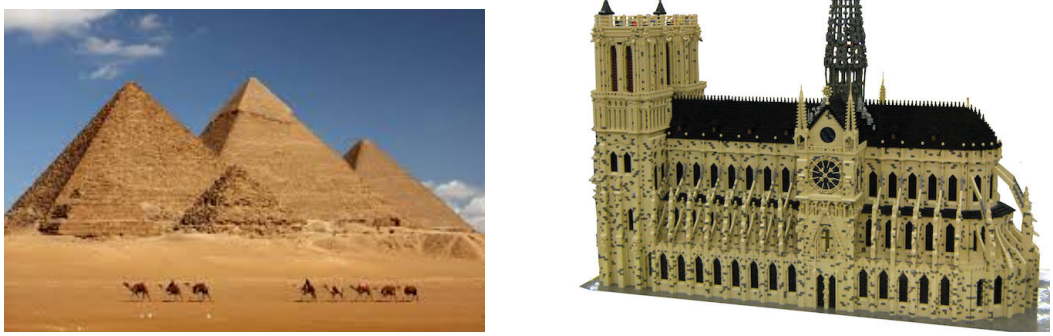
Then each piece is guaranteed to be disassemblable (Figure 1.5a). Each time we make a straight cut, the parts on two sides of the cut can always move along the normal directions of the line to infinity. However, if we impose a small constraint on the object, the problem gets harder. For example, if we require that some parts of the target object cannot be cut, how can we design components? See Figure 1.5b.

The untouchable area may be caused by another object the structure has to enclose, by a pre-existing substructure, or simply by the fact that the material is not cuttable.These design problems arise quite frequently in our lives. How should styrofoam packaging be assembled to support a delicate object for transportation? How can we cut cut rubble into smaller, movable pieces to allow us to rescue a person underneath? How many pieces a model ship must be broken into in order to be re-assembled in a bottle?

In this thesis, I explore the problem of designing assembling components under constraints.

1.2.2. General purpose component design

As different structures need different components to be assembled, our hardware stores are full of all kinds of parts. Is it possible to select only a few components and lay them out to form any arbitrary shape? Fewer components reduce the complexity of searching the right



(a) Pyramids in Egypt were built using rocks of similar sizes.

(b) A lego model of Notre Dame de Paris built using a few kinds of lego blocks.

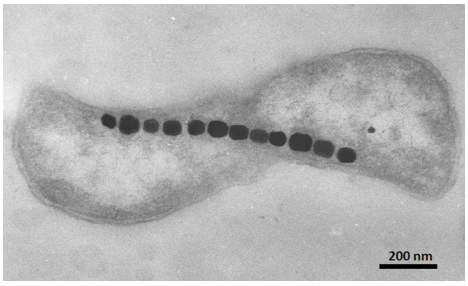
Figure 1.6: Two examples of structures built using only a few types of construction components.

parts to finish assembling a structure. When there are too many choices, some components are used very rarely and the average manufacturing cost for such components are likely to be high. Thus, reducing the kinds of components also reduces the overall cost.

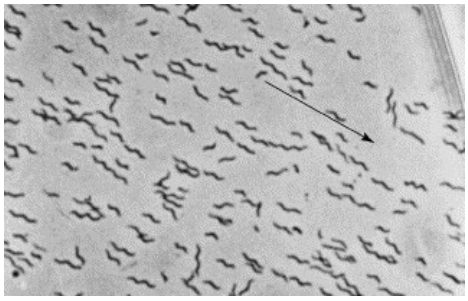
Due to the clear convenience of limiting the types of required construction elements, humans aim for designing structures with simpler elements that also enhances overall structural complexity. For example, most buildings are constructed using bricks; kids use Lego bricks to build various toys and statues; and wood logs are stacked together to form cabins. But using small components also implies a large quantity to form a target shape, making the assembly a boring and time-consuming job. Can we have robots perform the repetitive work and construct it for us? If so, what kinds of components are easy for robots to handle? One of the most basic features of a robot is the ability to move an object. Can we have some components that require only rigid body transformations to be assembled? To further reduce the cost of the components, we hope that they are re-usable. An unwanted structure can be disassembled and its components be re-used to build new structures. Cement and nails make this feature very difficult, so hopefully, our desired component can connect to each other based on its geometry instead of chemistry.

We will explore designs of such components. We will see two kinds of blocks that can be connected based on their geometry and be assembled into many structures. Without the use of extra connectors, our blocks are re-usable, and thus environment-friendly.

1.2.3. Assembly under global control constraints



(a) Image of a magnetotactic bacterium. [103]



(b) Magnetotactic bacteria aligned. [39]

Figure 1.7: An example of entities controlled by global signals. Magnetotactic bacteria orient themselves along the magnetic field lines

During structure assembly, each piece is typically assembled separately. For robots to perform the assembly, the path for each piece must be computed. The assumption behind is that the movement of each piece does not depend on the other pieces. This assumption can be broken when the sizes of the components are decreased to some extreme cases. For example, when the components are as small as sand, it is unlikely to have a gripper capable of grasping such small pieces and impractically to move sands one-by-one.

One way to solve the problem of moving small objects is to use global controls. A global control can be viewed as a command that once triggered, all the parts or particles must follow until stopped by obstacles or when a balance configuration is reached. The global controls can be from gravity, magnetic field, or chemical reactions. For example, paramecium, a creature extremely sensitive to slight changes, moves against the high-temperature water [42]; Magnetotactic bacteria orient themselves along the magnetic field lines [18]. See Figure 1.7. Following the same idea, we can assemble nanoscale structures using global control signals. The last chapter of this thesis discusses how some 2D matrix of small particles can be rearranged under these global controls.

Section 1.3

Key contributions

We are not the first one to study the concept of designing parts for assembly. Most of the previous work consider "assembly" as a single action of translation (rigid body motion) along one direction. We relaxed the constraint and consider a piece to be assemblable if there exists a collision-free path from infinity to a goal configuration. This relaxation makes our problem more difficult, while also more interesting. In many cases, it is not possible to find a single direction to translate a component under constraints, but a feasible path for assembly does exist.

We propose the concept of physical complexity as a measure of the difficulty for designing assemblable structures under constraints. Just like the time and space complexity for analyzing the difficulty of an algorithm, our physical complexity, which is measured by the lower-bound and the upper-bound on the number of pieces, serves as a fundamental piece for describing an assembly piece design problem.

There are also many existing works on designing interlocking puzzles. The purpose of these works is to generate geometric designs for each piece of a puzzle structure, such that they can be assembled into the desired shape in a way that no pieces are disassemblable if not followed in a specific order. These works are similar to our study on the general purpose interlocking blocks design. We raise a fundamental question about the interlocking structures. How many kinds of pieces/blocks/components are sufficient to build interlocking structures? How many kinds of components are necessary to build an interlocking structure? The second question has a trivial answer: one, since we need at least one component to build something. Our exploration in this thesis shows two kinds of blocks are enough. Thus we limit the range of the minimum kinds of components needed to either one or two. Although we do not know which one is the answer, our research will inspire further exploration in this direction.

Our study on interlocking blocks will encourage a wide range of interests in robotic construction, as we show rigid structures can be constructed using a series of simple pick-and-place actions, which are the simplest functions of robots. Automatic construction in an environment where the human can hardly survive, such as outer space and under water, becomes feasible using our materials.

There are also several papers on rearranging robots using global control. Some use the exact same model as we described in the thesis. We successfully improved the time and space complexity for rearranging a matrix of particle robots in a grid space, showing that fast rearrangement in a small space is possible.

Section 1.4

Structure of the thesis

The structure of this thesis is as follows:

In Chapter 2, we will discuss the problem of designing pieces to form a package for a given object. Given two interlocking parts with no gap between, one is divisible and the other is atomic, how can we divide the divisible part into pieces such that each piece can be moved to infinity? We will first explore the lower and upper bound on the number of pieces to cut into, which describe the physical complexity of the problem. The complexity of the optimal partition problem is then discussed. And finally, we will propose an algorithm to generate practical partitions with a reasonable time cost.

In Chapter 3, we will first see how joints simplify the constraints between parts and propose a method to determine which parts are movable. We then discuss how to computationally generate block designs with joints to form a target interlocking structure. A design of 2 kinds of blocks is later proposed along with a layout algorithm that assembles these blocks into arbitrary voxelized models such that only a small number of blocks are movable. We finally describe a robot system for doing automatic block assembly as our experiment with the blocks.

In the last chapter (Chapter 4), we will see how to rearrange a rectangular matrix of particle robots in a grid space using global control signals. Several board designs are proposed for different scenarios.

Chapter 2

Geometric design of divisible material for packaging and extraction

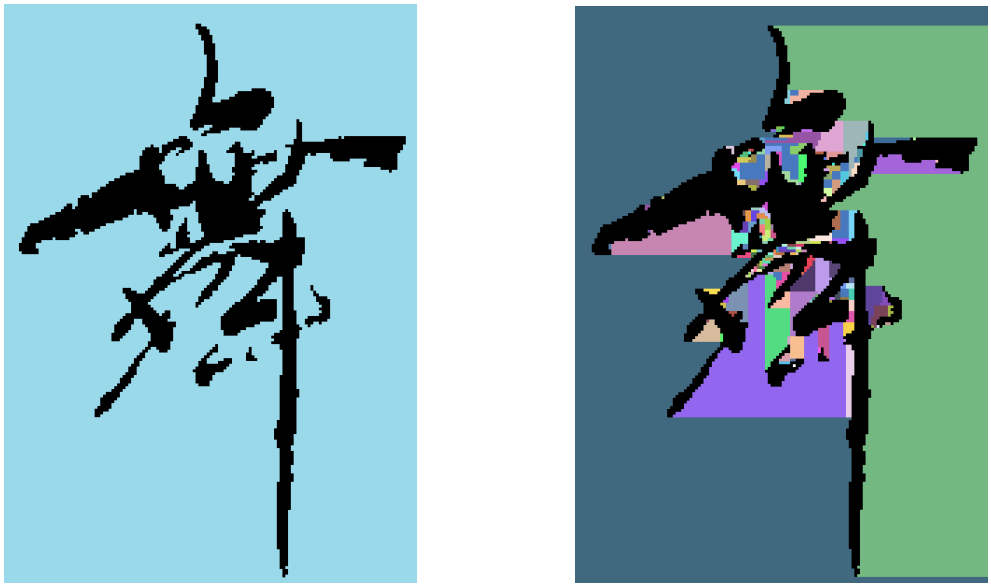
This chapter discusses a package assembling problem. We want to design a package to enclose an object firmly without a gap in-between. When the object is not convex, a package of two pieces might not be enough. Our goal is to derive an algorithm for designing each piece of the package. The discussion will start from planar cases, then will extend to three dimensions. We will first explore the lower-bound on the number of piece a package has to be cut into and then the upper-bound. Eventually, we will propose a practical algorithm for designing the package and plan assembly motions for each piece.

Figure 2.1a is an example of the designing problem where a black part needs to be packaged. There are many small open rooms formed by the object. It is hard to come up with a package design by hand. Figure 2.1b is one solution given by our algorithm. This chapter will go over details and explain how and why each piece is generated as shown.

Section 2.1

Introduction

In October 2017, a 1,000-year-old bowl from China's Song Dynasty was sold in Hong Kong at an auction by the Sotheby's. See Figure 2.2b. To protect an antique of such importance,



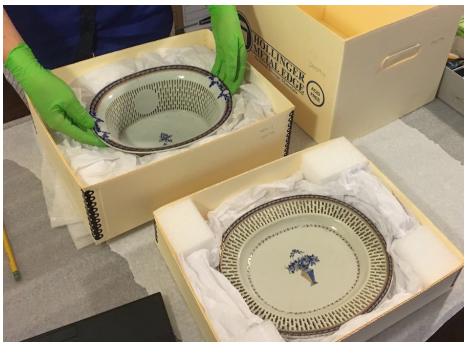
(a) The black part is the object we want to package. The blue part is the packaging material.

(b) One solution of package design with 173 pieces. Each piece can be moved to infinity in a given order.

Figure 2.1: Desiging a 2D package to hold an object such that there is no gap between the object and the package. How many pieces must the package be? The blue object is a Chinese character with the meaning of “dance”.

a special package was designed to enclose the object and prevent cracking. This package fully wraps the bowl with soft material and almost keeps the bowl from any movement.

Packaging and shipping fragile objects is a common but challenging task. Figure 2.2 is an example of the National Society of The Colonial Dames of America museum staff packaging antique porcelain. Similar to the case in Figure 2.2b, the porcelains are wrapped by soft material that immobilized the object and absorbs sudden movements which may damage the treasures. For the museum, a guaranteed safe way to pack porcelain is a styroform package able to hold every contacting part of the porcelain so tightly that there is no gap between the two materials, and the porcelain is immobilized. This problem seems trivial at first glance, especially, if the porcelain is a strict convex shape: we can simply design a styrofoam case divided into two pieces, as shown in Figure 2.3a. This strategy still holds as a solution if the object we are packing is a monotone polygon. A polygon is said to be monotone to line L if any line perpendicular to L intersects with the polygon

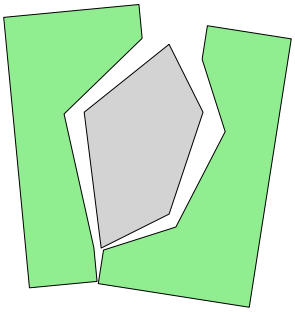


(a) The National Society of The Colonial Dames of America museum staff packaging an antique porcelain. [34]

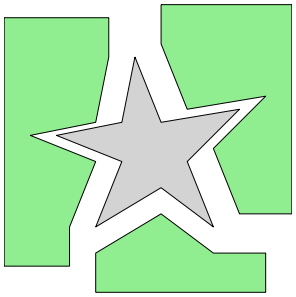
(b) A extremely rare Ru Guanyao brush washer (Northern Song Dynasty) carefully packed. [118]

Figure 2.2: How can we design a package to tightly hold a porcelain?

only twice. See Figure 2.4. The porcelains shown in Figure 2.2 are both monotone shapes. Therefore their packages consist of two pieces only. The design problem gets harder when the shape becomes more complex, such as shown in Figure 2.3b. How can we design each piece of styroform such that they can hold our treasure when assembled?



(a) A convex-shaped porcelain and its two-piece package.



(b) A more complicated shape whose pacakage has to be many pieces.

(c) A real example of porcelain package. The object is not convex and hard to design a firm wrapping package.

Figure 2.3: Examples of designing packages to firmly hold an object. As the shape gets more complicated, the package design gets harder.

To solve this package design problem, we are facing two challenges: (1) How can we partition styroform into a small number of pieces such that each piece can be moved to infinity? (2) How can we plan the motions for each piece to actually assemble or disassemble it?

When the set of ceramics is small and the shapes are simple, designing such styrofoam boxes can be done manually. However, when a museum or a porcelain company has to pack a large number of ceramic artworks of various shapes and seeks to automate the process, answering these questions become particularly helpful. Having an algorithm for finding solutions with a small number of package pieces will help to save a lot of time and effort for packaging.

In the above packaging problem, we consider two interlocking parts, one is divisible another is atomic. This chapter of study wants to solve this problem: How can we cut the divisible object into pieces such that every piece can be moved infinitely far from the atomic part?

In particular, we will explore these problems mainly in 2D. We will discuss how hard the first problem is in Section 2.3 by providing a lower bound on the number of pieces the package must have based on the given atomic shape. We will start from a 2D shape and then extend to 3D. Later, we will think about how to find pieces small enough that always guarantee paths to infinity. By uniformly cutting the divisible part into these small pieces, we have a naive but complete algorithm to design a package for an atomic object. Finally, we will consider grouping the small pieces into larger ones, such that only a small number of cuts are needed to design a package.

Our study can be extended and applied to 3D printing and prototyping where some parts cannot be printed as a whole and are supposed to contain atomic components such as motors, wires, circuit boards, and chips. In general, our study helps to automatically generate assembly pieces, and plan construction (destruction) motions for structures under constraints.

Section 2.2

Related work

Our work is closest in spirit to work on k-moldability. In the k-moldability problem, a separable k-piece mold is taken apart using a single translation per piece to expose a molded atomic part [52, 78]. Ravi and Srinivasan [79] give a list of criteria to aid the engineer in making decisions of parting surfaces. Priyadarshi and Gupta [78] used accessible directions to decompose molds into a small number of pieces. Exact-cast-mold design methods require models to be moldable or result in a large number of mold pieces. Herholz *et al.* [48] deform a model into an approximate but moldable shape, and then decompose mold pieces.

Instead of applying only translations to mold parts, our work will allow general rigid body motions to separate different parts. This allows study of the fundamental theoretical limits of disassembly. For a part to be k-moldable, every point of the divisible part has to be able to see a point infinitely far, so as to allow translations to separate. However, like most of the structures in our life, most of the parts studied in this chapter are not k-moldable, because there is no set of directions from which all of the divisible structure is visible; some portions of the structure are occluded. For example, Figure 2.9b. We show that in fact, any structure without inaccessible voids can be disassembled using only sequences of translations. The lower and upper bounds that we study are true physical bounds – they hold over *any* sequence of rigid body motions, not just single translations or rotations.

This chapter is also inspired by Snoeyink's work on the number of hands required to disassemble a collection of rigid parts [93]. Snoeyink showed that even if all shapes are convex, multiple parts may have to move simultaneously to disassemble a structure. Because we allow parts to be cut, simultaneous motions are not required for disassembly. In the *Carpenter's Rule* problem studied by Rote, Demaine, Connelly [26], Streinu [99], and others, the rigid pieces are also all atomic, and connected by joints, typically requiring many simultaneous motions.

This chapter is therefore somewhat closer in spirit to work by Wang [109] that studies the number of fingers needed to tie a knot – in that work, the string is treated as a collection of rigid bodies, but the joints may be placed arbitrarily, essentially cutting the carpenter's rule, without disconnecting the pieces. Our work is also close in spirit to work by Bell and coauthors [17] that studies the number of pieces that a mechanical knotting device must be cut into to extract the knotted string.

There has been significant work in the graphics community in computational fabrication. Song et al. [94]'s approach to fabricating large 3D objects was to break the shell of the object into pieces and assemble after fabrication. Hu et al. [51] presented a method to decompose a 3D object into a set of pyramidal shapes such that no support material is needed when 3D printing the model. Fu and Song [40, 97] studied computational interlocking furniture design, which requires segmenting a furniture into several parts and all parts can be re-assembled following a single translation motion.

Our approach to the lower-bounds problem in particular grows out of seminal work on immobilizing rigid bodies, or *grasping*. Traditional geometric approaches to grasping attempt to prevent all possible sliding and rotational motions of a polygonal object by placing fingers around the object. Reuleaux [80] is credited with the concept of *form closure*. Mishra et al. [70] proved the sufficiency of four fingers to immobilize any polygonal object. Czyzowicz et al. [28, 29] showed that polygons without parallel edges can be immobilized using three fingers. Rimon and Burdick [82, 83] showed how two-finger grasps can be analyzed using *second-order immobility*. Cheong [23] provided an algorithm to compute all immobilizing grasps of a simple polygon. Cheong et al. [24] also showed that $n+3$ contacts suffice to immobilize a chain of n hinged polygons.

A polygonal object can be *caged* by surrounding an object with fingers, such that the object has some freedom locally but cannot escape the cage. Some of the earliest work on caging was by Rimon and Blake [81]. Vahedi et al. and Allen et al. [107, 2] proposed algorithms to find all caging grasps of two disk fingers. The idea behind Allen et al. work

is that immobilization is a special case of caging with two fingers in contact with the object. And all possible two-finger contacts can be described using two parameters, thus constructed a 2D space called contact space. Escaping is also an extreme case of caging where the object is just able to leave obstacles formed by two fingers. These extreme cases are points in the contact space and computable. Bunis et al. later extended this idea of search contact space into equilateral three-finger caging problem [19]. They proved that the obstacles introduced by the third finger in the contact space (2D) is a polygon whose vertices and edges are computable. Erickson et al. [36] studied the case of three-finger caging for arbitrary convex polygon. They utilized the concept of *capture region* which is a set of configurations that may not immobilize the object but prevents it from moving to infinity. Erickson et al. showed that computing the capture region for convex shapes can be reduced to a visibility problem. Makita et al. [65] extend the caging problem from 2D to 3D with multi-fingers. Our work, instead of caging an object, can be viewed as removing contacting pieces to un-cage a polygon in the plane.

A classic problem of *self-assembly* is to move a set of small robots to specified target positions; some of the challenges with narrow corridors and coordination are similar to those faced in the current work. Kotay et al. [57], for example, designed a robotic module, groups of which aggregate into 3D structures. Rus and Vona's early work on the Crystalline robot [88] presented an algorithm to do self-reconfiguration. Recently, working on the problem of scale, Rubenstein et al. [87] provided an algorithm for moving kilo-bots one-by-one to form certain planar shapes. Arbuckle et al. [4] allowed identical memory-less agents to construct and repair arbitrary shapes in the plane. In the authors' own work [123] on assembly of interlocking structures, 9 kinds of blocks are used to build large-scale voxelized models such that all blocks are interlocked and the whole structure is rigid as a whole. Self-assembly and modular robots in the presence of obstacles has also been extensively studied. Becker et al. [14] proposed an algorithm to efficiently control a large population of robots in this scenario. Rubenstein et al. [86] used multiple robotic units to

manipulate the positions of obstacles.

Section 2.3

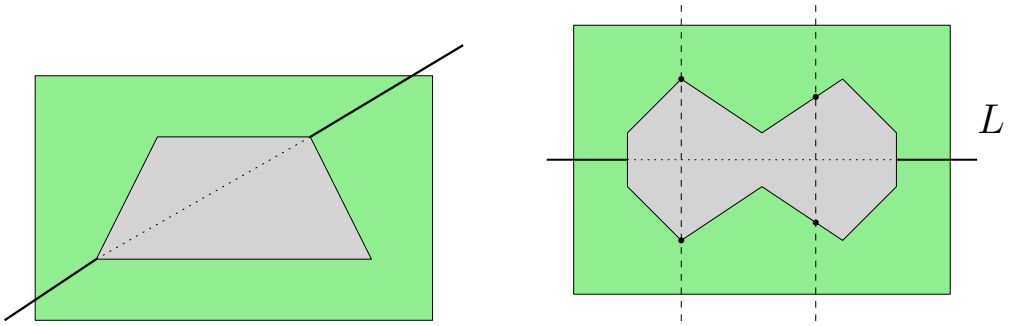
Problem complexity: Lower bound on the number of pieces

In this section, we are mainly curious about this problem: Given an atomic object interlocked with a divisible part, how many pieces *at the least* should we expect to cut the divisible part into so every piece can be moved out of the bounding box of the environment and to infinity? We will propose two methods for answering thing problem, one works for 2D, another is extendable to 3D.

When presented a collection of objects to package, if the time is limited, one may want to start from the simple shapes, so we can pack as many objects as possible in a short amount of time. So given an object, how hard is it to design a package for the object? Should we keep working on packaging the object or is it easier to move on to another one? Having an estimation of the physical complexity of each package is very important to answer these questions and to schedule appropriate arrangements.

Instead of thinking how to design pieces and *assemble* them to form a tight package, our analysis will attack the problem from the opposite direction and ask how to cut an enclosing package into pieces such that every piece can be *disassembled*. As an early exploration, we consider both components being rigid body polygons or polyhedra. A cut can be made inside the divisible part along any arbitrary curves.

Let $\mathscr{A}$ and $\mathscr{B}$ be two interlocked parts, where $\mathscr{A}$ is atomic and $\mathscr{B}$ is divisible. How many pieces *at least* should we expect to cut the divisible part into? Here, we use the number of required pieces to indicate *physical complexity*. Just like time and space complexity for analyzing computer science problems, physical complexity is a good basis for thinking about robotics problems. In many situations like search-and-rescue and packaging where time or labor is limited, knowing the complexity of each task will help resource allocation



(a) If the atomic shape is convex, one cut going though two farthest points along one direction can always make two divisible component movable to infinity.

(b) A monotone shaped atomic can also be hold by a package of two pieces. All lines perp to line L intersects with the shape only twice.

Figure 2.4: Two kinds of atomic shapes whose firm wrap package can be assembled using only two pieces. Gray parts are atomic and the green parts are divisible. Solid lines are the cuts.

and job scheduling.

A trivial answer to the lower bound on the number of pieces is two because one needs to make at least one cut to separate two interlocking objects. This happens to be the optimal solution for convex and monotone atomic parts. See Figure 2.4. More generally, the minimum number of pieces differs from shape to shape, so is the lower bound. We, therefore, cannot provide an interesting number directly but will propose several algorithms to compute such lower bounds for any given shapes. Two methods will be proposed, one geometric method based on Reuleaux's graphical method for analyzing the constrained motion of planar rigid bodies, and one linear-algebraic approach that can be extended to solve 3D cases. These algorithms will give us good insight on the complexity of a separation and disassembly problem, and help with our original package shape design question.

When a divisible part is stuck in an environment and has to be cut into small pieces to leave, there could be both local and global properties causing the stuck issue. Locally, two parts (divisible and atomic) could be mutually immobilized, in the same way that a hand grasps an object. Globally, the divisible could have some local freedom but faces narrow passages that allows small parts to go through. In this chapter of computing the necessary

number of pieces, we will focus only on the local property causing mutual immobilization, and used the term "interlocked" to indicate immobility before making any cuts.

2.3.1. A graphical method anlyzing the rotation centers of boundary points

This approach is inspired by some existing work [11, 69] on the contact modes analysis of 2D rigid body contacts, as well as Reuleaux's graphical method [80] for analyzing if a set of unmovable points on the boundary of an object constrains the motion of the rigid body.

We first relax this problem by replacing our divisible object $\mathscr{B}$ with a finite set of points $\mathscr{P}$ contacting the atomic rigid body $\mathscr{A}$ on the boudary. These points can be chosen by random, In our implementation, we pick all vertices and some more points along the boundary sampled uniformly. Our goal is to break the immobilization constraint and give $\mathscr{A}$ at least some local freedom by removing obstacle points. Consider a subset of point obstacles in the same piece, they will follow the same rigid body motion of the piece. In order for the piece to separate from $\mathscr{A}$, there must exists a single rigid body motion that does not cause any point obstacle to collide with the atomic object, at least instantaneously. The necessary number of pieces is the smallest number of subsets covering all obstacles.

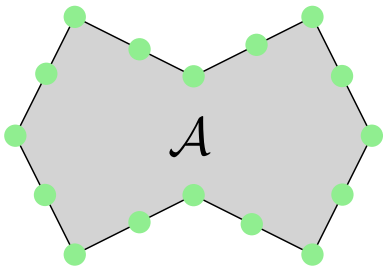


Figure 2.5: Simplify the divisible part into a set of points along the boundary of part $\mathscr{A}$. How many groups of points can move together without collision?

As we know, any single planar rigid body motion is a combination of translation and rotation. Looking at a specific point $x(t)$ on the rigid body, its motion with velocity $\dot{x}$ at any moment t can also be viewed as a single rotation about a center whose connection to $x(t)$ is perpendicular to the velocity. This means at time 0, when a motion is just about to happen, every point on the same piece is doing a rotation about the same center. Pure translation of a piece can be viewed as a rotation about a center infinitely far. If we can find a rotation center such that a subset of point obstacles can rotate about it in either clockwise

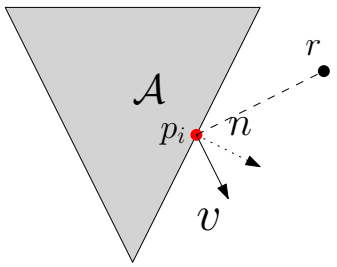


Figure 2.6: Instantaneously, a single motion of a rigid body that contains point p_i can be viewed as a rotation about a center r. The motion is feasible only if the velocity of p_i is not pointing into the atomic part.

or counter-clockwise direction without colliding with the atomic object instantaneously, we actually find at least one piece that hold the subset of point obstacles.

Based on the above observation, we immediately come up with a naive algorithm that can compute a lower bound as shown in Algorithm 1. We first generate the powerset over $\mathscr{P}$, then run a minimum set-cover algorithm over the generated subsets. The complexity of this algorithm is very bad. Computing the powerset of all point obstacles takes exponential time, and finding the minimum set cover is known to be *NP-complete*.

Algorithm 1 Graphical method to compute the lower bound.

```
1: procedure NAIVELOWERBOUND(𝒫, 𝒜)
2:     S ← generate powerset over 𝒫.
3:     Run minimum set-cover algorithm over S.
4: end procedure
```

Let's first attack the first issue of Algorithm 1. How can we compute large enough subsets of $\mathscr{P}$ that can move together without iterating over the powerset over $\mathscr{P}$? The following theorem gives us hope.

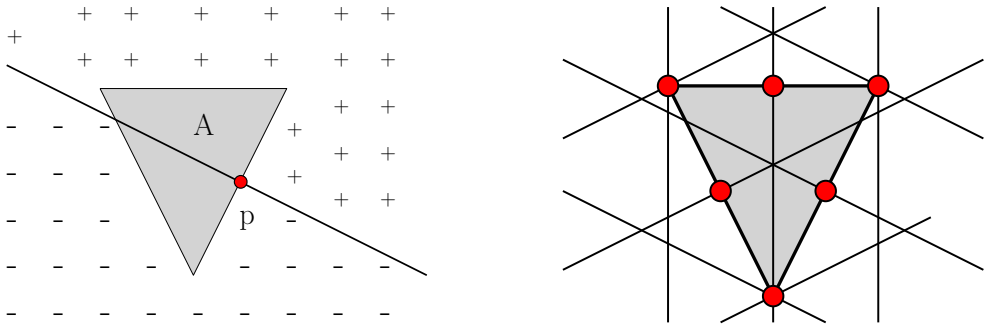
Theorem 2.1. *If n points $\mathscr{P}$ of a planar rigid body B are in contact with a polygonal rigid body $\mathscr{A}$, then there are at most $n(n-1)/2+2n$ maximal subsets of $\mathscr{P}$, such that each subset may be moved together as a rigid body without colliding with $\mathscr{A}$, and any other non-colliding subset is contained within one of the maximal subsets.*

Proof. We would like to find if a subset of points in $\mathscr{P}$ can be grouped as if in the same piece and be separated from $\mathscr{A}$. Let's first consider rotations. Given a rotation center and a

direction in either positive (counter-clockwise) or negative (clockwise) way, we can iterate over each point in $\mathscr{P}$ and test if the point collides with the atomic part under the rotation motion, forming a compatible subset. A rotation center and a rotation direction thus maps to a compatible subset of points. But how many rotation centers do we have to consider? And how many unique compatible subsets can we generate?

Consider a rotation center r in the plane, an associated rotation direction, either positive or negative, and a point $p_i \in \mathscr{P}$. The velocity of p_i at time 0 is a vector $\mathbf{v_i}$ perpendicular to the line throught r and p_i. Assuming p_i is sitting on an edge of $\mathscr{A}$ with an outward normal vector $\mathbf{n}$, for most of choices of rotation centers, the point under one direction either collides with the edge ($\mathbf{v} \cdot \mathbf{n} < 0$) or not ($\mathbf{v} \cdot \mathbf{n} \geq 0$). See Figure 2.6. Along the normal to the edge at p_i, rotation centers with both positive and negative directions are permissible ($\mathbf{v} \cdot \mathbf{n} = 0$). See Figure 2.7a. Keep the rotation direction and move r, p_i is still compatible with the motion until r goes across the normal line through p_i. The normal line actually partitions the space into two parts, one allows rotation centers with positive direction, another allows centers with negative direction. Two points p_i and p_j are compatible iff the rotation center sits in the overlapped direction regions defined by two points' normal lines. Let $P_r \subset \mathscr{P}$ be the set of points compatible with rotation center r and one direction. Similarly, we move r along a continuous trajectory. Membership of P_r changes only when the center r crosses a normal line through one point of $\mathscr{P}$. In order to compute the *maximal subsets* of compatible points, we only need to consider the non-overlapping regions of the planar space partitioned by all normal lines. See Figure 2.7b. The number of regions is smaller than the number of triangles formed by triangulating the intersections of normals. The normal lines form an *arrangement* [98] [105], and there are $n(n-1)/2$ possible intersections. We can now generate candidate maximal subsets and discard any that are contained within other existing subsets.

Next, let's consider translations. Any translation of a set of points can be viewed as a rotation about a center infinitely far. For any point p_i, we consider two translation directions along its connecting edges. (Any point on the edge of a polygon is also considered a vertex, thus has two connecting edges.) This is equivalent to choosing a rotation center infinitely far along the normal line with both positive and negative directions. Similarly, each point p_i



(a) Contacting point p can rotate about centers in $+$ area in positive direction or about centers in $-$ area in negative direction.

(b) The normal lines through the contact points form a set of cells, such that the number of points that may be separated from the gray part is maximized by choosing a rotation center inside a cell.

Figure 2.7: Reuleaux's graphical method for analyzing if a contact point or a collection of points fully constrain(s) the motion of a rigid body.

permits half-plane of translation directions, if included in a compatible subset. Therefore, it is sufficient to test two translation directions for each point, one long one edge direction and another along the outward normal direction. For n points in $\mathscr{P}$, we collect $2n$ directions, and for each direction, test the remaining points against that direction to generate candidate maximal subsets.

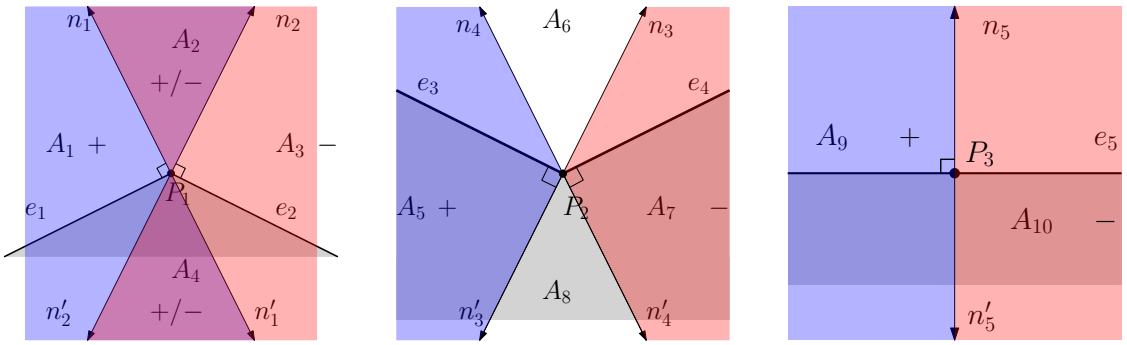
In total, we have done $n(n-1)/2+2n$ tests representing the same amount of subsets of $\mathscr{P}$. □

Theorem 2.1 suggests a better algorithm for computing a lower bound on the number of pieces the divisible part must be cut into to allow separation. Compute the maximal subsets as suggested in the above proof; then solve the minimum-set-cover problem to find the minimum number of such sets needed to separate all points in $\mathscr{P}$ from $\mathscr{A}$. If the number of maximal subsets is small, minimum-set-cover may be solved exactly. If there is a large number of subsets, then a greedy approach yields a solution in polynomial time, with guaranteed logarithmic approximation quality. The pseudo-code is described in Algorithm 2. The simple examples presented in this section were solved exactly using integer-linear programming.

Based on the angle of two edges connecting a point, contact points are classified into

Algorithm 2 Graphical method to compute the lower bound.

procedure COMPUTELOWERBOUND($\mathscr{P}$)
    for all point $p \in \mathscr{P}$ **do**
        Generate two lines going through p and perpendicular to each connected edge.
        Add the two lines to set L
    end for
    for every pair of non-parallel lines in L **do**
        Compute intersection i.
        $F(i,+) \leftarrow$ find all points in $\mathscr{P}$ compatible with rotation center i and the $+$ direction.
        $F(i,-) \leftarrow$ find all points in $\mathscr{P}$ compatible with rotation center i and the $-$ direction.
    end for
    Run minimum set cover algorithm over $F(i,+/-)$ for all intersections such that all points in P are covered.
end procedure



(a) Convex vertex. (b) Concave vertex. (c) Edge vertex.

Figure 2.8: Three kinds of vertices in a polygon. The blue areas allow positive rotation centers while the red area allows negative rotation centers. The shared region permits both negative and positive rotation centers. No points can rotate about centers in the empty areas. Arrows indicate the normal (inward and outward) directions.

three kinds: convex vertex, concave vertex, and edge vertex. Any point on the edge is considered an edge vertex connects two parallel edges. For each edge connecting a vertex, we draw a line parallel to the normal and going through the vertex (line 2 and 3). With two lines, the plane is divided into four areas allowing positive or negative rotation centers. Each area is closed on the boundaries. Figure 2.8a is an example of a convex vertex; the space is divided into four areas: A_1,A_2,A_3 and A_4, where A_1 allows positive rotation centers for vertex P_1, A_3 allow negative rotation centers, and A_2 and A_4 allow positive and negative rotation centers. Mathematically, each area is represented using two linear inequality constraints. Given a rotation center and a rotation direction, we test if a vertex can rotate about the center by checking which area the rotation center falls into. Each rotation center and rotation direction is tested for concave vertex and edge vertex separation in the same way.

