In 1964, when entering medical school, I found myself with a dream whose origins I do not know. I was new to biology then, as unfamiliar as many of you may be with its intertwined marvels of historical contingency, selection, design, drift, accident, and sheer wonder. I think as a young scientist I could not yet begin to fathom the power of natural selection, whose subtlety has grown more impressive to me over the intervening three decades. Yet the dream that welled up from whatever unknowable sources is one I still hold. If biologists have ignored self-organization, it is not because self-ordering is not pervasive and profound. It is because we biologists have yet to understand how to think about systems governed simultaneously by two sources of order. Yet who seeing the snowflake, who seeing simple lipid molecules cast adrift in water forming themselves into cell-like hollow lipid vesicles, who seeing the potential for the crystallization of life in swarms of reacting molecules, who seeing the stunning order for free in networks linking tens upon tens of thousands of variables, can fail to entertain a central thought: if ever we are to attain a final theory in biology, we will surely, surely have to understand the commingling of self-organization and selection. We will have to see that we are the natural expressions of a deeper order. Ultimately, we will discover in our creation myth that we are expected after all.

Stuart Kauffman, *At Home in the Universe*, 1995

University of Alberta

Algorithmic Distributed Assembly

by

Jonathan Scott Kelly ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2008

Your file Votre référence
ISBN: 978-0-494-45831-0
Our file Notre référence
ISBN: 978-0-494-45831-0

Abstract

This thesis describes a model for planar distributed assembly, in which unit-square *assembly components* move randomly and independently on a two-dimensional grid, binding together to form a desired *target* structure. The components are simple reactive agents, with limited capabilities including short-range sensing and rule-based control only, and operate in an entirely decentralized manner.

Using the model, we investigate two primary issues, *coordination* and *sensing*, from an algorithmic perspective. Our goal is to determine how a group of components can be reliably programmed to produce a *global* result (structure) from purely *local* interactions. Towards this end, we define the local spatiotemporal ordering constraints that must be satisfied for assembly to progress in a coordinated manner, and give a procedure for encoding these constraints in a rule set. When executed by the components, this rule set is guaranteed to produce the target structure, despite the random actions of group members. We then introduce an optimization algorithm which is able to significantly reduce the number of distinct environmental states that components must recognize in order to assemble into a structure. Experiments show that our optimization algorithm outperforms existing approaches.

For Frances, Barbara, Helen and John.

Acknowledgements

I would like to thank Dr. Hong Zhang for introducing me to the problem which led to the research described herein, and for supervising my work. His guidance and assistance were invaluable as I charted a course through graduate studies at the University of Alberta. I began my doctoral program in California before finishing my Master's degree, and I am particularly indebted to Dr. Zhang for his patience over the years that it took to complete this manuscript.

I would like to thank the members of my committee for their careful reading of the thesis, and to specially acknowledge Dr. C. Ronald Kube for providing much helpful advice and for reviewing and commenting on my early work in this area. I would also like to thank Dr. David Kempe at the University of Southern California for teaching an outstanding course on algorithm design and analysis.

I was fortunate, during my time at the U of A, to become involved with Team Canuck, the Computer Science department's small-size league entry in the worldwide RoboCup robot soccer competition. I participated in the 2002 contest in Fukuoka, Japan, and I am grateful to my teammates Axel von Bertoldi, Doug Kondor, Matt McNaughton, Chris Parker, Sean Verret, and Andrzej Zadorozny for making this a wonderful and memorable experience – may your sake cups always be full my friends.

Most importantly, I would like to thank my mother, Barbara Kelly, and my grandparents, John and Helen Aitken, for their support throughout my graduate career. Your love and encouragement mean the world to me.

This work was funded in part by grants from the National Sciences and Engineering Research Council of Canada and the Alberta Informatics Circle of Research Excellence.

Contents

List of Algorithms

List of Tables

List of Figures

Chapter 1

Introduction

Distributed assembly is the process by which a group of agents interact to assemble a coherent structure or pattern from individual components. An *agent*, in this context, is an autonomous entity that is *a*) capable of sensing its environment, and *b*) able to act on the sensory information it receives. These definitions are purposely broad, encompassing both *self-assembling* and *collective* systems.[1]

We are motivated to study this distributed approach because it offers several advantages over traditional, sequential assembly methods. Specifically, distributed systems are able to exploit parallelism to improve yield or throughput, and can also be made inherently redundant and failure tolerant. As such, distributed assembly is expected to become the dominant fabrication technique for nanoscale structures, which cannot be built in a top-down, linear manner [1]. Researchers have also suggested that self-assembly may be harnessed to build meso-scale and macro-scale objects [2]. Indeed, Nature very successfully utilizes distributed assembly across a range of spatial scales, from the manufacture of intricate biostructures inside living cells to the construction of elaborate nests by many social insect species. Although ubiquitous in the natural world, we have at present only a basic understanding of how many of these systems operate.

If we wish to employ distributed assembly for our own purposes, we must immediately deal with two fundamental and complementary issues: maintaining coordination of the assembly process, and ensuring that a desired structure is produced accurately and reliably. Some form of coordination is required so that the agents, operating in parallel, do not produce a disorganized or random result. The coordination problem is made more difficult because systems containing hundreds or thousands of agents are frequently *locally-interacting* only – the cost of communication, in terms of power, time, or complexity, between distant individuals is often too high to be practical. A challenge, then, is to solve the *local-to-global problem*: given a system of many components, interacting *locally*, how does one 'program' the system to produce a specific *global* result? Further, the accuracy and reliability of the

[1]We will have more to say about the distinction between these types of systems in Chapter 2.

assembly process depends on an agent's ability to correctly recognize and discriminate between members of a set of environmental features or states – for agents with limited sensing capabilities, discrimination becomes more difficult as the number of features increases. Is there a way to reduce the number of distinct features that agents are required to recognize?

In this thesis, we examine distributed assembly from an *algorithmic* perspective. Our goals are twofold. We wish to determine how a group of homogeneous agents with limited capabilities, including local sensing and rule-based reactive control only, can be programmed to assemble into complex two-dimensional structures without centralized coordination. We also seek to reduce the sensing demands placed on the agents, by minimizing the number of environmental features that individuals must recognize in order to complete an assembly task.

To address the above, we propose a model for distributed assembly in which unit-square *assembly components* (our agents) use local rules to self-assemble into planar structures. An assembly rule is defined by a local configuration of assembly components and an identifier (feature or state) associated with each component; we use the generic term *label* for such an identifier. We show how to generate a set of rules that, when executed by the agents, deterministically produces a pre-specified *target* structure. We then introduce a randomized optimization algorithm which attempts to minimize the number of unique labels appearing in the rules. The algorithm operates by iteratively refining a worst-case solution, verifying at each step that the modified rule set preserves all of the constraints necessary for successful assembly.

Ultimately, we hope to develop principles and techniques that will allow us to build complex structures from a large number of physically simple components, in an entirely distributed and yet precise and reliable way. Our work here is a step towards this goal.

1.1 Contributions

This thesis makes the following contributions:

- It presents a discrete model for rule-based planar distributed assembly, requiring very simple reactive agents (assembly components) only. We make no assumptions about the capabilities of the assembly components beyond their ability to sense their immediate *local* environment and to store and execute a set of rules. The components move randomly, do not communicate directly with each other, and maintain no history of past actions or observations.

- It formally defines the *global* spatiotemporal ordering constraints that must be satisfied for assembly to progress in a coordinated manner. We develop an algorithm for expressing these constraints as a directed acyclic graph, and show that the algorithm can correctly describe constraints for complex structures, including those with interior holes.

- It gives an algorithm for encoding the graph of spatiotemporal constraints in a set of local assembly rules, such that the production of a desired structure is guaranteed, despite the random actions of individual assembly components. That is, we show how to produce a *deterministic* and preprogrammed result in a system where interactions occur randomly. By doing so, we solve an instance of the local-to-global problem.
- It introduces the *Minimum Label Set* (MLS) problem, a combinatorial optimization problem which involves finding the minimum number of unique labels necessary (in a rule set) for components to exactly self-assemble into a specific structure. We describe an iterative, randomized algorithm which provides quantitatively 'good' solutions for the MLS problem in a reasonable amount of time.

The thesis deals with planar structures only, however the assembly model and the algorithms described herein can be readily extended to three dimensions. We briefly discuss these extensions in Chapter 7.

1.2 Organization

The thesis is organized as follows. We begin in Chapter 2 by discussing related work from a variety of fields, including biology, entomology, computer science and robotics. Chapter 3 describes our planar assembly model. In Chapter 4, we introduce a graph-theoretic formalism for expressing sets of spatiotemporal ordering constraints that, when satisfied, ensure the coordinated assembly of a structure. Chapter 5 gives a procedure for encoding these constraints in a set of local assembly rules, and develops our randomized label reduction (optimization) algorithm. We present a series of experiments in Chapter 6 which characterize the performance of the optimization algorithm, for a variety of structures. Finally, we offer some conclusions and directions for future research in Chapter 7.

Throughout the remaining chapters we introduce formal definitions as necessary. Basic definitions are often given directly in the section text, while more complicated and important terms are defined separately.

Chapter 2

Related Research

In this chapter, we explore relevant work in the broad areas of *self-assembly* and *collective assembly*. For the purpose of this thesis, we will define a self-assembling system as one in which the components involved *become part of the final structure*. A collective system, in contrast, will rely on autonomous agents to position and attach inert building materials.