For each normal line going through contact points, the algorithm finds intersections with all other normal lines. Therefore a set I of $O(n^2)$ intersection points are maintained, where n is the number of normal lines. For each intersection point, we test both positive and negative rotation centers against every contact points on the atomic part (line 8 and 9). Let $F(i,d)$ be the set of contact points that can rotate about intersection i in direction d without colliding with the atomic part instantaneously. The algorithm also tests $F(i,d)$ for all $i \in I$ and $d \in \{+,-\}$ and eliminates the sets containing the exact same contact points. (If a set is a subset of another, we only keep the larger set.)

Because the intersections are on the boundary of areas, testing if a point can rotate about an intersection point by checking linear inequalities might suffer numerical issues caused by floating point arithmetic. A solution is to triangulate the intersections and use the triangle centers as rotation centers to test against all contacts on the atomic part boundaries.

We now have a set of sets each containing contacts that can rotate about a same center in the same direction. With a minimum-set-cover algorithm, we find the minimum number of sets that covers all contacts (line 11). Contacts in the same set are partitioned into a same

piece of divisible material to be separated from the atomic part. The minimum-set-cover problem is NP-complete, so the algorithm to find an exact solution can take a long time if the number of subsets is large.

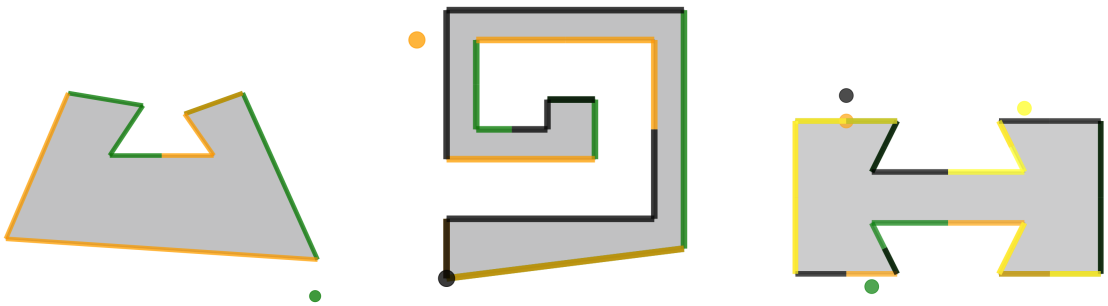
To accelerate the computation, we may instead use a greedy algorithm to rapidly solve the set cover problem approximately. Given a set of n subsets, Chvatal [25] showed the approximation ratio $H(n)$ of the greedy algorithm is $H(n) = \sum_{k=1}^{n} \frac{1}{k} \leq ln(n+1)$. Dividing the result from the greedy algorithm by the approximation factor, we have a lower bound on the number of pieces the divisible material must be cut into to separate from the atomic part.

Figure 2.9 shows some test cases and partition results. A few stastics are shown in Table 2.1. We wrote our code in Python2.7 and ran the code on a 2016-model MacBook Pro with a 2.6 GHz Intel processor and 8 GB 1600 MHz DDR3 memory, and are intended only to give a sense of the practicality of analysis of structures with various numbers of edges. From the table, we can see that with the increase of maximum number of rotation sets, the time cost increases dramatically, as we would expect for a $O(n^3)$ check of rotation centers and a linear-integer program optimal solution to minimum-set-cover; we expect that much larger problems could be solved with good approximation by using greedy minimum-set-cover techniques.

In the test cases, we choose 3 points for each edge, two on both ends and one in the middle. A segment is marked as in the same piece (color), if two end points of the segment rotates about the same center in the same direction. Edges containing points in the same set are not necessarily connected. Whether there exist cuts to separate edges exactly into the derived sets as connected rigid bodies is an open question. Whether those bodies can be extracted after initial separation, is also unknown. Here we naively assume there exist such cuts and edges containing points in the same set are in the same piece. This assumption will ensure a lower bound.

shape	edges	largest set size	set cover size	time cost
cavity	8	144	2	0.2018 s
spiral	14	612	3	1.8764 s
dumbbell	12	1104	4	7.2543 s
mammoth	64	12012	3	540.327 s

Table 2.1: Lower-bound analysis examples.



(a) Cavity. Orange edges translate to the left, and green edges rotate about the green point in negative direction.

(b) Spiral. Black edges rotate about the black point in positive direction and orange edges rotate about the orange point in negative direction. Green edges translate to the right.

(c) Dumbbell. 4 sets of edges in 4 colors. Each set of edges rotates about centers with the same color. Rotation directions: positive for black, positive for the rest.

Figure 2.9: Three examples of analyzing necessary number of pieces for the divisible part. In each example, edges with the same color are in the same set.

2.3.2. A linear-algebraic method for lower-bounding the number of pieces

The previous subsection introduced a graphical method for computing a lower bound on the number of pieces the divisible must be cut into to allow separation from the interlocking atomic part. The method however only works for 2D cases. In this subsection, we will introduce a linear-algebraic approach for the exact same problem but can be extended to 3D.

Similar to the previous problem, we consider a finite set of points $\mathscr{P}$ contacting the atomic part $\mathscr{A}$. How can we group a subset of points in $\mathscr{P}$ in a same rigid piece following the same rigid body motion and separate from $\mathscr{A}$? How many maximal subsets do we have to consider? We will attack this problem by analyzing the motion of contact points using a set of linear inequalities.

Let x_i be the Cartesian location of point p_i and $\mathbf{x} = [x_0, \ldots, x_n]$ be the vector representing locations of all n points in $\mathscr{P}$. The configuration of a rigid body can be expressed using q containing position and orientation of the object. Equation 2.1 defines the velocity $\dot{x}_i$ of the point p_i, where $J_i(q)$ is the Jacobian matrix of this point.

$$J_i(q)\dot{q}_i = \dot{x}_i \tag{2.1}$$

We describe the rigid body motion of the atomic part using Equation 2.2, where $J(q)$ is the collection of Jacobian matrices of all points in the current configuration.

$$J(q)\dot{q} = \dot{x} \tag{2.2}$$

Consider the point p_i on an edge or surface e. The point has a feasible motion (no collision) if the angle of between its speed $\dot{x}_i$ and the edge normal n_e is less than 90 degrees, or

$$n_e \cdot \dot{x}_i = J_i(q)\dot{q}_i \geq 0 \tag{2.3}$$

A convex or concave vertex has multiple edges / surfaces connecting to it. For convex vertices, a motion is feasible if the motion is not penetrating any edges / surfaces. For concave vertices, a motion is feasible if none of its connecting edges / surfaces is penetrated.

Let N be the matrix collecting all normals to $\mathscr{A}$ at he contacts, such that the non-penetrating motion of contacts can be described as:

$$N \cdot J\dot{q} \geq 0 \tag{2.4}$$

For simplicity, we phrase our problem of grouping points slight differently by considering the motion of the atomic part instead of the points, because the motion of $\mathscr{A}$ is symmetric to the motion of its contact points. How many non-static motions ($|\dot{q}| > 0$) of

the atomic part do we need, such that every point in $\mathscr{P}$ has at least one non-penetrating motion? As both parts $\mathscr{A}$ and $\mathscr{B}$ are interlocked, there is not a single non-static motion that satisfies Equation 2.4. Our goal is to find the smallest set of actions $Q = \{\dot{q}_0, \dot{q}_1, \dots\}$ such that for any single contact point, there is at least one feasible motion in Q satisfying Inequality 2.3.

We simplify Inequation 2.4 a little bit by replacing $N \cdot J$ with $M = NJ$. For any particular velocity $u = \dot{q}$, the product matrix $N \cdot J \cdot \dot{q} = M \cdot u$ has some rows with non-negative values satisfying some contacts' feasible motion constraints, and some negative rows causing penetration. Let the contact points with feasible motions be given by:

$$s(u) = \{p \in \mathscr{P} : \text{p has a feasible motion under u}\} \tag{2.5}$$

Any other velocity u_i being a positive scaling of u_i has the same set of compatible contacts, $s(u_i) = s(u_j)$. We therefore always normalize the configuration velocity of the atomic part, so $|u| = 1$. Similar to the graphical method mentioned in the previous subsection, the feasible motion constraints of a contact point partition the 3D configuration space into two parts with one half-space permits feasible motions. Two linearly independent configuration velocities u_i and u_k have the same set of contacts with feasible motions, if u_i and u_k fall into the same subspace formed by the same set of constraints.

Theorem 2.2. *The collection of all feasible motion constraints for every contact point is a set of at most* $2n$ *half-spaces in a 3D configuration space partitioning the space into* $O(n^2)$ *convex cones.*

Proof. Inequality 2.4 has a trivial solution $\dot{q} = \mathbf{0}$, implying the boundary of every feasible half-space (a plane) goes through the origin. So the 3D configuration space is partitioned into many convex cones.

How many convex cones are there? Let $f(d,n)$ denote the maximum number of cells in a d-dimensional space under n cuts. In 2D, n cuts going through the origin partition the planar space into at most $2n$ parts. Thus, $f(2,n) = 2n$. In d-dimensional space, the n-th cut

goes through at most $f(d-1,n-1)$ cells and adds as many new cells. Therefore

$$f(d,n) = f(d,n-1) + f(d-1,n-1). \tag{2.6}$$

So $f(3,n) - f(3,n-1) = f(2,n-1) = 2(n-1)$, $f(3,1) = 2$, and $f(3,0) = 0$. The following equations will prove us $f(3,n) = O(n^2)$.

$$\begin{aligned}
f(3,n) - f(3,n-1) &= 2(n-1) \\
f(3,n-1) - f(3,n-2) &= 2(n-2) \\
f(3,n-2) - f(3,n-3) &= 2(n-3) \\
&\cdots \\
f(3,2) - f(3,1) &= 2(n-(n-2)) \\
f(3,1) - f(3,0) &= 2(n-(n-1))
\end{aligned}$$

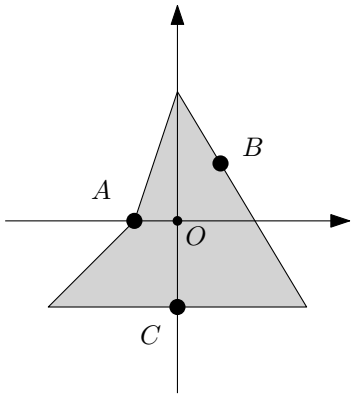
Adding all the equations us, we have

$$f(3,1) - 0 = 2 \cdot \sum_{i=1}^{n-1} i = O(n^2)$$

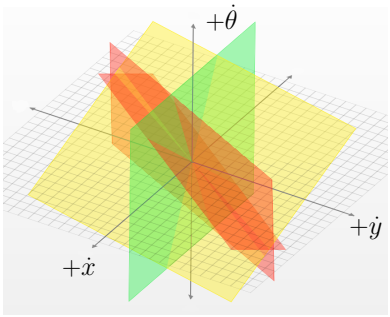
□

Figure 2.10a shows an example of analysis using method proposed in this subsection for an atomic shape with one concave vertex. The coordinates are: $A = (-1,0)$, $B = (1,4/3)$ and $C = (0,-2)$. Let (x,y,θ) be the configuration of the atomic shape, constraints for point A are $-3\dot{x} + \dot{y} + \dot{\theta} \geq 0$ and $-\dot{x} + \dot{y} + \dot{\theta} \geq 0$. The constraint for point B is $5\dot{x} + 3\dot{y} + 2\dot{\theta} \geq 0$ and the constraint for point C is $\dot{y} \geq 0$. Constraints are shown in Figure 2.10b.

Theorem 2.2 implies an algorithm to compute the lower bound on the number of pieces that the divisible part must be cut into to allow assembly or disassembly. We first construct the convex cones in the 3D space. In each convex cone, select a point (configuration velocity) and compute the set of contact points compatible with the selected velocity. We then run minimum set-cover algorithm to generate the minimum number of subsets such



(a) A simple atomic shape. 3 points, A, B and C are selected for analysis.



(b) The two red linear constraints are for point A because it is on a concave vertex. The yellow constraint is for point B and the green constraint is for point C.

Figure 2.10: Analysis of a simple atomic shape and visualization of constraints for point A, B and C.

that every contact point in $\mathscr{P}$ has at least one feasible motion. The pseudo-code of this approach is provided in Algorithm 3.

Algorithm 3 Linear algebraic method to compute the lower bound.

1: **procedure** COMPUTELOWERBOUND($\mathscr{P}$)
2: Split every convex and concave point into two overlapping points each on an edge connected to the original point.
3: Construct matrix $N \cdot J\dot{q} \geq 0$ as the matrix to describe non-penetrating motion of all contact points. Define $M = NJ$.
4: $R \leftarrow$ all combinations of two linearly independent rows in matrix M.
5: **for** combination in R **do**
6: Let x' be a solution for $M'u = 0$ and $|x'| = 1$, where M' is a matrix with two rows from the combination.
7: $F(x') \leftarrow$ the subset of points with feasible motions under x'.
8: **end for**
9: Run a set-cover algorithm over $F(x)$ for all x computed in line 6 to find a minimum number of configuration velocities whose sets of feasible motion points cover all contacts
10: **end procedure**

Line 6 of Algorithm 3 selects two linearly independent constraints and computes the null space. This is because any three linearly independent constraints, all going through the origin, intersects only at the origin. Intersections of the null space and a unit sphere are

candidates of configuration velocities and their corresponding contact points with feasible motions are computed (line 7). Since every intersection is on the boundary of a convex cone, using this velocity to compute contacts with feasible motions might also suffer numerical issues. The solution is similar to the graphical method. Since the null space of every two linearly independent constraints is a line, we list all these lines and intersect with a unit sphere, triangulate the points on the surface of the unit sphere, and use the center of each triangle to compute contact points with feasible motions.

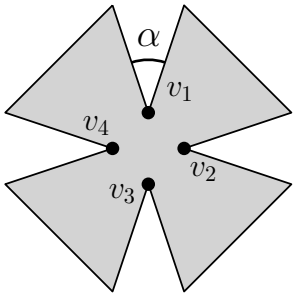
2.3.3. Shapes hard to cut and disassemble

As mentioned at the beginning of this chapter, if the atomic part is a convex polygon, its surrounding interlocking divisible part can be partitioned into two pieces and be separated from the atomic part. The partition gets harder when more concave vertices are introduced. Figure 2.11a shows a polygon shape with 4 concave vertices. To separate all boundary points of the atomic (gray) part, we must divide the divisible part into at least 4 pieces. Figure 2.11b analyzes the 4 concave vertices and their feasible motion regions using the graphical method. Each vertex rotate about a point in the blue region in positive direction or a point in the red region in negative direction. No two points have overlapping positive regions or overlapping negative regions. Therefore, no concave vertices share a same rotation center.

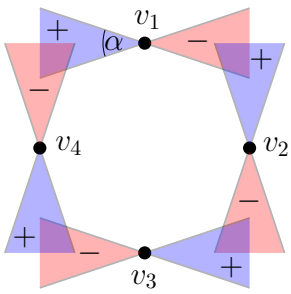
Extending the idea of having no overlapping rotation areas for any pair of concave points, we construct a shape with n concave vertices and require the divisible part to be partitioned into at least n pieces.

Theorem 2.3. *Given n concave vertices, there exists an atomic shape whose surrounding divisible material must be disassembled into at least n pieces to separate from the atomic shape.*

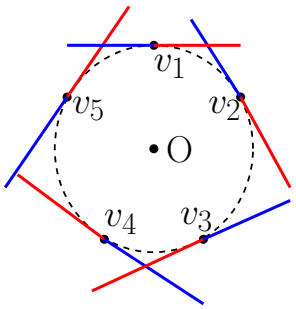
Proof. We first draw a circle, equally divide the circle into n arcs, and put a vertex at the center of each arc. The radius of each arc is $2\pi/n$. Let the angle between two edges of a convex vertex be $0 < \alpha < 2\pi/n$. The angles of cones allowing positive or negative



(a) Atomic part with 4 concave vertices. Points on the boundary must be partitioned into 4 pieces to separate from the atomic part.



(b) Graphical analysis of concave vertices in Figure 2.11a. Blue areas allow positive rotation centers while red areas allow negative rotation centers.



(c) Five concave vertices. No pair can rotate together. Blue areas allow positive rotation centers while red areas allow negative rotation centers.

Figure 2.11: Two examples of atomic shapes that require divisible part to be disassembled into at least 4 or 5 parts to be separated.

rotation centers are also α and no two positive areas or negative areas intersect. Thus the surrounding divisible material of any shape with these n concave vertices each has two edges forming an α angle must be broken into at least n pieces to be removed from the atomic part. □

Figure 2.11c shows an example with five concave vertices. The resulting shape is similar to Figure 2.11a but with 5 leaves.

Section 2.4

The upper-bound: break into particles

What is the maximum number of pieces a package may have? This section will answer this problem by developing an algorithm that gives the upper bound on the number of pieces. We will explore the geometric properties of both the divisible and atomic part and find some tools which can guide us to partition the divisible part into small pieces bounded by uniform square cells. These small square cells can move freely in the environment and escape to infinity. The main result of this section is a method to compute the size of the small cells such that they are guaranteed removable one-by-one.

The previous section discusses some theoretical analysis of the partitioning and extraction problem. Although the results are helpful for us to understand the problem, it is not helpful to make any actual cuts. Let us look back to the original problem. Suppose you are the package designer with cut-able styrofoam in hand. Now, you are given the tool to estimate the lower bound of the difficulty of designing a package. You know how much time and effort are needed *at the least* to assemble an enclosing package for a polygonal shape. However, the lower bound is just an estimate and can be very conservative and inaccurate. How much time or effort do we have to pay *at the most* to finish the packaging task? Is there a better estimation on the max amount of effort required? This section will help answer this question.

Intuitively, one can cut the divisible part into infinitely many particles of infinitely small size and reassemble all particles to form a solid package, which takes an infinite number of cuts. Removing the components one-by-one is also going to take an infinite amount of time, but at least we know each component, wherever it locates, is guaranteed escape-able from the environment. An interesting observation is, if we have only one bounding square for the whole divisible part (we make no cut at all in this case), the component can not get out. Between the size of infinitely small cells and the size of the bounding box, there seems to be a turning point that if we partition the divisible part into squares of that size, some pixel is just able (or just not able) to escape the maze. In this subsection, we want to find a cell size that is as big as possible but also smaller than the critical point.

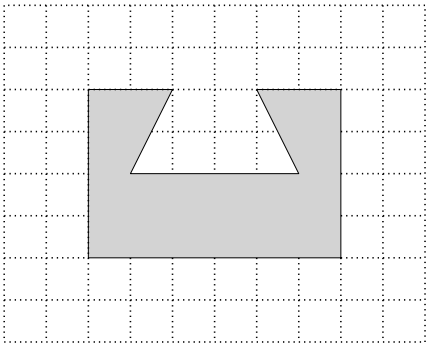


Figure 2.12: Two interlocked parts where the gray part is atomic and the rest is divisible. Gridding the divisible part into small enough cells and removing them will separate both parts.

Starting from this section, we will call a piece of divisible part bounded by a unit square a *component*, while the bounding unit square is a *pixel*.

Figure 2.12 is an example of gridding the divisible material into axis-aligned components. In this example, every divisible component bounded by a dotted pixel can move around the environment and escape to infinity. For example, moving the top left corner component to infinity is simply a translation along y positive direction. Its adjacent components can be first moved to the top-left corner, then follow the path. Similarly, other components can be removed by moving to an adjacent removed component then follow the removed component's path. Our goal in this section is to find a pixel whose bounded component can always find another component whose escape path is known.

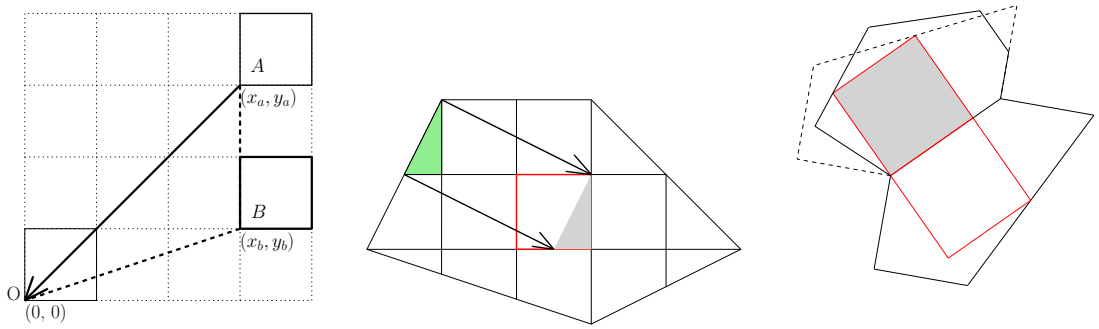
In a plane without obstacles, our first observation about axis-aligned pixels is the convenience of planning motions. Partitioning a planar divisible part into unit size pixels, there exists a motion for each pixel to move to a target location, if follow a specific order, such that there is no collision with other unmoved pixels.

Theorem 2.4. *In a planar grid of square cells with a designated target square, continuous translation of each grid cell to the target will not cause collision, if the cells are translated in order of L_2 distance from the target.*

Proof. The idea to prove this theorem is by showing contradiction. Assuming a pixel collides with another following the ascending L_2 distance order, we will show that the colliding pixel is actually removed before current one. Thus a collision is not possible.

Let A be a pixel whose bottom-left point is at position (x_a, y'_a), moving in one direction towards the target square O, with bottom-left point at (x_o, y_o). Without loss of generality, assume $x_a > 0, y_a > 0$ and $x_o = 0, y_o = 0$, and that the width of each cell is 1. We know that, during the motion, every point of square A is bounded by the rectangle R defined by its bottom-left point $(0,0)$ and top-right point $(x_a + 1, y_a + 1)$.

Assume A collides with a square B whose bottom-left point is at (x_b, y_b), $x_b, y_b \in \mathbb{Z}^+$. Then $0 \leq x_b \leq x_a$ and $0 \leq y_b \leq y_a$; otherwise, no point in the square is in R. Because OA is along the diagonal of R, OA is the longest edge in triangle $\triangle OAB$. So $|OB| < |OA|$, and square B would have been moved before A using the proposed order. Therefore the collision will not happen.



(a) Squares in a grid space can move to any target position with no collision following the order of their distances to the target.

(b) Cells can also move to any target square with no collision in the grid space bounded by a convex polygon using the same method.

(c) Between two adjacent convex shapes, there exists a square that can move freely from one to another without leaving either shape.

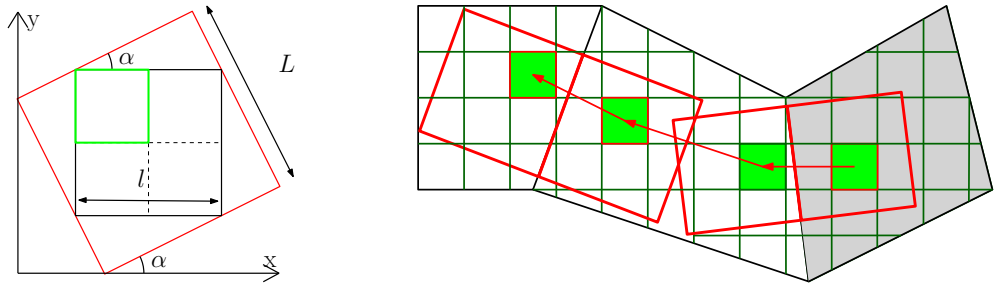
Figure 2.13: Axis-aligned pixels and their motions.

Above proof is also explained in Figure 2.13a. □

Although Theorem 2.4 is claimed and proved in the plane, using the similar reasoning, it can be easily extended into arbitrary dimensions.

Consider the simplest case of moving pixel-bounded components in an environment without obstacles and the plane is bounded in a convex shape as shown in Figure 2.13b. The same strategy of moving pieces based on their L_2 distance to the target also yields no collision between pixels. Because we can extend the convex shape to a bounding square with all components being complete, and no collision happens in the superset indicating the same result in the subset. Our environment, however, cannot always be convex shapes. Luckily, we can always decompose a non-convex shape into a set of disjoint convex shapes. Let $\mathscr{A}$ be our atomic part, $\mathscr{B}$ be the divisible part, where $\mathscr{A} \cup \mathscr{B}$ equals the bounding box of both $\mathscr{A}$ and $\mathscr{B}$. We can decomposes part $\mathscr{B}$ into a set of nonoverlapping convex shapes. We know how to move pixels in one convex shape, if we can also move pixels in adjacent convex shapes, we might be able to empty the part $\mathscr{B}$ following a sequence of pixel motions between adjacent convex shapes.

In the one convex shape environment, we did not consider the pixel size. To move



(a) Partitioning the space using a (green) square of width $l/2$ guarantees there exists at least one square in the transition square.

(b) Moving components through a chain of convex polygons. White polygons are empty and the gray one is filled. Using the pairs of transit squares, cells can move between any two adjacent polygons, or through a chain of connected polygons.

Figure 2.14: Moving pixels in a chain of convex shapes.

pixels between two adjacent convex shapes without leaving either one, pixel size became important. Figure 2.13c shows two convex shape sharing an edge. Flipping one polygon about the shared edge and intersecting with the other polygon gives a new convex shape. Inside the new convex polygon, we compute a largest inner square completely contained in the shape and also share part of the shared edge. We call this shape a *transit square*. A transit square can move freely between the two convex polygons without leaving the interior. Rotate the two convex shapes (or the local frame) such that the transit square aligns with the axes, then grid the space formed by the two convex shape using the transit square size. Now each component can move to a target square without colliding unmoved pieces following the order of their L_2 distance to the target.

The transit square's orientation is based on the shared edge of two adjacent convex polygons. We would like to compute a square aligned with the x- and y-axis and invariant to the orientation or both polygons. Assume the size (width) of the transit square is L and the smallest angle between edges of the square and x-axis. There exists another square of edge length $l = L\sqrt{1-2\cdot\tan\alpha/(1+\tan\alpha)^2}$ inside the outer transit square. See Figure 2.14a. Another general method without worrying about the angle with axes is to find an inner circle bounded by the transit square then find another inner square in the circle.

We now come to the main result of this subsection.

Theorem 2.5. *Given an intersection of a grid with a pair of adjacent convex polygons, such that at least one complete grid cell is completely contained within each of the transit squares of the polygons, one polygon can be emptied into the other without collisions, using the intersections of the cells with the polygons as components, assuming each component disappears once it has completely entered the empty polygon.*

Proof. Theorem 2.4 indicates that cells in a grid can be translated to a target in order of distance without collision. Since all motions of all points in cells are along straight lines during translation, the result extends trivially to a case where the space is constrained to a convex polygon, and the cells are clipped by the convex polygon, as are the previously-defined *components*.

To move components from the full polygon into the empty polygon, we first look at the area bounded by two adjacent transit squares. The union of the two transit squares is also a convex shape with at least two complete components one in each transit square. Now we move components, one-by-one in ascending L_2 distance order, in the full transit square into one complete component in the empty transit square which can be emptied as the complete component previous was. This process causes no collision as it all happens inside a convex shape.

Since the transit square in the previous filled polygon is now empty, and it contains a complete component, we can move components in the now half-filled convex polygon to the empty complete component in the transit square using the same strategy. Thus components in the full polygon can be emptied one-by-one.

□

Theorem 2.5 suggests a simple but complete algorithm for cutting and extracting the interlocked parts. First, decompose the divisible part $\mathscr{B}$ into a set of nonoverlapping convex polygons. (In our implementation, we used Delauney triangulation.) Remove the convex outside the convex hull of the atomic part $\mathscr{A}$, thus we have an empty convex of $\mathscr{B}$. Compute the smallest axis-aligned transit square over all adjacent convex shapes, and grid the remaining space using the square size. Define a *boundary polygon* as a polygon that is connected by a sequence of adjacent empty convex polygons (the exit sequence) to the outside

of the containing square (the exit).

Figure 2.14b shows how each pixel is extracted into the exit sequence and leave the environment. Choose a boundary polygon (the gray part in Figure 2.14b), and an exit sequence (the white empty convex polygons). Extract square-bounded pieces from the boundary polygon one at a time in the order of L_2 distance to a target next to the shared edge of the boundary polygon and another empty polygon. As each piece enters the exit sequence, move it through the sequence following the motion of the previous piece sitting at the same location. Each piece can thus be extracted from the sequence of connected polygons.

Using this partitioning strategy, we have an upper-bound on the number of piece to cut the divisible part into. Although the particles can be small, especially it considers the global property of the atomic part and local extreme features spread over the environment, we have complete algorithm better than cutting infinitely many particles. Next, we will consider how to reduce the effect of local extreme features and make pieces larger.

Section 2.5

Practical division algorithms

Previously, we looked at two problems: (1) what is the minimum number of pieces the divisible material must be cut into to be separated from the atomic part; (2) how small should a piece be so it is guaranteed to be escapable from the environment with atomic obstacles? Answering the first question gives us a sense of the physical complexity of the problem while answering the second question helps to establish an upper-bound on the number of pieces to cut. None of these are actually going to make practical solutions to our original problem. In this section, we will think about how to make real cuts and separate these two parts $\mathscr{A}$ and $\mathscr{B}$.

We will propose two algorithms for this task, both based on Theorem 2.4 and 2.5. As mentioned in Section 2.4, we can compute a square small enough that no matter where it is placed, it can be moved out of the convex hull of the atomic part and move to infinity. The divisible part is then partitioned into components bounded by the squares. The issue is, if some local features of the atomic part have unexpected geometry and allow only very narrow passage, we will compute a very small square size globally. Thus a lot of small components will be created while most of the divisible part is easy to separate. For example, in Figure 2.15, we only need to partition the divisible part into 2 pieces, orange and green, then pull each piece in opposite directions along y-axis to separate them infinitely far. However, using the algorithm proposed in Section 2.4, we will partition the divisible part into very small pixels as it describes a global property rather than a local one. Hundreds, instead of two, pieces will be created.

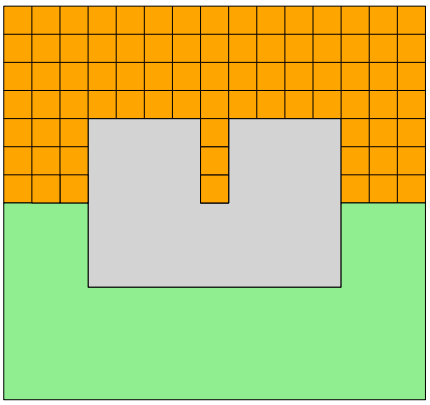


Figure 2.15: An atomic part (gray) with a narrow passage. The divisible part can be separated into two parts, orange and green. However, the narrow passage limits grid size to be very small.

To have a better and practical cut algorithm, our approach optimizes from the worst case. First, cut the divisible part into many small components, then group some components so they can move together as if in the same piece. Remove the piece and repeat the process until all components are removed. An exhaustive approach for grouping components is to grow a piece pixel-by-pixel. Given a piece which is initially empty or has only one component, include neighbor components such that an escape path to infinity exists. The different order of new components being added may result in pieces of different shapes. Among all possible shapes, select the largest one. As a result, we have exhaustively explored all possible expansions that contain the given initially piece. Next, we remove the largest grown piece and repeat the process until all components are removed. This is de-

scribed in Algorithm 4.

Algorithm 4 An exhaustive piece growing algorithm

```
1:  function GROWPIECE(P ← one piece )
2:      candidate ← ∅
3:      for each neighbor component c_n of the piece do
4:          expanded_piece ← piece ∪ {c_n}
5:          if expanded_piece can escape from the environment then
6:              candidate.add(c_n)
7:          end if
8:      end for
9:      if candidate is empty then
10:         return P.
11:     else
12:         grown_set ← ∅.
13:         for c_n ∈ candidate do
14:             grown_set.add(GrowPiece(P ∪ {c_n})).
15:         end for
16:         return the largest piece in grown_set
17:     end if
18: end function
```

One issue with the algorithm is line 5 being expensive, as the pathfinding typically requires searching the configuration space and we have to do the search so many times. However, this is not the worst issue. Even if we can find a path in $O(1)$ time, the runtime of this algorithm is still unfortunately exponential, since we exhaustively iterated all possible combinations of neighbor components.

2.5.1. The complexity for finding an optimal solution

Is an exponential time complexity algorithm unavoidable, if we want to find the optimal solution or the minimum number of pieces to group the components into? Or is it simply because Algorithm 4 is terrible and a good one has not been found? Here we will show the complexity of the optimal expansion search problem is at least *NP-complete*.

Lemma 2.6. *Given an atomic part $\mathscr{A}$ and a set of non-overlapping components $\mathscr{C}$ bounded by squares. Grouping components into a minimum number of pieces such that each piece can be moved to infinity is at least NP-complete.*

Proof. Let c_i be a component, $G(c_i)$ be the largest group of components containing c_i whose escape path does not collide with any atomic components, and $G(c_i)^*$ be the optimal (largest) piece containing c_i whose escape path does not collide with neither the atomic components nor other divisible components not included in the piece. $G(c_i)^*$ is a subset of $G(c_i)$. But in the best case $G(c_i)^* = G(c_i)$.

Suppose there exists an algorithm Alg which solves the optimal partitioning problem and gives the minimum number of pieces covering all components, then Alg also solves a special case of the minimum set cover problem to cover all components c_i over subsets $G(c_i)$ for all $i \in [0, \ldots, n]$ where $G(c_i)^* = G(c_i)$. However, the minimum set cover problem is *NP*-complete.

So our problem of finding minimum number of pieces is at least as hard as finding a minimum set cover for all components, which is *NP*-complete. □

Lemma 2.6 suggests that a fast algorithm for finding the optimal partition is unlikely to exist if the number of components is large. Our best hope is to come up with an algorithm that finds us one feasible and reasonable solution in a good runtime.

2.5.2. A greedy algorithm

Algorithm 4 will give a feasible solution, although the runtime is bad. One idea to improve Algorithm 4 is to completely avoid constructing the configuration space and search escape path in another manner. We give up the desire to find the largest piece containing one component, and only care about finding one piece reasonably large.

Among many divisible components, some are very easy to be moved out of the convex hull of the atomic part. For example, the green part in Figure 2.15. If other components can be moved to the green region, they can be treated as if carried by the green piece, then follow the path of the green piece and escape the environment. Our first algorithm will be based on this observation.

We constrain the motion to be only 4 translation along x and y-axis. We observe that if some components are in the same piece, they will have the exact same path. This observa-

tion suggests a reverse process to group components: based on the path of each component, we can aggregate components into *pieces* and cut only along component divisions. Connected components with the same path will be in the same group.

Let the path of a component be $P = [p_0, p_1, p_2, \ldots, p_n]$ where $p_i \in P$ is the intermediate position after the i-th control (or action), and p_0 is the initial position of the component. Since we define the feasible motions to be only 4 translations, the i-th control, or c_i, can be computed by $c_i = p_i - p_{i-1}$. We define the *control sequence* as $C = [c_1, c_2, c_3, \ldots, c_n]$. If two components have the exact same path, they have the exact same control sequence.

When grid the divisible parts into components, some are of the same size as a complete grid (*whole*), some are smaller than a grid (*partial*). A simple greedy approach to group components is as follows: We first deal with whole components in each of the four cardinal directions. For each direction, count the number of components that reach the exit area which is initially outside the convex hull of the atomic part, and may be grouped together based on 4-connectivity; greedily choose the direction that give the fewest such grouped pieces. For example, after the first time we do this operation, we will find one control direction that moves the most number of components out of the convex hull region. See the orange part in Figure 2.15.

Add the resulting cleared squares (grids) as an exit area, too, and attempt motion in each of the four cardinal translation directions, potentially creating new piece groups; attempt to move the largest group, then add the cleared squares as an exit area.

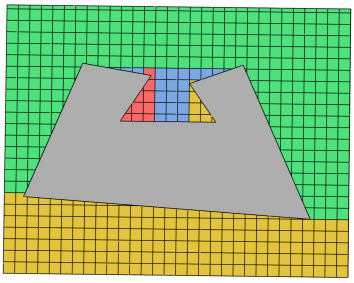
Since each square in the exit area is bounded to a control that moves it to a further exit area square, the path of each pixel is constructed by simply adding the control of the exit area square it sits on. In this grouping algorithm, each action creates one component group.

Algorithm 5 describes the greedy grouping algorithm.

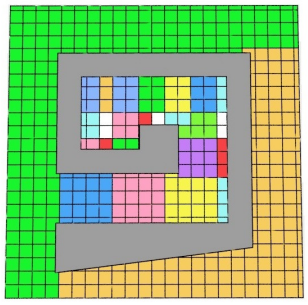
Table 2.2 shows some statistics for the sufficient decomposition for some example shapes, and a few illustrative run times. The main time cost is spent on testing collisions. Decomposition solutions are shown in Figure 2.16.

Algorithm 5 Greedy component grouping algorithm

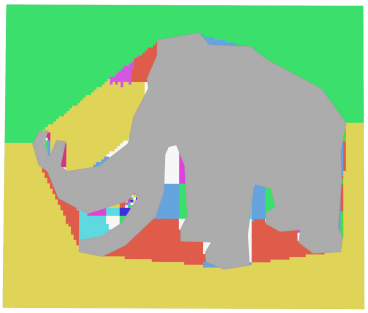
```
1: function GROUPCOMPONENTS(component_set )
2:     exit_area ← areas infinitely far from the atomic part.
3:     while component_set is not empty do
4:         for action in { y+, x+, y-, x- } do
5:             new_group ← Find the max number of components movable to the
   exit_area.
6:         end for
7:         Move the new_group.
8:         exit_area ← exit_area ∩ new_group.
9:         component_set ← component_set \ new_group
10:    end while
11: end function
```



(a) Cavity. The divisible part is divided into 6 pieces.



(b) Spiral. The divisible part is divided into 26 pieces.



(c) Mammoth. The divisible part is divided into 61 pieces.

Figure 2.16: Decompositions of separable parts into pieces. Colors are reused as needed; each set of same-color components is a single piece.

shape	components	pieces	decompose time	planning time
cavity	144	6	0.862 s	0.822 s
spiral	462	26	0.786 s	1.500 s
dumbbell	269	8	0.529 s	0.496 s
mammoth	9954	61	16.816 s	149.192 s

Table 2.2: Analysis of sufficient number of pieces for a few example shapes.

The algorithm is able to reduce the cuts from a large number of components to a small number of pieces, and each piece is guaranteed to be moved out of the convex hull of the atomic part. The problem is some components are still unnecessarily small. For example, some pieces in the hallway of the spiral example should be able to stick with each other and move together. There are two reasons causing pieces to be small: first, we allow only limited translations (4 along axes) and no rotations; second, each piece follows the path of its predecessor. Consequently, each piece can only be smaller than or equal to the size of its predecessor. The benefit is the algorithm runs very fast, each component will be visited at most k times, where k is the number of final pieces.

2.5.3. Piece expansion algorithm

As mentioned above, Algorithm 5 is still not very satisfying, since every new piece will only be smaller than or equal to a previous piece, although the algorithm is pretty fast and complete. To have better results, we will improve Algorithm 4 from another perspective. We may spend some time on pathfinding (line 5), but we are still not willing to do exhaustive iteration over all possible pieces.

A slightly better algorithm is as follows: Start from a divisible component on the boundary as a single piece, if one of the neighbor component can be added to the piece without ruining the escapability (a path to infinity exists), add the component and expand the piece. Repeat until no more component can be added. Then we have a reasonably large piece. Remove the piece, and repeat the process for the rest unmoved components until all are removed. The algorithm pseudocode is described in Algorithm 6. The algorithm iterates

each component at most twice and will finish in a polynomial time.

How much time will it cost for Algorithm 6 to find a piece (stop the inner while loop)? The runtime consists of two parts: one for building the c-space (line 13); another for finding the path (line 14) in the c-space. Assume a piece has N_p number of pixels, T_c is the time complexity of building the c-space when a pixel is added, and T_s is the time for searching a path for a piece. Let T_p be the time for finding a piece of N_p pixels, $T_p = N_p \cdot (T_c + T_s)$. In the rest of this section, we will discuss method that reduces either T_c or T_s in order to improve the overall performance of the piece expansion algorithm.

Algorithm 6 A naive component grouping algorithm

```
1:  function ImprovedExtraction(B ← divisible components )
2:      while B is not empty do
3:          piece ← {one extractable component}
4:          tested ← ∅
5:          neighbors ← {neighbor components of the piece}
6:          while neighbors is not empty do
7:              c_n ← neighbors.pop()
8:              if c_n ∈ tested then
9:                  continue                ▷ avoid checking the same component again
10:             end if
11:             expanded_piece ← piece ∪ {c_n}
12:             tested.add(c_n)
13:             cspace ← build cspace for the expanded piece.
14:             if expanded_piece can escape from the environment then
15:                 piece ← expanded_piece                          ▷ expand the piece
16:                 add new neighbor components to neighbors
17:             else
18:                 remove c_n from expanded_piece          ▷ Restore the previous stage
19:             end if
20:         end while          ▷ All neighbor components of the piece have been tested.
21:         Remove the piece.
22:         piece ← ∅
23:     end while                          ▷ All divisible components have been removed
24: end function
```

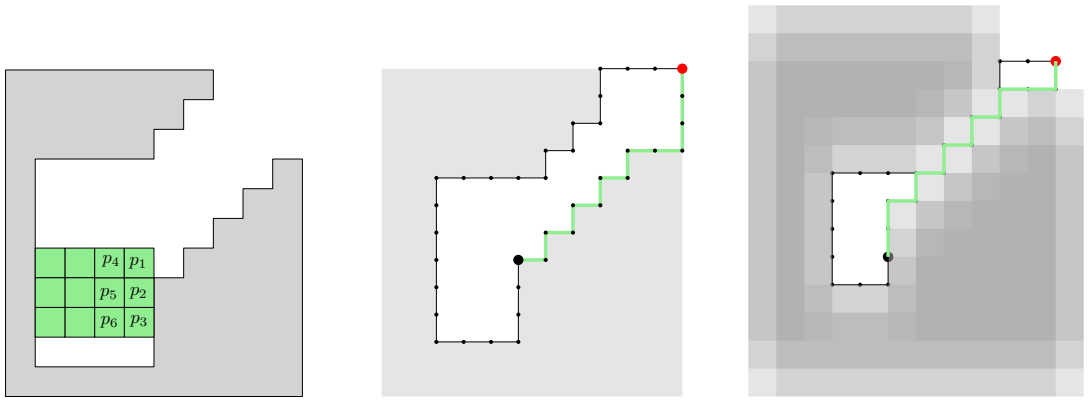
To help understand how the configuration space changes when new components are added or deleted, we will look at a simple example as in Figure 2.17. Here we assume both the atomic part and the divisible part are pixelized, meaning that they are formed by a set

of squares. All divisible components are complete components with the same size of the pixel squares. We limit the motions to only 4 translations along x and y-axis. Each step of translation is only one unit (square width) distance. Later we will relax these constraints to allow also rotations and a fixed degree of rotation.

Figure 2.17a shows an environment of 12 divisible components (green). Our goal is to aggregate them into a small number of pieces and move all pieces out of the convex hull of the atomic part. Figure 2.17b is the configuration space of a single-component piece $\{p_1\}$ without considering other divisible components as temporary obstacles. By banning rotations, this c-space can be viewed as a slice of the complete c-space allowing rotations. We later call this slice a *sub-c-space*. Sliding the piece along the boundary of other components, we will find one path that moves the piece out. In the sub-c-space as in Figure 2.17b, the escape route corresponds to a path connected by the green line segments. See Figure 2.17b. Since the path has pretty wide clearance (empty space around), we can try to add more components into the piece. We add pixels one-by-one such that the piece includes p_1, p_2, p_3, p_4, p_5 and p_6 the configuration space becomes a shape as in Figure 2.17c. At this point, we know no more pixels can be added, because some part of the escape path (green line segments) is in a tight channel of width 0.

The workspace as in Figure 2.17a and the configuration space (Figure 2.17b) of p_1 have very similar shapes. In fact, the sub-c-space obstacles is a Minkowski sum of the p_1 and obstacle components, which is also pixelized! Assuming the top-right corner represents the origin of a component, Figure 2.17b is constructed by extending each pixel at (x_o, y_o) in Figure 2.17a into $\{(x_o, y_o), (x_o + 1, y_o), (x_o, y_o + 1), (x_o + 1, y_o + 1)\}$ four components. Similarly the configuration space for 6 pixels is not too different from the configuration space of one pixel. Figure 2.17c can be viewed as the overlay of 6 c-space in Figure 2.17b with each one being slightly shifted according to the newly added component position, each layer represents the configuration space of one divisible component.

Now that we have some understanding of the configuration space, how can we find a



(a) An environment with some divisible components (green).

(b) The configuration space of component p_1 without considering other divisible components as obstacles.

(c) The configuration space of piece $\{p_1, p_2, p_3, p_4, p_5, p_6\}$ without considering other divisible components as obstacles.

Figure 2.17: Finding a feasible large peice to escape the atomic part. Green line is a feasible path for the piece to move out of the environment.

path in the space that leads the piece to the boundary of c-space obstacles? Next, we will introduce two methods for this task. One tries to avoid building the sub-c-space at all and directly search for a path, the other will spend a little effort in constructing c-space, then search for a path.

Naive pathfinding and time complexity. With both atomic and divisible parts pixelized, the configuration space of a divisible piece allowing only translation motions can also be described using a set of pixels, as shown in Figure 2.17. Since each pixel must move at least one-pixel distance to collide with obstacles, continuous translation motions are also decomposed into a sequence of discrete one-pixel per step translations. The free configurations of a piece are therefore captured using a set of discrete points. Searching a path in the c-space does not necessarily require reconstructing the c-space obstacles. Algorithm 7 is a simple depth-first search for a path. In the worst case where no path exists, the algorithm will visit all free configurations reachable from the start point.

Algorithm 7 is complete. The collision check operation in line 6 can be done by iteratively checking if one divisible component in a piece is in the atomic components set, or

Algorithm 7 A simple pathfinding algorithm for moving a divisible piece.

```
1:  function DFSPATHSEARCH( start, goal )
2:      if start = goal then
3:          return true.                                   ▷ Path found!
4:      end if
5:      for each neighbor n of do
6:          if n was not visited and n is free  then       ▷ O(n) time collision check
7:              if n is reachable from start then          ▷ Assume it is O(1)
8:                  if DFSPathSearch(n, goal) then
9:                      return true.
10:                 end if
11:             end if
12:         end if
13:     end for
14:     return false.                                      ▷ Path not found
15: end function
```

reversely check if an atomic component is in the piece. This operation takes $O(N')$ time where N' is the minimum between the number of obstacle pixels and the number of divisible components. If there are K free configurations reachable from the start point, in the worst case where an escape path does not exist, the algorithm takes $O(KN')$ time. Line 7, which checks the reachability from start configuration to one of its 4 neighbors, can be done in $O(1)$ if there are only translations because the c-space obstacles are also a set of pixels.

As a result, the time complexity for Algorithm 6 to find a piece of N_p pixels becomes $T_p = N_p \cdot (T_c + T_s) = N_p \cdot (O(1) + O(KN')) \leq O(N_p) + O(N_p^2 \cdot K)$.

Avoiding the collision check. If we can avoid the $O(N')$ collision check operation or make it faster, the pathfinding algorithm can potentially be boosted. Notice that the configuration space is approximately the same size of the workspace. If we can keep a track of the c-space obstacles and invalid configurations, we can easily test collision by checking if the current configuration is in the invalid configurations set. This strategy will reduce collision detection to a constant time operation, however, require efforts to build the sub-c-space, which sacrifices some space complexity. Does the trade-off give us actual benefits? We

will now show an implement of this strategy and analyze its time complexity.

We introduce a method for rapid c-space construction with the support of adding and removing components. The algorithm starts with an empty piece and keeps expanding using neighbor divisible components. Each time a new component is added, our algorithm keeps a record of c-space obstacles and invalid configurations, so later, we can quickly answer if a configuration is in collision or not. Although this extra step takes time and space in building sub-c-space, later we will show it actually speeds the component expansion algorithm up. We take the top-right corner of a pixel as the origin. Configuration (x_0, y_0) of a piece means the first added component of a piece sit at a location where its top-right corner is at (x_0, y_0) and its edges are aligned with $x-$ and y-axis.

Initially, a piece is an empty set of divisible components, both divisible and atomic components are obstacles to the piece because the piece cannot collide with any static components while moving. Our piece grow algorithm works by repetitively expanding a piece by one component and test if the expanded piece can escape. If so, keep the component in the piece, otherwise, forget this component, restore to the previous stage and try other ones. When expanding a piece, we are sequentially doing three operations:

1) Remove one divisible component not in the piece from the obstacles (Algorithm 8).

2) Add the removed divisible component to the current piece (Algorithm 9).

3) Search for an escape path.

If the search is not successful, we will perform two more operations to restore the previous configuration space:

1) Remove the previously added divisible component from the piece (Algorithm 10).

2) Add the removed divisible component back to the obstacles set (Algorithm 11).

Details of these operations are described below. To better visualize the configuration space, our figures in 2.19 and 2.20 show both the configuration obstacles (gray pixels with

numbers indicating the number of obstacles in collision) and invalid configurations (black dots). But in the real implementation, we only keep a track of the invalid configurations. The invalid configurations can be viewed as a set allowing duplicated elements. It is implemented using a dictionary with configurations being the keys and their number of appearance as values.

- **Remove a component from obstacles (Algorithm 8).** Let the removed component be at (x_c, y_c). This operation removes 4 c-space obstacles $\{(x_c - x_p, y_c - y_p), (x_c - x_p + 1, y_c - y_p), (x_c - x_p, y_c - y_p + 1), (x_c - x_p + 1, y_c - y_p + 1)\}$ for each component (x_p, y_p) in the piece, as well as one invalid configuration $(x_c - x_p, y_c - y_p)$. Figure 2.20d is the result of removing one obstacle from Figure 2.20a. Orange circled area is where the removed c-space obstacles sat. This operation takes $O(N_p)$ time where N_p is the number of components in a piece.

- **Add a component to piece (Algorithm 9).** Let the component be at (x_c, y_c). For each obstacle (x_o, y_o) in the obstacles set, 4 c-space obstacles $\{(x_o - x_c, y_o - y_c), (x_o - x_c + 1, y_o - y_c), (x_o - x_c, y_o - y_c + 1), (x_o - x_c + 1, y_o - y_c + 1)\}$ are created and added to the c-obstacles set. Similarly, one invalid configuration $(x_o - x_c, y_o - y_c)$ is added for each workspace obstacle (x_o, y_o). This operation takes $O(N_o)$ time, where N_o is the number of workspace obstacles. Figure 2.19d is the result of adding one component to the green piece in Figure 2.19a. The circled 4 pixels in Figure 2.19b are c-space obstacles created by the bottom-left workspace obstacle. The circled pixels in Figure 2.19d are all newly created c-space obstacles.

- **Remove a component from piece (Algorithm 10).** Since we only remove one component from the piece when there is no path found, the pixel to remove was the one just added in Algorithm 9. This operation is simply deleting c-space obstacles and invalid configurations created in the previous step, thus have the same time complexity.

- **Add a component to the obstacles (Algorithm 11).** Similarly to Algorithm 10, this operation is executed when a path is not found, hence can be viewed as a reverse process of Algorithm 8. We simply add the removed c-space obstacles and invalid configurations back. The time complexity is the same, $O(N_p)$.

All the operations above take at most $O(\max(N_p, N_o))$ time, where N_p is the largest piece size and N_o is the number of workspace obstacles. When doing path searching, the collision check can be done in $O(1)$ by querying if a configuration is in the invalid configurations set, therefore path searching Algorithm 7 takes $O(K)$ time in the worst case. Recall that K is the number of valid configurations reachable from the starting point. As a result, the time complexity for Algorithm 6 to find a piece of N_p pixels becomes $T_p = N_p(T_c + T_s) = O(N_p \cdot \max(N_p, N_o)) + O(K \cdot N_p)$. As a comparison, the previous algorithm which does not build c-space will take at most $O(N_P) + O(K \cdot N_p^2)$. This algorithm could be better than the previous one, unless there are much more atomic components than the divisible component, making N_o significantly larger than $K \cdot N_p$.

One way to reduce the number of atomic components is to only consider the boundary components of the atomic part, digging out the atomic components surrounded by other atomic components in all four orthogonal directions. Because the pathfinding is not going to explore configurations deeply inside a c-space obstacle, which corresponds to deep penetration between the piece and obstacles, knowing the surface of workspace obstacles is enough to prevent invalid path. In the worst case where the atomic part is a single shell of pixels and the divisible part is also a hollowed part, this improvement will not change the runtime of the algorithm. However, practically this improvement reduces the obstacle pixels iterated in Algorithm 8 and 9 from a filled two-dimensional shape to just the boundary of the shape, and greatly accelerates the piece grow algorithm.

A result of this algorithm is shown in Figure 2.18 where we want to remove some divisible parts from a floor plan. This environment is of 2000×1044 pixels large with 5920 divisible components (pixels). The atomic part (black) is hollowed inside to speed up

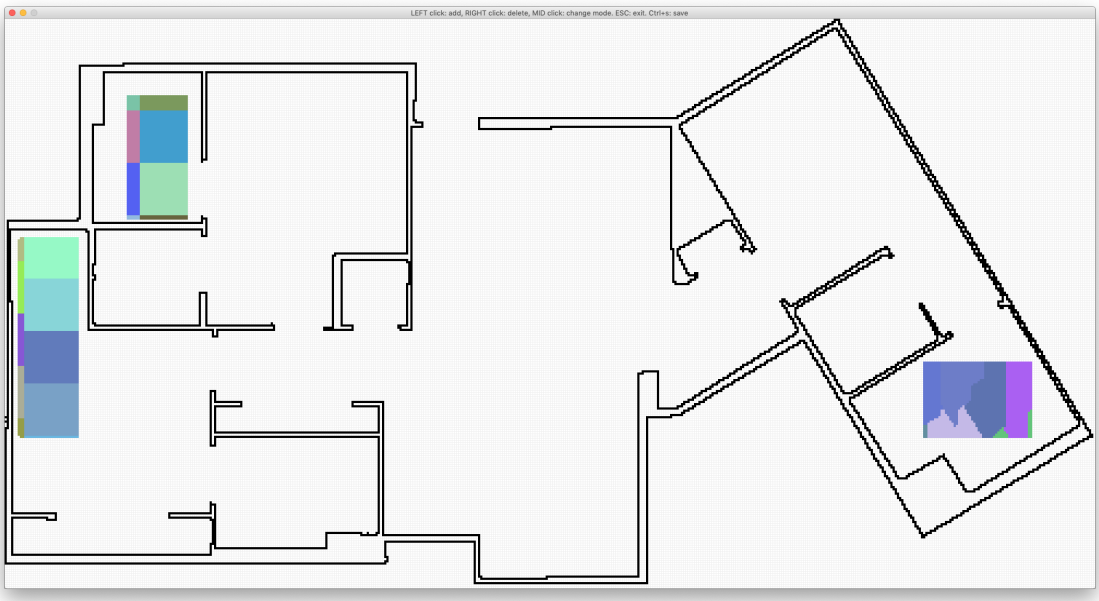


Figure 2.18: Partitioning and extracting some parts (colored) in a floor plan. Black pixels are atomic. The environment is of 2000×1044 size. This environment allows only translations. Colored divisible materials are partitioned into 25 pieces.

the algorithm.

Algorithm 8 Removing a component from the workspace obstacles.

```
1: function REMOVEOBSTACLECOMPONENT((x_c, y_c))
2:     remove_history ← empty list.
3:     for each component (x_p, y_p) in the piece do
4:         invalid_configs.add((x_c − x_p, y_c − y_p))
5:         remove_history.add((x_c − x_p, y_c − y_p))
6:     end for
7: end function
```

Integrating rotations. One reason the configuration space can be quickly constructed is because we only allow translations. The configuration space is the Minkowski sum of the workspace object and obstacles under this constraint. The problem caused by such convenience is obvious: the piece computed in the previous method is smaller than the actually piece that can be moved out of the environment without rotation constraints. For example in Figure 2.21, the divisible piece (colored parts) is partitioned into two pieces

Algorithm 9 Adding a component to the piece.

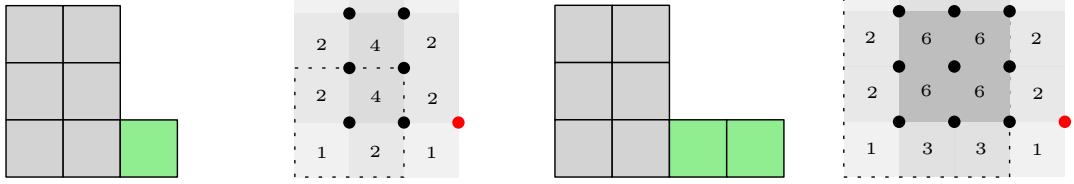
```
1: function AddComponentToPiece((x_c, y_c))
2:     add_history ← empty list.
3:     for each workspace obstacle (x_o, y_o) do
4:         invalid_configs.add((x_o − x_c, y_o − y_c))
5:         add_history.add((x_o − x_c, y_o − y_c))
6:     end for
7: end function
```

Algorithm 10 Removing a component to the piece.

```
1: function RemovePieceComponent
2:     for (x, y) ∈ add_history do
3:         invalid_configs.remove((x, y))
4:     end for
5: end function
```

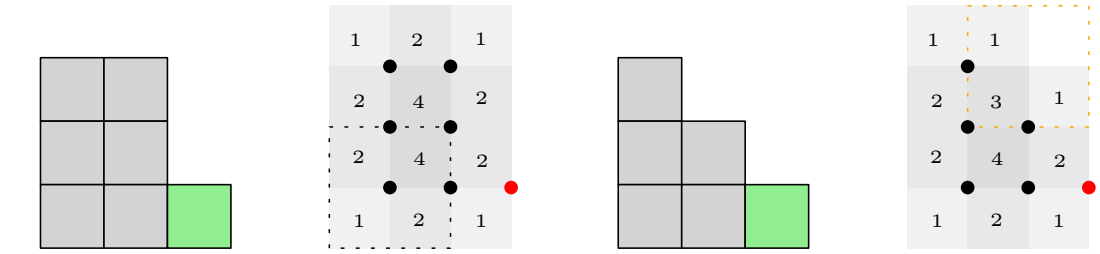
Algorithm 11 A naive component grouping algorithm

```
1: function AddComponentToObstacle
2:     for (x, y) ∈ remove_history do
3:         invalid_configs.add((x, y))
4:     end for
5: end function
```



(a) A green piece with one component. (b) The c-space for the 1-component piece. (c) A green piece with two components. (d) The c-space for the 2-component piece.

Figure 2.19: Configuration space construction for adding a divisible component. Gray pixels are obstacles. Numbers (opacity) indicate the number of obstacle pixels in collision. Red dot is the current configuration of the green piece. Black dots are invalid configurations.



(a) A green piece with one component. (b) The c-space for the 1-component piece. (c) A green piece with two components. (d) The c-space for the 2-component piece.

Figure 2.20: Configuration space construction for adding an obstacle component. Gray pixels are obstacles. Numbers (opacity) indicate the number of obstacle pixels in collision. Red dot is the current configuration of the green piece. Black dots are invalid configurations.

when only translation is allowed. However, if rotation is permitted, the two pieces can actually move together to get out of the maze.

The complete configuration space including rotation and translation is hard to construct explicitly and exactly, although it is only three dimensional. In order to relax the constraint on the permitted motions, we select a discrete set of orientations of an object to represent rotations. For example, if select 18 orientations $\{0, 20, 40, \ldots 320, 340\}$, under each fixed orientation, the sub-configuration space (a slice of the complete c-space) is also a Minkowski sum of the rotated object and the obstacles. Therefore we can quickly build 18 sub-c-spaces, one for each orientation. They are 18 slices of the complete configuration space. Figure 2.22 is a configuration space constructed using 60 slices for the green piece in Figure 2.22a.

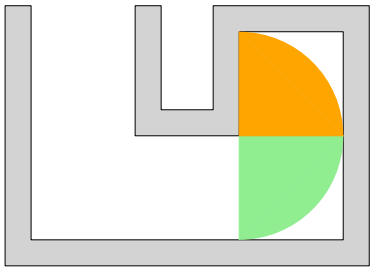
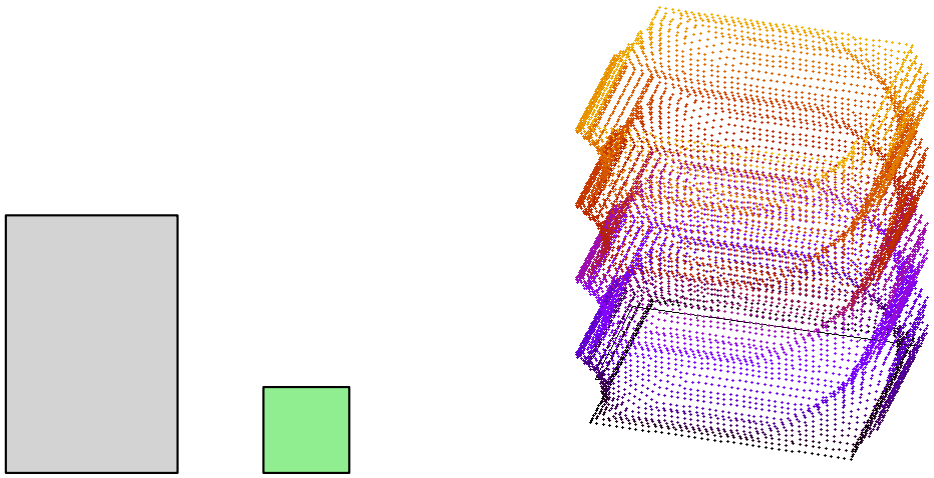


Figure 2.21: Without rotation, the divisible part must be cut into two pieces (orange and blue) to move out of the maze.

Any rigid body motion can be viewed as a translation combined with rotation. In our method, we will simplify this model by only allowing either one unit of translation or one unit of small rotation in every single motion. In other words, we use two actions (translation or rotation, one after another) to approximate a rigid body motion. This simplification



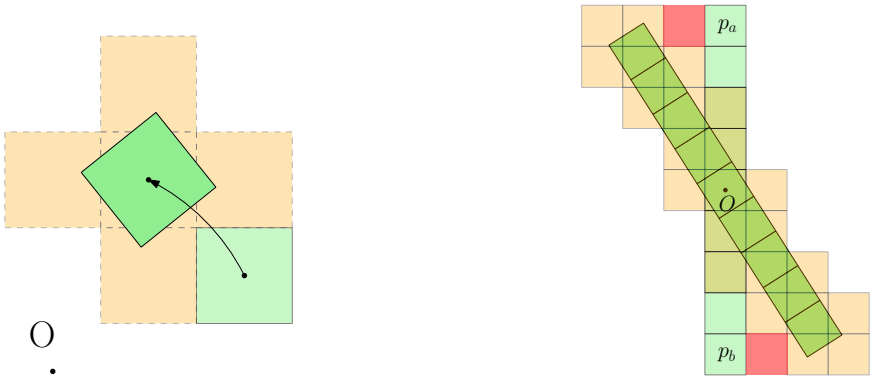
(a) Workspace of a piece (green) and some atomic components (gray). The piece's configuration space obstacle is shown in the right figure.

(b) The obstacle is represented using a set of discret invalid configurations in a grid space. There are 60 layers representing 60 orientations.

Figure 2.22: The configuration space is approximated using a set of sub-c-space each representing a fixed orientation.

allows us to treat the configuration space as a grid space of discrete valid and invalid configurations. Motion planning is a pathfinding problem in the grid space.

Our previous method is based on the fact that when the obstacle and divisible pixels are all axis-aligned, the Minkowski sum is also an axis-aligned shape, thus building a configuration space can be done quickly. However, if some pixels are rotated about a point, the axis-aligning property does not hold anymore. To preserve this convenience, we take a conservative estimate of the rotated object by creating a bounding shape that includes all pixels intersecting with the rotated object. The invalid configurations of the corresponding sub-c-space form a superset of the actually invalid configurations. Thus, we guarantee no collision will occur. Figure 2.23a is an example where a component is rotated 30 degrees about a point O. The resulting component intersects with 5 pixels in the space as shown in orange. We consider all 5 pixels are in the rotated component to conservatively avoid collisions. As long as these 5 orange pixels are not colliding with obstacles when moving, the original component is safe. Define $R_O(P, \theta)$ as the set of pixels intersecting with piece



(a) Rotating a component about point O for some degrees. Five pixels (orange) are intersecting with the result component. We take the five pixels as a conservative estimation of the rotated shape.

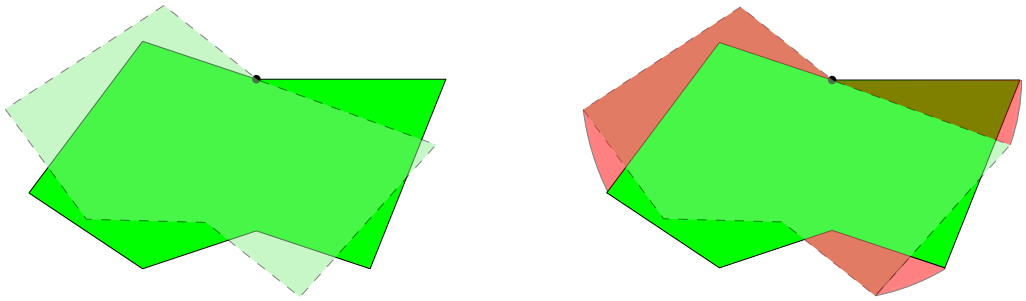
(b) Rotating a vertical piece by 30 degrees. To make sure the rotation is feasible, we must check (1) the green components are free, (2) the orange components free, and (3) components p_a and p_b do not collide with obstacles.

Figure 2.23: We use the set of orange pixels intersecting with a rotated piece to represent a conservative estimation of the rotated shape. The two red pixels are swept when rotating the piece, but not included in neither the original piece nor the rotated one.

P when it is rotated by θ about point O. $R_O(P,\theta)$ can be computed by first figuring out which pixel the rotated component center sits in, then iterate over its 8 neighbor pixels to check collisions. In the worst case, a rotated component intersects with at most 5 neighbor pixels. This computation takes $O(1)$ time.

Inside a sub-c-space, Algorithm 7 will find a path between two configurations just like before. However, connecting configurations between sub-c-spaces needs some extra work. The challenge is to quickly test if a small amount of rotation will make the piece to collide with obstacles.

An intuitive way for checking collision is to rotate each component at the boundary of a piece, compute all pixels swept by the rotation, then check if there is collision against any obstacle boundary pixels. This is a bad idea. Figure 2.24a shows the rotation of a green piece by about 20 degrees. The boundary of the shape has swept a large area as shown in Figure 2.24b. Any pixel intersecting with the red area must be tested and see if the obstacles set contains a same pixel. Assuming the number of boundary components in a piece is $N_{pb} \leq N_p$, checking if a configuration can reach another neighbor configuration takes at



(a) A piece before and after rotating about a center (block dot).

(b) The shaded red area is where the shape boundary sweeps.

Figure 2.24: Testing if a rotation will cause collision requires testing if the swept area (red) intersects with any obstacles.

the least $O(N_{pb})$ time. Algorithm 7 total runtime becomes $O(K \cdot N_{pb})$ or worse. In other words, this intuitive rotation feasibility check makes our effort to reduce the collision check time complexity inside each slice of sub-c-space almost pointless. So can we speed up this process? Notice that the red areas in Figure 2.24b have a lot of overlaps with the rotated piece, and collision detection for the rotated piece is done prior to testing the rotation feasibility. Can we select only the non-overlap region and reduce the time complexity?

Figure 2.23b shows a rotation of a vertical piece by 30 degrees. The resulting shape is covered by some orange pixels as a conservative estimation. We are able to check if the original piece is in collision in $O(1)$ time and the same for the rotated piece. However, this is not enough, because the pixels p_a and p_b each swept one more pixels next to them and these two pixels are not included in neither the original piece nor the rotated one. If we can find out which components will sweep pixels not included in the original shape and the rotated shape, we can do a conservative rotation feasibility check by testing the swept pixels against the obstacles after making sure the original and rotated pieces are both not in collision.

Let the rotation center of a piece P be point O. We use $B_O(P,\theta)$ to represent the set of components in P whose sweep area intersects with pixels neither in $R_O(P,0)$ nor in $R_O(P,\theta)$, when the piece is rotated by θ degree(s) about point O. $B_O(P,\theta)$ is conservatively computed by checking if one of four corner points travels more than a pixel unit

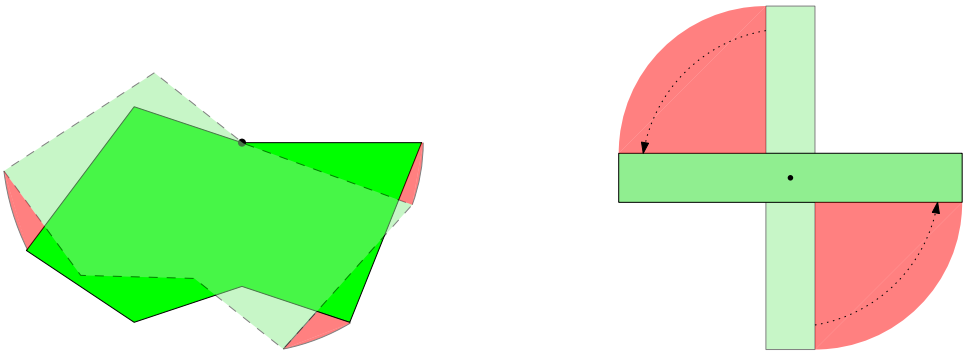
distance, and the intersecting pixel not in $R_O(P,\theta)$. This computation includes slightly more components than $B_O(P,\theta)$ actually has. $B_O(P,\theta)$ are the set of pixels that needs extra work to make sure a rotation of θ degrees is collision-free. We first compute the swept pixels by components in $B_O(P,\theta)$, then iteratively check if any swept pixel is also in the obstacle pixels set.

Algorithm 9 is now updated as in Algorithm 12 to reflect the computation of $B_O(P,\theta)$ and its swept pixels. A history record is kept in case the component will be removed later. Assuming the special components and its swept area is smaller than or equivalent to the number of obstacles or the piece itself, this does not change the performance of the algorithm. The swept area is computed only once in the local frame of the piece, and can be reused later.

Algorithm 12 Adding a component to the piece.

```
 1: function ADDCOMPONENTTOPIECE((x_c, y_c))
 2:     add_history ← empty list.
 3:     for each orientation θ_i do
 4:         rotated_component ← R_O((x_c, y_c), θ_i)
 5:         for each workspace obstacle (x_o, y_o) do
 6:             for (x_i, y_i) ∈ rotated_component do
 7:                 invalid_configs.add((x_o − x_i, y_o − y_i))
 8:                 add_history.add((x_o − x_i, y_o − y_i))
 9:             end for
10:         end for
11:         if the added component is a special component in B_O(P, θ_i − θ_{i−1}) then
12:             for pixel intersects with the component when rotated θ_i − θ_{i−1} do
13:                 if pixel not in original piece nor in the rotated piece then
14:                     swept_area.add(pixel)
15:                     swept_area_history.add(pixel)
16:                 end if
17:             end for
18:         end if
19:     end for
20: end function
```

Later in pathfinding Algorithm 7, Line 7 will take $O(N_s)$ time, where N_s is the number of swept pixels not covered in $R_O(P,0)$ nor in $R_O(P,\theta)$, to iteratively check if a swept pixel



(a) Regular case of rotating a piece by 20 degrees. Black dot is the rotation center. Red areas are swept by rotation and need extra collision check. The swept area is very small.

(b) Worst case of swept pixels collision check. The piece has spiky structures and rotates by a very large degree. The swept area is equivalent or larger then the original piece.

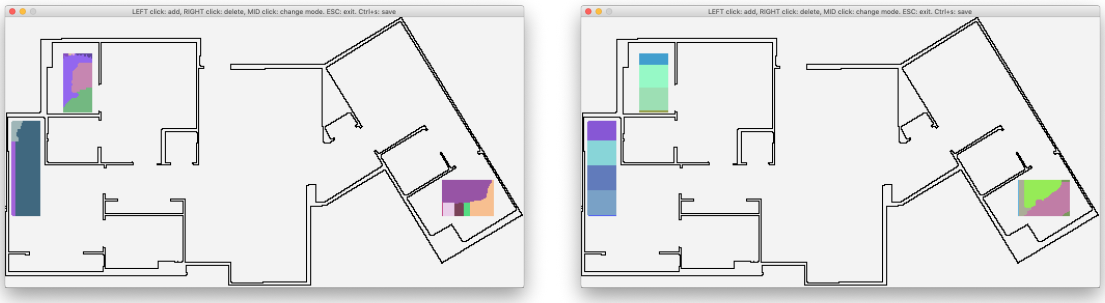
Figure 2.25: When rotating a piece, swept pixels (red areas) need extra effort to confirm rotation feasibility. Normally, the swept area is very small compared to the piece size, but in the worst case, it can be large and slow down the algorithm.

is in obstacle components set. In the idea. case, if the rotation is small enough and no component is far from the piece rotation center, $R_O(P,\theta)$ will be of size 0. Then we spend no extra time for checking rotation feasibility. In the worst case, as shown in Figure 2.25b, the swept area is about the same as or even more than the original piece. The pathfinding algorithm is still slow. But this only happens to extreme cases where these conditions are satisfied: (1) the rotations are big, (2) components in a piece are distributed biasly, and (3) many rotations are performed along the path. Normally, we are more likely to see situations where the number of swept pixels is small and makes almost no impact to the algorithm performance.

2.5.4. Performance improvement

The algorithm presented above is complete. However, it is going to consume a huge amount of time and computing resource. Figure 2.26a is an example of floor plan environment similar to Figure 2.18 but shrunk to 1000×522 size. With 72 orientations, the problem is solved in 28048 seconds or 7 hours and 47 minutes.

The major issue is there are still too many divisible components, every step of piece expansion brings only one more component to the current piece, and this process has to be



(a) 16 pieces, 72 orientations, solved in 28048s. (b) 15 pieces, 120 orientations, solved in 1410s.