Research in self-assembly and collective assembly spans the diverse fields of molecular biology, entomology, computer science and robotics, among others. Our review below highlights theoretical contributions and initial steps towards developing controlled (programmable), artificial implementations of self-assembling and collective systems. We focus primarily on the *modelling* of these systems here, as a first step to understanding their operation.

2.1 Self-Assembly

Self-assembly has traditionally been studied as a chemical process, where interactions between components (individual molecules) are governed by inter- and intramolecular forces. It is well known that groups of certain molecules will, in reaching their lowest-energy configuration, spontaneously form into symmetric, ordered aggregates. Our interest, however, is in complex and potentially asymmetric self-assembled structures, built from more sophisticated components. We begin by surveying various *passive* models for asymmetric self-assembly, and then consider several *active* (robotic) approaches.

2.1.1 Theoretical Contributions

There have been significant recent efforts toward developing an *algorithmic* theory of self-assembly, in which the process is viewed as a form of computation. This approach permits analysis using tools from algorithmic complexity theory. The input to a self-assembly 'program' is a set of individual atomic (irreducible) components or parts, and the output is a final, aggregate structure. Programs may be analyzed in terms of both their size complexity, i.e. the cardinality of the input set and/or number of assembly rules, and their time complexity, i.e. average or asymptotic time required to assemble the complete structure.

The formal study of algorithmic self-assembly began with the *Domino Tiling Problem* described by Wang in the early 1960s [3]. A *domino tile* or *Wang tile* is a square planar tile with a colour assigned to each edge. The fundamental problem Wang posed was to determine if a particular set of tiles could fill the infinite plane without leaving any gaps. Tiles may be placed according to the following two rules: attachments must be made along edges of like colour, and tiles may not be rotated or flipped. The Domino Tiling Problem has been shown to be equivalent to the Turing machine halting problem, and is therefore undecidable.

Wang's early work influenced Adleman's model [4] for the one-dimensional self-assembly of linear tile chains (a process called *tile polymerization*). In this model, the edges of a tile are assigned *glue* values (from a countably infinite set of glues), analogous to the coloured edges of a Wang tile. Tiles will bind along abutting edges if the glues are compatible; rotation of the tiles is not allowed. Adleman provides a metric for the time complexity of linear polymerization based on 'step-counting', where a step involves the binding of a tile with any adjacent tiles sharing compatible glues. The results in [4] include a proof of the asymptotic bound on the number of steps required for a certain proportion of the tiles to become incorporated into a single polymer.

Rothemund and Winfree [5] describes a two-dimensional *Tile Assembly Model*, in which unit-square tiles move randomly on the plane and join together to form larger compound structures. As in [4], each tile edge is assigned a glue value; a *tile type* is a unique assignment of glues to edges. Unlike [4], tiles will bind along abutting edges only if the total strength (sum) of their pairwise edge interactions, defined by a *glue strength function*, is larger than a fixed temperature parameter τ. The τ value models the thermal energy of the system, with a higher temperature making binding less likely. A structure's *program size complexity* is defined as the number of tile types required to uniquely assemble the structure. Adleman et al. [6] generalizes the model in [5] and gives an assembly algorithm for $n \times n$ squares that is optimal in terms of asymptotic assembly time and program size (tile types). The problem of determining whether a certain number of tile types is the minimum required to assemble an arbitrary planar structure is shown to be NP-complete in [7].

Saitou [8] introduces an alternative model for one-dimensional self-assembly, derived from automata theory. Here, assembly is guided by *conformational switching*, the (physical) process whereby two components, when brought into close proximity, bind to one another and change their overall conformation (or shape) as a result. Conformation changes then promote or inhibit binding with other components or assemblages.[1] A self-assembling

[1]Many biological reactions are mediated by conformation changes, caused by e.g. ligand docking etc.

automaton is a simplified rule-based machine (in the Turing machine sense) that operates on one-dimensional strings of components or parts. The parts are initially divided into a series of hypothetical 'slots' in a component bin; an assembly operation involves picking two parts from random slots, applying an assembly rule (if a suitable rule exists), and returning the resulting assemblage to another random slot. One or more of the parts in the assemblage may undergo a conformation change as a result of the operation. For the one-dimensional case, [8] shows that three conformations for every part are sufficient to exactly assemble any linear string of parts.

Klavins et al. [9] presents a model similar to [8] for self-assembly and distributed robotic assembly based on graph grammars. Here, graph vertices represent parts, and an edge between two vertices indicates that the corresponding parts are attached. Assembly rules are specified by a grammar consisting of a set of graph pairs, where each pair describes a local replacement rule: if a conformation of parts matching the first graph in a pair exists, the conformation can be replaced with a new conformation defined by the second graph. This procedure defines an assembly action. A particular grammar can be programmed into the system, by proper selection of the parts, to synthesize a desired structure. One difficulty, however, is that, although graphs naturally express topological relationships, they do not readily encode geometric information. Ghrist and Lipsky [10] successfully extends graph grammars to tile assembly systems, and gives examples of grammars that generate only planar outputs, but does not address the problem of designing grammars for arbitrary planar structures.

2.1.2 Models for Self-Assembling Robotic Systems

The models described above are *passive*, in the sense that assembly events occur as *a natural result of the design of the components themselves.* That is, the components cannot choose whether or not to participate in an assembly activity – all possible interactions are a priori specified by the components' intrinsic characteristics. We now discuss a series of models for *active* systems, in which the components (agents) have some level of basic intelligence and are able to sense, deliberate, and act – making them, effectively, simple robots. Our model, presented in Chapter 3, is an example of this type.

Guo et al. [12] suggests a model for active self-assembly involving cubic blocks, where each block face is assigned a *polarity* from the set {*like, unalike, neutral*}. Like polarities repel, while unlike polarities attract; neutral polarities neither attract nor repel. Every block contains an internal state machine that is able to change the polarity of a face in response to a 'sticking' or 'unsticking' event. Simulation results in [12] demonstrate that a genetic algorithm can evolve sets of state machine rules that allow stable shapes to form. A shape is stable if it stops growing at some point because no further sticking events can occur. The largest stable structure given in [12] consists of only 13 blocks, however.

Closely related to our own work is the *Intelligent Self Assembly* (ISA) model proposed by Jones and Matarić [13]. In ISA, autonomous unit-square Assembly Agents (AAs) move randomly on a two-dimensional grid and self-assemble into planar structures under local, rule-based control. Each AA maintains an internal *state*, and is able to detect the states of agents in adjacent cells. An AA will bind to its neighbours when it discovers a state pattern that matches one of a series of internally-stored assembly rules. The AA then transitions to a new state specified by the activated rule. Rules are generated offline by a Transition Rule Set (TRS) compiler, which takes the goal structure as an input and produces a set of assembly rules as an output. A rule set which builds the goal structure only is termed *consistent.* The TRS compiler operates by iteratively assigning larger state values to AAs in the complete structure until a consistent rule set is built. As described, the compiler is able to generate rules for a subset of planar structures without complex interior regions (e.g. certain types of 'holes' with non-convex boundaries). Also, the compiler does not attempt to minimize the number of states appearing in the rule set, and does not exploit symmetry or self-similarity within the goal structure to reduce the number of states used.

Li and Zhang [14] addresses the problem of state reduction in [13] by partitioning the goal structure into rectangular regions and generating rules independently for each region. Individual rectangles are encoded using the minimum number of states necessary. The rectangles are then 'stitched' together by introducing additional states to handle assembly along the boundaries. This partitioning approach is effective for structures composed of large, contiguous regions, however the algorithm does not take advantage of symmetry and does not provide improvement for degenerate rectangles (those with a length or width of one unit).

Arbuckle and Requicha [15] proposes a *communicative* model for self-assembly, in which agents are able to send messages to connected neighbours. Communication enables a global ordering to be imposed on the entire assembly process, and also for the assembly of sacrificial scaffolds and of partially-specified structures. In the case of partially-specified structures, assembly is adaptive – a structure may be defined in terms of constraints (e.g. "there must be a connected bridge from point A to point B"), and message-passing allows the components to self-assemble around unknown obstacles. The drawback of communication is that it requires significant resources (for example, power) from each assembly agent.

The work in [15] is extended to shape restoration in [16]. Given a desired, solid planar shape and an existing, partially-complete sub-shape, [15] describes an algorithm for programming assembly agents to restore the complete shape from the unknown sub-shape. The algorithm begins by assembling an outer perimeter for the complete shape. Agents on the perimeter then shift inwards until no further space is available in the interior region. A similar extension to self-repairing structures is examined in [17], where agents send messages to connected neighbours at regular intervals. If no message is received from a neighbouring agent within a specified time period, the receiving agent assumes that some type of failure has occurred, and disconnects from the neighbour. This allows another, operating agent to move into position, repairing the structure. It also implies that continuous message exchange is required to maintain structural cohesion, which may be an unrealistic requirement for many physical systems.