Figure 2.26: The same floor plan environment of 1000×522 size. We solve the same problem of moving divisible parts (colored) out of the rooms using different algorithms. With our improved algorithm, the performance is speed up by a lot even with more orientations.

repeated many many times. If we can expand multiple components in one single step, we can potentially speed up the algorithm.

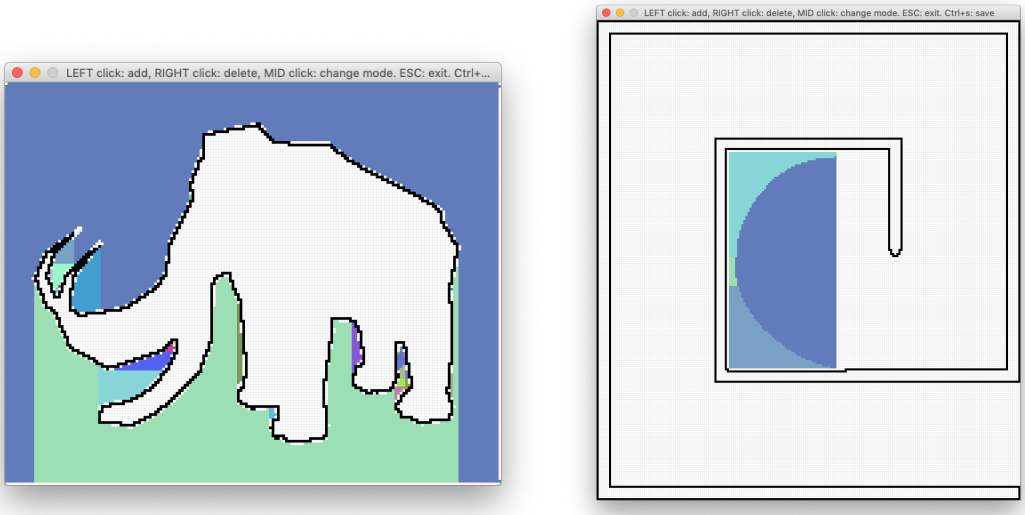
To solve this issue, we downsample the environment into a smaller resolution. For example, an environment of 400×400 pixels is down sampled into 100×100 size. Each pixel in the shrunk environment represents 4×4 pixels in the original environment. A pixel is atomic if one original pixel in the the original environment is atomic; it is divisible, if all original pixels are divisible. Some divisible component will be left neither atomic or divisible in the shrunk environment. We solve the shrunk environment first, then project back to the original environment to expand pieces using left over divisible components.

If the environment is large, the shrunk environment can be further shrunk to an even smaller environment.

Figure 2.26 shows an example environment of 1000×522 size. Without the multi-resolution approach, we solve the problem in 28048 seconds. The multi-resolution approach boosts the algorithm to 1410 seconds.

2.5.5. Experiments and results

Our experiments are run on a Lenovo Yoga 910-13IKB laptop with an Intel Core i7-7500U CPU and 16GB memory. Our code is written in C++ and runs on top of a Windows 10



(a) Mammoth environment with 36 orientations. 558×453 size. 17106 divisible components. Solved in 1564 seconds. 29 pieces.

(b) Spiral environment with 120 orientations. 615×699 size. 5460 divisible components. Solved in 283.5 seconds. 4 pieces.

Figure 2.27: Packaging a mammoth and extracting materials from a spiral environment.

operating system.

Figure 2.27a is the mammoth environment same as in Figure 2.16c. Previously, we need to partition the divisible part into 61 pieces. Now the number is improved to 29, although we spent much more time (1564 seconds vs 166 seconds).

The Spiral environment in Figure 2.27b is larger than the mammoth environment, and a lot more orientations (120 vs 36) are considered. However, the number of divisible components is much smaller, so the run time is faster.

Figure 2.28 is a large environment of 1698×822 size with 34914 divisible components.

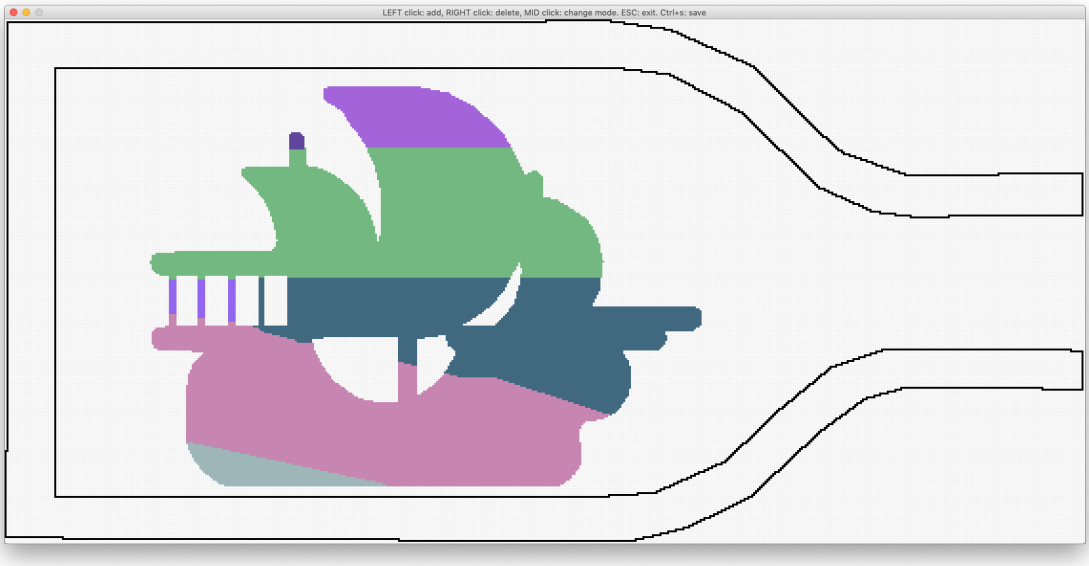


Figure 2.28: Ship-in-a-bottle environment with 180 orientations. 1698×822 size. 34914 divisible components. Solved in 17041 seconds. 9 pieces.

Chapter 3

Interlocking block design and assembly

This chapter presents a design for interlocking blocks and algorithms that allows these blocks to be assembled into desired shapes. During and after assembly, the structure is kinematically interlocked if a small number of blocks are immobilized relative to other blocks.

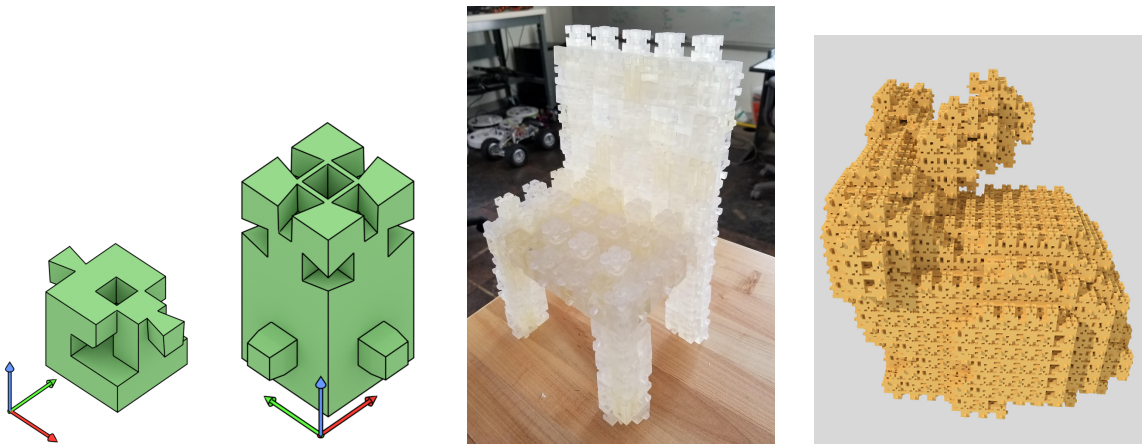
There are two types of blocks: cubes (Figure 3.1a) and double-height posts (Figure 3.1b), each with a particular set of male and female joints. Figure 3.1c and 3.1d demonstrate two structures assembled using these two kinds of blocks. The chair model of over 400 pieces is assembled by hand following orders generated by our algorithm. The Stanford bunny model has over ten thousand pieces and is assembled in simulation.

As a proof of concept, we also used a robot system to assemble 48 blocks into an interlocking cube-like structure.

Section 3.1

Motivation

Many structures, such as furniture, houses and machines are large. Directly manufacturing as single pieces is neither practical nor preferable. Shipping, repairing and remodeling are challenging as well. Practically, these structures are built from smaller pieces. These small components are selected to be easily manufacturable then connected by screws, nuts or



(a) A short block with joints.

(b) A tall block with joints.

(c) A chair assembled using two kinds of blocks.

(d) Stanford bunny assembled using two kinds of blocks.

Figure 3.1: General purpose interlocking blocks and structures assembled using the blocks. The chair model has only one piece movable. The bunny has two movable piece on top of each ear.

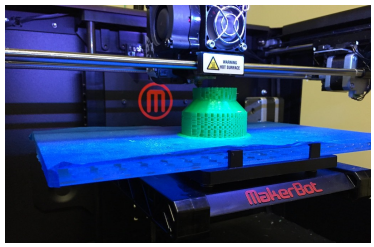
nails to form a desired shape.

Going into a hardware store like Home Depot, we can find thousands of different parts for building different things. Each kind may require a production line in a factory to manufacture. Even organizing so many different parts for storing is a challenging task. Complicated robot systems are built for storing and accessing different parts in warehouses. Is it possible to select only a small number of components whose combination can be assembled into a very large number of different structures? If so, a lot of these troubles can be avoided.

Lego is probably a component closest to our imagine. Figure 3.2a is a department store model assembled using many Lego bricks. One problem is no one seems willing to use Lego for building serious structures like bridges, houses or even cabins. This is because Lego bricks connect relying on friction, and friction is not reliable. We can disassemble blocks or even some layers easily. What we want is something simple like Lego, but reliable and hard to disassemble. Another issue is so many bricks are needed to build a single structure using Legos. Assembling with human labor is an extremely time consuming and

(a) A department store model assembled using hundreds of lego blocks.

(b) Many Dartmouth buildings are constructed using bricks.



(c) ABS is used as a 3D printing material to building interesting shapes.

Figure 3.2: Some simple components capable of forming complex structures.

boring job. It will be better if these components can be handled easily by robots such that simple pick-and-place actions can form us desired shapes.

Bricks is another possible option. As one of the most common construction materials that has been used for centuries, it is simple to manufacture and easy to lay out. Many buildings at Dartmouth College are built using bricks and concrete. See Figure 3.2b. Similar to Lego, bricks are super simple in shape and can be manufactured in large scale. Connected by concrete, brick built architectures are super strong and reliable. Bricks may be ideal for human, but cement is not the best for robots to handle. Also, cement glues bricks permanently, making the construction component not reusable.

3D printers and 3D printing materials are probably another option (Figure 3.2c). ABS material is our component and a 3D printer is a robot that builds structures using the single component. Recent development of 3D printing technology grants the ability to fast prototype parts in arbitrary shapes. Any structure can be concentrated in an STL file and shared any where in the world. The sizes of the fabricated objects are, however, bounded by the working volume or the size of the printer.

In this chapter, we will examine the problem of building structures of arbitrary geometric shape (specified by voxels) using a small set of components. The goal of our work is to enable robotic assembly of large structures from blocks that interlock without need for glue,

cement, screws or other connectors. We hope the blocks are simple in shape, so no complicated production lines are needed to manufacture. Robots should be able to manipulate each block with very little effort, for example, using simple rigid body motions to pick and place components without specially designed attachments for gluing or pasting concrete. Lastly, the finished structure must be reliable that disassembly is almost impossible.

Our work in this chapter will try to overcome all the problems mentioned above. We will present some designs of blocks with joints. Kinematic interlock presents some advantages over traditional connection methods such as glue, cement, screws, nails, or friction locks. The assembly requires pure translations and interlocks blocks with each other using only geometric constraints, providing convenience for robotic construction. The blocks will be reusable and pre-fabricated to support massive production. The completed structure will be interlocked with only a small number of movable pieces, ensuring rigidity and stability. Relative to 3D printing, fabricating in parts has several advantages, including efficient manufacturing process, simple storage, transportation and replace-ability.

An example of interlocking structure with only one key is shown in Figure 3.3a. Figure 3.3b is a burr-puzzle inspired pavilion by architect Kengo Kuma. Figure 3.3c is an interlocking house built out of over 10,000 blocks.

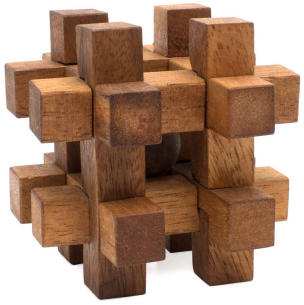
In this chapter, we will briefly review the blocks design that we have done in the past two years, the algorithm to layout blocks to form an interlocking structure, and some completed experiments.

Section 3.2

Related work

3.2.1. Interlocking structures

Interlocking structures have a long history. Wood joints such as the dovetail and mortise and tenon are used in carpentry around the world; in China and Japan, complex interlocking designs have permitted the construction of wooden buildings with no screws or nails [125];

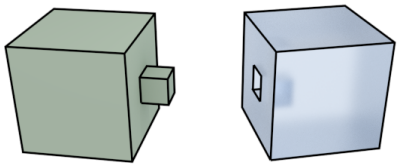


(a) An interlocking puzzle caging a ball at the center.

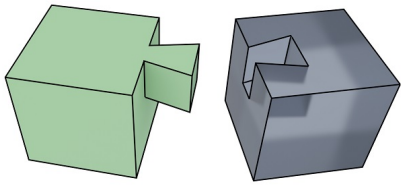
(b) A burr-puzzle inspired architect by Kengo Kuma.

(c) A structure built out of over 10000 blocks.

Figure 3.3: Interlocking puzzle and puzzle-like interlocking structures.



(a) A simple joint design allowing motions only in the normal directions of the contact surfaces.



(b) A simple joint design allowing motions only in the tangential directions of the contact surfaces.

Figure 3.4: Two kinds of joints.

In the paleontology community, evidence has recently been presented that supports a hypothesis that the backbones of theropod dinosaurs interlocked to provide support for the extremely large body mass [116].

The present work is closest in spirit to Song *et al.* [95, 96, 94, 40, 110] which consider the problem of designing reusable components to be assembled into different forms relying on geometric constraints; the primary contribution of our work is a universal block design and layout algorithm that allows construction of arbitrary geometries. Yao *et al.* [119] proposed a method for interactively designing joints for structures and analyzing the stability. Kong *et al.* applied curve matching techniques for finding solutions for assembly of 2D and 3D interlocking puzzles; the layout algorithms considered in the current paper generate assembly motions together with the design.

3.2.2. Computational design

Computational methods have been applied in many design problems to automatically generate functional objects or optimal structure. With the development of advanced manufacturing techniques, computational approaches help to build physical artifacts that was never possible in the past. Bächer *et al.* [8] design spinnable object by optimizing the mass distribution of 3D printed model. A similar approach was take by Wang *et al.* [108] for designing objects with different buoyancy properties.

Computational method is also used for robot designs. Felton *et al.* [37] designed a crawling robot that can fold itself from a flat sheet. Sung *et al.* [100] took the same philosophy and built a framework for building robots using foldable materials. They designed some fold patterns of joints and showed these patterns combined can build entire robots. Du *et al.* [33] presented an interactive system for design, optimization and fabrication of multi-copters that allow non-expert to build a wide range of multi-copters with different specifications.

3.2.3. Robotic construction

Robotic construction research dates back to the 1990s [3] when Andres *et al.* created a prototype, ROCCO, capable of gripping and laying bricks. The same robotic system was later applied to site assembly operations by Balaguer *et al.* [10]. Many robot systems are developed for construction ever since [5]. Cousineau and Miura's book [27] surveyed over 100 construction robots and five automated systems in Japan. These robot systems already have significant impact on Japanese architecture. Pritschow *et al.* [77] used a mobile bricklaying robot to help some skilled workers to do masonry constructions.

More recent works include DimRob, a system with an industrial robot arm mounted on a mobile platform (Helm *et al.* [47]) used for construction tasks. This prototype was later developed into a mobile robot, In situ Fabricator, for construction at 1:1 scale (Giftthaler *et al.* [41]). Wismer *et al.* [117] designed a mobile system to place blocks embedded with magnets to for a roofed structure which the robot can later enter.

Drones are also used for automatic construction. Willmann *et al.* [115], for example, used autonomous flying vehicles to lift and position small building elements. Augugliaro *et al.* [7] demonstrated a system of multiple quadrocopters precisely laying out foam blocks forming a desired shape. Lindsey *et al.* [59] built cubic structures using quadrocopters. Augugliaro *et al.* [6] explored another approach of construction: quadcopters assembled a rope bridge capable of supporting people.

Giant 3D printers are another option. Keating *et al* [54] built a large mobile 3D printer using a robot arm to extrude adhesive materials. WinSun Decoration Design Engineering 3D-printed a five-story build in Suzhou, Jiangsu, China [71]. Malaeb *et al.* [67, 66] presented a framework for designing 3D printers extruding concrete for architecture construction. They introduced two 3D concrete machine and nozzle setups. Cesaretti *et al.* [21] proposed a more novel approach of building 3D printers in the Moon using only lunar soil and eventually build habitats for human.

3.2.4. Modular robots

Instead of focusing on the robot control system to carry building elements, some researchers designed new building elements. Terada *et al.* [102, 101] proposed a novel concept of a fully automated construction system. They built a prototype to verify the construction capability, then introduced distributed control methods to improve the performance. Werfel *et al.* [113, 112, 75] used mobile robots and modular blocks together for 3D structure constructions.

Rus and Vona [88] developed Crystalline, a modular robot with a 3-DoF actuation mechanism allowing it to make and break connections with other identical units; a set of such robots form a self-reconfigurable robot system. White *et al.* [114] introduced two three-dimensional stochastic modular robot systems that are self reconfigurable and self assemble-able by successive bonding and release of free-floating units.

Romanishin *et al.* [84] proposed a momentum-driven modular robot. SamBot, a cube shaped modular robots with rotation mechanism was introduced by Wei *et al.* [111]. Daudelin

et al. [30] present a self reconfigurable system that integrates perception, mission planning, and modular robot hardware. Tosun *et al.* [104] created a design framework for rapid creation and verification of modular robots.

Swarm robot can also be used to assemble structures in the micro scale where controlling each individual nano robot is difficult. One approach is to use global control signals such as gravity and magnetic field. Becker *et al.* [12, 14, 16] took this path and showed the feasibility of re-configuring massive particle swarm robots with limited controls. The authors own work [124] shows some reconfiguration tasks can be performed efficiently in a small space. Manzoor and Schmidt *et al.* [68, 89] proposed parallel mechanisms to make the assembly even more efficient.

Inspired by LEGO, Schweikardt *et al.* [90] proposed a robotic construction kit, roBlocks, with programmable cubic blocks for educational purpose. Kilobot, a swarm of 1000 crawling mobile robots, was introduced by Rubenstein *et al.* [87], along with algorithms for planning mechanisms allowing kilobots to form 2D shapes.

3.2.5. Robot grasping

As an important part of robotic assembly, robot grasping has a long history of research. Recent interests from industry, such as warehouse robotics, and the development of computer vision and deep learning advanced the research considerably. Robot grasping methods are typically classified into two categories: *analytical methods* and *empirical methods*.

Analytical methods assume the object shape is known. Contact locations could be computed or predefined, then selected by searching contact points that resist external wrenches (force closure) [50, 72, 45] or immobilize the object (form closure) [76]. A grasps form by form closure is also called a power grasp [73] or a enveloping grasp [106]. Form closure occurs when the contact fingers form configuration space obstacles so the contact configuration is isolated from other freespace instantaneously. One interesting form closure is called *second-order immobility*. Rimon and Burdick [82, 83] showed an object with curvature boundaries may be grasped by two fingers.

Empirical methods typically utilize pre-labeled grasping data of different 3D objects, then apply machine learning models to map a sensor data (usually RGB-D images or simply point clouds) to grasping locations. A well known data set is provided by Goldfeder et al. [43, 44] called the Columbia grasp database. The data set contains over 1800 models and 200,000 force closure grasps generated using a sampling-based grasp planner GraspIt!. Some work ([31, 49]) shows the method can also find grasping location for unknown objects if their shapes are similar to some objects labeled in the training set. To help train large data set and improve the accuracy of machine learning models, which usually requires a lot of time, Kehoe *et al.* [55] took advantage Google Cloud's storage and parallel computing resources. They first trained an object recognition server for a set of objects, link a database of CAD models with labeled grasping sets, then optimize the grasp for different poses. Mahler et al. [64, 62, 63] extended this idea and developed the Dexterity Network (Dex-Net), a data set of 3D models as a sampling-based planning utilizing cloud computing for finding robust grasping. Their framework works for two-finger grippers as well as vacuum suction grippers.

Section 3.3

Interlocking structures and constraint graph

In Section 2.3.1 and 2.3.2 of Chapter 1, we discussed how the geometry of parts determines how they are assembled or disassembled. However, purely relying on the geometry analysis is computationally expensive, since many geometric patterns are actually representing the same motion constraint. For example, the joint design in Figure 3.4a allows motions along the normal direction of the contact surface. Selecting 10 or 100 vertices from the joint and running the analysis proposed in the previous chapter will not make the motion analysis more accurate, as they eventually yield the same motion constraint, while densely sample points requires more computation resources to be applied to solve the linear programming problem. In this chapter, we discuss how to simplify the analysis given the parts being

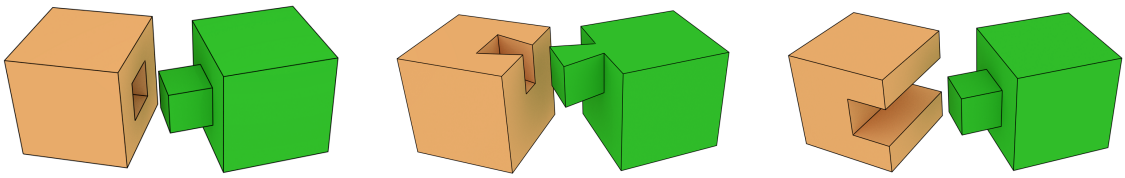
connected by joints.

3.3.1. The joints

A *joint pair* is a geometric design pattern that allows two connected parts to move only following a single motion. A joint pair has a *male* and a *female* part on each of two shapes connected. A block is a rigid body that has male, female or both joints allowing assembly with other blocks. In this chapter, we typically present block joints that allow a simple translation to assemble, for example by sliding one block against another. Figure 3.5 demonstrates three joints that are used as the only joints to make interlocking blocks. A *mortise and tenon* joint pair (Figure 3.5a) allows blocks to be disassembled only in the non-penetrating normal direction of the contact surface. A *dovetail* joint pair (Figure 3.5b) allows block motion only in a particular tangential direction of the contact surface. The third joint pair we use is a *two-way* joint (Figure 3.5c) that allows motions of associated blocks in both normal and tangential directions. Both the mortise and tenon joint and the dovetail joint fully constrain rotational motions, while the two-way joint permits one rotational degree of freedom. We will later see how this rotation motion is actually prevented when a block with two-way joint is connected with two other blocks.

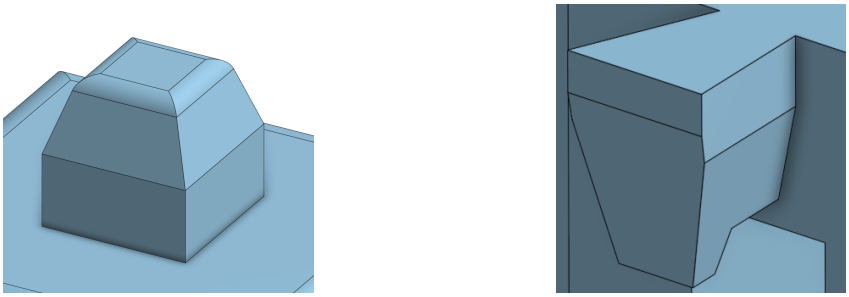
Joint manufacturing details are shown in Figure 3.6. Only the male joint design are shown in the figures, female joint geometry is a result of a Boolean difference operation between a solid part and the male joint. The male joints are made so the front part in the insertion direction is thinner than the bottom part, thus reduce early surface-surface contact to prevent friction between parts in the early stage of joint insertion. This design provides better error tolerance for assembly.

Blocks with joints can be assembled into a relatively rigid structure following a specific assembly order. Figure 3.7 is a 2D projection of a 3D structure assembled using four blocks. In the assembly process shown in Figure 3.7, we first connect block 2 to block 1, using a tenon joint on the top of the block and moving block 2 in the y-positive direction, assuming a coordinate frame aligned with the page. Block 3 then slides in and connects with block



(a) Mortise and tenon joints (b) Dovetail joints (c) Two-way joints

Figure 3.5: Three different joint pairs and detailed design.



(a) Tenon joint design. (b) Dovetail joint design.

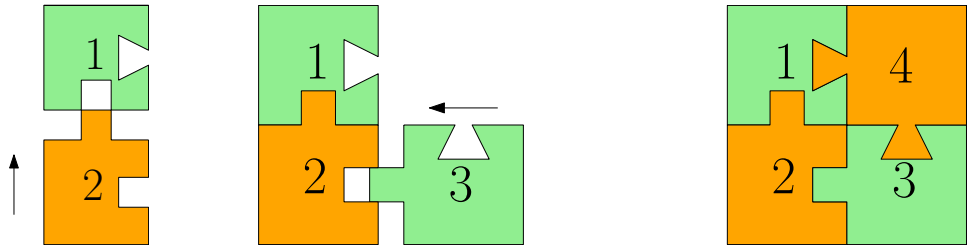
Figure 3.6: Joint design details.

2 using another tenon joint. The last block is assembled from the z positive direction (top down) connecting block 1 and 3 with two dovetail joints, which limits block 2 and 3 to move in y negative or x positive directions.

The last block assembled can be removed by applying a reverse motion of the most recent translation assembly. However, if it is attached to its neighbors using glue, friction, screws or any other external methods, the structure cannot be disassembled. We call such a block a *key*.

3.3.2. The constraint graph

To better understand the motion constraints introduced by joints, we represent an assembled structure using a directed graph $G = (V, E)$ with weighted edges. V is the set of vertices representing the blocks of the structure. A pair of edge $e_{i,j}$ is added between two vertices i and j if the corresponding blocks are in contact. The weight of an edge $w_{i,j}$ denotes the set of permitted motions of j relative to i. Figure 3.8 is an example of a simple interlocking square and its constraint graph.



(a) Tenon joint. (b) Tenon joint. (c) Dovetail joint assembled from top down.

Figure 3.7: Assembling an interlocking 4-block square. Arrow indicates the assembly direction. Block 4 is the key.

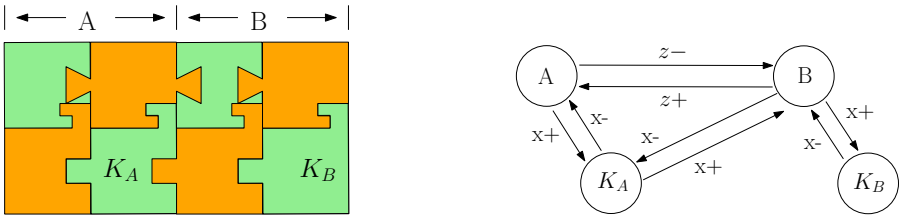
(a) A 2×2 interlocking structure. Numbers indicate the order. Block 4 is the key.

(b) Graph representation. Each edge allows some motions for associated blocks.

Figure 3.8: A four-block interlocking structure and its graph representation.

How can we tell which blocks are movable? Consider a *partition* of the graph into some non-overlapping subsets of vertices. A partition is separable if there exists a motion not violating constraints by all in-edges along the boundary of the subset of vertices. For example, in Figure 3.8b, consider partitioning the graph into two parts $\{1,2\}$ and $\{3,4\}$. The in-edges for piece $\{3,4\}$ are $e_{2,3}$ and $e_{1,4}$. This partition is inseparable, since $w_{2,3} \cap w_{1,4} = \{x+\} \cap \{z+\} = \emptyset$. By checking more partitions of the structure, we found $\{1,2,3\}$ and $\{4\}$ are the only separable subsets. If block 4 is attached (glued) to either of its neighbors, the structure becomes rigid and inseparable. Here we only considered the partitions of a graph into two subsets of vertices, we will later discuss how to analyze partitions of more subsets.

A structured is called *k-interlocked*, if when k keys are attached to its neighbors, no partition is separable. In the example above, block 4 is the key piece, and the structure is 1-interlocked.



(a) Two interlocking substructures connected by a dovetail joint and a mortise & tenon joint.

(b) Forming a larger interlocked structure from two interlocked structures.

Figure 3.9: Any interlocking substructure can be viewed as two nodes in the graph, which simplifies the graph representation.

When a structure grows in size, analyzing the constraint graph gets more complicated as the number of possible partitions expands exponentially. Fortunately, showing a structure is interlocked can be accomplished in a hierarchical fashion: we first show that smaller components are interlocked, and then treat each interlocking component as a whole rigid piece, use those components to build larger interlocking structures. This is particularly useful when some subsets of blocks also form sub-interlocking structures.

Figure 3.9 shows an example of how a larger interlocking structure is built from two smaller interlocking substructures. In the figure, substructure A and B are very similar to the structure shown in Figure 3.8a which is interlocked. Some extra geometric structures are added between the two substructures. One is a dovetail joint preventing motions along x- and y-axis, another is a mortise and tenon joint preventing motions in y- and z-axis. Together, these two joints permit no separation motions of the two substructures. The structure is still disassemblable as we can remove bottom-right, top-right, bottom-left, and top-left blocks of part B respectively then the same with part A.

We now simplify the constraint graph from 8 vertices to 4 vertices by uniting all vertices of an interlocking substructure, except for the key(s), as one single vertex. The keys of the substructures are K_A and K_B, and are not considered as part of A or B. To show that the entire structure is 1-interlocked by K_B, we only need to considers all possible partitions that separate K_A from A or K_B from B since all other partitions are inseparable for the interlocked substructures. Figure 3.9b is the corresponding simplified constraint graph.

3.3.3. Two-hand assembly

One advantage of representing the constraints using the constraint graph is the convenience for finding two-piece separation. Many structures are practically assembled one piece or one component at a time. In each step of the process, the intermediate substructure and the piece / component to be assembled can both be treated as one rigid body. Assuming a piece will not move once connected to its neighbor(s), these structures require at least two robot hands to get assembled or disassembled, one to move the next piece, another to hold the current substructure.

A structure is two-piece separable if it can be separated into only two pieces. Given a structure, how can we tell if the structure can be separated into two pieces? Consider a *cut* of the constraint graph that artitions the vertices into two non-overlapping subsets. If the two subsets of piece can be separated, there exists a motion for one subset, assuming the other is static, that does not violate any constraints on the cut. Based on this observation, one way to iterate over all possible cuts and check if the constraints on a cut are the same. However, we will have to check at the most 2^n cuts, where n is the number of vertices.

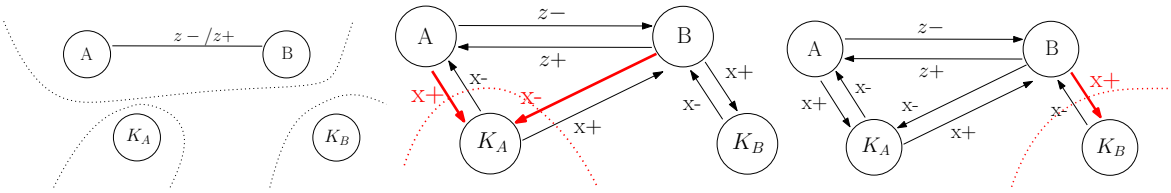
On the other hand, we have at most $|E|$ pairs of motion constraints, one for each pair of adjacent blocks. A simpler method is to check if a pair of motion constraint and the corresponding edges form a cut of the graph.

Algorithm 13 provides the pseudocode for this check. The idea is to first build an indirected graph G' by removing a constraint pair and corresponding edges. If the motion can separate the structure into two or more parts, G' has multiple connected components. We later check if in-edges of each component in the original directed graph G has the same motion constraint. If not, the component is still not separable using the constrained motion.

One example is shown in Figure 3.10 which analyzes the structure in Figure 3.9. When we consider the constraint pair $x+$ and $x-$, we can build a new indirected graph as in Figure 3.10a. The new graph has three components $\{A,B\}$, $\{K_A\}$ and $\{K_B\}$. By checking component $\{K_A\}$, we found its in-edges in the original directed graph (Figure 3.10b) have

Algorithm 13 Determine if a structure is two-piece separable.

```
1:  procedure TwoPieceSeparable(G = (V, E))
2:      for e_{i,j}, e_{j,i} ∈ E do
3:          E' ← {e' ∈ E | w(e) = w_{i,j} or w(e) = w_{j,i}}
4:          Build indirected graph G' = (V, E \ E').
5:          if G' is has one connected component then
6:              return False
7:          end if
8:          for Component C of G' do
9:              V_C ← vertices of C
10:             if In-edges to V_C in graph G have different motion constraints then
11:                 return False
12:             end if
13:         end for
14:         return True
15:     end for
16: end procedure
```



(a) Removing edges of constraint $x+$ and $x-$ and build an indirected new graph. The new graph has three components.

(b) The in-edges of component $\{K_A\}$ do not have the same motion constraint in the original graph, so it cannot be separated.

(c) The in-edges of component $\{K_B\}$ has only one motion constraint. It is separable by moving along $x+$ direction.

Figure 3.10: Steps for analyzing if the structure in Figure 3.9 is two-piece separable.

different constraints suggesting it cannot move along $x+$ or $x-$ direction. We will find the same thing for the component $\{A,B\}$. But the component $\{K_B\}$ passed the test as its in-edge indicates it can move along $x+$ direction freely. The structure is thus two-piece separable.

3.3.4. Multi-piece separation

Although the constraint graph is convenient for checking two-piece separation, Algorithm 13 does not detect the case where a structure can only be separated into more than two components. Figure 3.11 shows a structure that can only be disassembled when multiple com-

ponents are moving at the same time.

Consider assigning velocities $\dot{q}_1, \dot{q}_2$, and $\dot{q}_3$ to the orange, green and blue pieces in Figure 3.11a respectively. In order for the orange piece to move without violating the constraint of $w_{3,1}$, the velocities must satisfy $\dot{q}_1 - \dot{q}_3 \in w_{3,1}$. The velocity of the orange piece relative to the blue piece cannot violate the joint motion constraint between these two blocks. Similarly, we can write down these constraints for every piece and its velocity relative to its neighbors.

How can we test how many robot arms are required in the worst case to disassemble a structure? Or how many components at the most can move at the same time? We found this problem can be solved by considering a linear program.

Generally, given a structure of n pieces, solving the following linear constraint systems will tell which pieces of the structure are disassemblable and their corresponding velocities to move.

$$\dot{q}_i - \dot{q}_j \in w_{j,i}, \forall i, j \in \{1, 2, \ldots, n\} \tag{3.1}$$

To separate components, we also require pieces to not all move together.

$$\sum_{\forall i,j} (\dot{q}_i - \dot{q}_j) \neq \mathbf{0} \tag{3.2}$$

We would like to constrain the speed two pieces are separating, too, since we only care about if components are separable (relative speed equals $\mathbf{0}$ or not) instead of how fast they separate from each other.

$$-\mathbf{1} \leq \dot{q}_i - \dot{q}_j \leq \mathbf{1}, \forall i, j \in \{1, 2, \ldots, n\} \tag{3.3}$$

So how can we disassemble as many components as possible at the same time? To find the answer, we may optimize the velocities of each piece to achieve the largest difference of their relative velocities.

$$max \sum |\dot{q}_i - \dot{q}_j|, \forall i, j \in \{1, 2, \ldots, n\} \tag{3.4}$$

If a structure has no separable piece, constraints 3.1 and 3.2 can not be satisfied together. Thus, if the linear constraints can not be satisfied, we know the structure is not separable.

Although the above linear constraints will tell us if a structure is separable and the max number of movable components, it does not tell us the minimum number of separable components. For example, one can design a more complicated puzzle by cutting each piece in Figure 3.11a into three smaller pieces connected by the same 3 joints. The minimum number of separable components is still 3, same as the original puzzle. However, the maximum number of separable components increased.

In order to find the minimum number of separable components, we first test if the structure is separable. If so, we change constraint 3.2 to

$$\sum |\dot{q}_i - \dot{q}_j| > \mathbf{1}, \forall i, j \in \{1, 2, \ldots, n\} \tag{3.5}$$

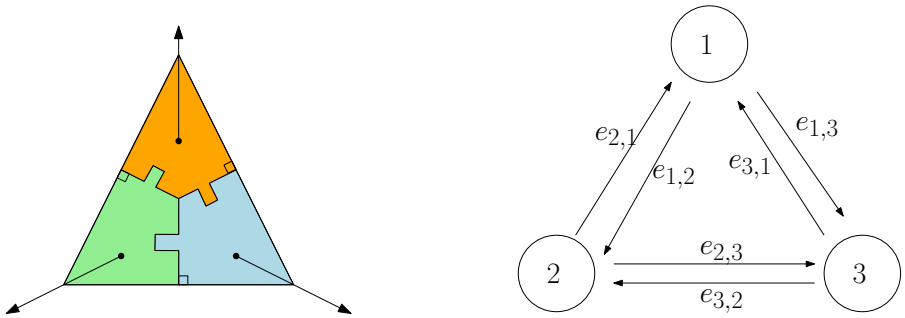
enforcing the components to separate. Objective function will be modified, too, so that we minimize the velocity difference between pieces, making them more willing to stick together.

$$min \sum |\dot{q}_i - \dot{q}_j|, \forall i, j \in \{1, 2, \ldots, n\} \tag{3.6}$$

Section 3.4

Designing interlocking blocks

The previous section introduced the concept of *interlocking structure* and how to check if a structure is interlocked. In this section, we will consider a design problem: given the shape of a desired structure, how can we partition it into pieces with joints such that the pieces can be assembled into one interlocking structure?



(a) A puzzle that can only be disassembled when multiple pieces move at the same time.

(b) Constraint graph of the 3-piece puzzle.

Figure 3.11: A structure that cannot be separated into only two components and its constraint graph. Multiple pieces have to move at the same time to get disassembled.

Our question is vague since the possible ways of cutting a structure is infinitely many. Let us simplify the problem by considering only voxel models. How can we assign joints (mortise and tenon joint, dovetail joint or neither) between every two adjacent voxels, such that the assembled structure has only one movable piece?

3.4.1. Computational design

Figure 3.13 is an example puzzle design problem of 16 blocks. It is an overhead view of a layer of unit blocks in 3D. Our task is to assign joints to each pair of blocks, so the structure is (1) disassemblable and (2) 1-interlocked.

The joint pairs we use are mortise and tenon joint and dovetail joint. A pair of blocks do not always have to be assigned a joint, there can also be no motion constraints between them. Thus for each pair we have 5 choices: 2 ways of assigning mortise and tenon joint, two ways of assigning dovetail joint, plus one option of assigning no joint. In total, there are 24 adjacent pairs in Figure 3.13, the number of possible assignments is 5^{24}.

So given a joint pairs assignment for this structure, how can we tell if it is an interlocking structure? We do the following tests:

(a) Iterate over each on the boundary, there should be exactly one block removable without moving other adjacent blocks.

(b) Disassembling movable blocks one-by-one, there should be no blocks left un-removed.

(c) If glue the first movable piece with its neighbors, the structure should not be disassemblable.

A naive way for computationally design joinery blocks is to list all possible joint pair assignments, perform the three tests for each assignment until we find one successfully passed all three tests. As mentioned above, this process can take very long time.

One important observation could help to speed up the search. Let joint assignment $a = [j_0, j_1, \ldots, j_{24}]$ be a joint assignment, and $j_i \in \{0, 1, 2, 3, 4\}$ represents 5 kinds of joint pairs for a pair of adjacent blocks. If a is not disassemblable, changing any unconstrained pairs j_i in the assignment will not make the structure disassemblable. In other words, adding joint motion constraints is only going to make a structure less likely to be disassemblable.

Based on this observation, we model the design problem as a tree search. The i-th layer of the tree represents the joint assignment for the i-th pair of blocks. In the root node, we do not assign any motion constraints. Each node has 5 children, since each pair of blocks has 5 possible joint assignments. If a node is not disassemblable, its sub-trees are pruned since none of them will have disassemblable nodes. In theory, the tree still have at the most 24 layers, and the complexity is still $O(5^{24})$, but since we pruned a lot of branches, the tree search is much faster than the naive algorithm. This method is shown in Algorithm 14 which searches the joint assignment recursively.

An interlocking blocks design is shown in Figure 3.12. Our algorithm was run to design blocks that can be assembled into a letter *D*. 34 blocks were generated. An animation of the disassembly process can be found at [121]. The animation was rendered using Blender 2.76. We 3D printed the designed blocks and assembled them by hand. See Figure 3.12b.

3.4.2. Our block design

Section 3.3.2 suggested another approach for building interlocking structures: first build smaller interlocking structures, then connect them to form a large interlocking structure.

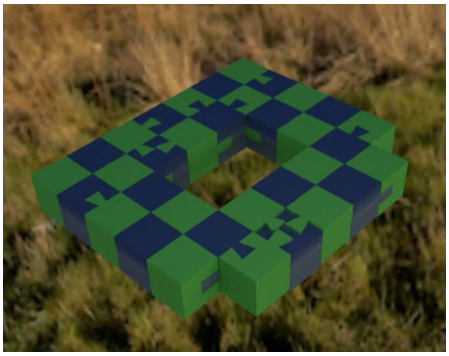
Algorithm 14 Algorithm overview

```
1: function AssignJointPairs(a = [Null, Null, ..., Null], depth = 0)
2:     if not isDisassemblable(a) or depth ≥ max_depth then
3:         return Null
4:     end if
5:     if lockedByOneKey(a) then
6:         return a
7:     else
8:         for joint ∈ {all possible joint pairs plus no constraint} do
9:             if AssignJointPairs(a[depth] = joint, depth + 1)! = Null then
10:                return a
11:            end if
12:        end for
13:    end if
14:    return Null
15: end function
```

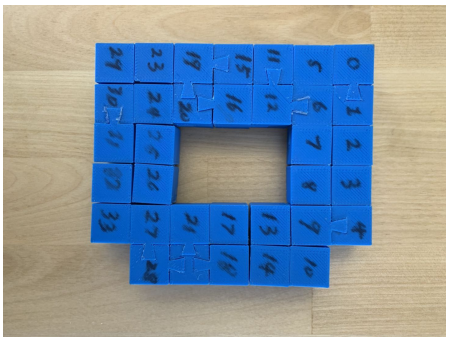
This idea simplifies our block design search: instead of assigning every block pair, we only assign in the construction order and make sure substructures are interlocked. Since the substructures are smaller, its search space is much smaller, and once an interlocking substructure is found, we do not search for new joint pair assignments unless a higher level substructure's interlocking property cannot be satisfied latter.

One critical observation on block design is that joint types for a pair of blocks actually determines the assembly order of blocks. On the opposite, if the layout order is known, the joint assignment to a block is also limited. Since there are 2 joint types, each has male or female, and six faces are on a cube, this suggests there might be at most $5^6 = 15,625$ designs. If we also add two-way joints into our candidates, the number becomes $7^6 = 117,649$. Fortunately, by patterning the assembly order, not all of the possible designs appear. Further tricks allow reducing the number of block type by even more. As an example, consider the block in Figure 3.9a, adding a mortise joint on the right side of block K_B makes it identical to K_A, reducing the number of block types by 1.

In our approach, the smallest interlocking structure is a square of 2×2 unit blocks. Squares are connected to form an interlocking segment. And segments are connected to



(a) Interlocking blocks designed to be assembled into a letter *D*. Rendered using Blender 2.76.



(b) 3D printed blocks for the interlocking blocks design. Numbers indicates assembly order.

Figure 3.12: Interlocking blocks designed by the computational method (Algorithm 14). Disassembly animation can be viewed in [121].

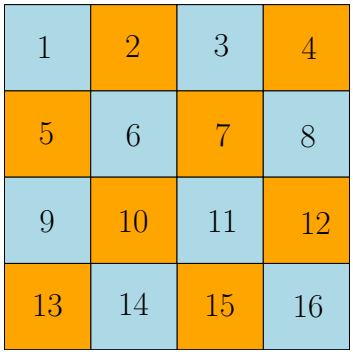
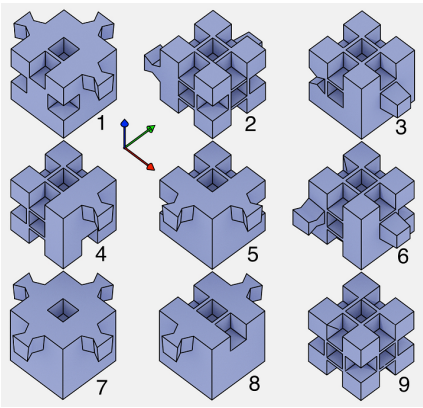


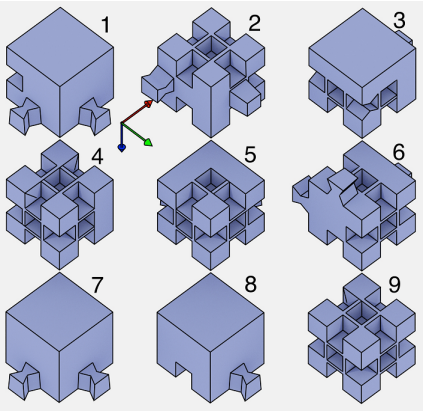
Figure 3.13: A simple puzzle design problem with 16 blocks and two kinds of joints.

form a layer. Layer-by-layer, we can build an interlocking structure. Following this path, we first computationally designed some blocks capable of assembling a cube of $4 \times 4 \times 4$ unit blocks. We simplify the design by adding unused female joints to some blocks and reduced the number of blocks to 9. Figure 3.14 shows the design of 9 blocks. These blocks can be used to assemble any interlocking structures.

We latter improved our design and reduced the number of blocks to 2. Figure 3.15 shows the new design. Similarly, these blocks are also general purpose. The details of the layout algorithm is explained in Section 3.5.



(a) Front view of 9-block design.



(b) Back view of 9-block design.

Figure 3.14: A design of 9 kinds of blocks that can be assembled into any voxelized structure in an interlocking manner.

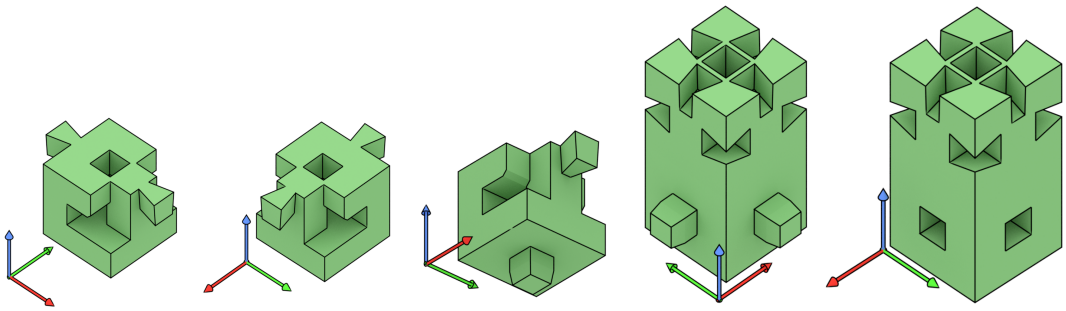


Figure 3.15: Different views of cube and post blocks.

Section 3.5

Layout algorithm

With the blocks designed in Section 3.4.2, how can we assemble them into any structure in an interlocking manner? The example in Figure 3.8a is a square of 2×2 blocks, Figure 3.9a links two squares together to form a longer interlocking segment. Inductively, we can extend the segment as long as we want by connecting more interlocking squares to the right end of the segment.

Intuitively, some additional connections might be added to connect two segments to form a flat structure, we call a *layer*. Larger 3D structures are assembled by connecting interlocking layers. Figure 3.16 is a visualization of the conceptual process. Keys are

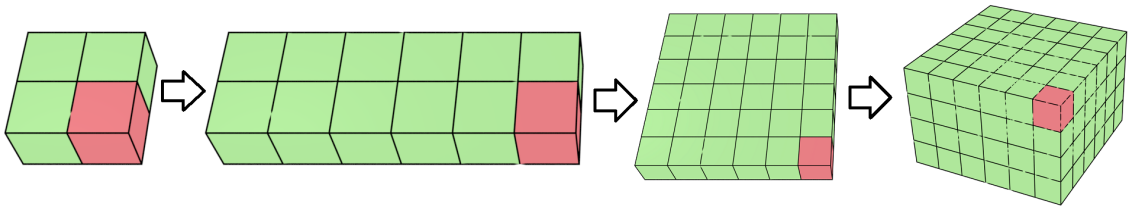


Figure 3.16: Building an interlocking 3D structure. Blocks interlock to form a square; squares form an interlocking segment; segments form a layer; layers interlock to form the structure. The keys are marked red.

marked red.

In the following subsections, we will address the details on how to implement the assembly planner for our two block design. Mainly, we will look at how segments interconnect to form a layer, and how layers interconnect to build a higher and higher structure. We will also see how something more interesting than a large cubical volume can be built, how some blocks will be arranged to deal with special situations to ensure interlocking property.

Algorithm 15 presents an overview of the layout algorithm. Details of each line will be discussed in the next several subsection, as indicated by the section numbering in the algorithm. It is worth noting that our input to the algorithm is a *voxelized* model imitating the desired shape of structures to be built. Each voxel is further subdivided into eight $(2 \times 2 \times 2)$ subvoxels of the same cubical size. Each subvoxel will effectively be instantiated by a block.

The block types are assigned depending on the layer and segment. The assignment is selected to allow the adjacent substructures to be connected and inseparable once assembled in a specific order. The output is a sequence of block assembly order that constructs an interlocking structure shaped as the desired model. Since our model is built layer-by-layer, the final structure could have more than one keys. The number of keys is the number of layers that do not have any other adjacent layer on their top.

It is also worth pointing out the approach we have taken to connect adjacent layer. To provide a firm connection, we use a block of height two as a connector between layers; we call this block a *post*.

Algorithm 15 Algorithm overview

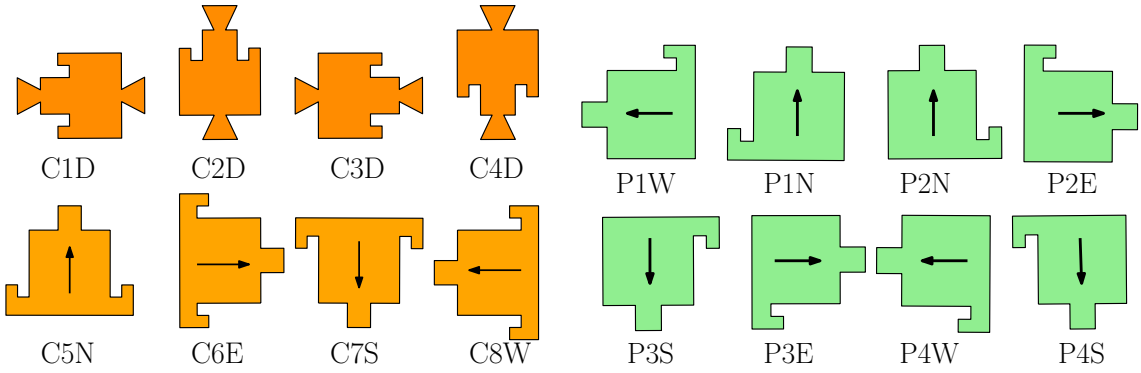
```
1: function ConstructVoxelModel(M)
2:     M′ ← split every voxel into eight dimension-1 cubes.
3:     for each layer L_i of M′ from bottom to top do
4:         Lay out any missing posts.
5:         if L_i is an even layer then
6:             Set all segment types to X_l−Y_+. (Section 3.5.2)
7:         else
8:             Determine the key to each layer component. (Section 3.5.4)
9:             Order segments in each layer component. (Section 3.5.4)
10:            Determine the key(s) to each segment. (Section 3.5.4)
11:            Determine the type of each segment. (Section 3.5.4)
12:            Find special cases. (Section 3.5.5)
13:            Modify L_i and L_i+1 if necessary. (Section 3.5.5)
14:        end if
15:        Assemble blocks, potentially in parallel. (Section 3.5.6)
16:    end for
17: end function
```

3.5.1. Blocks and squares

Our layout algorithm in 15 uses two types of blocks: a *cube* is a unit-cube sized block for filling empty space in a layer and locking with existing blocks, and a *post* is a two-unit high block. See Figure 3.15. The bottom half of a post block connects to cubes in the same layer, while the upper half connects to cubes in the upper layer, making two layers connected and eventually inseparable. The post blocks also act as key blocks of substructures.

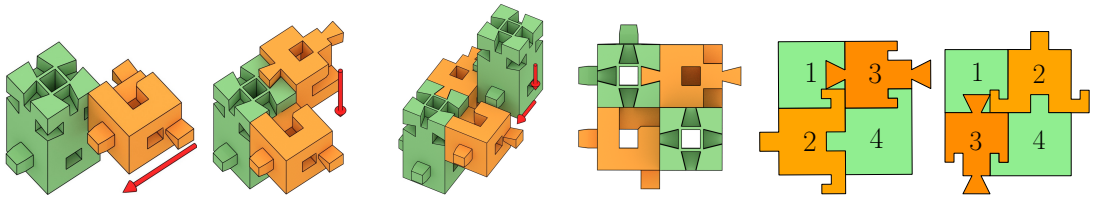
A *cube* block has two dovetail male joints on two opposite sides that can connect with female joints in post blocks allowing motions only in the assembly direction tangential to the contact surface. Figures 3.18a and 3.18b show how side male joints of a cube connect with female joints of a post in two different direction. A cube also has a male joint on the bottom that connects, in the normal direction, either with the top of a cube or post. The female joints on the opposite sides of the cube block allow post blocks' male joint to drop and slide to connect, which allows the post block to disassemble only in two directions.

We carefully analyzed segments and layers to determine how joints might be arranged on cubes and post. To describe a layout and an assembly process, some notation is helpful.



(a) Cube orientations and assembly directions. (b) Post orientations and assembly directions.

Figure 3.17: Different ways to assemble cube and post blocks and corresponding notations.



(a) Slide in a C8W block. (b) Drop in a C3D block. (c) Assemble a P1W block. (d) Top view of the square. (e) S_a square. (f) S_b square.

Figure 3.18: Assembly process of a square, and two designs for a square.

For each block, we use a triplet of characters indicating block type, orientation of the block, and assembly direction. Figure 3.17 shows all of the triplets used in assembly of structures in this paper. For example, *C1D* means "**C**ube in orientation 1, assembled by moving **d**own". Figure 3.17b shows all of the notation triplets used in the current approach. Not all axis-aligned orientations of cubes and posts are needed to construct structures; for example, posts only occur in the four orientations generated by rotating the post in Figure 3.15 around the z axis in ninety-degree increments.

Block designs are crafted to allow assembly of squares, segment, and layers. A *square* is the smallest interlocking structure we consider, composed of four blocks: two posts and two cubes. By using posts and cubes in different orientations, different squares may be constructed, as shown in Figure 3.18e and 3.18f. These two kinds of squares are denoted as S_a and S_b. Different squares will be used in Section 3.5.4 to constrain key block motions of other adjacent segments in the same layer, making the layer interlocked.

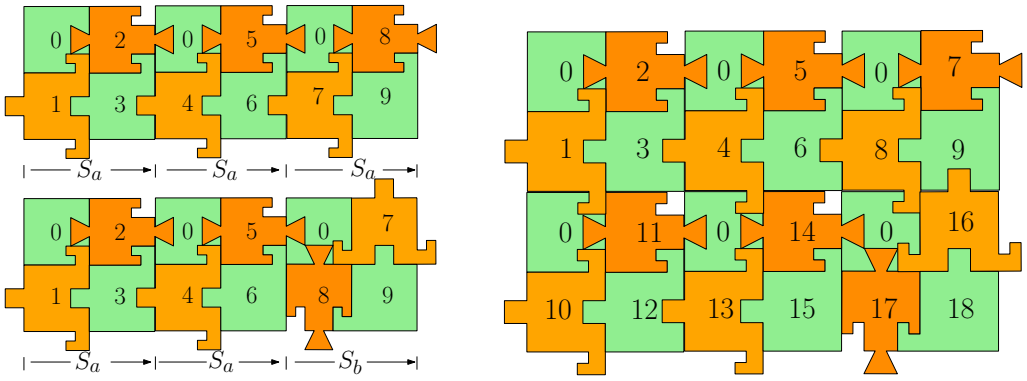
Figure 3.18 shows the process of assembling one kind of square. The first piece, a post, is usually from a layer below the current one. Two cubes are connected to the top half of the post. The second post acts as a key, and the top half of this post extends above the square to provide connection to a square that may be later built above the current one.

3.5.2. Segments

We now introduce a method to link multiple squares into a long sequence of interlocking structure called a *segment*. A segment is composed of n squares in a $1 \times n$ pattern. The size of a segment is thus $2 \times 2n$. To build a segment, we assume n posts are already pre-placed in the prior layer, so the top half of each post appears in the same x or y-coordinate position. Building squares using these posts connects the squares to the prior layer.

Segments can be built in many orientations, left to right, up to down, or the opposite. We here only discuss how to assemble a segment from left to right in the direction of x axis, assuming posts are in the upper (or $y+$) half of each line. Other segments are symmetric and will not be detailed. Their construction can simply be viewed as a rotation of a left-to-right segment. We denote a segment as $Y_{l+}X_{+}$ if the posts are in the left position of the y positive half and the segment is built from x negative towards x positive direction withe the key block at the right end of the last square.

Figure 3.19a visualizes the process of assembling a $Y_{l+}X_{+}$ segment of 3 squares. We connect, from left to right, $n-1$ S_a squares. The final square of a sub-segment can be of type S_a or S_b. The key piece of the segment is the last assembled post block. A sub-segment with a S_b final square is not interlocking, but when connected with previous segment(s), the S_b square prevents the adjacent block in the y positive direction from moving and interlocks the structure. Building another $Y_{l+}X_{+}$ segment on the y negative side will create an interlocking layer (Figure 3.19b). In Section 3.5.4, we will discuss how to constrain the motion of the key in different types of segments.



(a) Two $Y_{l+}X_{+}$ segments. (b) A layer built by two $Y_{l+}X_{+}$ segments.

Figure 3.19: A simple segment and an example layer built by connecting two segments.

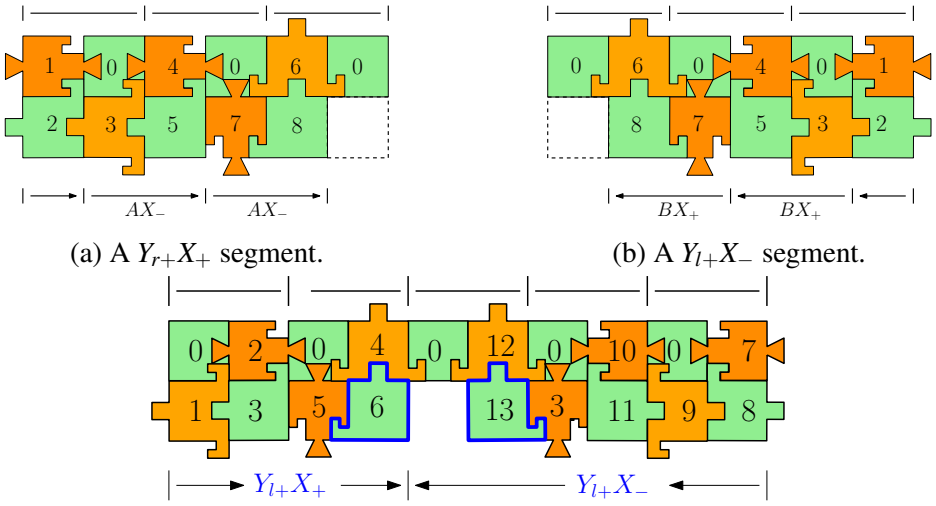
3.5.3. Structure mirrors

Following the process to build a $Y_{l+}X_{+}$ segment as in Figure 3.19a, one can lay out an array of segments one-by-one and create interlocking planar structures as in Figure 3.19b. However, this structure requires the key to every segment to be in the x positive end and constrained by the next adjacent segment. In practice, we want to build some planar structures in different orientations with different key positions. To built more complicated structure, we first introduce the concept of *mirrors*.

Definition 3.1 (*x-mirror*)**.** Object A is an x-mirror, $m_x(B)$, of another object B if one is a reflection of the other with reflection plane perpendicular to the x-axis.

Analogously, we have a y-mirror operation, too. Cube and post designs are symmetric in such a way that x and y-mirror operations can be accomplished by simply rotating the block into another orientation. Construction of a mirrored structure follows the same order of the original structure with opposite directions along the same axis; for example, we may build a $Y_{r+}X_{-}$ segment by x-mirroring a $Y_{l+}X_{+}$ segment.

We are going to need two more types of segments to do arbitrary layer construction: $Y_{r+}X_{+}$ (Figure 3.20a) and its x-mirror $Y_{l+}X_{-}$ (Figure 3.20b). To build a $Y_{r+}X_{+}$ segment with $n \geq 2$ squares, , we first assemble two blocks ($C3D$ and $P1W$) in the left two positions. Then assemble a $Y_{l+}X_{+}$ segment of $n-1$ squares. When all pre-existing posts are prevented



(a) A $Y_{r+}X_+$ segment.

(b) A $Y_{l+}X_-$ segment.

(c) A pair of segments with keys in the middle.

Figure 3.20: Construction of two x-mirrored segments. Arrows indicate construction direction. Numbers indicate assembly order.

from moving along z axis, the segment is interlocked.

In many input geometries, segments may not always have the nice property of being adjacent to other segments at either end. Instead, segments could only be connected to other segments in the middle, causing the key to be exposed if we use the $Y_{r+}X_+$, $Y_{l+}X_+$ types or their mirrors. To solve this problem, we may replace a single segment by two segments grown from the ends, effectively allowing placement of a pair of keys at an arbitrary position in the middle, as shown in Figure 3.20c. These keys may then be immobilized by later segments.

3.5.4. Layers

Now that we know how to build different kinds of segments, we can connect a set of segments on the same plane to create complicated interlocking 3D structures, by careful assignment of subvoxels from the original model into layers, segments, squares, and blocks.

A *layer* is a set of squares with the same z-coordinate. A set of **connected** squares with the same z-coordinate is a *layer component*. We assume all layer posts are provided by the prior layer. This assumption can immediately become a problem: we cannot build hanging structures without having additional supports underneath.

We will first introduce the ordering of segments' construction in one component. Once ordered, segments are ready to be assigned square types and latter assembled accordingly. In the next subsection, we will discuss some special cases caused by the nature of our block design and square structures, and provide techniques to still ensure interlock.

Layer key(s). As the first step of building any interlocking structure, we determine the key(s) of the layer. A layer is immobilized if the key(s) is fixed with respect to its neighbors. Since every even layer has an upper layer with the exact same shape, based on the division of voxels into subvoxels, post blocks that connect the upper layer will be immobilized as long as the upper layer is interlocked, preventing the horizontal motion of any posts. Therefore, we only consider the odd layers in this section.

For any odd layer component without adjacent upper layer blocks, we select a post block at the x negative end of a boundary segment as the key, where a boundary segment is a segment with adjacent neighbors on only one side. If the odd layer component has an adjacent upper layer, the key can be any post block covered by an upper layer square.

Under this rule, every layer component constrains the key to its lower component. Any layer components that do not have an immediate upper layer introduce a new key that will not be covered. The number of key pieces of the whole structure is thus the number of layer components without an immediate upper layer. This introduces an interesting effect of the orientation of the object to be constructed. For example, the chair in Figure 3.1c has a single key, but if the chair were built upside-down, then there would be four keys: one in each leg.

Segment construction order. Once a layer's key square and all starting posts of squares are known, the second step of assembling a layer is to determine the order and the type of each segment. Segment types indicates the block types and block assembly order.

In the preprocessing step, every voxel is split into two squares, making every layer of voxels two layers in the real assembly. The bottom layer has an even z-coordinate value,

while the up layer has an odd z-coordinate. Every segment in an even layer is constructed along y-axis directions. We simply assemble every segment as $X_{l-}Y_{+}$, or 90° clockwise rotation of a $Y_{r+}X_{-}$ segment, from left to right. An even layer component is not necessarily interlocked, because there can be many segment keys unconstrained and able to move in the x positive direction. However, all square keys are posts in the upper layer, and as long as the upper layer is interlocked, or all posts are prevented from moving in x positive direction, the two-layer structure is interlocked.

Each square in an odd layer component is initially assumed to have a post in the bottom-right position. This, however, could change after the segment types have been assigned. We first build a set of post lists where each list contains posts with the same y-coordinate, and two adjacent posts are 2 units away. Each *list* will be built into a segment. Two posts are considered adjacent if their x or y-coordinates have a difference of 2, as their associated squares will be adjacent. Two lists are considered adjacent if they have adjacent posts. Lists are ordered by their shortest distances to the final list that contains the post of the key square, where the distance between two adjacent lists is 1.

Given a list l and the next-built adjacent list l_n, the type of the segment S_l associated with l is determined as described below:

- If l_n is at $y-$ side of l & the left end post of l is adjacent to l_n, S_l is $Y_{r+}X_{-}$.
- If l_n is at $y-$ side of l & the right end post of l is adjacent to l_n, S_l is $Y_{r+}X_{+}$.
- If l_n is at $y-$ side of l & neither ends of l is adjacent to l_n, S_l is broken into a $Y_{r+}X_{-}$ and a $Y_{r+}X_{+}$ segment.
- If l_n is at y positive side of l & the left end post of l is adjacent to l_n, S_l is $Y_{r-}X_{-}$.
- If l_n is at y positive side of l & the right end post of l is adjacent to l_n, S_l is $Y_{r-}X_{+}$.
- If l_n is at y positive side of l & neither ends of l is adjacent to l_n, S_l is broken into a $Y_{r-}X_{-}$ and a $Y_{r-}X_{+}$ segment.

The segment associated with the last built list has been specified a key (line 8 of Algorithm 15). Its type is thus determined.

3.5.5. Special cases

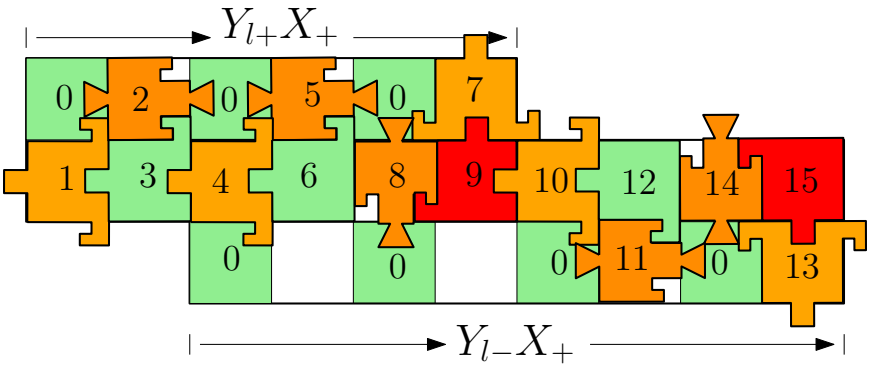
At this point, each segment has been assigned a type. Their construction order in each layer is also determined. Many interlocking layer structures can be assembled by directly following the construction of each segment as specified in Section 3.5.2. However, depending on the successor segments, some small modifications might be applied to insure the interlock of adjacent segments.

Consider a segment with key(s) in the y negative side, for example $Y_{l+}X_{+}$. Its successor can be (1) a segment whose key will be constrained by further segments in y negative side, (2) a segment with key being constrained in y positive side, or (3) a segment whose key will be constrained by the upper layer. We now list all possible cases that need modifications.

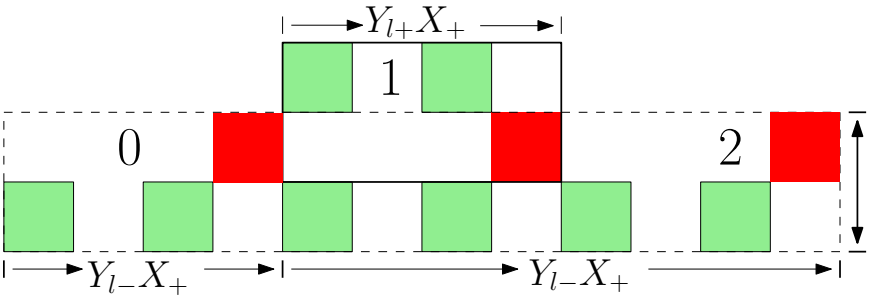
Case (1): A $Y_{l+}X_{+}$ segment followed by another $Y_{l+}X_{+}$ segment. We use a S_b squares at the later built segment to prevent the segment's key from moving. See Figure 3.19b. Otherwise, a $Y_{l+}X_{+}$ segment always uses a S_a end square.

Case (2) contains four subcases where the current segment has one or both ends adjacent to its successor whose key is in the x positive or negative side. Figure 3.21a shows one subcase. The first $Y_{l+}X_{+}$ segment is still assembled as usual. We leave some positions adjacent to the first segment unfilled, and assembled the rest part. The segment is interlocked. Some bottom posts are prevented from moving in y positive, y negative and x negative directions by the lower layer.

Figure3.21b is a similar subcase where both ends of the segment are adjacent to the successor. We divide the lower segment into two segments, one containing no posts adjacent to the upper segment will be built first, the other containing the rest posts will be built after the upper segment. In the other subcases, the successor has a key in the x negative direction, we change the upper segment to $Y_{l+}X-$ and create an x-mirror of the previous case.



(a) A $Y_{l-}X_+$ segment built after a $Y_{l+}X_+$ segment. Some positions are left empty.



(b) A longer lower segment. The lower segment is broken into two segments.

Figure 3.21: Two special cases of building adjacent segments. Green blocks are posts, and red blocks are keys of each segment. Numbers indicates the assembly order.

Case (3) is shown in Figure 3.22 where a $Y_{r+}X$ segment and a $Y_{r-}X$ segment are assembled before the segment in the middle. We require the upper and lower segments' keys to be in different x positions. To ensure interlocking, we firstly finish the upper segment, then assemble two C5N blocks in the middle segment. After the lower segment is assembled, we put in C3D block(s) in the middle to constrain the motion of the lower segment key(s). The last assembled blocks (keys) in the middle will be constrained by its upper layer. If the upper layer is not wide enough to cover the keys, we must expand the upper layer (Line 13 in Algorithm 15).

3.5.6. Parallel construction

Algorithm 15 gives an overview of the construction process. Our construction starts from the bottom layer to the top. For each layer, we first check if all required posts exist. If not, we lay out these posts before starting the assembly (Line 4). Even layers are constructed

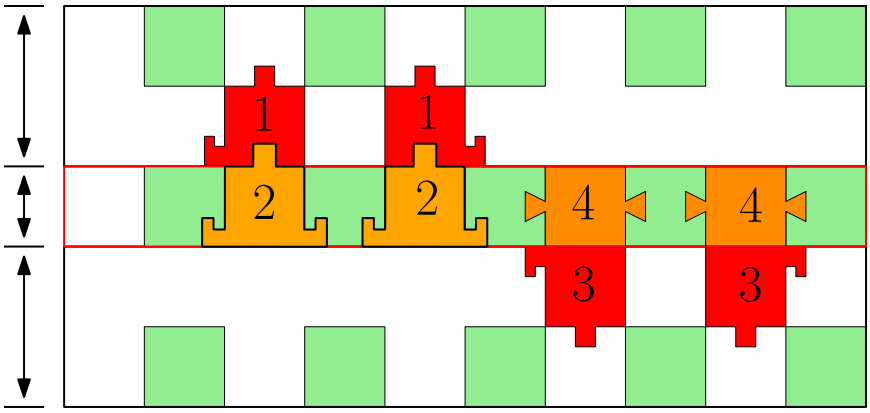


Figure 3.22: Special cases where two segments with posts in different sides are finished before the segment in the middle. Red blocks are keys of two segments ($Y_{r+}X$ and $Y_{r-}X$ types). Numbers indicates the assembly order.

using $X_{l-}Y_{+}$ segments (Line 5, 6). Odd layers need to find the keys first (Line 8). Based on the key to each layer component, we order segments (Line 9) then determine segment keys and segment types (Line 10, 11). Before assembling, we check if any special cases exist as mentioned in Section 3.5.5 (Line 12). Since $Y_{r+}X$ and $Y_{r-}X$ segments require at least two adjacent square, we need to modify the current layer if the condition is not satisfied. The special case as in Figure 3.22 can also require the upper layer to expand and cover lower layer keys (Line 13). We then finally assemble blocks based on block types and special cases.

Laying out blocks one-by-one is time-consuming when a structure has a large number of blocks. This section provides an algorithm that generates a parallel construction order to accelerate the process. We first consider preliminaries blocks of assembling each new block, and build a graph between blocks. By querying the graph for blocks whose preliminaries are satisfied, we can have multiple agents to lay out the blocks.

Consider a block b to be assembled in a layer. Any adjacent block(s) to be assembled later should not be prevented by the male joint(s) of b, meaning the joints of a block connect to only the pre-existing blocks. Along the assembly direction of b, the male joints of b should not be able to touch any blocks. The blocks that must be assembled before a new block to prevent collision are called *predecessors* of the new block. Every block has a

predecessor below it if an adjacent block exists in the lower layer. Consider a block at position (x,y) in any layer. Table 3.1 is a list of predecessors of different types of blocks in the same layer.