Other approaches more closely parallel biological processes, such as cell morphogenesis. Kondacs [18] describes a method for programming a group of synthetic 'cells' to replicate and grow into an arbitrary two-dimensional shape. There is no centralized control in the system – each agent relies solely on an identical internal program that is pre-compiled from a covering-disc decomposition of the input shape. The assembly process is mediated by long-range gradient signaling, which allows cells to triangulate their current positions in the overall structure, and by short-range messaging between adjacent cells. A spanning tree formed from the covering disc set is used to generate the individual programs that determine when cells should send specific long- and short-range messages. The system has an attractive self-containment property: because cells determine their positions via triangulation, they do not require any external, global orientation information to self-organize.

2.1.3 Physical Implementations

Theoretical models can provide significant insight into the design of self-assembling systems. However, as abstractions, these models often do not fully describe the complexities of physical assembly processes. For example, high-level models usually do not consider assembly errors that may result from contamination, misalignment of individual components, environmental noise or other factors. We therefore review several attempts to develop artificial self-assembling systems in this section.

Rothemund [19] demonstrates that a simple tile assembly system can be physically realized. In this implementation, binding between small, square plastic tiles is mediated by lateral capillary forces, which are attractive or repulsive depending on the hydrophilic or hydrophobic wetting characteristics of the individual tile edges. When placed in a liquid consisting of two immiscible layers (one hydrophilic, the other hydrophobic), the tiles naturally deform the interface between the layers, resulting in capillary forces that cause tiles with opposite wetting characteristics to move towards each other and bond together. A critical finding in the work is that the model or simulation used to predict the outcome of these experiments cannot be over-simplified relative to the physics of the system. For example, a model which assumes that tile edge characteristics may be represented by digital values $\{0, 1\}$ fails to capture several important physical effects, such as bonding variations that can cause tiles to tilt and inhibit aggregation.

Reif et al. [20] reports on self-assembly using deoxyribonucleic acid (DNA) tiles with 'sticky ends'. Each DNA tile is formed from multiple single-strand DNA anti-parallel crossovers. At the end of every strand is an unbound base pair sequence (the sticky end); this sequence will naturally bind to a complementary sequence at the corner of another tile. By designing the selective affinity of the ends, a series of individual tiles can be made to reliably form into a larger tiling lattice. The tiles may be homogeneous, producing a regular periodic structure, or heterogeneous for application-specific designs. The primary challenge for DNA tile systems is error control – error rates from 0.5% to 5% are common.

Burden et al. [21, 22] describes a real-world testbed for the grammatical self-assembly model defined in [9]. Triangular programmable parts, which are small, self-contained robots, float on an air table and interact through random collisions. A part is equipped with an electromagnetic latch and an infrared transceiver along each of its edges, allowing it to reversibly bind and communicate with neighbouring parts. The parts are 'stirred' (to simulate thermal agitation) by several fans located around the border of the table. Initially, the edges of a part are assigned a triple of labels; when two parts come into contact, they compare their label states and decide collectively whether to remain bound together or to separate. The decision is based on a common graph grammar stored internally by each part. Interactions between the parts can be modelled using reaction-diffusion dynamics from chemical kinetics theory, with an appropriate kinetic rate constant.

2.2 Collective Assembly

In collective assembly[2], a number of agents work in parallel to assemble a structure from inert building materials. The size of the group may vary from just a few to hundreds or thousands of individuals. We are concerned with minimalist systems in which the agents are relatively simple and operate without centralized control. In these cases, the agents usually do not possess explicit, internal representations of the world or of their actions within it – instead, as described by Brooks, they "use the world as its own model"[3] [23].

Much of the research on collective robotics has been inspired by entomological studies of social insect species (bees, wasps etc.). Although individuals in these societies possess limited capabilities and small behavioural repertoires, reinforcing interactions will often produce coherent group behaviour. Pioneering work by Kube and Zhang [24, 25] demonstrated that a group of insect-like, behaviour-based robots can collectively accomplish certain tasks, such as box pushing, which would be impossible for a single robot. Coordination and cooperation emerge from simple, local rules that regulate the interactions between group members; the

[2]Collective assembly is also called *collective construction* in some literature.

[3]A different, and more famous, quotation from Rodney Brooks is that "the world is its own best model."

robots do not need to communicate directly with each other. Kube emphasizes in [24] that designers of collective tasks must explicitly consider an interaction loop involving both the system (i.e. the robots) and the environment.

Assembly tasks, in particular, require some method for coordinating the actions of the agents involved. Without coordination, misassembly of one portion of a structure may impede, or worse, prevent, successful assembly of the structure as a whole. One possible coordination strategy is biologically-inspired, hormone-based control [26, 27], in which different types of digital 'hormone' messages diffuse throughout a colony according to specific propagation and decay rules. Hormone-based approaches typically require the transmission of electromagnetic or chemical signals, however, and are therefore energy-intensive. Below, we focus on *stigmergy*, an alternative, powerful coordination mechanism used by many social insects.

2.2.1 Stigmergy

Our work on distributed assembly [28] was originally motivated by a desire to apply models for social insect nest construction to robotic systems. Insect workers do not follow a global blueprint and do not normally communicate with each other – instead, the workers convey information through modification of the environment itself. New environment configurations, in turn, stimulate other actions by the same worker or another worker in the colony. French entomologist Pierre-Paul Grassé coined the term *stigmergy* in 1959 to define this type of behaviour [29], which he studied initially in termite species. The word is derived from the Greek roots 'stigma' (urge or goad) and 'ergon' (action or work), and is meant to convey the idea of "incitement to work by the products of work" [30].

Stigmergy can be generally classified as either quantitative and qualitative. In quantitative stigmergy, the intensity of the interactions between agents and the environment takes on a continuous range of values. Pheromone gradients are one example of a quantitative stimulus, and have been suggested as a useful coordination mechanism to guide robotic construction [31]. Qualitative stigmergy, in contrast, uses a set of discrete stimulus types – stimuli that are qualitatively different trigger different responses from an agent.

An example of qualitative stigmergy relevant to this thesis is the model of wasp nest construction proposed by Bonabeau et al. [32]. In this model, reactive agents move randomly and asynchronously on a three-dimensional cubic or hexagonal lattice, depositing elementary bricks when they sense certain configurations of bricks in adjacent cells. Each agent carries an identical, internal lookup table specifying which of two types of bricks to deposit when it encounters an appropriate *stimulating configuration*. Entries in the table are called micro-rules, and together they form a building algorithm. To avoid random deposits, every algorithm begins with a single, pre-placed seed brick.

The model in [32] has three important characteristics: 1) construction can occur in parallel at several locations simultaneously, 2) there is no centralized control or direct (agent-to-agent) communication, and 3) agents have access only to local information about the portion of a structure in their immediate vicinity. The rule space is also extremely large: for the 26-cell local neighbourhood used in the model, and with only two types of elementary bricks, there are $2^{26} \approx 10^8$ unique micro-rules. Each of these rules can potentially be combined with thousands of others, generating an enormous number of possible combinations. However, there are relatively few subsets of compatible rules which generate coherent architectures [33]; a significant challenge is to efficiently find such *coordinated* subsets of micro-rules.

Bonabeau et al. [34,35] explores the rule space using a genetic algorithm (GA). The GA fitness function is based on subjective human evaluation of the 'structuredness' of a series of example architectures, and incorporates the following observations: coherent architectures are compact, consist of repeating, modular patterns, and tend to make use of many of micro-rules. Although the structures produced by the GA are visibly organized, [35] does not consider the problem of generating micro-rules for structures that are a priori specified.

Werfel et al. [36] proposes endowing the environment itself with basic information processing capabilities, as a way to improve the robustness and efficiency of collective assembly. In this approach, called *extended stigmergy* in [36], square structural blocks are able to communicate state information to their neighbours and to nearby robots. The blocks can also react to this information by attaching to or detaching from one another. Using the information provided by the blocks, mobile robots are able to assembly any solid, two-dimensional structure, specified by either a complete design or by a set of more general constraints (e.g. that every block of type A must have a block of type B within a radius of three units). Communication within the structure also provides some capacity for error correction via disassembly.

2.2.2 Physical Implementations

The literature contains several examples of collective assembly involving physical robots. Wawerla et al. [37] studies the role of communication in a multi-robot construction task, where mobile robots assemble linear barriers composed of coloured cylinders. Robots construct a barrier by placing cylinders of alternating colour adjacent to one another, starting with an initial 'seed' cylinder that is pre-positioned. The authors carry out several experiments to verify that a single robot is able to complete the construction task successfully. Simulation studies with multiple robots indicate that, as the number of robots increases, there is a corresponding decrease in individual robot performance, due to interference and competition for access to the construction site. A robot will often have to wait to access to the end of the barrier, only to discover that it is carrying a cylinder of the wrong colour. Overall performance improves significantly if the last robot to successfully add a cylinder is able to communicate one bit of state information (the colour of the attached cylinder) to other, nearby robots.

Werfel et al. [38] describes several algorithms for collective robotic construction of planar structures, where each algorithm is distinguished by the capabilities it requires from the structural blocks – blocks are either inert and identical, inert and distinct, or 'writable' (blocks which can be labelled dynamically). In all cases, the algorithms are limited to the construction of solid shapes without internal holes. Experiments using a mobile robot equipped with a gripper, colour camera and Radio Frequency Identification (RFID) tag reader/writer verify that the robot is able to successfully build to small planar structures from self-aligning, writable blocks.

Parker and Zhang [39] demonstrates a robotic construction system modelled on the behaviour of a particular species of ant (*Leptothorax tuberointerruptus*). This species employs a building method known as *blind bulldozing*, in which the ants push material outwards from a central location to form a circular nest surrounded by a uniform wall. In the corresponding robotic implementation, each member of a group executes the same simple algorithm: a robot moves directly forwards in a straight line, pushing gravel with its plow, until the resistive force on the plow exceeds a predetermined threshold. When this occurs, the robot backs up, randomly reorients itself, and resumes moving forwards in a new direction. Repetition of the pushing action, over a period of time, produces a cleared nest area that is approximately round. The robots are entirely reactive and have no knowledge of the structure they are building. Although robot-robot and robot-wall interactions occur randomly, the time-evolution of the nest structure is statistically predictable.

2.3 Summary

Our review above focused primarily on the modelling of self-assembling and collective systems. We also briefly described several initial attempts to develop artificial implementations of these types of systems.

We saw that theoretical models for self-assembly are at present largely simplified abstractions, designed to permit tractable analysis using, for example, algorithmic complexity theory. The models can generally be grouped based on whether they are *passive* or *active*; active systems are typically more capable, but also more complex, than their passive counterparts. Examples in the passive category include tile assembly, conformational switching, and graph grammar-based approaches. Examples in the active category include intelligent state-based methods and techniques inspired by cell morphogenesis. The addition of direct short-range communication in some active models enables capabilities such as environmental adaptation, disassembly and error correction.

Researchers have begun to form a bridge from self-assembly theory to practice, using a variety of mediums and components. Existing physical implementations rely on capillary forces, selective affinity of DNA molecules, or more sophisticated macro-scale components such as robotic programmable parts.

Our discussion of collective assembly emphasized that designers of collective systems must consider an interaction loop involving both the agents and the environment in which they operate. In some cases, this interaction loop produces a gestalt effect, allowing a group of agents to complete a task which would be impossible for a single individual.

We described methods for collective coordination that rely on digital hormones and on stigmergy. Stigmergy, or environment-mediated coordination, is an attractive approach because it does not require agents to perform any actions beyond those already necessary for the construction task.

Much of the work on collective assembly has been inspired by the stigmergic, distributed nest-building activities of social insect species. For these types of systems, the range of distinct environmental configurations and possible agent-environment interactions is typically extremely large. A challenge, then, is to determine how to constrain the assembly process enough to ensure that a coherent result is produced. Because of the additional complexity involved, collective robotic implementations have thus far been limited to the assembly of planar structures by small numbers of robots.

Chapter 3

A Model for Distributed Assembly

We now introduce a discretized model which captures many of the essential properties of real-world distributed assembly systems. In our model, homogeneous unit-square *assembly components* move randomly on a two-dimensional grid of cells, binding together to form a desired *target structure* (or simply *target*). The assembly process is entirely *local* – assembly decisions depend only on the information available within a component's immediate environment.

We begin in the next section by defining the elements of the model, and proceed to formalize how a structure is assembled over time. The chapter closes with a discussion of several ways in which our abstraction can be modified to describe a wider variety of self-assembling, and also certain types of collective, systems.

3.1 Elements of the Model

Our model consists of two elements: a planar *assembly grid* and a set of *assembly components.* The assembly grid is a finite, two-dimensional Cartesian lattice on which the components move and upon which the final structure is built. An assembly components (or simply *component*), is an autonomous agent that is able to recognize and respond to a limited set of features within its local environment. At any time, a grid cell may be empty or may contain a single assembly component which completely fills the cell.

The position of a cell is represented by a coordinate pair (i, j) of integers, relative to an arbitrary origin $(0, 0)$. We use the cardinal directions north, east, south and west to define spatial relationships between the cells: for coordinate pair (i, j), the first value i specifies the position of the cell along an axis increasing from west to east, while the second value j specifies the position of the cell along an axis increasing from south to north. When there is no risk of ambiguity, we will often refer to a cell by its position, as 'cell (i, j)', instead of explicitly writing 'the cell at position (i, j)'. Likewise, we will generally not distinguish

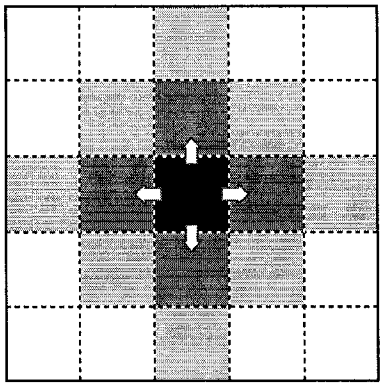


Figure 3.1: The assembly component (black) may move to any one of the four adjacent cells (medium grey) at the next time step, as indicated by the arrows. After two time steps, the component is able to reach any one of the light grey cells.

between a set of cells and the set of coordinate pairs that identify those cells on the grid; this notational convenience will allow us to simplify a number of the definitions below.[1]

Each cell has four adjacent neighbours to the north, east, south and west, as shown in Figure 3.1. Given two adjacent cells at positions (i, j) and (m, n), we describe the position (m, n) with respect to (i, j) by writing $(m, n)^d_{(i,j)}$, where $d \in \{n, e, s, w\}$ is an abbreviation for the direction (north, east, south or west), respectively, of (m, n) relative to (i, j) on the grid.

Definition 3.1 Grid cells at positions (m, n) and (i, j) are ***adjacent*** if $|m-i|+|n-j| = 1$.

Although all of the assembly components have the same size and shape, each may be distinguished by an intrinsic feature called a *label*. In a physical system, this label might be any characteristic that is readily discernible at the scale of the components, such as a colour (macro-scale), inscription (micro-scale), or particular surface molecule (nano-scale). For our purposes, labels will be values from the set $\mathbb{N}^+$ of positive integers. Initially, every component is assigned the *null* label, which is '0' by our convention. Each component is able to undergo a single *label update*, from the null label to a label $l \geq 1$; this change occurs as the result of an assembly action.

An assembly component has minimal sensing capabilities: it is able to determine which direction is north, and to recognize the labels assigned to other components in adjacent, occupied cells. Heading information is used only to maintain a fixed orientation while traversing the grid – an individual does not know its current position, and has no capacity to acquire or compute this information. Additionally, the components have no memory of past actions or observations and cannot communicate directly with one another.

[1]This does (subtly) confuse ontology with representation, however it is acceptable in this case because there is a bijective mapping between cells and coordinate pairs.

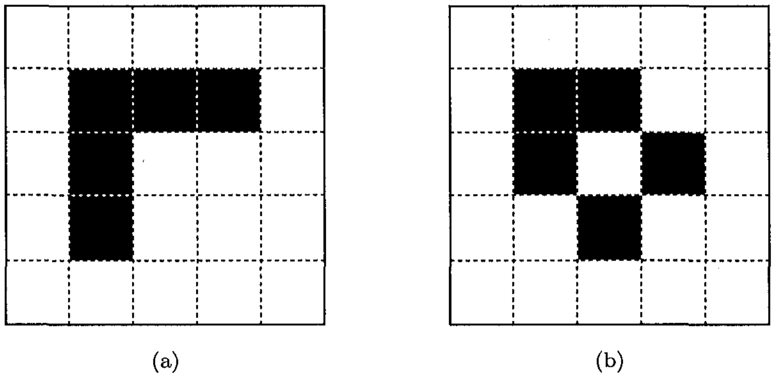


Figure 3.2: A simple connected structure (a), and a similar unconnected structure (b).

3.2 The Assembly Process

Assembly begins with a group of $n \geq 1$ *free components* at a set $\mathcal{F}$ of random grid positions, and a single *seed component*, which is fixed at the origin and given the initial label '1'. Growth of a structure occurs outwards from the seed. We assume that the grid is of sufficient size to accommodate the target structure that we wish to build, with an additional border of cells (at least one unit wide) around the entire grid perimeter.

At each discrete time step, every free component performs a random walk on the grid, moving to an adjacent empty cell if possible. As a component moves, it compares the occupancy pattern in the four adjacent cells with entries in an internal lookup table of *assembly rules*, stored in read-only memory. Each rule specifies a *binding configuration* that triggers an assembly action, and a *resultant* label. The binding configuration enumerates *a*) which adjacent cells must be occupied, and *b*) what labels must appear on components in the occupied cells. At least one cell in a binding configuration must be occupied, and up to three cells may be occupied. When a binding configuration is identified, the component *applies* the corresponding rule by attaching (binding) itself to the adjacent components at its current grid position. Binding is irreversible – assembly actions are never undone. The component is then *bound* to the structure, and the number of free components is reduced by one. Immediately after binding, the component performs a label update, transitioning from the null label ('0') to the resultant label defined by the activated rule. The rules are identical for all components, making each component redundant and interchangeable.

Formally, we define a structure as a set $\mathcal{S}$ of grid cells, with the understanding that the cells are to become occupied as part of the assembly process, and a *label set* $\mathcal{L}$ as $\mathcal{L} \subset \mathbb{N}^+$, with $|\mathcal{L}|$ finite. We model the assembly of *connected* structures only; Figure 3.2 presents examples of a connected and an unconnected structure.

Definition 3.2 A ***connected structure*** is a set $\mathcal{S}$ of $n > 1$ cells (represented by their positions on the grid), such that there is a path (or sequence of pairwise-adjacent cells) between any two cells in $\mathcal{S}$.