Block type	Predecessors	Block type	Predecessors
C1D, C3D	$(x-1,y)$, $(x+1,y)$	C8W	$(x,y+1)$, $(x,y-1)$, $(x-1,y)$
C2D, C4D	$(x,y-1)$, $(x,y+1)$	P1W, P1N	$(x-1,y)$, $(x,y+1)$
C5N	$(x-1,y)$, $(x+1,y)$, $(x,y+1)$	P2N, P2E	$(x,y+1)$, $(x+1,y)$
C6E	$(x,y+1)$, $(x,y-1)$, $(x+1,y)$	P3S, P3E	$(x+1,y)$, $(x,y-1)$
C7S	$(x-1,y)$, $(x+1,y)$, $(x,y-1)$	P4W, P4S	$(x-1,y-1)$, (x,y)

Table 3.1: Predecessors of each type of block.

Besides predecessors listed above, inside each square, cube blocks with mortise joints connecting blocks in the same layer (C5N, C6E, C7S or C8W blocks) must be assembled before others (C1D, C2D, C3D or C4D blocks).

With the predecessors of each block, we then construct a directed graph $G = \{V,E\}$, where V is the set of blocks, and directed edge $e_{i,j} \in E$ indicates block i being a predecessor of block j. The construction follows the order of removing nodes with in-degree of 0. Each construction agent/thread will take a block whose predecessors have been placed, and remove the node from the graph when the block assembly is finished.

A simple observation with the parallel construction is, after the construction of one square s, all the adjacent squares to be assembled after s in the sequential order are ready to assemble. A visualization can be found in Figure 3.23. When the first square, labeled 0 with key pieces marked red, is built, one adjacent square in the same segment can also be assembled, another adjacent square in next segment is also free to assemble without conflict any blocks. Thus two more squares can be built simultaneously. As a result, in the next assembly step, three more squares can be built. The process goes on and on. The whole layer with 6×6 squares are built in only 11 ($\leq 2 \times 6$) steps, instead of 36.

3D structures can be built using the same strategy. We therefore have the following theorem:

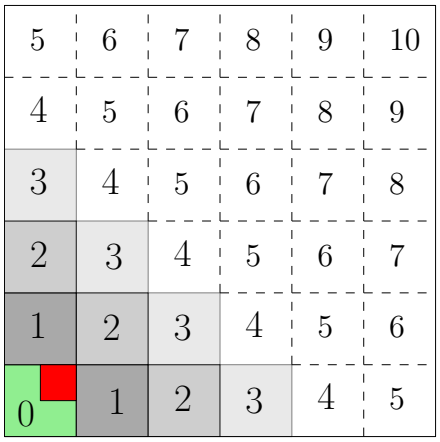


Figure 3.23: Parallel construction visualization. When the first square, labeled 0 with red key block, is built, two more adjacent squares can be built without conflicting other square assembly.

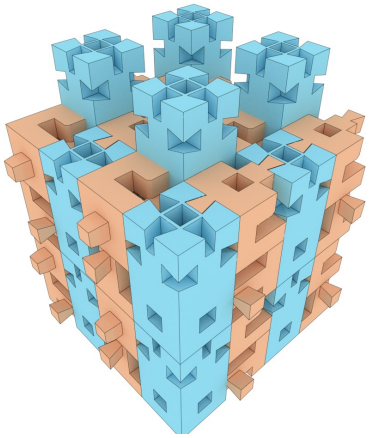
Theorem 3.2. *Parallel construction of a solid cube of N squares takes* $O(\sqrt[3]{N})$ *time.*

Proof. First consider constructing a solid layer of $n \times n$ squares. For simplicity, we scale the width of each square to one. After assembling the square at the corner $(0,0)$, two adjacent squares in x and y positive directions will be assembled at the next time step, then three, four, and so on. See Figure 3.23. It takes k steps to construct $k(k+1)/2$ squares. When $k = n$, over $n^2/2$ squares are constructed. So constructing a layer takes at most $2n$ steps. In a cube, since finishing every square allows all adjacent squares in x, y and z positive directions to assemble. When the last square of the bottom layer is done, it takes one more step to finish the upper layer. So $2n-1$ more steps will finish all upper layers. Therefore a solid cube of $2n \times n \times n$ squares takes $O(n) = O(\sqrt[3]{N})$ time to assemble. □

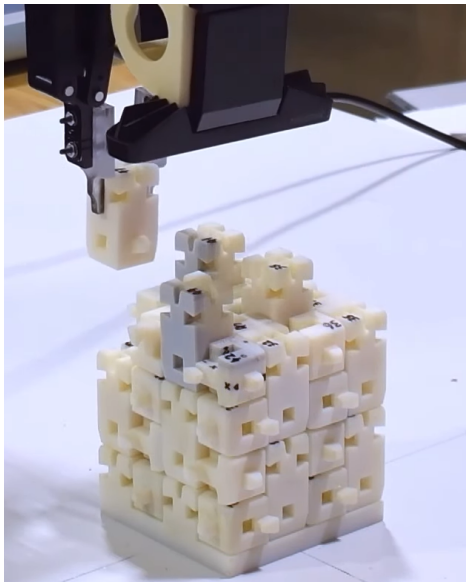
Section 3.6

Robotic assembly experiment

To demonstrate feasibility of the assembly approach, we developed a system to allow two 6-DoF robot arms to perform a simple example of interlocking structure assembly using our blocks. With this system, we assembled a 4-layer cube-like structure of 48 blocks. The structure does not have overhang, and can be constructed without support material.



(a) Rendered structure in simulation.



(b) Assembled in real-world.

Figure 3.24: A 48-block cube-like interlocking structure is assembled using a two-arm robot system. Blue blocks are posts and orange blocks are cubes.

To correctly assemble a block into its target position, the system must solve the following problems: (1) recognize the position and orientation of the construction base; (2) distinguish between cube and post blocks; (3) pick up blocks that initially sit in a specified area; (4) precisely estimate the position and orientation of the held block precisely; (5) re-orient the block if it cannot be picked up in the desired orientation (6) and follow a path to assemble the block. In this section, we will introduce the robot system, how each problem mentioned above is solved and experimental results.

The infrastructure, including network communication, trajectory generation, math library, and neural network framework were built by Dr. Yotto Koga. Yinan Zhang's work is limited to machine learning model training, neural network tuning, re-grasping and assembly experiments.

3.6.1. Experiment setup and assembly process

The environment setup is rendered in Figure 3.25. A photo of the live platform is shown in Figure 3.29. The system includes the following hardware:

(a) **One flat table**. The table was used as the platform for assembly. A construction base is attached to the center of the table. The base is our origin. Blocks are initially placed randomly close to the bottom right corner of the table.

(b) **Two 6-DoF robot arms**. We used two Universal Robots' UR-10 robots, each with six degrees of freedom. The arms cooperate to regrasp the block sequentially to reorient blocks and remove configuration error. Details of re-grasping are presented in Section 3.6.3.

(c) **Two depth cameras**, one for each robot arm. We use the Intel SR-300 structured light depth cameras to obtain a 3D point cloud reconstruction of the environment. The point cloud is represented as a 2D heightmap image. Heightmaps nicely capture the 3D structure of the scene and show less sensitivity to lighting conditions compared to images from RGB cameras.

(d) **Two two-finger grippers**. The arms are equipped with Robotiq 2F-85 and 2F-140 model grippers. We designed and 3D printed finger tips for grasping small areas on the blocks.

(e) **One force-torque sensor**. One Robotiq FT-300 force torque sensor was installed on one robot arm that does all block insertion. The force/torque sensor gives some information about the contacts one block might experience when inserted. We utilize this information to prevent jamming caused by friction.

Robot arms are mounted on two diagonal corners of the table to maximize space utilization. The right robot, R_A, is responsible for picking up blocks and placing them into desired locations, and is equipped with the force-torque sensor. The other arm, R_B, is used during the regrasping and pose estimation process.

We placed a base of four posts at the center of the table as the starting point for structure assembly. Waiting blocks are typically placed at the bottom-right corner of the table, so R_A can move to check that area and pick up a block if presented.

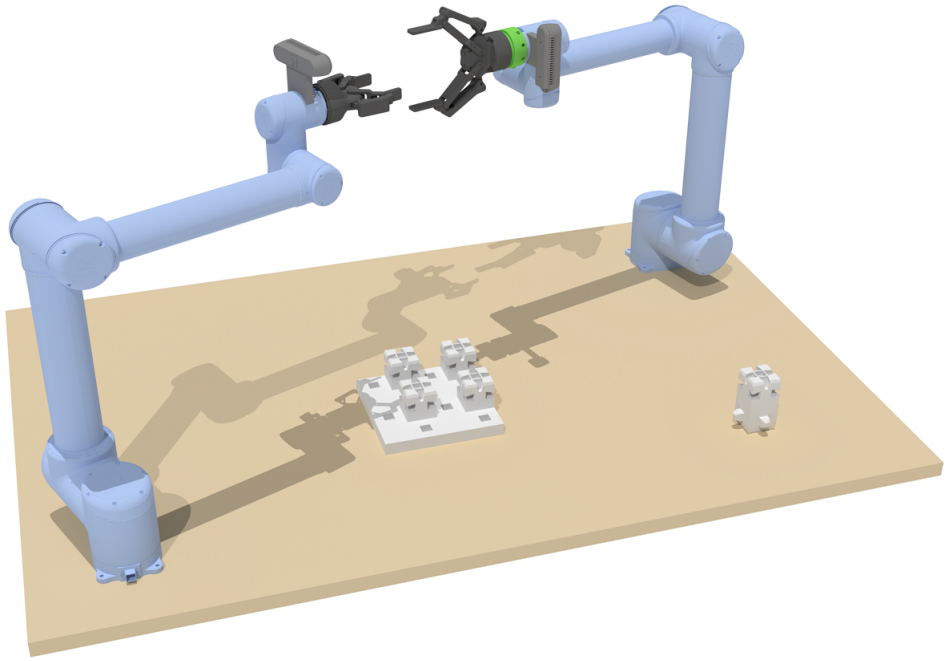


Figure 3.25: Our experiment environment setup. Robot arms are blue, the force torque sensor is green, the depth cameras are in dark gray, while the grippers are black. The blocks and structure base are the white part.

Our assembly process is described in Algorithm 16. The input is a sequence of block types and a desired final configuration. We call a tuple of block type and configurations a *command*. Commands are generated based on the assembly rules described in previous sections. For each command, the right side robot R_A will first pick up a block based on the input type. The block is then moved above the center of the table so the camera mounted on R_B can see the block and estimate its current configuration. If the block is not currently grasped in the desired orientation, we must find a sequence of re-grasping actions and execute them to re-orient the block correctly. Finally, we place the block into the desired location.

3.6.2. Pose estimation and block picking

The precise configuration of waiting blocks is unpredictable without measurements. A reliable method for recognizing the position and orientation is critical. To provide enough friction for holding blocks, gripper fingers are equipped with small rubber pads. The com-

Algorithm 16 High-level robotic assembly process

function ASSEMBLE($C \leftarrow [(b_1, c_1), \ldots, (b_n, c_n)]$)
  for Block b_i and orientation o_i in C **do**
    Move R_A to bottom-right corner of the table.
    Pick up a block (same type as b_i). (Sec 3.6.2)
    Move R_A and R_B above the table center so two arms are facing each other.
    $o_e \leftarrow R_B.EstimateBlockOrient()$. (Sec.3.6.2)
    if $o_e \neq c_i.orientation$ **then**
      $Acts \leftarrow$ generate re-grasping actions for each robot arm. (Sec.3.6.3)
      $ExecuteRegrasp(Acts, R_A, R_B)$. (Sec.3.6.3)
    end if
    Plan path for block assembly.
    R_A executes the path and transform the block into a desired configuration.
  end for
end function

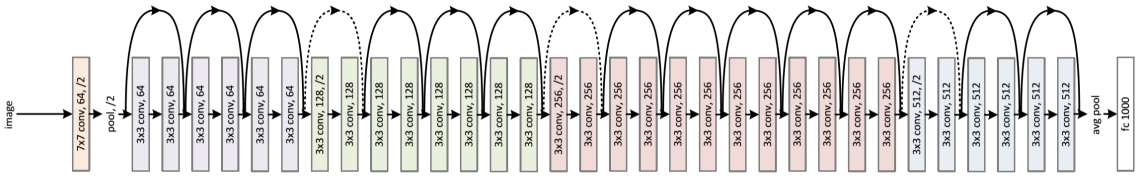


Figure 3.26: An example residual network of 34 layers. The links between layers represent the residual blocks.

pliance also introduces unexpected small block movements or rotations when closing the gripper. These errors can cause assembly to fail if the pose is not re-estimated precisely.

In this chapter, we used a neural network for estimating the configuration of a block. In specific, we trained a ResNet-101 Residual Network to map a 128×128 depth image to a 9-element vector which contains the block type, the surface id, the block position (3 elements) and orientation (4 elements representing a quaternion). The neural network was trained using labeled data generated virtually.

The neural network architecture we used to learn the mapping is a ResNet-Unet configuration [46]. The encoder layers are arranged as ResNet-101 with a mirroring of the structure in the decoder. Figure 3.26 is an example of 34-layer residual network. To make this fully convolutional, we replace the max pool layers with convolution layers. Skip connections are added between corresponding blocks of the ResNet structure. We use a

multi-class cross entropy loss per pixel. We train for 3 epochs.

Our training data was generated in a virtual 3D environment. To generate labeled data, we first loaded a block model into the virtual scene, set a random configuration, placed a camera to look at the block, then generated an orthogonal depth image of 256×256 pixels. The depth image can be viewed as a record of a point cloud with each pixel value indicating the distance from a point on model to camera center. The depth value of each pixel was clipped to allow a maximum distance of 20cm from the virtual camera because the real depth camera provides very noisy readings when looking at far points. We added Perlin noise to the generated depth images to simulate noisy real depth camera readings. The noisy images are then downsampled to 128×128 size to smooth the surfaces of the 3D point cloud.

Block grasping locations are manually specified in each block's local frame. When the pose of a block is estimated, the grasping locations in global coordinate arm are then computed so a robot arm can move to grasp. For the purpose of precise pose estimation later, we only pick up the blocks such that the gripper normal direction (from the center of gripper bottom to the center of fingertips) is parallel to a surface normal. Thus, knowing which grasping point is selected also gives a guess of the block configuration in the gripper frame and the surface the gripper is facing.

Since our compliance fingertips will introduce error to the block pose, to improve the accuracy of pose estimation, we always re-estimate the block pose after it is picked up. One robot arm places a block in front of the other robot's depth camera such that the camera is looking at the center of a block surface with the surface normal roughly parallel to the camera normal direction. Then the pose estimation model will recognize the precise block pose. Hence, training data were generated much more densely around configurations that surface normals are parallel to the camera direction. In the experiment, gripper fingertips were also detected by the depth camera; when generating training data, we also had to include the fingertip locations with specified grasping locations.

Since we assume the camera can be placed roughly facing the opposite face of the grasped block, the offsets for the generated data are within a modest +/- 3 cm and +/- 0.3 radians of the centered view. Figure 3.27 is an example of the generated heightmap of a face with the associated labeled reference rectangle (in blue) superimposed in the same heightmap. We generate roughly 800,000 pairs of 128x128 heightmap images and train for 3 epochs. Figure 3.27 is an example height map from the training data.

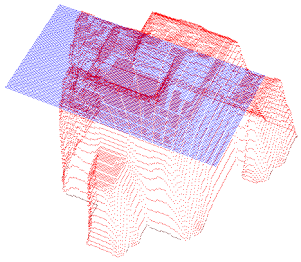


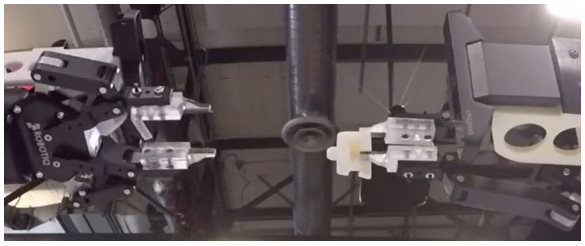
Figure 3.27: An example height map from the training data. The blue rectangle area indicates the face the camera is looking at.

Our model was trained using a separate work station with 4 Nvidia GTX 1080Ti graphics cards. When using the perception model, we make sure the blocks are always presented with one surface normal nearly parallel to the camera normal, and around 17cm from the center of the camera to the surface. The depth camera then takes an image, crops the image to 128×128 pixels, and clip the pixel distance value to 20cm to avoid noisy background readings. The image is then sent to the perception model to evaluate the block configuration with respect to the camera. If the block type and surface id align with our initial guess, we adjust the block to move the surface center closer to the camera view center, and surface normal closer to the camera normal. This process is repeated several times until the adjustment is smaller than an empirically defined threshold. We are thus very sure about the configuration of the block.

3.6.3. Re-grasping

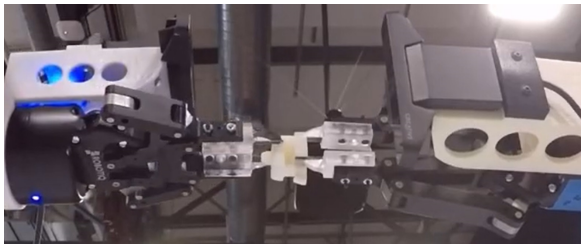
When robot R_A picks up a block, there is no guarantee that the block is held at the desired location. However, the gripper does not allow rotations of a block while holding that block. To solve this problem, we used the second robot R_B to temporarily hold the block, so R_A can re-grasp the block in a different location (See Figure 3.28). We may have to repeat this process for many times to achieve the goal orientation.

Grasping locations for each of the six faces of a block are specified manually. Initially,

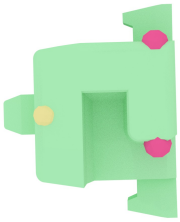


(a) Right robot R_B estimating the block pose.

(b) Block held by R_A at red points.

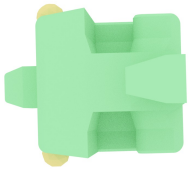


(c) R_A transfering the block to R_B. R_A will release.

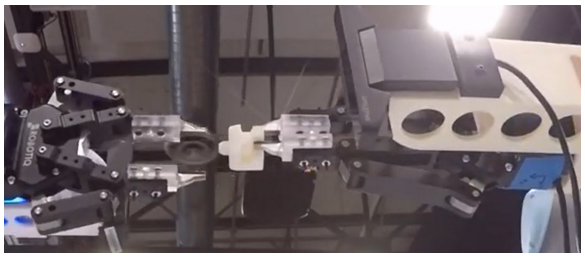


(d) Block grasped by two grippers.

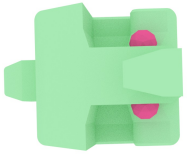
(e) R_B rotated. R_A estimating the block pose.



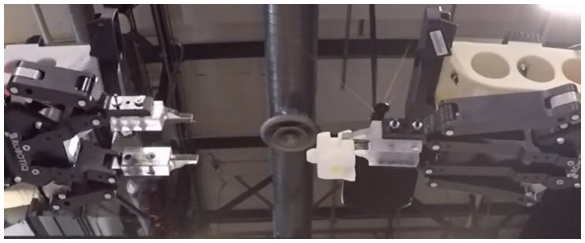
(f) Block pose when rotated.



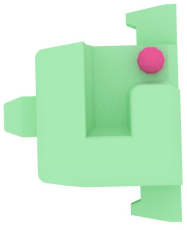
(g) R_B transferred the block back to R_A, then released.



(h) Block held by R_A, R_B released.



(i) Robot R_A rotating back. The block is grasped at the desired location.



(j) Block grasped by R_A in the right location.

Figure 3.28: A re-grasping process that allows the right robot R_A to hold the block in different locations. The left robot is named R_B. Red dots represent grasping locations of R_A, and yellow dots are grasping points by R_B.

we move robots to home configurations C_{ra} and C_{rb} respectively where the grippers are 40cm above the center of the table facing each other, and the cameras are up. The configuration C_{rb} is shown as the left robot configuration in Figure 3.25. Every time a regrasping action is executed, the robot holding the block is moved back to home configuration. Our goal is to have the robot R_A hold the block in the desired orientation while the robot is in the home configuration.

To make the process as fast as possible, we constructed this problem as a tree search, then used breadth-first search to find the shortest sequence of re-grasping actions. The algorithm is shown in Algorithm 17. In this algorithm, we assume the block is initially held by R_A and the initial and goal block orientations are in the frame of the gripper attached to R_A. We first check if the block is held in the desired orientation, if not, a set of valid grasping locations are generated for the other robot. The other robot will imagine re-grasping the block according to each valid grasping location, then transform to its home configuration. If a valid re-grasp will hold the block in the desired orientation, the algorithm will stop, otherwise, children valid re-grasps will be further explored. This process is continued until R_A is holding the block correctly.

There are several configurations a robot cannot achieve, making many grasping locations invalid. For example, our environment requires that no gripper is facing up, as the gripper may collide with the assembling structure in these configurations. Also, due to the nature of how we mount the cameras, we don't want the grippers to be perpendicular to each other with cameras on the same side, because cameras may collide in these configurations.

Algorithm 17 generates a sequence of re-grasping actions since it is planned virtually. We then have the robots to execute the actions. The execution process is straight forward but includes an extra part of block pose estimation before grasping the block. This is because the block is not strictly static with respect to the gripper when and after the previous grasping. So a pose estimation will help to reduce the error.

Algorithm 17 Regrasping actions generation.

1: **function** PLANREGRASP(init orient o_s, goal orient o_g)
2: Move R_A to home configuration C_{ra}.
3: Move R_B to home configuration C_{rb}.
4: $orients \leftarrow$ a stack of block orientations and corresponding holding robot
5: $orients.push((c_s, R_A))$.
6: **while** $orients$ is not empty **do**
7: $o, r \leftarrow orients.pop()$
8: **if** $r = R_A$ and $o = o_g$ **then**
9: **return** $TraceBack(o, r)$
10: **end if**
11: **if** $r = R_A$ **then**
12: valid_grasps $\leftarrow$ valid grasps for R_B
13: **for** $grasp \in valid\_grasps$ **do**
14: $o_{next} \leftarrow$Transform block according to $grasp$, assuming R_B will be moved to C_{rb}.
15: $orients.push((o_{next}, R_B))$.
16: **end for**
17: **else**
18: valid_grasps $\leftarrow$ valid grasps for R_A
19: **for** $grasp \in valid\_grasps$ **do**
20: $o_{next} \leftarrow$Transform block according to $grasp$, assuming R_A will be moved to C_{ra}.
21: $orients.push((o_{next}, R_A))$.
22: **end for**
23: **end if**
24: **end while**
25: **end function**

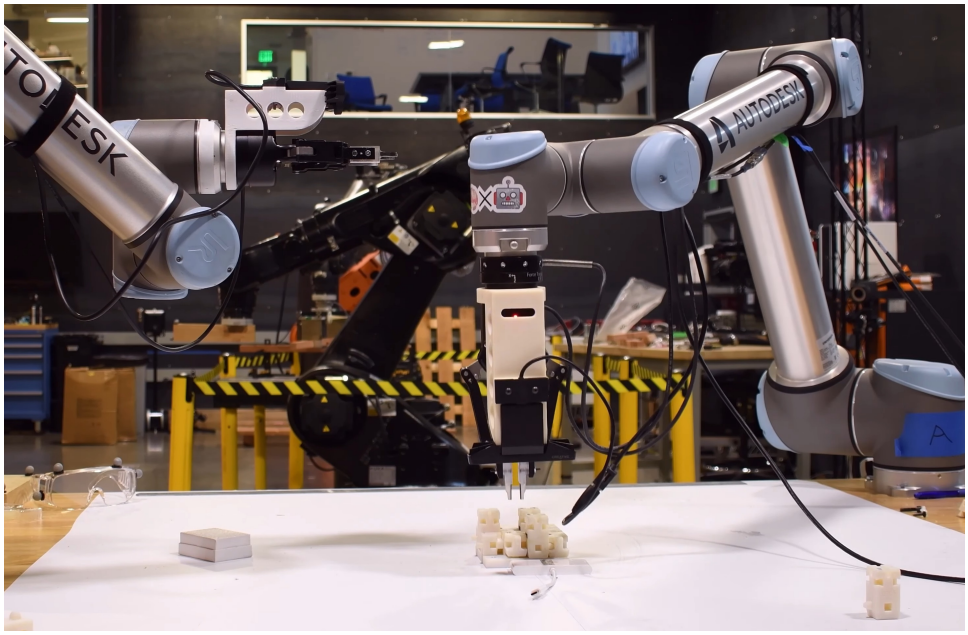


Figure 3.29: Real platform setup. The robots are assembling the first layer of an interlocking cube.

3.6.4. Experiment results

Using the robot system, we assembled a cube-like interlocking structure of 48 blocks. The rendered structure is shown in Figure 3.24a where blue blocks are posts and orange blocks are cubes. This experiment includes all the operations described above. Figure 3.29 is a screenshot of the robots executing assembly commands.

A re-grasping operation can be found at Video [120]. In this video, 4 re-grasping actions were performed (excluding the initial pickup) to re-orient a cube held by the right robot R_A. This operation took over 5 minutes for one block, because the robot is set to move slowly to ensure safety.

A complete video of assembling the cube structure can be found in Video [122]. To prevent making the experiment video too long, we avoided some re-grasping by hand-feeding the robot blocks in correct grasping locations. In total, the experiment took less than 2 hours. Because of the limitation of my camera (30-mininute max video recording), the assembly of each layer is executed and recorded separately in two days.

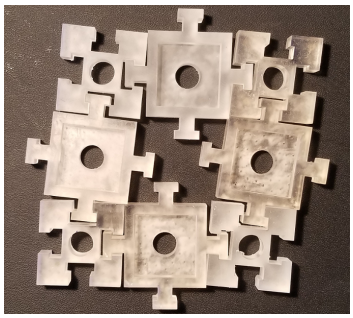
3.6.5. Challenges

We faced many of challenges during the experiment. In this experiment, we made two very strong assumptions. First, we assumed a block can be moved perfectly precisely such that there is no contact during insertion. Second, we assume block joints are fabricated precisely so that the gap between a pair is quite small. These assumptions are unfortunately not satisfied in the real world.

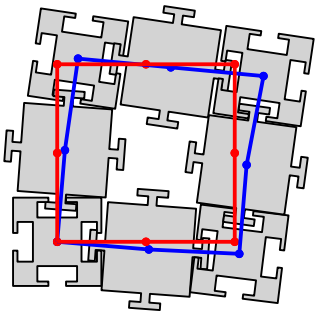
Our robots were fairly precise in terms of repeatability. However, in order to coordinate two robots to work in a shared workspace, a calibration process was performed to align two robots' local frames. This calibration is not perfect and could introduce errors causing the block movement not exactly following a designed trajectory. The pose estimation can also introduce errors due to the nature of neural networks. When grasping a block, the gripper is not always holding near the center of mass, which might cause a tall block to rotate slightly. These errors combined could cause a block to touch other blocks during the insertion process.

The blocks were 3D printed and the printer we had was very precise. However, the support material covered outside each block is problematic. It is impossible to remove completely, and environmental variables such as time, temperature and humidity could cause the support material to expand or shrink slightly, making the insertion very difficult. To avoid this issue, we designed the male joint slightly smaller than the female joint such that not all contact faces of a joint give resistant friction. This design, however, makes the blocks slightly flexible causing the jamming problem later.

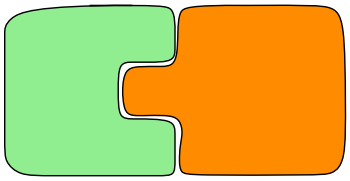
The biggest resulting issue of the unsatisfied assumptions was the friction. Friction can be from many different contact surfaces or points whose number and location are unpredictable. The friction led to some slight rotation of blocks which in some trials caused jamming.



(a) The joints manufactured are unlikely to match perfectly. The assembled structure is slightly flexible.



(b) Blocks are designed to sit along the red rectangle. Due the joint flexibility, there are small displacements. Current positions are marked in blue.



(c) Due to manufacturing error or the purpose of ensuring assemblability, joints do not match perfectly in real world, causing small local movements.

Figure 3.30: Joint manufacturing error causes the assembled structure to be flexible.

Section 3.7

Future work: Flexibility analysis

Ideally, our block are designed so the male and the female joints match perfectly. The blocks connected by a joint pair is almost not able to move, unless strictly following one translation. In the real world, manufacturing process could introduce errors to both joints, a perfect joint pair is unlikely to exist. Even if perfect manufacturing method exists, male joints are purposely designed slightly smaller than the female joint allows, so as to ensure assemblability. Figure 3.30c is an example of a possible joint manufacturing. Assuming the female part (green) is fixed, the male part (orange) is able to move slightly along y-axis instead of strictly along x-axis. Small rotations are also possible.

In our real block assembly, we noticed observable displacements of the blocks. Although the structure is still hard to disassemble if not follow a specific sequence, the structure as a whole is not strictly *stable* or *immobilized*. We would like to know how bad is the displacement? How much can blocks move assuming we know exactly their manufactured geometry? Is the movement big enough to damage the interlocking property of the

structure making some substructures actually separable? In this section, we will introduce a method for analyzing the flexibility of blocks.

The work [58] described in this section is done by Samuel Lensgraf, Karim Itani, Yinan Zhang, Zezhou Sun, Yijia Wu, Alberto Quattrini Li, Bo Zhu, Emily Whiting, Weifu Wang, and Devin Balkcom. My contribution is the limitation subsection 3.7.3. The work is included here, because it is a natural extension of the current work on interlocking blocks and general assemble-able structures.

3.7.1. Block movement in a two-block structure

Assuming our structure is a single pair of blocks as in Figure 3.31 with the female joint fixed. The male joint can have slight movement in the local space. But however it moves, its vertices will not collide with the edges on the female joint. In this case, point $\mathbf{p}$ cannot collide with edges E_1, E_2 or E_3.

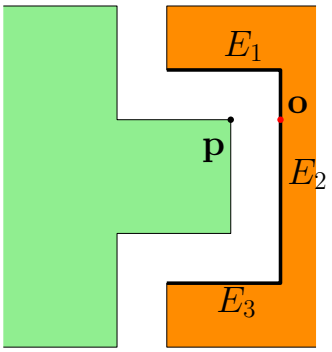


Figure 3.31: Moving the male joint should not collide point p with edges E_1, E_2 and E_3.

Let the configuration of the structure be $\mathbf{q} \in Q$. We consider how a vector of vertices $\mathbf{o}(\mathbf{q})$ will collide with some near point $\mathbf{p}(\mathbf{q})$. A signed distance function $\mathbf{d}(\mathbf{o},\mathbf{p})$ is used to give the distance information between the set of vertices and near collision points. Since no collision is allowed, we enforce $\mathbf{d}(\mathbf{o},\mathbf{p}) \geq \mathbf{0}$. Since both $\mathbf{o}$ and $\mathbf{p}$ rely on the configuration $\mathbf{q}$. The signed distance function can also be viewed as a function of the configuration, or $\mathbf{d}(\mathbf{q})$.

As the two-block system is moving slightly, we describe the configuration of the system using a function of time: $\mathbf{q}(t) \in Q$. At a specific moment t' of time, the instantaneous rate of change of the distance function is

$$\dot{\mathbf{d}}(\mathbf{q}(t'),\dot{\mathbf{q}}(t')) = J_d(\mathbf{q}(t'))\dot{\mathbf{q}}(t') \tag{3.7}$$

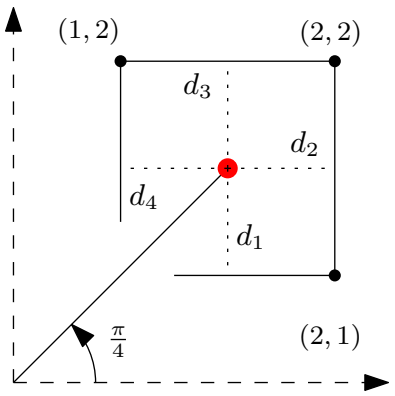


Figure 3.32: An example joint pair of a peg and a square-like hole. To analyze the flexibility of the peg, we compute how much the end (red) of the peg can move in the square-like hole.

For simplicity, we write

$$\dot{\mathbf{d}}(\mathbf{q}, \dot{\mathbf{q}}) = J_d(\mathbf{q})\dot{\mathbf{q}} \tag{3.8}$$

For a short amount of time, we can treat the rate of change of the signed distance function a constant. Thus the change of distance is approximated using

$$\triangle \mathbf{d}(t) \approx \triangle t \cdot \dot{\mathbf{d}}(\mathbf{q}, \dot{\mathbf{q}}) \tag{3.9}$$

Let $\mathbf{d_0} = \mathbf{d}(\mathbf{o_0}, \mathbf{p_0})$ be the initial distance vector at time 0. A valid small movement of the system cannot have vertices collide with edges, thus $\triangle \mathbf{d}(t) \leq -\mathbf{d_0}$. Combined with Equation 3.8, we have

$$\triangle t \cdot J_d(\mathbf{q})\dot{\mathbf{q}} \geq \mathbf{d_0} \tag{3.10}$$

The scalar factor $\triangle t$ can be dropped since we may scale the configuration change $\dot{\mathbf{q}}$ and time unit such that $\triangle t = 1$. The above equation is simplified as

$$J_d(\mathbf{q})\dot{\mathbf{q}} + \mathbf{d_0} \geq \mathbf{0} \tag{3.11}$$

3.7.2. A simple example

To better visualize the idea, see an example in Figure 3.32 where a peg is inserted into a hole. The peg is a straight line segment with one end at point $(0,0)$ another end at point $(1.5,1.5)$. We use $q = \pi/4$ to represent the peg's configuration. The square-like hole has four edges parallel to the axes and three corners $\mathbf{o} = \{(2,1),(2,2),(1,2)\}$. To analyze how much the peg can move with respect to the square-like hole, we consider the red end point p of the peg and its distance to four edges of the hole.

$$p(q) = (2 \cdot cos\theta, 2 \cdot sin\theta) \tag{3.12}$$

is the coordinate of the red point p, and θ is the angle between the segment peg and x-axis. There are four distance functions:

$$d_1 = p_y - o_{1_y} = 2sin\theta - 1 \tag{3.13}$$

$$d_2 = -(p_x - o_{2_x}) = -2cos\theta + 2 \tag{3.14}$$

$$d_3 = -(p_y - o_{2_y}) = -2sin\theta + 2 \tag{3.15}$$

$$d_4 = p_x - o_{3_x} = 2cos\theta - 1 \tag{3.16}$$

corresponding to the distance from point p to the bottom, right, top and left walls. Computing the partial derivatives with respect to θ,

$$J_d(q)\dot{q}+\mathbf{d}=\begin{pmatrix}2cos\theta\\2sin\theta\\-2cos\theta\\2sin\theta\end{pmatrix}\cdot\dot{q}+\mathbf{d}\geq\mathbf{0} \tag{3.17}$$

Since $\theta=\pi/4$, we have

$$\begin{pmatrix}\sqrt{2}\\\sqrt{2}\\-\sqrt{2}\\-\sqrt{2}\end{pmatrix}\cdot\dot{q}+\begin{pmatrix}\sqrt{2}-1\\2-\sqrt{2}\\2-\sqrt{2}\\\sqrt{2}-1\end{pmatrix}\geq\mathbf{0} \tag{3.18}$$

Candidate values for $\dot{q}$, or equivalent, Δq, are

$$\begin{pmatrix}-(\sqrt{2}-1)/\sqrt{2}\\-(2-\sqrt{2})/\sqrt{2}\\-(2-\sqrt{2})/\sqrt{2}\\-(\sqrt{2}-1)/\sqrt{2}\end{pmatrix}\approx\begin{pmatrix}-0.29\\-0.41\\0.41\\0.29\end{pmatrix} \tag{3.19}$$

The values here correspond to how much the peg should rotate to have its end point collide with walls of the hole. For example, $\Delta\theta=-0.29$ means if the peg rotates 0.29 radian, point p will collide with the bottom wall. Taking the smallest absolute value of the solution, we know $\Delta\theta=-/+0.29$ will be the first to cause collisions. The values are not exactly correct due to the linerization of J around the initial configuration.

3.7.3. Flexibility analysis based on linear programming

In previous subsections, we developed linear constraints for the motion of vertices. With the constraints, we can simulate how a 2D structure of loosely connected parts moves to extreme configurations by solving the following linear program

$$\begin{aligned} \max_{\dot{q}} \quad & \mathbf{c}^T \dot{\mathbf{q}} \\ \text{st.} \quad & J(q)\dot{\mathbf{q}} + \mathbf{d_0} \geq \mathbf{0} \end{aligned} \tag{3.20}$$

where vector $\mathbf{c}$ is a list of weights. By turning the weights, we allow the blocks to move in different directions.

In our implementation, in each pair of blocks in adjacent, we choose one to provide the edges, and the other to provide vertices. Same as in Figure 3.31. Our distance function and constraints as in Equation 3.9 and 3.11 assume each time step to be small and the motion to be only in a small local region, therefore, each vertex is only paired with edges in a small distance ε.

To help understand the computation of the Jacobian matrix $J(q)$, let us see the following example of a pair of blocks in Figure 3.33.