Definition 3.3 Two structures $\mathcal{S}_1$ and $\mathcal{S}_2$ are ***isomorphic*** if there is a translation vector $v = (v_1, v_2)$ such that adding v to every element of $\mathcal{S}_1$ results in $\mathcal{S}_2$.

Definition 3.4 The ***bounding box*** for a structure $\mathcal{S}$ is an ordered pair of coordinate pairs $((p,q),(r,t))$, of the largest possible values p, q and the smallest possible values r, t such that for every coordinate pair $(i,j) \in \mathcal{S}$, $p \leq i$ and $q \leq j$ and $i \leq r$ and $j \leq t$.

The assembly process yields a labelled structure $(\mathcal{S}, \mathcal{L}, \lambda)$, where $\mathcal{S}$ is a connected structure, $\mathcal{L}$ is a label set, and λ is a surjection $\lambda : \mathcal{S} \longrightarrow \mathcal{L}$. We note that, while a structure is defined as a set of cells, a labelled structure is defined as a set of *occupied* cells. This is a subtle but important distinction – labels are assigned to *components*, and so the cells in a labelled structure must be occupied.

We will normally represent a labelled structure as a set of ((*coordinates*), *label*) pairs, or, for brevity, by a symbol $\bar{\mathcal{S}}$ with an over-bar. An $|\mathcal{L}|$-*labelling* of a structure $\bar{\mathcal{S}}$ is an assignment of exactly $|\mathcal{L}|$ unique labels to the elements in $\mathcal{S}$ (for example, an assignment using two labels is a two-labelling etc.). For a grid defined by the set of cells M, we use the notation $M(i,j)$ for the label on the component in the cell at position (i,j) if the cell is occupied, or '–' if the cell is empty. Likewise, since a labelled structure occupies a subset $\mathcal{S} \subset M$ of the cells on a grid M, we also use the notation $\bar{\mathcal{S}}(i,j)$ for the label on the component in cell (i,j) which is part of $\bar{\mathcal{S}}$.

Our goal is to produce a specific *target structure* $\mathcal{T}$, or a set of cells that, when occupied, form a desired shape. The set $\mathcal{T}$ always includes the origin cell $(0,0)$, which is the position of the seed. A labelled structure is *complete* when it contains exactly the same set of occupied cells as the target $\mathcal{T}$.

Definition 3.5 For a target structure $\mathcal{T}$ and a labelled structure $\bar{\mathcal{S}} = (\mathcal{S}, \mathcal{L}, \lambda)$, we say $\bar{\mathcal{S}}$ is ***complete*** when $\mathcal{S} = \mathcal{T}$.

Definition 3.6 A ***binding configuration*** is an ordered four-tuple (l_n, l_e, l_s, l_w) of values from the set $\mathbb{N}^+ \cup \{-\}$, where the values represent labels on components in adjacent cells to the north, east, south and west, respectively, of a component's grid position. The symbol '–' is used to identify cells in the configuration that are either empty or occupied by a free component.

Rule Number	Expanded Format	Tuple Format
1	$\begin{pmatrix} & - & \\ - & \bullet & - \\ & 1 & \end{pmatrix} \implies 2$	$(-,-,1,-,2)$
2	$\begin{pmatrix} & - & \\ 1 & \bullet & - \\ & - & \end{pmatrix} \implies 3$	$(-,-,-,1,3)$
3	$\begin{pmatrix} & - & \\ 2 & \bullet & - \\ & 3 & \end{pmatrix} \implies 4$	$(-,-,2,3,4)$

Table 3.1: A simple, three-entry rule set. Each rule is shown in expanded format (second column) and as a five-tuple (third column). In the expanded format, the right arrow is interpreted as 'leads to binding, with the resultant label'. The topmost entry in the expanded binding configuration refers to the cell to the north. The label set here is $\mathcal{L} = \{1, 2, 3, 4\}$.

Definition 3.7 An ***assembly rule*** is an ordered five-tuple $(l_n, l_e, l_s, l_w, \gamma)$, where $l_n, l_e, l_s, l_w \in \mathcal{L} \cup \{-\}$ define the binding configuration and $\gamma \in \mathcal{L}$ is the resultant label.

Definition 3.8 Two rules ***conflict*** if they share the same binding configuration but have different resultant labels.

A rule $r = (l_n, l_e, l_s, l_w, \gamma)$ may be applied by a free component in cell (i, j) on grid M if $l_n = M(i, j+1)$ and $l_e = M(i+1, j)$ and $l_s = M(i, j-1)$ and $l_w = M(i-1, j)$. When r is applied, we update the set $\mathcal{F}$ of positions of free components as $\mathcal{F} \leftarrow \mathcal{F} \setminus \{(i, j)\}$ and the structure $\bar{S}$ as $\bar{S} \leftarrow \bar{S} \cup \{((i, j), \gamma)\}$. At the start of the assembly process, $\bar{S} = \{((0, 0), 1)\}$. When we say that a set of rules *produces* a specific structure, we mean that a group of assembly components executing the rules will form the structure on the grid.

It is possible for a set of bound components to form a barrier around an empty cell, preventing free components from reaching the cell (see Figure 3.3 for an example). In this case, the empty cell is *blocked* by the bound components, and therefore *inaccessible* – if the cell is part of the target structure then the target cannot be completed. This problem, in particular, motivates our discussion of spatiotemporal coordination in Chapter 4.

Definition 3.9 A cell (i, j) is ***accessible*** if, with all free components removed from the grid and starting at an empty cell at the grid edge, there is a connected set of pairwise-adjacent empty cells that includes (i, j). Conversely, a cell is empty but ***inaccessible*** if no such set of pairwise adjacent cells including (i, j) exists.

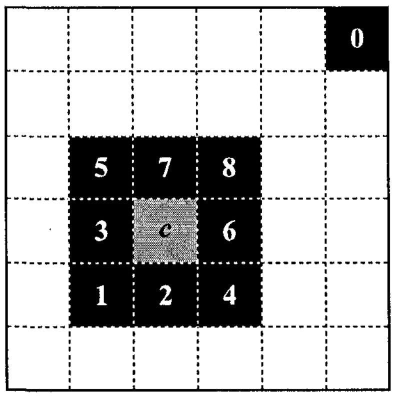


Figure 3.3: The cell *c* (light grey) is blocked by eight bound components (labelled 1 – 8). The free component in the northeast corner of the grid will be unable to access *c*.

Definition 3.10 Each grid cell is always in one of three mutually exclusive ***states***: ***accessible***, ***inaccessible***, or ***occupied*** by a bound component.

An instance of the model is fully specified by the set $\mathcal{F}$ of the positions of free components on the grid, a target structure $\mathcal{T}$, an initial labelled structure $\bar{\mathcal{S}} = \{((0,0),1)\}$, and a rule set $\mathcal{R}$. Table 3.1 gives both graphical and shorthand (tuple) notations for a small rule set. We will primarily use the tuple notation throughout the remainder of the thesis. Figure 3.4 shows four views, each at a different time index, of the structure assembled using these rules.

We do not directly mention assembly time in the exposition above. The rate at which assembly proceeds depends on the size of the grid, the number of free components, and the shape of the target structure, among other factors. We will assume that assembly occurs over some finite (but possibly very large) amount of time.[2]

3.3 Perspectives on Distributed Assembly

According to the taxonomy in Chapter 2, we have defined a model for *active robotic self-assembly* – however our formulation is general enough to be applied more broadly. For example, we specified that the assembly components update their own labels immediately after an assembly action; we could instead assign a fixed label to every component beforehand, and thereby permit each component to implement only one assembly rule. We would then have a heterogeneous set of components, and the resulting model would be very similar to the Tile Assembly Model defined in [5]. This variation is more suitable for describing passive assembly at the micro- or nano-scale.

[2]It is also possible that the assembly process may never terminate, because the components move randomly, although this becomes statistically less likely over longer intervals of time.

With several additional assumptions[3], our model is also valid from a *collective assembly* perspective, where agents transport and attach inert building blocks. If we are able to ignore the spatial extent of the agents themselves, then the theorems regarding spatiotemporal coordination (and the associated assembly guarantees), derived later in the thesis, also hold for the collective system. This variation may be useful for describing assembly tasks in minimalist, macroscopic robotic systems.

In summary, small changes to the model often lead to wide variations in its domain of applicability. The aspect common to all of these domains is that only *local* information is available to an agent at any stage of the assembly process. From an algorithmic point of view, our model is flexible with respect to the way in which labels are assigned and the way in which components are positioned.

[3]We omit the details of these assumptions, except to note that they involve the size and maneuverability of the agents, e.g. an agent must be able to 'fit' into the same spaces and through the same openings as the inert building blocks.

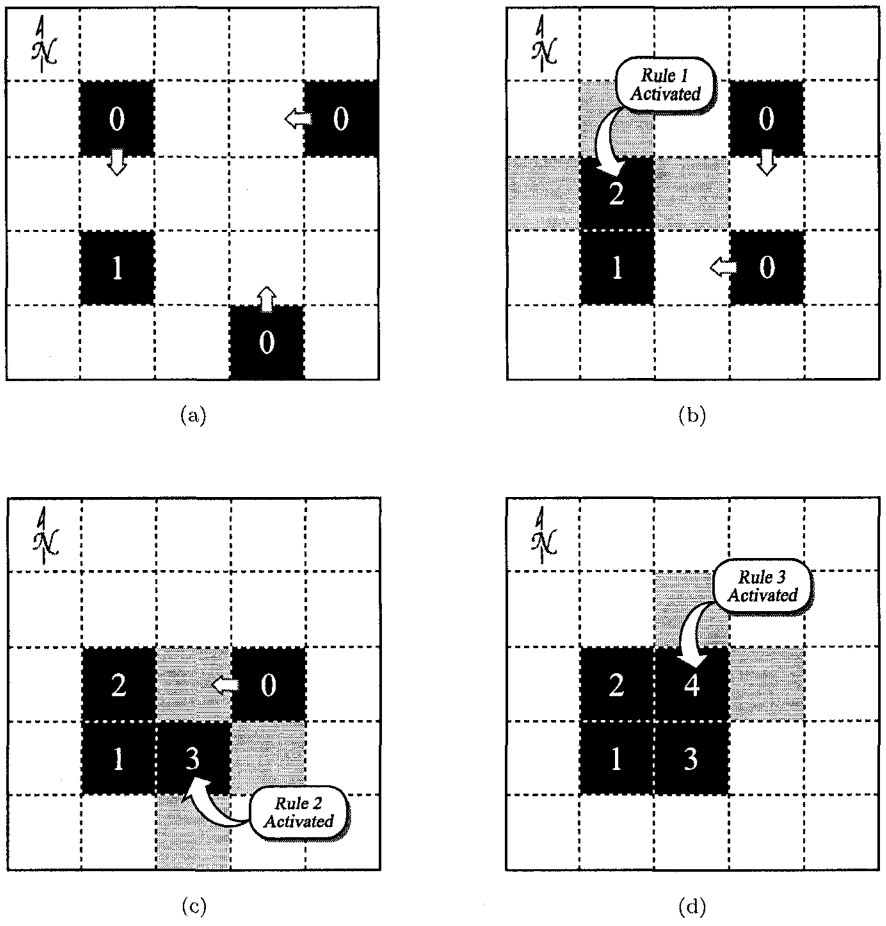


Figure 3.4: Incremental snapshots of the assembly of a 2 × 2 square target structure using the rule set in Table 3.1. The compass rose in the upper left-hand corner cell points north. Small arrows indicate the direction of motion of each assembly component at the next time step. The light grey (empty) cells are part of the binding configurations that trigger rule activations.

Chapter 4

Spatiotemporal Coordination

In our model, assembly components move randomly on the grid, and therefore the exact sequence in which binding events occur cannot usually be determined in advance. However, for assembly to be successful, the geometry and topology of a target structure will necessarily impose certain spatiotemporal ordering constraints on this sequence. As an example, if we consider a solid 3×3 square target with the seed positioned in the southwest corner (as shown in Figure 3.3), the centre component must be attached before the outer border of the structure is closed. Otherwise, free components will not be able to access the interior, making completion of the structure impossible. How can we guarantee that situations such as this, in which a cell becomes inaccessible, cannot occur? A solution to the problem is to embed sufficient coordination information, in the form of specific labels and rules, in the rule set itself.

The first step is to define a set of pairwise spatiotemporal ordering constraints between adjacent cells in the target structure. These constraints determine the order, with respect to time, in which the components should bind to the growing structure. Together, the constraints form an *assembly ordering*, which guides the assembly process. We begin in the section below by formally specifying the conditions that an assembly ordering must satisfy to ensure that no sequence of assembly steps will ever allow a target cell to become inaccessible. In the remainder of the chapter, we present a graph-theoretic approach for finding orderings for a large class of structures, including structures that contain interior holes. Chapter 5 shows how to produce a set of local assembly rules which are guaranteed to follow this ordering, and thus to produce a specific target structure.

4.1 Assembly Orderings

The assembly process we have described cannot occur in a completely arbitrary way – components must already be positioned in certain grid cells before other components can bind to the growing structure. Further, since binding is irreversible, it is necessary to ensure that groups of components do not prematurely form 'barriers' that prevent free components from moving to fill vacant target cells. The notion of an *assembly ordering* captures these dependencies and constraints. We seek to define this ordering, such that any *assembly sequence*, or sequence of component bindings indexed by time step, which respects the ordering will produce the desired target structure. Two separate experimental trials which produce the same structure may have different assembly sequences, depending upon the random actions of the assembly components during each trial.

Definition 4.1 An ***assembly ordering*** for a target structure $\mathcal{T}$ is a set of temporal ordering constraints over the cells in $\mathcal{T}$. Let $\prec$ be the binary ***predecessor*** relation, such that for any two cells $(i,j),(m,n) \in \mathcal{T}$, $(i,j) \prec (m,n)$ if a component in cell (i,j) must bind to the structure before a component in cell (m,n), or $i = m$ and $j = n$. We call (i,j) a *predecessor cell* (or simply *predecessor*) of (m,n), and (m,n) a *successor cell* (or simply *successor*) of (i,j). Together, the set $\mathcal{T}$ and relation $\prec$ define an assembly ordering $(\mathcal{T},\prec)$.

Because the structure grows outward from the seed, the seed cell is a predecessor of every other cell in $\mathcal{T}$. An assembly ordering is not necessarily fully constrained – for all but the simplest structures, assembly operations will often occur in multiple locations simultaneously.

Theorem 4.1 An assembly ordering $(\mathcal{T},\prec)$ is a partial ordering over the cells in $\mathcal{T}$.

Proof. We must show that an assembly ordering is reflexive, antisymmetric and transitive. We show each property in turn.

- The ordering is *reflexive*: $(i,j) \prec (m,n)$ if $i = m$ and $j = n$, by definition.
- The ordering is *antisymmetric*: if $(i,j) \prec (m,n)$ and $(m,n) \prec (i,j)$, then $i = m$ and $j = n$. Here, we appeal to an 'arrow of time' argument – binding events are required to be well-ordered with respect to time. Assume that the ordering is not antisymmetric – then binding at (i,j) must precede binding at (m,n), and binding at (m,n) must precede binding at (i,j). Clearly, because time flows in one direction, only one of these events can occur 'first', and our assumption is incorrect.
- The ordering is *transitive*: if $(i,j) \prec (m,n)$ and $(m,n) \prec (p,q)$, then $(i,j) \prec (p,q)$. Again, using the arrow of time argument, binding must occur in a temporal sequence. That is, if binding at (i,j) must precede binding at (m,n), and binding at (m,n) must precede binding at (p,q), then binding at (i,j) must precede binding at (p,q).

Therefore, an assembly ordering is a partial ordering (or partially-ordered set). □

We say that an assembly ordering is *violated* if an assembly rule allows a component to bind to the structure before one of the predecessor cells is occupied. Assembly rules are evaluated *under* (or with respect to) a particular ordering – a rule which violates one ordering may or may not violate a different ordering, for the same target structure.

As discussed in Chapter 3, a grid cell is always in one of three states: *accessible*, *inaccessible*, or *occupied.* If a particular cell in $\mathcal{T}$ becomes inaccessible, then it is not possible for a free assembly component to reach the cell, and hence $\mathcal{T}$ cannot be completed. We seek to include a sufficient number of constraints in the assembly ordering to prevent this from occurring, i.e. to ensure that assembly can always proceed to completion.

Definition 4.2 An assembly ordering $(\mathcal{T}, \prec)$ for target structure $\mathcal{T}$ is ***valid*** if, for any assembly sequence that respects $(\mathcal{T}, \prec)$ and at any stage of the assembly process, every cell in $\mathcal{T}$ is either accessible or occupied by a bound component.

Stated in a different but equivalent way, an assembly ordering is valid if there does not exist an assembly sequence that both respects the ordering and allows a target cell to become inaccessible. To define a valid ordering, we will divide the cells in the target into two mutually exclusive subsets: *interior cells* and *exterior boundary cells.*

Definition 4.3 Consider a grid containing a complete target structure $\mathcal{T}$, with all other cells empty. Choose a cell (m, n) outside of the bounding box that encloses the target. A cell $(i, j) \in \mathcal{T}$ is an ***exterior boundary cell*** of $\mathcal{T}$ if, with (i, j) empty and all other cells in $\mathcal{T}$ occupied, (i, j) is accessible from (m, n). The set of exterior boundary cells in $\mathcal{T}$ is called the ***exterior boundary set.***

Definition 4.4 Let $\mathcal{T}$ be a target structure, and $\mathcal{B} \subseteq \mathcal{T}$ be the exterior boundary set for $\mathcal{T}$. An ***interior cell*** is a cell $(m, n) \in \mathcal{T} \setminus \mathcal{B}$. The set of interior cells in $\mathcal{T}$ is called the ***interior set.*** Let $\mathcal{I} \subset \mathcal{T}$ be the interior set in $\mathcal{T}$. Then $\mathcal{I} \cup \mathcal{B} = \mathcal{T}$ and $\mathcal{I} \cap \mathcal{B} = \emptyset$.

A straightforward algorithm (which we describe informally) can be used to find the exterior boundary set. As above, consider a grid containing a complete target structure $\mathcal{T}$, with all other cells empty. Choose a cell (m, n) outside of the bounding box enclosing the target $\mathcal{T}$. Then, perform a breadth- or depth-first search over all accessible cells, starting at (m, n). Let this set of empty cells be $\mathcal{C}$. Any cell in $\mathcal{T}$ adjacent to a cell in $\mathcal{C}$ is a member of the exterior boundary set. Figure 4.1 shows the set of exterior boundary cells for an example structure.