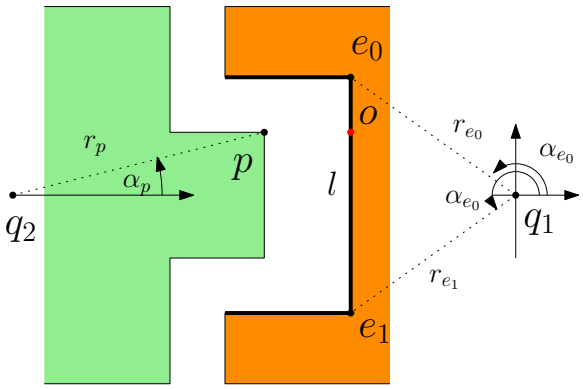


Figure 3.33: A joint pair and variables used for computing the Jacobian matrix. Block 1 is orange, and block 2 is green.

Two blocks are presented in Figure 3.33 where the orange block on the right side is named block 1 and the green block on the left is block 2. They are under configurations $q_1 = \{x_1, y_1, \theta_1\}$ and $q_2 = \{x_2, y_2, \theta_2\}$ respectively. In this pair, we will have multiple distances in vector $\mathbf{d}(\mathbf{o}, \mathbf{p})$ where p is a point on block 1 and p is a nearby collision vertex on block 2. Let $\mathbf{n}(q_1)$ be the normal direction pointing outwards and going through point o. The distance d_{ij} between points p and n can be written as

$$d_{i,j}(q_1, q_2) = \mathbf{n}(q_1) \cdot (p(q_2) - o(q_1)) \tag{3.21}$$

The edge containing point o has two ends e_0 and e_1 whose distances to the origin of block 1 are r_{e_0} and r_{e_1} respectively and the length of the edge is l. The angle from the

x-axis in object 1's local frame to e_0 and e_1 are α_{e_0} and α_{e_1}. Similarly, let the distance from the block 2 origin to p be r_p, and the angle from the x-axis to p be α_p. To make our equations more readable, we define the following helper variables: $s_{e_0} = sin(\theta_1 + \alpha_{e_0})$, $c_{e_0} = cos(\theta_1 + \alpha_{e_0})$, $s_{e_1} = sin(\theta_1 + \alpha_{e_1})$, $c_{e_1} = cos(\theta_1 + \alpha_{e_1})$. Then

$$p(q_2) = \begin{pmatrix} x_2 + r_p c_p \\ y_2 + r_p s_p \end{pmatrix} \tag{3.22}$$

$$n(q_1) = 1/l \cdot \begin{pmatrix} r_{e_1} s_{e_1} - r_{e_0} s_{e_0} \\ -r_{e_1} c_{e_1} + r_{e_0} c_{e_0} \end{pmatrix} \tag{3.23}$$

$$o(q_1) = \begin{pmatrix} x_1 + r_{e_0} c_{e_0} \\ y_1 + r_{e_0} s_{e_0} \end{pmatrix}. \tag{3.24}$$

The Jacobian matrix is a matrix of the gradients of the distance function:

$$J(\mathbf{q}) = \begin{bmatrix} \frac{\partial d_{ij}}{\partial x_1}, \frac{\partial d_{ij}}{\partial y_1}, \frac{\partial d_{ij}}{\partial \theta_1}, \\ \frac{\partial d_{ij}}{\partial x_2}, \frac{\partial d_{ij}}{\partial y_2}, \frac{\partial d_{ij}}{\partial \theta_2} \end{bmatrix} \tag{3.25}$$

where $\mathbf{q} = [q_1, q_2]$ and

$$\frac{\partial d_{ij}}{\partial x_1} = \frac{1}{l} \cdot (-r_{e_0} s_{e_0} + r_{e_1} s_{e_1}) \tag{3.26}$$

$$\frac{\partial d_{ij}}{\partial y_1} = \frac{1}{l} \cdot (r_{e_0} c_{e_0} - r_{e_1} c_{e_1}) \tag{3.27}$$

$$\begin{aligned} \frac{\partial d_{ij}}{\partial \theta_1} =& \frac{1}{l} \cdot (c_{e_0} r_{e_0} (c_{e_0} r_{e_0} - c_{e_1} r_{e_1}) - r_{e_0} s_{e_0} (-r_{e_0} s_{e_0} + r_{e_1} s_{e_1}) \\ &+ (-c_{e_0} r_{e_0} + c_{e_1} r_{e_1})(-c_p r_p + c_{e_0} r_{e_0} + x_1 - x_2) \\ &+ (-r_{e_0} s_{e_0} + r_{e_1} s_{e_1})(-r_p s_p + r_{e_0} s_{e_0} + y_1 - y_2)) \end{aligned} \tag{3.28}$$

$$\frac{\partial d_{ij}}{\partial x_2} = -\frac{1}{l} \cdot (-r_{e_0} s_{e_0} + r_{e_1} s_{e_1}) \tag{3.29}$$

$$\frac{\partial d_{ij}}{\partial y_2} = -\frac{1}{l} \cdot (c_{e_0} r_{e_0} - c_{e_1} r_{e_1}) \tag{3.30}$$

$$\frac{\partial d_{ij}}{\partial \theta_2} = \frac{1}{l} \cdot (-c_p r_p (c_{e_0} r_{e_0} - c_{e_1} r_{e_1}) + r_p s_p (-r_{e_0} s_{e_0} + r_{e_1} s_{e_1})) \tag{3.31}$$

Modeling convex corners . Our signed distance function introduced a set of linear constraints as in Equation 3.20, which means the constrained region of valid movements is bounded in a convex region. This can be particularly bad for some geometries. For example, in Figure 3.34, point p_1 is closest to point o_1, a convex corner vertex. Two distance functions are created to constrain the movement of p_1 to edges e_1 and e_2. However, these two constraints keep the point inside a convex cone region defined by the extension of e_1 and e_2. This example suggests that our solution provides a lower-bound of the movement of the point, instead of the exact constraint. Constraints in each row of the Jacobian express an *AND* relationship, while in this example a more precise constraint is "point p_1 cannot collide with edges e_1 *OR* e_2".

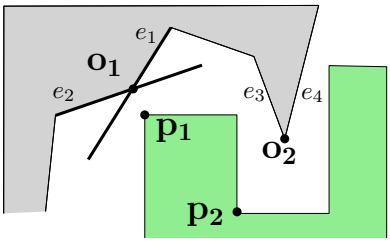
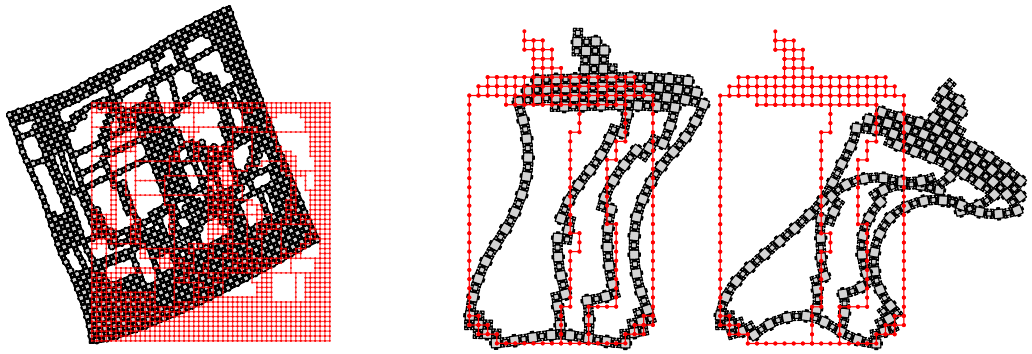


Figure 3.34: The edge-vertex distance constraint limits the valid motion in a small convex region. p_1 distance constraints to e_1 and e_2 makes it impossible to move beyond the ling extended from the edges.

The problem can be avoided if only one of the nearby vertices is convex. For example, in Figure 3.34, setting up constraints for point p_2 against e_3 and e_4 causes the same problem, however, if we swap p_2 with o_2, the constraints become precise.

3.7.4. Results

Our method can be used to analyze how individual joint flexibility can change the shape of a whole structure.



(a) A structure of 1703 rigid bodies flexing upwards.

(b) Crushing a soda can with tight (left) and loose (right) joints.

Figure 3.35: Some maximum flexibility analysis results. Red polygons denote the initial configuration.

Figure 3.35a shows the deformation of a shape with loose joints. By optimizing the objective function to move blocks upwards, the structure of 1703 blocks looks very different from its designed shape. This indicates too much flexibility of joint pairs. The Jacobian matrix size of this example is 52655×5106, with 14 iterations, the problem was solved in 223.341 seconds.

Figure 3.35b shows the structure deformation change when joints of different flexibilities are used.

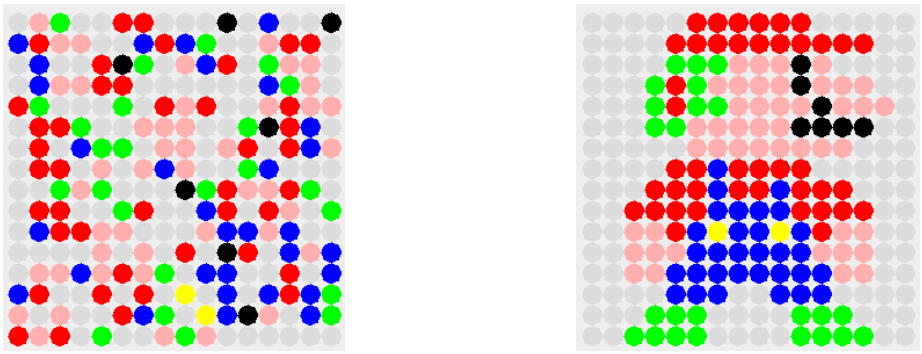
Chapter 4

Rearranging agents using global controls

In the previous assembly problems, we assume every piece can move individually. We can apply any rigid body motion to any object, and the motion of one object does not depend on other objects. This assumption is not always true in the real world, especially when we are dealing with small elements. We cannot command a bacterial cell; we do not want to move sands one-by-one; we cannot control individual biomolecule. Many similar objects are not individually actuate-able nor even reachable, which makes it challenging to rearranging a swarm of these elements.

The same issue occurs with swarm robot systems where each agent is small and under actuated. Swarm robots are commonly expected to perform complicated tasks such as shape forming and transforming. However, limitations in computing power, the lack of actuation, or the restriction of incomplete information about the environment make individual planning and control difficult. A practical solution is to have a centralized planner and actuate robots (agents) using global controls, e.g., gravity force or magnetic field. For example, by rotating a package, gravity makes all packing peanuts move in the same direction.

In this chapter, we will study how global control signals can be used to rearrange pieces/agents. We first explore a simpler problem in 2D. We simplify the setting into a 2D grid space every square piece (or agent) occupies a unit cell in every discrete time step. A global command *left, right, up* or *down* moves all agents in the same direction until an



(a) Agents are initially randomly sitting in a rectangle matrix.

(b) The goal is to rearrange each agents into a target configuration.

Figure 4.1: Given a set of agnets sitting in a rectangle matrix, rearrange them by placing obstacles in the grid space, and applying 4 global control signals *left, right, up* or *down*.

agent hits an obstacle or another stopped agent. The study focuses on a special case where agents form a rectangle matrix initially, and each agent needs to be rearranged into a different position in the matrix. Figure 4.1 is an example of our setup. Initially, the agents are sitting in a rectangular matrix with random configurations. Our goal is to transform them into a target configuration using the 4 global commands. How should we place obstacles in the space? How much time is needed to finish the rearrangement? How large should the workspace be?

This chapter is primarily inspired by a collection of prior papers by Becker *et al.* ([12, 13] and [15]) using the same model, and show that by placing obstacles cleverly, sequences of global controls can allow essentially arbitrary re-configuration of the agents. In this chapter, we will see that with different placement of obstacles, the space requirements (a rectangle-bounded space where the agents and obstacles live in) may be much smaller than the previous work thought. Specifically, general re-arrangement of an $n \cdot m$ rectangle of agents requires a board that is only a constant factor larger than the number of agents. For a specific $n \cdot m$ matrix of agents, constant-time rearrangement can be done in a $O(n \cdot m \cdot (n + m))$ workspace.

We present two approaches to design the workspace. In the first *task-specific* approach, we design the board based on the start and goal configuration of the agent matrices. This

approach requires very small time complexity in completing the transformation but could be impractical for cases where both the start and goal configurations are random. In the second approach, *multi-purpose* design, obstacles are placed in fixed positions, requiring much more complicated sequences of controls are required to finish the reconfiguration.

We hope further studies on how to transform agents from arbitrary shapes to matrices, then vise versa, will grant the ability to perform arbitrary reconfiguration of swarm robots using global controls. By studying these problems, we are exploring a new assembling technique. One can imagine an automatic assembly device: a black box that takes some small elements as input, after rotating the box several times, the elements stack up forming a desired structure.

Section 4.1

Related work

Becker *et al.* [12] examined the same model of particle swarms and proved it is NP-hard to decide whether a given initial configuration can be transformed into a desired target configuration, if the obstacles are fixed. Becker *et al.* designed an algorithm to construct AND and OR logic gates, when allowed to place obstacles, and also provided an algorithm to place obstacles in a $O(N^2)$ space to perform arbitrary permutations, where N is the number of robots. [13] later proved a stronger result: the problem of finding an optimal control sequence is PSPACE-complete. Shad *et al.* [91] extended the idea of building logic gates using unit-size swarm robots with global control into building a binary memory unit and showed it is possible to have nano-robots perform arbitrarily complex operations without using external computational devices.

Collecting objects in a compact manner has been studied using a similar model. Dhagat and O'Rourke [32] considered the problem of pushing square objects in a grid world. Mahadev *et al.* [61] provided algorithms to concentrate under-actuated swarm robots despite obstacles. Akitaya *et al.* [1] considered the problem of sweeping a line to compact a set of

objects setting in a grid world, and showed that deciding whether the set of objects can be pushed to form a square is NP-hard.

Over the past several years, researchers have built swarms of biologically-inspired small-scale robots; papers include [9, 60, 74, 53, 22]. Caprari *et al.* [20] built programmable ultra-low-power robots. Kornienko *et al.* [56] used the Jasmine micro-robots to study the re-embodiment of biological aggregation behavior of honeybees. Rubenstein *et al.* [85] built a thousand-robot swarm, Kilobot. Becker [14] later studied manipulation of large populations of simple robots with common input signals using kilobots. Rubenstein *et al.* [86] also used kilobots to study object transportation. Manipulation of droplets in lab-on-a-chip designs is an emerging application [92], and global controls may serve as one approach.

Seminal work on minimalist manipulation by Erdmann and Mason used tray tilting and interaction of a part with obstacles to coerce the part into a desired configuration [35]. Rubenstein *et al.* [87] designed an algorithm to control kilobots to self-assemble shapes in 2D. Kotay *et al.* [57] designed a robotic module, groups of which aggregate into 3D structures. Rus and Vona's early work [88] on the Crystalline robot presented an algorithm to do self-reconfiguration. Arbuckle *et al.* [4] allowed identical memory-less agents to construct and repair arbitrary shapes in the plane. In the authors' own work [123] on assembly of interlocking structures, nine kinds of blocks are used to build large-scale voxelized models such that all blocks are interlocked and the whole structure is rigid as a whole.

Section 4.2

Task specific board design

Given a start matrix of $n \cdot m$ agents, and a goal matrix, how can we design a board by placing obstacles in the grid space that is able to transform the start matrix into the goal matrix? This section will provide two algorithms for generating such boards for specific start and goal matrices. Alghouth each different task requires re-designing a workspace,

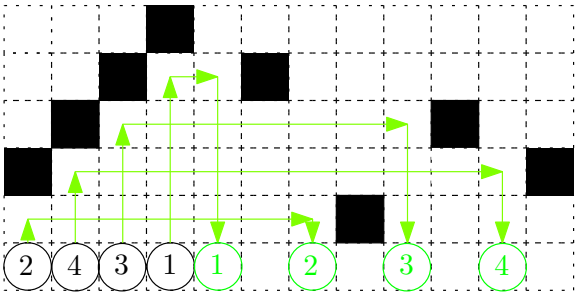


Figure 4.2: Rearranging a row of n agents using $2n$ obstacles in a $(n+2)\cdot(3n)$ space. Green spheres are in desired order.

this approach requires very small time complexity and can be used for repetitive jobs.

We will first discuss the simplest case where there is only a row of agents, then apply the techniques derived from this case to 2D $n \times m$ matrices.

4.2.1. Rearranging a row of agents

Consider the simplest case where the matrix of unit-sized agents has only one row. Figure 4.2 is an example with a row of 4 elements arranged as $[2,4,3,1]$.

To help illustrate the process, we represent each agent using a ball marked by a unique number from 1 to n indicating its final order in the row. The goal is to sort the array in ascending order. Through out this chapter, the $(0,0)$ position of the space is always the left-bottom corner. Coordinate (x,y) is a position at the x-th column and the y-th row. In Figure 4.2, the four agents can be rearranged using four controls: *up, right, down* and *left*.

The idea here is to first separate each agent into different rows (y coordinates) in the plane with an *up* command. Each agent can then be moved to the sorted relative positions using a single *right* control and then re-assembled into the same row with a *down* and *left* command.

To design a board that allows such agent movement, assume an agent is sitting at the i-th position of the initial row and is supposed to move to the j-th position in the sorted array (indices start from 0), two obstacles will be placed at $(i,i+2)$ and $(n+2(j+1)-1,i+2)$ positions, where n is the number of agents in the array. This design is illustrated in Figure 4.2.

In the above design, separating n agents into different rows requires $(n+2)\cdot n$ space. Moving each agent into its desired relative position requires at most $(n+2)\cdot 2n$ space. The total space requirement is thus $(n+2)\cdot 3n$. Since we placed two obstacles for each agent, the total number of obstacles is thus $2n$.

Re-configuring a vertical column of agents can be viewed as a 90-degree rotation of a horizontal array.

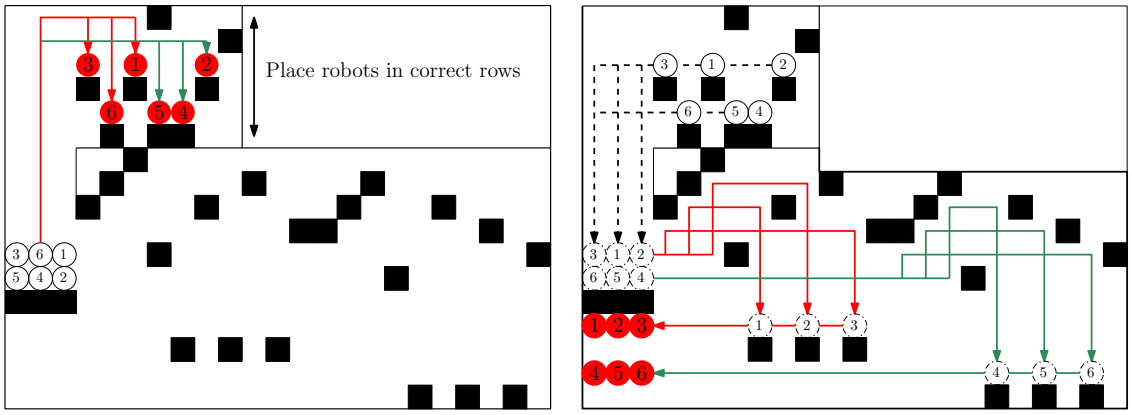
4.2.2. Rearranging a matrix using eleven controls

The method mentioned in the previous subsection can also be extended to apply to cases where more than one row/column agents are present. Here, we present an algorithm for rearranging a matrix of $n\cdot m$ agents. Figure 4.4 is an example of a 7×7 size heart image being rearranged using this algorithm.

Taking advantage of the fact that all row rearrangements require the same controls to sort rows simultaneously, we separate each agent into different rows, then move each into the desired relative place in the target arrangement, finally agents are re-assembled into a rows and then a matrix. Example in Figure 4.3 explains how this algorithm works visually. Let the initial agents be in a matrix M and the goal configuration be in a matrix M'. (All indices start from 0.) The algorithm is described in two parts. The first part places agents into the correct rows, while the second part rearranges every row simultaneously.

1) **Place agents in correct rows.** First place obstacles at $(m+m*i, 6n+m+2-i)$ for $i\in[1,n]$ to separate rows. Then for each agent, if at $M[x,y]$ and to be moved to $M'[x',y']$, place an obstacle at $(m+x, 3n+m+2y'+2)$. Thus every agent can be transported to its desired row. We denote the matrix after completing the first step as M''.

2) **Rearrange agent arrays.** We place m obstacles at position $(i,2n)$ for $i\in[0,m-1]$ to stop agents from the last *down* command. Similarly, we place obstacles to separate rows at $(2m+(3m+1)\cdot i, 3n-i)$ for $i\in[0,n-1]$. Then for each agent, if at $M''[x,y]$



(a) The first step moves every agent to its final using a 5-control sequence: *up*, *right*, *down*, *left*, *down*.

(b) With agents in their correct rows, we now rearrange all rows together using a 5-control sequence: *right*, *up*, *right*, *down*, *left*.

Figure 4.3: A board design to rearrange a 2×3 matrix in constant steps. The rearrangement is done in two parts; one moves agents to correct rows, another rearranges all rows at the same time.

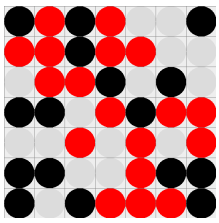
and to be moved to $M'[x', y]$, place three obstacles at $(m + (3m+1) \cdot (n-1-y) + x, 3n+1+x+y)$, $(2m + (3m+1) \cdot (n-1-y) + 2x' + 2, 3n + y + (m-1-x'))$ and $(2m + (3m+1)(n-1-y) + 2x' + 1, 2y)$.

As we can see from the algorithm, the width of the board is $m + (3m+1) \cdot n$, and the height of the board is $6n + m + 2$. The space complexity of the board is thus $O(m \cdot n \cdot (m + n))$, and $2n + 6m$ obstacles are placed in the board.

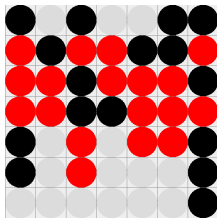
Section 4.3

General purpose board design

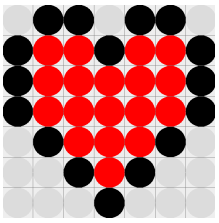
Previously, we have discussed how to design boards for some specific rearrangement tasks. Each board is tight to one task and cannot be used for other purpose. Although the designed boards provide fast solution (a constant number of controls for rearrangement), they require a space that is relatively large, and the boards cannot be re-used for other tasks. In this section, we will propose reusable board designs for general purpose agent matrix rearrangements.



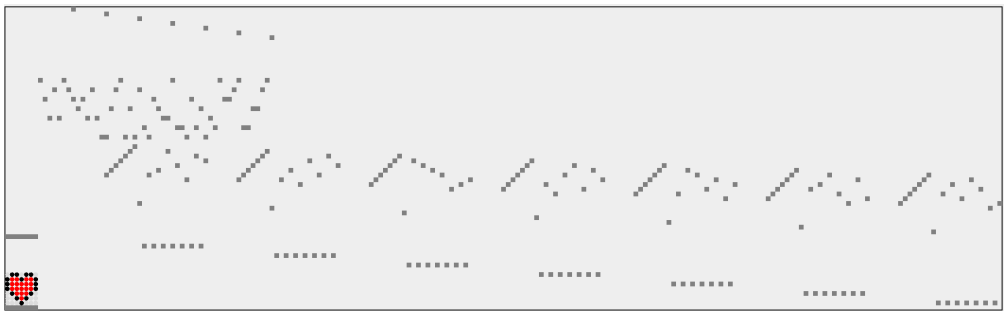
(a) A randomized arrangement of a heart image.



(b) Agents are placed in target rows after 5 moves.



(c) The target rearrangement of the heart image.



(d) The space designed for rearranging the 7×7 heart image. Dark gray squares are obstacles

Figure 4.4: Rearranging a heart image (7×7) in a grid space in eleven moves. Initially, agent positions are randomized in Figure 4.4a. After 5 moves, all agents are moved to their target positions, Figure 4.4b. Figure 4.4c shows the final result of a rearranged heart image.

Inspired by the bubble sort algorithm, we treat the matrix as a folded array, and sort the array by swapping adjacent agents. We will demonstrate the swapping mechanism first, then show how agents are shifted so we only have to swap between two fixed positions of a matrix. The two parts are combined in one single board capable of rearranging any matrices of $n \times m$ size or smaller. The rearrangement using this board will require $O(N^2)$ time, where $N = n \times m$ is the number of agents.

In the last part of this section, we will also present a small board design for sorting a column/row of agents in $N \cdot \log N$ time by imitating quicksort algorithm.

4.3.1. Swapping adjacent agents

We first present the technique to swap two agents at the end of a row using three commands: *up, right* and *down*. Our board design uses the first command (*up*) to separate two target agents and the rest of the agents into three rows. The second command (*right*) changes

the relative positions of the target agents. Then the last command (*down*) reassembles all agents into the same row. Figure 4.5 shows how the process is done for a row of four agents. The relative positions of other agents (gray) remain unchanged.

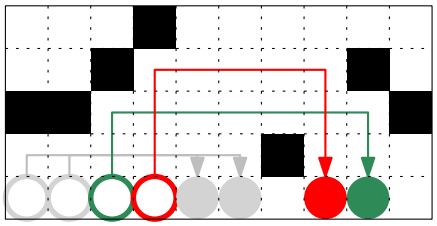


Figure 4.5: Swapping the green and red agents. Other agents remain in the same order. This design requires $5+(n-1)$ obstacle and $5\cdot(2n+1)$ space, where n is the number of agents in the array.

The obstacles are placed in two steps:

1) Place obstacles at $(i,2)$ for $i \in [0,n-3]$,

2) Place obstacles at $(n-2,3)$, $(n-1,4)$, $(2n-2,1)$, $(2n,3)$ and $(2n+1,2)$.

The width of the grid space is $2n+2$, and the height is a constant (5). Thus we have in total $n+3$ obstacle on the board.

This board design can easily be modified to swap two elements in the first row of a matrix with multiple rows and columns. The idea is to first separate the first row from others, then swap elements in the separated row while keeping the rest elements' relative positions unchanged. We will later show how it can be done in Section 4.3.3.

4.3.2. Shifting agent positions

Since the swap operation only applies to two agents in two fixed positions (the last two elements of a row), to swap other adjacent agents, we have to first shift positions of all agents. Consider an array of agents arranged as $[a_1,a_2,a_3,\ldots,a_n]$, a shift operation will transform the array into $[a_n,a_1,a_2,\ldots,a_{n-1}]$. For a rectangular matrix of agents, we treat it as a folded one dimensional array, the shift operation will do the exact same rearrangement to the matrix.

The shift operation board design is based on one key observation: shifting agents by one position will only change the relative positions of the first and the last elements of an array. All other elements' relative positions remain unchanged. Figure 4.6a demonstrates this observation. Similarly, shifting a matrix of agents will move all columns other than the first one by one position to the left. The first column will become the last column and the first and last elements of the row will shift positions. Thus, instead of moving agents one-by-one, which will take $O(N)$ steps, the matrix arrangement shifting can be done in four steps:

1) Pop the first element of the first column.

2) Push the element back to the end of the first column.

3) Pop the first column.

4) Push the column to the end.

If a column of agents is treated as a whole agent, steps 3 and 4 can also be viewed as a rearrangement of a single line of agents. So our design is a combination of two agent line rearrangements. This is shown in Figure 4.6 where a matrix of 3×3 agents are shifted by one position.

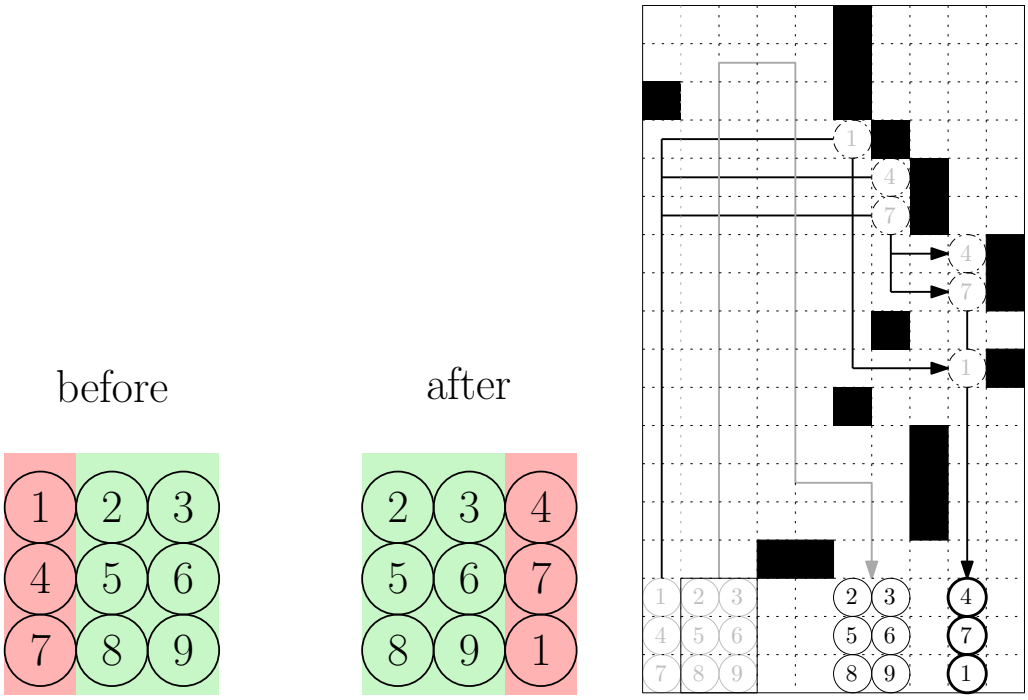
We place obstacles in the following way:

1) One obstacle at at $(0, 4n+3)$ to separate the first column from other columns.

2) $m-1$ obstacles at $(m+i, n)$ for $i \in [0, m-2]$ and n obstacles at $(2m-1, 4n+3+i)$ for $i \in [0, n-1]$ for elements not in the first column

3) Three obstacles at $(2m, 4n+2)$, $(2m-1, 2n+1)$ and $(3m, 2n+2)$ for the first element of the first column.

4) For other elements of the first column, put $n-1$ obstacles at (3m - 2, 4n + 2 - i) for $i \in [1, n-1]$, $n-1$ obstacles at $(3m, 3n+2-j)$ for $j \in [1, n-1]$ and one obstacle at $(3m-3, 2n+3)$.



(a) Shifting elements in a matrix only changes the position of elements in the first column (red). Other elements remain the same.

(b) The board design and its process for shifting a matrix of 3×3 elements.

Figure 4.6: Shifting all agents to predecessor positions. The first agent is shifted to the end. The shifting can be viewed as popping the first element of the first column and pushing it back to the end of the column, then popping and pushing the first column to the end.

In the above board design algorithm, the space is $3m+1$ wide and $5n+3$ tall, so the space complexity is also $O(n \cdot m)$. A total of $4n+m+1$ obstacles are required.

4.3.3. The design of a $O(N)$ workspace

With the capability of shifting and swapping agents, we are able to apply the bubble sort algorithm to any matrix of agents. But first, we need to combine the two boards designed for different operations into one.

Figure 4.7 is an example board designed for rearranging 3×3 agent matrix. The left part of that space is a shifting room and the right part is a swapping room. The bottom of the right part is designed to constrain changes of relative positions of agents not in the first row. Although this board is generated for 3×3 matrices, it also works for smaller matrices. We can transform any matrix A into B using a bubble sort algorithm. We treat each matrix

as a folded one dimensional array whose first element is the top-left element while the last is at the bottom-right. Agents in matrix B are labeled as a sorted array in ascending order.

The sorting algorithm is simple: If the last two elements of the first row are in descending order and they are not the end and the head of matrix B, swap. Otherwise, shift. Once we found $n \times m$ times of consecutive shifts are performed, the matrix is sorted, and we stop any more operations.

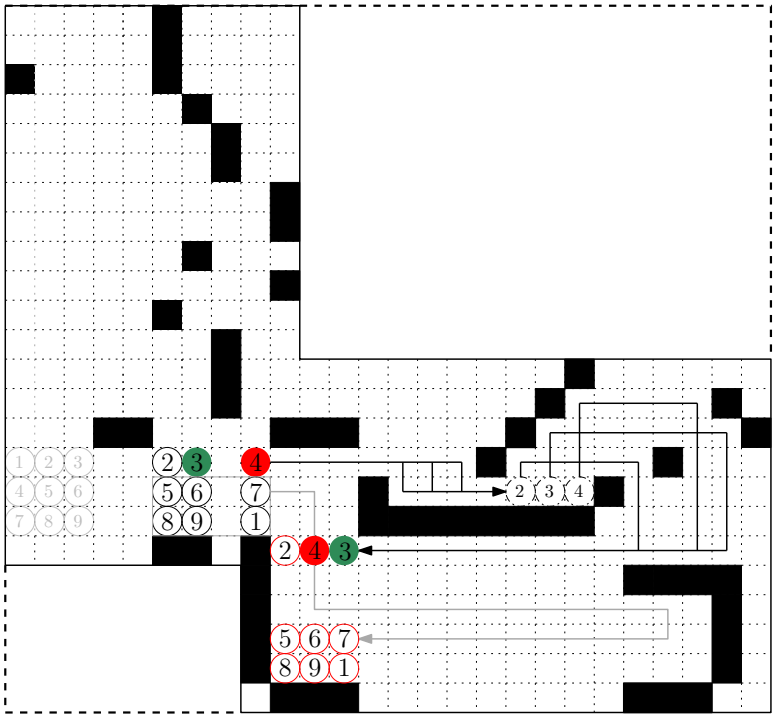


Figure 4.7: The design of a $O(n \cdot m)$ size board for rearranging arbitrary agent matrices smaller than $n \cdot m$. The board has two parts: the left part does shifting and the right part does swapping. Here the agent labeled 3 and 4 exchanged positions, others remain the same position in the matrix.

The complexity. The shifting part of the space is $O(n)$ tall and the swapping part is $O(m)$ wide. Both parts have $O(n)$ obstacles. So the space complexity is $O(n \cdot m)$. We are using a bubble sort algorithm to transform a matrix, and bubble sort requires $O(N^2)$ number of swaps, where N is the total number of agents $(n \cdot m)$. Each swap is followed by a shift operation. To place an agent to its correct position, the algorithm has to do at most N shifts to position the agent to swap positions, and the worst case will do N swaps. So the total

number of shifts is $N \cdot N + N \cdot (N-1)/2$. The total number of operations is $O(N^2)$, which is the time complexity. Thus, we have Theorem 4.1.

Theorem 4.1. *Transforming a matrix A into matrix B can be done by a set of $O(n)$ obstacles in a $O(n \cdot m)$ grid space. The transformation requires at most $O((n \cdot m)^2)$ moves.*

Simply holding $n \cdot m$ agents requires a $n \cdot m$-size grid; our grid design also uses $O(n \cdot m)$ space, achieving a tight bound.

Section 4.4

Sorting agents with fewer actions and less space

Section 4.3 reduced the space requirement for rearranging agent matrix to $O(N)$, however increased the time complexity from constant time (Section 4.2) to $O(N^2)$ since the bubble-sort is the underlying rearranging algorithm. In this section, we will discuss board design methods aimed at reducing the time complexity while keeping the space complexity.

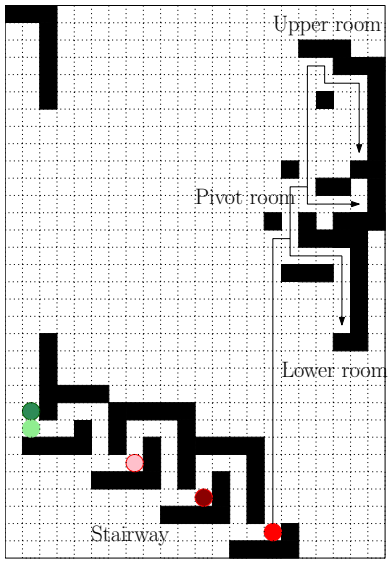
In this section, we will discuss a case where the agents form a one dimensional array. A board design based on quicksort algorithm is proposed and sorts the array in guaranteed $N \cdot \log N$ time. We hope this design can inspire more exploration on board designs for sorting matrices in $N \cdot \log N$ steps using $O(N)$ space.

The board designed in Section 4.3 is based on bubble sort algorithm, which yields a $O(N^2)$ time complexity. Can we improve the performance? The quicksort algorithm can achieve a time complexity of $N \cdot \log N$ in average. Can we design a board capable of performing an algorithm like quicksort?

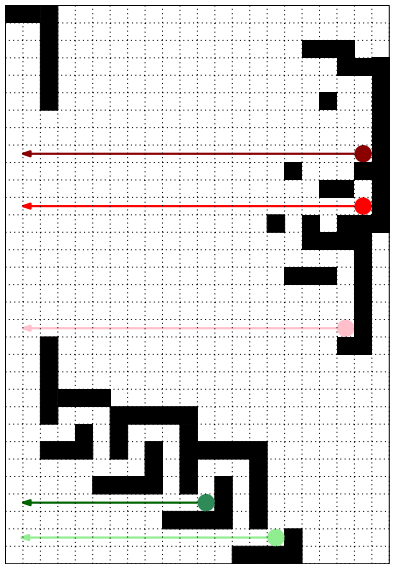
Figure 4.8 is a board design intended to do quicksort for an array of agents. In our design, a space is divided into five parts:

(a) **Entrance**. The left-most part is called the *entrance* where agents are kept in a row.