With these definitions in hand, we now state an important theorem about valid assembly orderings.

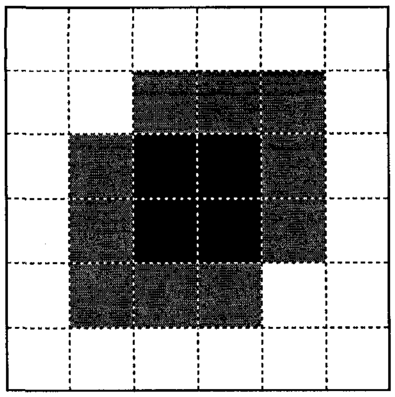


Figure 4.1: Set of exterior boundary cells for an example structure; boundary cells are shown in medium grey.

Theorem 4.2 An assembly ordering $(\mathcal{T}, \prec)$ is valid iff, for each cell in $\mathcal{T}$ except the seed, there is at least one adjacent predecessor cell, and for each interior cell there is at least one adjacent successor cell.

Proof. Our connectivity constraint requires that *every* cell except the seed have at least one adjacent predecessor, so the first clause of Theorem 4.2 is immediately satisfied. Further, cells in the exterior boundary set for $\mathcal{T}$ cannot, by definition, become inaccessible (assuming components only bind to the structure at positions in $\mathcal{T}$), and can therefore be removed from consideration.[1] So we must now show that an ordering is valid iff each interior cell has an adjacent successor cell. We prove by contradiction.

Consider an interior cell (i, j), and assume that the assembly ordering is valid but that (i, j) has no adjacent successors. The $\prec$ relation is transitive, and so it follows that (i, j) cannot not have *any* successors. If (i, j) has no successors, then we must be able to assemble the remaining components in $\mathcal{T}$, regardless of whether cell (i, j) is occupied or not (there are no constraints in $(\mathcal{T}, \prec)$ that prevent this). Assume that (i, j) is empty and, according to $(\mathcal{T}, \prec)$, we place components in the remaining cells in $\mathcal{T}$. Cell (i, j) is an interior cell, and the final structure is connected, so there must exist a subset of cells $Q \subset \mathcal{T}$ that block (i, j) – otherwise (i, j) would be an exterior boundary cell. But if $(\mathcal{T}, \prec)$ is valid, then such a subset cannot exist, and we have reached a contradiction.

Likewise, assume that every interior cell in $\mathcal{T}$ has an adjacent successor cell, but that the assembly ordering is not valid. Consider an inaccessible interior cell (i, j), and let $Q \subset \mathcal{T}$ be a set of occupied cells that block (i, j). Let W be the set containing (i, j) and any successors of (i, j) that are also blocked by Q. There must exist a cell $(p, q) \in W$ that does not have an adjacent successor, since W is finite and the predecessor relation is anti-symmetric. According to $(\mathcal{T}, \prec)$, (p, q) must therefore be an exterior boundary cell. But if

[1]In Chapter 5, we show how to ensure that components only bind at positions in $\mathcal{T}$.

(p, q) is an exterior boundary cell, it is always accessible, and cannot be blocked by Q. We have again reached a contradiction, which completes the proof. $\square$

For solid structures which do not contain any interior holes, generating valid assembly orderings is relatively straightforward. We can create a valid ordering simply by moving outward from the seed, because each cell incrementally farther from the seed is incrementally closer to an exterior boundary cell. The cells in structures which incorporate holes do not necessarily satisfy the above relationship, and this in turn can complicate the ordering problem. We discuss this issue briefly in the next section, after giving a formal definition of an interior hole.

Definition 4.5 A set is ***maximal*** if it is not a subset of a larger set.

Definition 4.6 A ***hole*** in target $\mathcal{T}$ is a maximal set H of connected empty cells, such that for all $(i, j) \in H$, $(i, j) \notin \mathcal{T}$ and for each direction $d \in \{n, e, s, w\}$, either (i, j) is adjacent to another empty cell $(m, n) \in H$ in direction d, or (i, j) is adjacent to a cell $(p, q) \in \mathcal{T}$ in direction d, and there exists a set $\mathcal{W} \subseteq \mathcal{T}$ of cells that, when occupied, block all cells in H.

4.2 Assembly Graphs

We now show how an assembly ordering can be expressed as a *directed graph*, consisting of a set of vertices, corresponding to the cells in the target structure, and a set of directed edges between the vertices, corresponding to pairwise ordering constraints. We call such a graph an *assembly graph*; this formalism will allow us to use an efficient graph algorithm to produce valid assembly orderings for a large class of structures. We begin with several definitions.

Definition 4.7 A ***graph*** $G = (V, E)$ consists of a vertex set V and an edge set E, where each edge is associated with two vertices known as the ***endpoints*** of the edge. An edge is said to be ***incident*** on both of its endpoints. If the vertex pairs that identify edges are unordered (where (v_a, v_b), $v_a, v_b \in V$ refers to the same edge as (v_b, v_a)), the graph is said to be ***undirected***. If the vertex pairs that identify edges are ordered, the graph is said to be ***directed***, and the directed edge (v_a, v_b) has the ***tail*** vertex v_a and the ***head*** vertex v_b. The edge (v_a, v_b) is ***outgoing*** from vertex v_a, and ***incoming*** to vertex v_b.

Definition 4.8 The ***degree*** of a vertex v, denoted $\mathtt{deg}(v)$, is the number of edges that are incident on v.

Definition 4.9 A ***path*** through a graph G is a traversal of consecutive vertices along a sequence of edges in G. The vertices that begin and end the path are the ***initial*** vertex and ***terminal*** vertex, respectively. The length of the path is the number of edges that are traversed along the path.

Definition 4.10 An undirected graph is ***connected*** if there is a path between every pair of vertices.

Definition 4.11 A path is ***simple*** if it does not contain any repeated vertices.

Definition 4.12 A ***directed path*** is an oriented, simple path such that all edges go the same direction.

Definition 4.13 A ***directed cycle*** is a directed path where the initial vertex is the same as the terminal vertex.

Definition 4.14 A ***directed acyclic graph*** (or DAG), is a directed graph that contains no directed cycles.

Our ordering algorithm requires an initial, undirected *adjacency graph* for a target structure $\mathcal{T}$. We form this adjacency graph G by:

- adding a vertex $v_{(i,j)}$ to the set V for each cell in $(i,j) \in \mathcal{T}$, and
- adding an undirected edge $(v_{(i,j)}, v_{(m,n)})$ to the set E for every pair of adjacent cells $(i,j), (m,n) \in \mathcal{T}$, between the corresponding vertices $v_{(i,j)}$ and $v_{(m,n)}$.

As indicated above, we will identify a vertex using the subscripted coordinate pair which defines the corresponding grid cell, e.g. $v_{(1,2)}$ for the vertex that maps to grid cell $(1,2)$. When we describe a directed path between two vertices, we will use two coordinate pairs and an arrow, where the first coordinate pair identifies the initial vertex in the path, and the last coordinate pair identifies the terminal vertex, e.g. $p_{(0,0)\rightarrow(i,j)}$ for a path from the seed vertex to a vertex $v_{(i,j)}$.

The relationship between cells in $\mathcal{T}$ and vertices in G can be expressed as a bijection $\pi : \mathcal{T} \rightarrow V$. Because there is an enforced adjacency relationship between cells in the target structure, our graphs are always connected.

Using directed edges, we can transform the undirected graph G, which defines the static adjacency relationships between components in the final structure, into a directed graph G' that contains information about the dynamic relationships that exist as the structure evolves from the seed.[2] The directed graphs we work with will always be acyclic. A cycle

[2]We will use the prime marker, $'$, to indicate that a graph is directed.

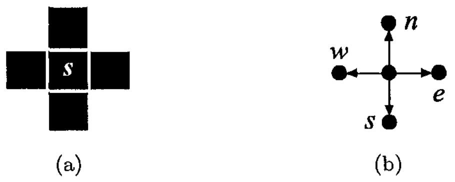


Figure 4.2: A small structure (a) and its edge-oriented assembly graph (b). The edges of the graph in (b) have been marked with their directions.

(of minimum length four) implies that a successor cell must be occupied before one of its predecessor cells; this contradicts an allowable temporal ordering. Stated equivalently, only directed acyclic graphs have vertex partial orders.

Definition 4.15 A ***source vertex*** in a directed graph $G' = (V, E')$ is a vertex $v \in V$ that has only outgoing edges.

Definition 4.16 A ***sink vertex*** in a directed graph $G' = (V, E')$ is a vertex $v \in V$ that has only incoming edges.

We form a directed assembly graph G' for the structure $\mathcal{T}$ from the same set of vertices V, by adding a directed edge from vertex $v_{(i,j)}$ to vertex $v_{(p,q)}$ if the corresponding cell (i,j) is adjacent to cell (p,q) and (i,j) is a predecessor of (p,q). A sink vertex in the graph represents a cell with no successors. There is a single source vertex in the graph corresponding to the seed cell. The set of exterior boundary vertices in V is simply the set of vertices that correspond to the exterior boundary cells in $\mathcal{T}$; the same correspondence exists for the interior vertices and interior cells.