(b) **Stairway**. The lowest eigtht rows next to the entrance are a *stairway* which ensures only one agent is able to exit and be further rearranged.



(a) The stairway ensures single-agent exit. At the exit of the stair way, an agent can be moved to go to pivot room, lower room or upper room.



(b) After re-arranging a segment, move all agents to the left, then the just rearranged segment will be pushed to the top of the row.

Figure 4.8: A board design to perform quick sort for a row of agents. The board is able to perform pivoting for any sub-sequences of agents and shift them to the top of the row.

(c) **Pivot room**. The middle space in right side is called the *pivot* room, which holds the pivot agent for quicksort.

(d) **Lower room**. To the bottom of the pivot room is the *lower room*, which holds the agents marked smaller than the pivot agent.

(e) **Upper room**. The space above the pivot room is the *upper room*, which is for agents larger than the pivot agent.

The rearrangement operates as follows: A sequence of *down-right-up-right* commands will make an agent leave the stairway. After exiting the stairway, an agent can be controlled to enter the pivot room, if it is the pivot agent, or the lower or upper rooms, if it is smaller or larger than the pivot agent. See Figure 4.8a. When a segment of the array is pivoted, move all *left* to recompose all agents into a single array again. This operation also puts the rearranged segments onto the top of the left-over agents. See Figure 4.8b.

After the partition step of the quicksort algorithm, all agents smaller or larger than the pivot element are placed on different sides of the pivot agent. Repeating the process, we can perform a full quicksort algorithm on the array of agents using this board.

The stairway design is fixed for any length array of agents. Its width and height are both fixed, too. The pivot room, the upper room and the lower room all have fixed width of five. Their heights are at most $N/2$. The board space is thus at most $O(N)$.

Although the quicksort can take $O(N^2)$ time in the worst case based on the pivot element selection, we can select the median element of the partition to pivot around in each partitioning step to avoid the worst-case scenario. From a computational perspective, finding the median is itself an expensive operation, so quicksort usually select a random pivot element from the sub-array to be partitioned. However, in our case, we are not particularly concerned with computational time needed to generate the sequence of actions, instead we care more about the physical number of moves required for the agents. Thus, it is quite reasonable to select the median for pivoting, allowing the modified quicksort to use something like $\Theta(n \log n)$ moves.

An example is shown as follow: Assume we are given an array [8,7,6,5,4,3,2,1]. The first step chooses 4 as pivot and partitions the array into two parts: [3,2,1],[4],[8,7,6,5]. Shift the lower part and pivot to the end, the array becomes [8,7,6,5],[3,2,1,4]. Now partition [8,7,6,5] with pivot 6, yielding: [5],[6],[8,7],[3,2,1,4]. Shift the lower part and pivot, we have: [8,7],[3,2,1,4],[5],[6]. Partition [8,7] with pivot 7, the array becomes: [7],[8],[3,2,1,4],[5],[6]. Shift the lower part and pivot: [3,2,1,4],[5],[6],[7],[8]. Partition [3,2,1,4],[5],[6],[7],[8] with pivot 2: [1],[2],[3,4],[5],[6],[7],[8]. The array is sorted!

4.4.1. Fast rearranging a matrix of agents in a small space

One easy way to extend this quicksort idea to fast rearranging a matrix of agents is to first unfold the matrix to an array, then utilize the quicksort board to sort the array, and finally reassemble the array back to a matrix. In this subsection, we present such a design. However, this design will have a space complexity of larger than $O(N)$.

Ideally, we would like to have a board design of $O(N = n \times m)$ size for a matrix of $n \times m$ agents, and rearrange the matrix in a time complexity of $O(N \cdot \log N)$. We hope this section can inspire some future exploration in that direction.

Figure 4.9 is a board capable of sorting a matrix is relatively small amount of time. On the left side, we issue an *up* and a *right* command to separate the matrix into different columns then merge into one single array. The left part has a width of $min(n,m)$ and a height of $N = n \times m$. Later, the array can also be re-separated and form a matrix using a *left* and a *down* command. The right side of the board does quicksort for the array generated. As shown earlier, the right side of the board has a size of $O(N)$.

The space complexity of the board is $O(n \cdot m \cdot min(n,m))$, which is better than $O(N^2)$ but worse than the ideal target of $O(N)$. The time complexity to use the board for rearranging is still $O(N \cdot \log N)$.

Section 4.5

Experiments

4.5.1. Constant-move grid design

Figure 4.4 shows an example of rearranging a heart image (7×7 size) using a 51×161 grid. In the initial image (Figure 4.4a), agent positions are randomized. After running the first 5 commands, every agent is moved into its target row, as shown in Figure 4.4b. The next 5 commands rearrange all 7 rows simultaneously, and the last command recomposes all rows into a matrix. Figure 4.4c is the goal configuration of all agents, and Figure 4.4d demonstrates the board generated by our algorithm based on the randomized agent positions.

4.5.2. Bubble sort

We ran our algorithm to generate $O(N)$ grid space and bubble-sort matrices of 3×3, 6×6, 9×9, 13×13 (duck image) and 16×16 (Super Mario image). For each matrix, we ran-

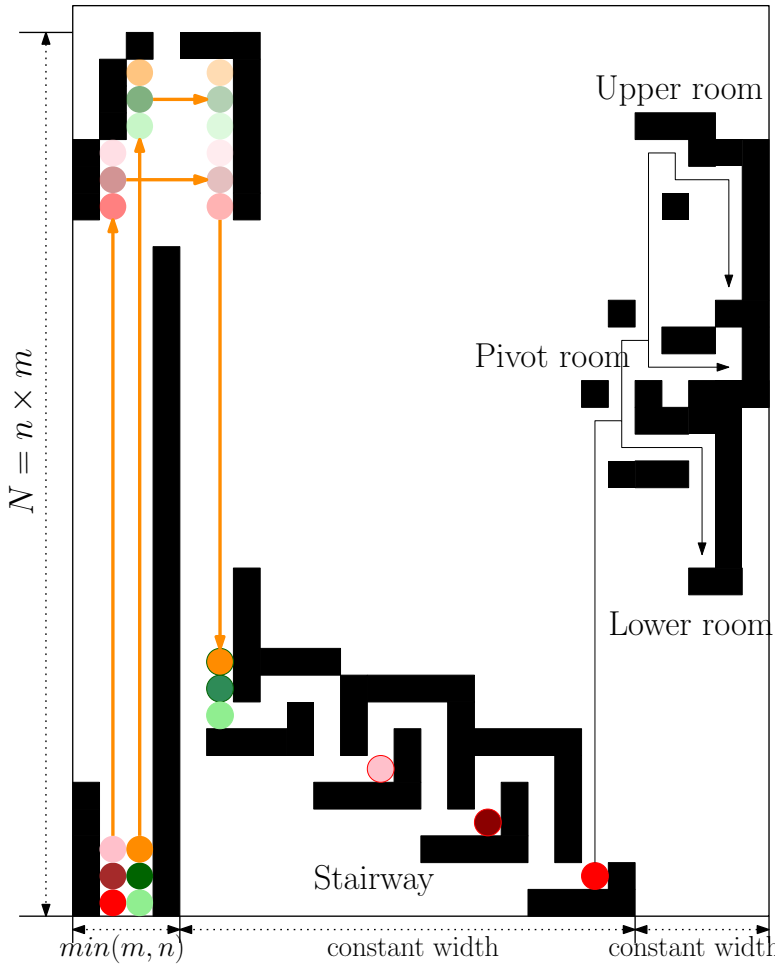


Figure 4.9: The design of a $n \cdot m \cdot min(n, m)$ sized board capable of performing quicksort on a matrix of agents. The left side of the board separates each column and merge them into a single array. The right side performs quicksort for the array.

domize its initial arrangement 10 times and do bubble-sorting for each random arrangement. The following chart (Figure 4.10) and table (Table 4.1) show the average number of shifts, swaps, and moves, and their standard deviations.

Figure 4.11 and 4.1 shows the process of rearranging a duck image (13×13) and a Super Mario image (16×16). They both use the same 100×119 grid space generated for 16×16 agent matrices. When 66% of commands are executed, the last several rows of both images are sorted and we are able to see the shape of the final image.

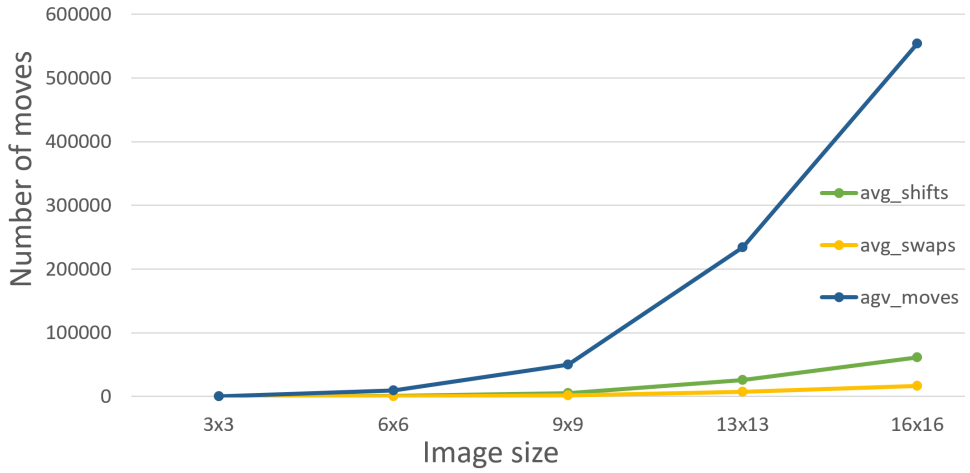
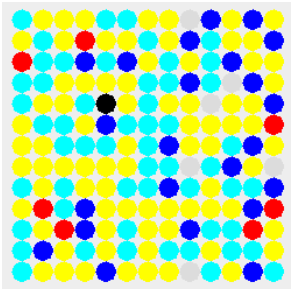


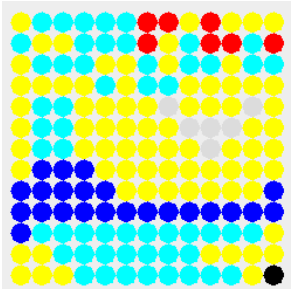
Figure 4.10: Average number of shifts, swaps and moves for rearranging 3×3, 6×6, 9×9, 13×13 (duck image) and 16×16 matrices.

Table 4.1: Bubble sort experiment results with different sized matrices.

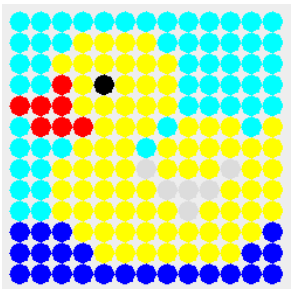
	shifts	stdev	swaps	stdev	moves	stdev
3×3	38.7	15.326	13.7	5.697	382.9	149.674
6×6	1047.6	93.646	316.7	33.869	9768.3	778.590
9×9	5508	402.291	1555.1	62.836	50154.1	2388.549
13×13	25941.5	1403.254	7129.4	482.695	234072.4	11498.934
16×16	61593.6	2023.678	16809.8	635.403	554469.4	14164.014



(a) A randomized arrangement of duck image.



(b) Duck image rearrangement 66% done.



(c) The target rearrangement of duck image.

Figure 4.11: Rearranging a duck image (13×13) in a $O(N)$ size board using bubble sort with limited global controls.

Bibliography

[1] Hugo Akitaya, Greg Aloupis, Maarten Löffler, and Anika Rounds, *Trash compaction*.

[2] Thomas F Allen, Joel W Burdick, and Elon Rimon, *Two-finger caging of polygonal objects using contact space search*, IEEE Transactions on Robotics **31** (2015), no. 5, 1164–1179.

[3] Jürgen Andres, Thomas Bock, Friedrich Gebhart, and Werner Steck, *First results of the development of the masonry robot system rocco: a fault tolerant assembly tool*, Automation and Robotics in Construction XI, Elsevier, 1994, pp. 87–93.

[4] DJ Arbuckle and Aristides AG Requicha, *Self-assembly and self-repair of arbitrary shapes by a swarm of reactive robots: algorithms and simulations*, Autonomous Robots **28** (2010), no. 2, 197–211.

[5] Hadi Ardiny, Stefan Witwicki, and Francesco Mondada, *Construction automation with autonomous mobile robots: A review*, 2015 3rd RSI International Conference on Robotics and Mechatronics (ICROM), IEEE, 2015, pp. 418–424.

[6] Federico Augugliaro, Ammar Mirjan, Fabio Gramazio, Matthias Kohler, and Raffaello D'Andrea, *Building tensile structures with flying machines*, 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2013, pp. 3487–3492.

[7] Frederico Augugliaro, Sergei Lupashin, Michael Hamer, Cason Male, Markus Hehn, Mark W Mueller, Jan Sebastian Willmann, Fabio Gramazio, Matthias Kohler, and Raffaello D'Andrea, *The flight assembled architecture installation: Cooperative construction with flying machines*, IEEE Control Systems **34** (2014), no. 4, 46–64.

[8] Moritz Bächer, Bernd Bickel, Emily Whiting, and Olga Sorkine-Hornung, *Spin-it: optimizing moment of inertia for spinnable objects*, Communications of the ACM **60** (2017), no. 8, 92–99.

[9] Andrew T Baisch and Robert J Wood, *Pop-up assembly of a quadrupedal ambulatory microrobot*, 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2013, pp. 1518–1524.

[10] C Balaguer, E Gambao, A Barrientos, EA Puente, and R Aracil, *Site assembly in construction industry by means of a large range advanced robot*, Proc. 13th Int. Symp. Automat. Robotics in Construction (ISARC'96), 1996, pp. 65–72.

[11] Devin J Balkcom and Jeffrey C Trinkle, *Computing wrench cones for planar rigid body contact tasks*, The International Journal of Robotics Research **21** (2002), no. 12, 1053–1066.

[12] Aaron Becker, Erik D Demaine, Sándor P Fekete, Golnaz Habibi, and James McLurkin, *Reconfiguring massive particle swarms with limited, global control*, International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics, Springer, 2013, pp. 51–66.

[13] Aaron Becker, Erik D Demaine, Sándor P Fekete, and James McLurkin, *Particle computation: Designing worlds to control robot swarms with only global signals*, 2014 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2014, pp. 6751–6756.

[14] Aaron Becker, Golnaz Habibi, Justin Werfel, Michael Rubenstein, and James McLurkin, *Massive uniform manipulation: Controlling large populations of simple robots with a common input signal*, 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2013, pp. 520–527.

[15] Aaron T Becker, Erik D Demaine, Sándor P Fekete, Hamed Mohtasham Shad, and Rose Morris-Wright, *Tilt: The video-designing worlds to control robot swarms with only global signals*, LIPIcs-Leibniz International Proceedings in Informatics, vol. 34, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[16] Aaron T Becker, Sándor P Fekete, Phillip Keldenich, Dominik Krupke, Christian Rieck, Christian Scheffer, and Arne Schmidt, *Tilt assembly: algorithms for micro-factories that build objects with uniform external forces*, Algorithmica (2017), 1–23.

[17] Matthew P Bell, Weifu Wang, Jordan Kunzika, and Devin Balkcom, *Knot-tying with four-piece fixtures*, The International Journal of Robotics Research **33** (2014), no. 11, 1481–1489.

[18] Richard Blakemore, *Magnetotactic bacteria*, Science **190** (1975), no. 4212, 377–379.

[19] Hallel A Bunis, Elon D Rimon, Thomas F Allen, and Joel W Burdick, *Equilateral three-finger caging of polygonal objects using contact space search*, IEEE Transactions on Automation Science and Engineering **15** (2018), no. 3, 919–931.

[20] Gilles Caprari, Patrick Balmer, Ralph Piguet, and Roland Siegwart, *The autonomous micro robot" alice": a platform for scientific and commercial applications*, Micromechatronics and Human Science, 1998. MHS'98. Proceedings of the 1998 International Symposium on, IEEE, 1998, pp. 231–235.

[21] Giovanni Cesaretti, Enrico Dini, Xavier De Kestelier, Valentina Colla, and Laurent Pambaguian, *Building components for an outpost on the lunar soil by means of a novel 3d printing technology*, Acta Astronautica **93** (2014), no. 93, 430–450.

[22] Yufeng Chen, E Farrell Helbling, Nick Gravish, Kevin Ma, and Robert J Wood, *Hybrid aerial and aquatic locomotion in an at-scale robotic insect*, 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2015, pp. 331–338.

[23] Jae-Sook Cheong, Herman J Haverkort, and A Frank van der Stappen, *Computing all immobilizing grasps of a simple polygon with few contacts*, Algorithmica **44** (2006), no. 2, 117–136.

[24] Jae-Sook Cheong, A Frank Van Der Stappen, Ken Goldberg, Mark H Overmars, and Elon Rimon, *Immobilizing hinged polygons*, International Journal of Computational Geometry & Applications **17** (2007), no. 01, 45–69.

[25] Vasek Chvatal, *A greedy heuristic for the set-covering problem*, Mathematics of operations research **4** (1979), no. 3, 233–235.

[26] Robert Connelly, Erik D Demaine, and Günter Rote, *Straightening polygonal arcs and convexifying polygonal cycles*, Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on, IEEE, 2000, pp. 432–442.

[27] Leslie Cousineau and Nobuyasu Miura, *Construction robots: The search for new building technology in japan*, Amer Society of Civil Engineers (February 1, 1998), 1998.

[28] Jurek Czyzowicz, Ivan Stojmenovic, and Jorge Urrutia, *Immobilizing a polytope*, Workshop on Algorithms and Data Structures, Springer, 1991, pp. 214–227.

[29] ———, *Immobilizing a shape*, International Journal of Computational Geometry & Applications **9** (1999), no. 02, 181–206.

[30] Jonathan Daudelin, Gangyuan Jing, Tarik Tosun, Mark Yim, Hadas Kress-Gazit, and Mark Campbell, *An integrated system for perception-driven autonomy with modular robots*, arXiv preprint arXiv:1709.05435 (2017).

[31] Renaud Detry, Carl Henrik Ek, Marianna Madry, and Danica Kragic, *Learning a dictionary of prototypical grasp-predicting parts from grasping experience*, 2013 IEEE International Conference on Robotics and Automation, IEEE, 2013, pp. 601–608.

[32] Arundhati Dhagat and Joseph ORourke, *Motion planning amidst movable square blocks*, Ph.D. thesis, Smith College, Northampton, Mass., 1992.

[33] Tao Du, Adriana Schulz, Bo Zhu, Bernd Bickel, and Wojciech Matusik, *Computational multicopter design*, (2016).

[34] Dumbarton House, the national headquarters of The National Society of The Colonial Dames of America (NSCDA), *The collection move: Plan, plan, and plan some more*, 2019, [Online; accessed April 29th, 2019].

[35] Michael A. Erdmann and Matthew T. Mason, *An exploration of sensorless manipulation*, IEEE Journal of Robotics and Automation **4** (1988), no. 4, 369–379.

[36] Jeff Erickson, Shripad Thite, Fred Rothganger, and Jean Ponce, *Capturing a convex object with three discs*, vol. 23, IEEE, 2007, pp. 1133–1140.

[37] Samuel Felton, Michael Tolley, Erik Demaine, Daniela Rus, and Robert Wood, *A method for building self-folding machines*, Science **345** (2014), no. 6197, 644–646.

[38] For Economic Justice, *The all-american iphone*, 2016, [Online; accessed April 29th, 2019].

[39] Richard B Frankel and Dennis A Bazylinski, *How magnetotactic bacteria make magnetosomes queue up*, Trends in microbiology **14** (2006), no. 8, 329–331.

[40] Chi-Wing Fu, Peng Song, Xiaoqi Yan, Lee Wei Yang, Pradeep Kumar Jayaraman, and Daniel Cohen-Or, *Computational interlocking furniture assembly*, ACM Transactions on Graphics (TOG) **34** (2015), no. 4, 91.

[41] Markus Giftthaler, Timothy Sandy, Kathrin Dörfler, Ian Brooks, Mark Buckingham, Gonzalo Rey, Matthias Kohler, Fabio Gramazio, and Jonas Buchli, *Mobile robotic fabrication at 1: 1 scale: the in situ fabricator*, Construction Robotics **1** (2017), no. 1-4, 3–14.

[42] Otto Glaser, *Temperature and forward movement of paramecium*, The Journal of general physiology **7** (1924), no. 2, 177–188.

[43] Corey Goldfeder and Peter K Allen, *Data-driven grasping*, Autonomous Robots **31** (2011), no. 1, 1–20.

[44] Corey Goldfeder, Matei Ciocarlie, Hao Dang, and Peter K Allen, *The columbia grasp database*, (2008).

[45] Li Han, Jeffrey C Trinkle, and Zexiang X Li, *Grasp analysis as linear matrix inequality problems*, IEEE Transactions on Robotics and Automation **16** (2000), no. 6, 663–674.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Deep residual learning for image recognition*, Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[47] Volker Helm, Selen Ercan, Fabio Gramazio, and Matthias Kohler, *Mobile robotic fabrication on construction sites: Dimrob*, 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2012, pp. 4335–4341.

[48] Philipp Herholz, Wojciech Matusik, and Marc Alexa, *Approximating Free-form Geometry with Height Fields for Manufacturing*, Computer Graphics Forum (Proc. of Eurographics) **34** (2015), no. 2, 239–251.

[49] Alexander Herzog, Peter Pastor, Mrinal Kalakrishnan, Ludovic Righetti, Jeannette Bohg, Tamim Asfour, and Stefan Schaal, *Learning of grasp selection based on shape-templates*, Autonomous Robots **36** (2014), no. 1-2, 51–65.

[50] Robert D Howe and Mark R Cutkosky, *Practical force-motion models for sliding manipulation*, The International Journal of Robotics Research **15** (1996), no. 6, 557–572.

[51] Ruizhen Hu, Honghua Li, Hao Zhang, and Daniel Cohen-Or, *Approximate pyramidal shape decomposition.*, ACM Trans. Graph. **33** (2014), no. 6, 213–1.

[52] Jun Huang, Satyandra K Gupta, and Klaus Stoppel, *Generating sacrificial multi-piece molds using accessibility driven spatial partitioning*, Computer-Aided Design **35** (2003), no. 13, 1147–1160.

[53] Michael Karpelson, Benjamin H Waters, Benjamin Goldberg, Brody Mahoney, Onur Ozcan, Andrew Baisch, Pierre-Marie Meyitang, Joshua R Smith, and Robert J Wood, *A wirelessly powered, biologically inspired ambulatory microrobot*, 2014 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2014, pp. 2384–2391.

[54] Steven J Keating, Julian C Leland, Levi Cai, and Neri Oxman, *Toward site-specific and self-sufficient robotic fabrication on architectural scales*, Science Robotics **2** (2017), no. 5, eaam8986.

[55] Ben Kehoe, Akihiro Matsukawa, Sal Candido, James Kuffner, and Ken Goldberg, *Cloud-based robot grasping with the google object recognition engine*, 2013 IEEE International Conference on Robotics and Automation, IEEE, 2013, pp. 4263–4270.

[56] S Kornienko, R Thenius, O Kornienko, and T Schmickl, *Reembodiment of honeybee aggregation behavior in artificial microrobotic system*, Adaptive Behavior (accepted for publication).

[57] Keith Kotay, Daniela Rus, Marsette Vona, and Craig McGray, *The self-reconfiguring robotic molecule: Design and control algorithms*, Workshop on Algorithmic Foundations of Robotics, Citeseer, 1998, pp. 376–386.

[58] Samuel Lensgraf, Karim Itani, Yinan Zhang, Zezhou Sun, Yijia Wu, Alberto Quattrini Li, Bo Zhu, Emily Whiting, Weifu Wang, and Devin Balkcom, *Puzzleflex: kinematic motion of chains with loose joints*, arXiv preprint arXiv:1906.08708 (2019).

[59] Quentin Lindsey, Daniel Mellinger, and Vijay Kumar, *Construction of cubic structures with quadrotor teams*, Proc. Robotics: Science & Systems VII (2011).

[60] Kevin Y Ma, Pakpong Chirarattananon, Sawyer B Fuller, and Robert J Wood, *Controlled flight of a biologically inspired, insect-scale robot*, Science **340** (2013), no. 6132, 603–607.

[61] Arun V Mahadev, Dominik Krupke, Jan-Marc Reinhardt, Sándor P Fekete, and Aaron T Becker, *Collecting a swarm in a grid environment using shared, global inputs*, 2016 IEEE International Conference on Automation Science and Engineering (CASE), IEEE, 2016, pp. 1231–1236.

[62] Jeffrey Mahler, Jacky Liang, Sherdil Niyaz, Michael Laskey, Richard Doan, Xinyu Liu, Juan Aparicio Ojea, and Ken Goldberg, *Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics*, arXiv preprint arXiv:1703.09312 (2017).

[63] Jeffrey Mahler, Matthew Matl, Xinyu Liu, Albert Li, David Gealy, and Ken Goldberg, *Dex-net 3.0: Computing robust vacuum suction grasp targets in point clouds*

using a new analytic model and deep learning, 2018 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2018, pp. 1–8.

[64] Jeffrey Mahler, Florian T Pokorny, Brian Hou, Melrose Roderick, Michael Laskey, Mathieu Aubry, Kai Kohlhoff, Torsten Kröger, James Kuffner, and Ken Goldberg, *Dex-net 1.0: A cloud-based network of 3d objects for robust grasp planning using a multi-armed bandit model with correlated rewards*, 2016 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2016, pp. 1957–1964.

[65] Satoshi Makita and Yusuke Maeda, *3d multifingered caging: Basic formulation and planning*, 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2008, pp. 2697–2702.

[66] Zeina Malaeb, Fatima AlSakka, and Farook Hamzeh, *3d concrete printing: Machine design, mix proportioning, and mix comparison between different machine setups*, In 3D Concrete Printing Technology (2019), 115–136.

[67] Zeina Malaeb, Hussein Hachem, Adel Tourbah, Toufic Maalouf, Nader El Zarwi, and Farook Hamzeh, *3d concrete printing: Machine and mix design*, International Journal of Civil Engineering & Technology (IJCIET) (2016).

[68] Sheryl Manzoor, Samuel Sheckman, Jarrett Lonsford, Hoyeon Kim, Min Jun Kim, and Aaron T Becker, *Parallel self-assembly of polyominoes under uniform control inputs*, IEEE Robotics and Automation Letters **2** (2017), no. 4, 2040–2047.

[69] Matthew T Mason, *Mechanics of robotic manipulation*, MIT press, 2001.

[70] Bhubaneswar Mishra, Jacob T Schwartz, and Micha Sharir, *On the existence and synthesis of multifinger positive grips*, Algorithmica **2** (1987), no. 1-4, 541–558.

[71] Tuan C. Nguyen, *Yes, that 3d-printed mansion is safe to live in*, 5 February, 2015.

[72] Van-Duc Nguyen, *The synthesis of force closure grasps in the plane.*, Tech. report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[73] Toru Omata and Kazuyuki Nagata, *Rigid body analysis of the indeterminate grasp force in power grasps*, IEEE Transactions on Robotics and Automation **16** (2000), no. 1, 46–54.

[74] Cagdas D Onal, Robert J Wood, and Daniela Rus, *An origami-inspired approach to worm robots*, IEEE/ASME Transactions on Mechatronics **18** (2013), no. 2, 430–438.

[75] Kirstin Hagelskjaer Petersen, Radhika Nagpal, and Justin K Werfel, *Termes: An autonomous robotic system for three-dimensional collective construction*, Robotics: Science and Systems 2011, vol. 7, 2011.

[76] Domenico Prattichizzo and Jeffrey C Trinkle, *Grasping*, Springer handbook of robotics (2008), 671–700.

[77] G. Pritschow, M. Dalacker, J. Kurz, and M. Gaenssle, *Technological aspects in the development of a mobile bricklaying robot*, Automation in Construction **5** (1996), no. 1, 3–13.

[78] Alok K Priyadarshi and Satyandra K Gupta, *Geometric algorithms for automated design of multi-piece permanent molds*, Computer-Aided Design **36** (2004), no. 3, 241–260.

[79] R. Ravi and M. N. Srinivasan, *Decision criteria for computer-aided parting surface design*, Comput. Aided Des. **22** (1990), no. 1, 11–17.

[80] Franz Reuleaux, *Theoretische kinematik: Grundzüge einer theorie des maschinenwesens*, vol. 1, F. Vieweg und Sohn, 1875.

[81] Elon Rimon and Andrew Blake, *Caging 2d bodies by 1-parameter two-fingered gripping systems*, 1996 IEEE International Conference on Robotics and Automation., vol. 2, IEEE, 1996, pp. 1458–1464.

[82] Elon Rimon and Joel W Burdick, *Mobility of bodies in contact. i. a 2nd-order mobility index for multiple-finger grasps*, IEEE transactions on Robotics and Automation **14** (1998), no. 5, 696–708.

[83] ———, *Mobility of bodies in contact. ii. how forces are generated by curvature effects*, IEEE Transactions on Robotics and Automation **14** (1998), no. 5, 709–717.

[84] John W Romanishin, Kyle Gilpin, and Daniela Rus, *M-blocks: Momentum-driven, magnetic modular robots*, 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2013, pp. 4288–4295.

[85] Michael Rubenstein, Christian Ahler, and Radhika Nagpal, *Kilobot: A low cost scalable robot system for collective behaviors*, 2012 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2012, pp. 3293–3298.

[86] Michael Rubenstein, Adrian Cabrera, Justin Werfel, Golnaz Habibi, James McLurkin, and Radhika Nagpal, *Collective transport of complex objects by simple robots: theory and experiments*, Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems, International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 47–54.

[87] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal, *Programmable self-assembly in a thousand-robot swarm*, Science **345** (2014), no. 6198, 795–799.

[88] Daniela Rus and Marsette Vona, *Crystalline robots: Self-reconfiguration with compressible unit modules*, Autonomous Robots **10** (2001), no. 1, 107–124.

[89] Arne Schmidt, Sheryl Manzoor, Li Huang, Aaron T Becker, and Sándor P Fekete, *Efficient parallel self-assembly under uniform control inputs*, IEEE Robotics and Automation Letters **3** (2018), no. 4, 3521–3528.

[90] Eric Schweikardt and Mark D Gross, *roblocks: a robotic construction kit for mathematics and science education*, Proceedings of the 8th international conference on Multimodal interfaces, ACM, 2006, pp. 72–75.

[91] Hamed Mohtasham Shad, Rose Morris-Wright, Erik D Demaine, Sándor P Fekete, and Aaron T Becker, *Particle computation: Device fan-out and binary memory*, 2015 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2015, pp. 5384–5389.

[92] V. Shekar, M. Campbell, and S. Akella, *Towards automated optoelectrowetting on dielectric devices for multi-axis droplet manipulation*, 2012 IEEE International Conference on Robotics and Automation (ICRA) (Karlsruhe, Germany), May 2013, pp. 1431–1437.

[93] Jack Snoeyink and Jorge Stolfi, *Objects that cannot be taken apart with two hands*, Proceedings of the ninth annual symposium on Computational geometry, ACM, 1993, pp. 247–256.

[94] Peng Song, Bailin Deng, Ziqi Wang, Zhichao Dong, Wei Li, Chi-Wing Fu, and Ligang Liu, *Cofifab: Coarse-to-fine fabrication of large 3d objects*, ACM Transactions on Graphics.

[95] Peng Song, Chi-Wing Fu, and Daniel Cohen-Or, *Recursive interlocking puzzles*, ACM Transactions on Graphics (TOG) **31** (2012), no. 6, 128.

[96] Peng Song, Chi-Wing Fu, Yueming Jin, Hongfei Xu, Ligang Liu, Pheng-Ann Heng, and Daniel Cohen-Or, *Reconfigurable interlocking furniture*, ACM Transactions on Graphics (TOG) **36** (2017), no. 6, 174.

[97] Peng Song, Zhongqi Fu, Ligang Liu, and Chi-Wing Fu, *Printing 3d objects with interlocking parts*, Computer Aided Geometric design (Proc. of GMP 2015) **35-36** (2015), 137–148.

[98] Jacob Steiner, *Einige gesetze über die theilung der ebene und des raumes.*, Journal für die reine und angewandte Mathematik **1** (1826), 349–364.

[99] Ileana Streinu, *A combinatorial approach to planar non-colliding robot arm motion planning*, Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on, IEEE, 2000, pp. 443–453.

[100] Cynthia Sung and Daniela Rus, *Foldable joints for foldable robots*, Journal of Mechanisms and Robotics **7** (2015), no. 2, 021012.

[101] Yuzuru Terada and Satoshi Murata, *Automatic assembly system for a large-scale modular structure - hardware design of module and assembler robot*, 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566), vol. 3, 2004, pp. 2349–2355.

[102] ———, *Automatic modular assembly system and its distributed control*, The International Journal of Robotics Research **27** (2008), no. 3, 445–462.

[103] The Nature Education, *Bacteria that synthesize nano-sized compasses to navigate using earth's geomagnetic field*, 2010, [Online; accessed April 29th, 2019].

[104] Tarik Tosun, Gangyuan Jing, Hadas Kress-Gazit, and Mark Yim, *Computer-aided compositional design and verification for modular robots*, Robotics Research, Springer, 2018, pp. 237–252.

[105] Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman, *Handbook of discrete and computational geometry*, CRC press, 2004.

[106] Jeffrey Coates Trinkle, *The mechanics and planning of enveloping grasps*, (1987).

[107] Mostafa Vahedi and A Frank van der Stappen, *Caging polygons with two and three fingers*, The International Journal of Robotics Research **27** (2008), no. 11-12, 1308–1324.

[108] Lingfeng Wang and Emily Whiting, *Buoyancy optimization for computational fabrication*, Computer Graphics Forum, vol. 35, Wiley Online Library, 2016, pp. 49–58.

[109] Weifu Wang and Devin Balkcom, *Grasping and folding knots*, 2016 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2016, pp. 3647–3654.

[110] Ziqi Wang, Peng Song, and Mark Pauly, *DESIA: A general framework for designing interlocking assemblies*, ACM Transactions on Graphics (SIGGRAPH Asia) **37** (2018), no. 6, Article No. 191.

[111] Hongxing Wei, Youdong Chen, Jindong Tan, and Tianmiao Wang, *Sambot: A self-assembly modular robot system*, IEEE/ASME Transactions on Mechatronics **16** (2011), no. 4, 745–757.

[112] Justin Werfel and Radhika Nagpal, *Three-dimensional construction with mobile robots and modular blocks*, The International Journal of Robotics Research **27** (2008), no. 3, 463–479.

[113] Justin Werfel, Kirstin Petersen, and Radhika Nagpal, *Designing collective behavior in a termite-inspired robot construction team.*, Science **343** (2014), no. 6172, 754–758.

[114] Paul White, Viktor Zykov, Josh C Bongard, and Hod Lipson, *Three dimensional stochastic reconfiguration of modular robots.*, Robotics: Science and Systems, Cambridge, 2005, pp. 161–168.

[115] Jan Willmann, Federico Augugliaro, Thomas Cadalbert, Raffaello D'Andrea, Fabio Gramazio, and Matthias Kohler, *Aerial robotic construction towards a new field of architectural research*, International journal of architectural computing **10** (2012), no. 3, 439–459.

[116] JP Wilson, DC Woodruff, JD Gardner, HM Flora, JR Horner, and CL Organ, *Vertebral adaptations to large body size in theropod dinosaurs*, PLoS ONE **11** (2016), no. 7.

[117] Stefan Wismer, Gregory Hitz, Michael Bonani, Alexey Gribovskiy, and Stphane Magnenat, *Autonomous construction of a roofed structure: Synthesizing planning and stigmergy on a mobile robot*, 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2012, pp. 5436–5437.

[118] Xinhua News, *Chinese ru-ware bow sets $38m auction record in hong kong*, 2019, [Online; accessed April 29th, 2019].

[119] Jiaxian Yao, Danny M. Kaufman, Yotam Gingold, and Maneesh Agrawala, *Interactive design and stability analysis of decorative joinery for furniture*, ACM Trans. Graph. **36** (2017), no. 2, 20:1–20:16.

[120] Yinan Zhang, *Block regrasping using two ur-10 robot arms. (1080p)*, `https://youtu.be/DyMBOK-Pz1I`.

[121] ———, *Computational interlocking blocks design and assembly for a voxel model.*, `https://youtu.be/qQQIe0ScLQg`.

[122] ———, *Interlocking cube assembly using two ur-10 robots (4k 60fps)*, `https://youtu.be/IjW0oxDqrYY`.

[123] Yinan Zhang and Devin Balkcom, *Interlocking structure assembly with voxels*, 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2016, pp. 2173–2180.

[124] Yinan Zhang, Xiaolei Chen, Hang Qi, and Devin Balkcom, *Rearranging agents in a small space using global controls*, 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2017, pp. 3576–3582.

[125] K. Zwerger and V. Olgiati, *Wood and wood joints: Building traditions of europe, japan and china*, Birkhäuser, 2012.