Since our assembly graphs are planar (a fact which we do not prove here, but which is obvious from the planarity of the target structure), we can assign each edge in G' an orientation based on the direction in which the edge points (north, east, south or west). We will refer to an assembly graph in which each edge has an explicit orientation as an *edge-oriented assembly graph.* When we draw an assembly graph, we will designate north to point towards the top of the page, and the orientation of each edge will therefore be implicit.

Definition 4.17 An ***edge-oriented assembly graph*** $G^* = (V, E', \phi)$ is an ordered triple, where V is a set of vertices, E' is a set of directed edges, and ϕ is a mapping $\phi : E' \rightarrow d$, $d \in \{n, e, s, w\}$ from edges to the directions north, east, south and west, respectively.

Orientation is significant because two portions of a structure which share the same edge-oriented assembly subgraphs can be assembled using the same rules (and labels). Edge orientations will be important when we compare assembly graphs in Chapter 6.

Using the above definitions, we can re-express Theorem 4.2 in terms of the properties of an assembly graph.

Theorem 4.3 Let G' be an assembly graph for target structure $\mathcal{T}$. Let $v_{(0,0)}$ be the seed vertex in G', and let $\mathcal{B}$ be the set of exterior boundary vertices. Then G' represents a valid assembly ordering for $\mathcal{T}$ iff for every non-seed vertex $v_{(i,j)} \in V$ there is a directed path $p_{(0,0)\to(i,j)}$ from $v_{(0,0)}$ to $v_{(i,j)}$, and, if $v_{(i,j)}$ is an interior vertex, there is also a directed path $p_{(i,j)\to(m,n)}$ from $v_{(i,j)}$ to a vertex $v_{(m,n)} \in \mathcal{B}$ and $p_{(0,0)\to(i,j)}$ and $p_{(i,j)\to(m,n)}$ are vertex-disjoint except for $v_{(i,j)}$.

Proof. The proof follows from Theorem 4.2 and the equivalence between directed edges in an assembly graph and pairwise ordering constraints in an assembly ordering. By Theorem 4.2, every cell in $\mathcal{T}$ except the seed cell must have at least one adjacent predecessor. Let (i,j) be a non-seed cell in $\mathcal{T}$, and let (p,q) be an adjacent predecessor cell. We represent the predecessor relation in the assembly graph by a directed edge from the vertex $v_{(p,q)}$ to vertex $v_{(i,j)}$. Therefore, every non-seed vertex in G' must have at least one incoming directed edge from a predecessor vertex. By transitivity, there must therefore be a simple, directed path $p_{(0,0)\to(i,j)}$ from the seed vertex $v_{(0,0)}$ to every other vertex in G'.[3]

Likewise, every interior cell in $\mathcal{T}$ must have at least one adjacent successor. Let (i,j) be an interior cell in $\mathcal{T}$, and let (m,n) be an adjacent successor cell. We represent the successor relation in the assembly graph by a directed edge from the vertex $v_{(i,j)}$ to vertex $v_{(m,n)}$. Therefore, every interior vertex must have at least one outgoing directed edge to a successor vertex. Let $W \subset V$ be the set of all interior vertices in G'. Choose an interior vertex $v_{(i,j)} \in W$. If we apply the transitivity property of the assembly ordering, starting at $v_{(i,j)}$, we will eventually reach a successor $v_{(m,n)}$ of $v_{(i,j)}$ which is not in W, because there are only a finite number of vertices in W and there are no cycles in the graph. The vertex $v_{(m,n)}$ is an exterior boundary vertex. So there must be a simple, directed path $p_{(i,j)\to(m,n)}$ from each interior vertex $v_{(i,j)}$ to an exterior boundary vertex $v_{(m,n)}$.

The paths $p_{(0,0)\to(i,j)}$ and $p_{(i,j)\to(m,n)}$ from the seed vertex to an interior vertex $v_{(i,j)}$, and from $v_{(i,j)}$ to an exterior boundary vertex $v_{(m,n)}$ must be vertex-disjoint except for $v_{(i,j)}$ because the graph G' cannot contain a cycle.

For the reverse case, it is immediately clear that if there is a directed path from the seed vertex to each vertex $v \in V$ in G', then every cell in $\mathcal{T}$ has an adjacent predecessor cell (corresponding to the incoming edge incident on v). And if there is a directed path from each interior vertex to an exterior boundary vertex, then each interior vertex has an outgoing edge in E' corresponding to an adjacent successor cell in $\mathcal{T}$. This completes the proof. $\square$

[3]There may in fact be several directed paths, the proof requires only that there be at least one.

We will say that an assembly graph is valid if and only if the ordering that the graph represents is valid.

For an interior vertex $v \in V$ in assembly graph G', if there is both a path from the seed to v, and another path from v to an exterior boundary vertex, and the paths are vertex-disjoint except for v, then we say that vertex v satisfies the *disjoint-paths condition* in G'. In a valid assembly graph, every interior vertex must satisfy this condition.

It is always possible to find a directed path from the seed vertex to any other vertex in the assembly graph – however, it may *not* be possible to find both a directed path from the seed vertex to an interior vertex *and* also a path from that interior vertex to an exterior boundary vertex. In particular, we can sometimes determine whether a valid assembly ordering exists for a structure simply by examining the adjacency graph, as stated by the following theorem.

Theorem 4.4 Let $\mathcal{T}$ be a target structure and G be the adjacency graph for $\mathcal{T}$. If G contains an interior, non-seed vertex $v \in V$ such that $\texttt{deg}(v) = 1$, then no valid assembly ordering exists for $\mathcal{T}$.

Proof. By Theorem 4.3, a valid assembly ordering for $\mathcal{T}$ exists iff there is a directed path from the seed vertex to every interior vertex and a directed path from every interior vertex to an exterior boundary vertex, and the paths are vertex-disjoint. Clearly, if an interior vertex has degree one, then it is not possible to assign both an incoming edge (along a path from the seed) and an outgoing edge (along a path to an exterior boundary vertex). Therefore, no valid assembly ordering exists for $\mathcal{T}$. $\square$

4.3 Generating Valid Assembly Orderings

We now turn to the problem of generating valid assembly orderings for specific target structures. In general, there may be many valid orderings for a given structure; the algorithm presented below generates one such ordering. Also, we choose to impose one constraint between every adjacent pair of cells in the target structure, although this is often not necessary to satisfy the validity requirement.

Our algorithm generates a *breadth-first* assembly graph. The graph is produced by traversing the target's (undirected) adjacency graph in breadth-first order, starting from the seed. The directed graph returned by Algorithm 4.1 is always acyclic, however it may not be a valid assembly graph.

Algorithm 4.1 is a modified version of a standard breadth-first graph search, which traverses all of the vertices in the adjacency graph G, rather than searching for a particular vertex. As we encounter each vertex in G, we incrementally build a corresponding assembly

Algorithm 4.1 Breadth-First Assembly Graph for Target Structure

Input: Target structure $\mathcal{T}$

Output: Breadth-first directed assembly graph G'

```
1: $V \leftarrow$ set of vertices, one vertex for each cell in $\mathcal{T}$
2: $E \leftarrow$ set of undirected edges, one edge for every pair of adjacent cells in $\mathcal{T}$
3: $G \leftarrow (V, E)$    // undirected adjacency graph
4: $E' \leftarrow \emptyset$
5: $open \leftarrow \{v_{(0,0)}\}$    // FIFO queue
6: $closed \leftarrow \emptyset$
7: while $open$ is not empty do
8:   $v_{(i,j)} \leftarrow$ dequeued vertex from front of $open$
9:   $closed \leftarrow closed \cup \{v_{(i,j)}\}$
10:  for each vertex $v_{(p,q)}$ adjacent to $v_{(i,j)}$ in $G$ do
11:    if $v_{(p,q)} \notin closed$ then
12:      if $v_{(p,q)} \notin open$ then
13:        add $v_{(p,q)}$ to end of $open$
14:      end if
15:    else
16:      $E' \leftarrow E' \cup \{(v_{(p,q)}, v_{(i,j)})\}$    // add directed edge
17:    end if
18:  end for
19: end while
20: return $G' = (V, E')$
```

graph, G', using the same set of vertices V, but adding a set of directed, rather than undirected, edges. When we arrive at a vertex v, we add directed edges to E' from all adjacent, previously-visited vertices. This procedure differs from standard breadth-first search in that we perform an action (adding directed edges to E') *after* visiting *all* predecessors vertices, instead of immediately after visiting a single predecessor. This modification ensures that there is a directed edge in the final assembly graph between every adjacent pair of vertices.

Theorem 4.5 Algorithm 4.1 generates an assembly graph for target $\mathcal{T}$ in $\Theta(|\mathcal{T}|)$ time. The graph is a valid assembly graph iff every sink vertex in G' is an exterior boundary vertex.

Proof. We show that the algorithm always produces a directed, acyclic graph, which is an assembly graph. First, we show that each vertex in the adjacency graph G is visited exactly once. The algorithm examines one vertex on every iteration of the `while` loop, starting with the seed vertex, and adds all *unvisited* adjacent vertices, which are not already in the *closed* set or the *open* queue, to the *open* queue. A vertex can therefore be examined once, at most, when it is removed from the front of the *open* queue. Note, also, that the algorithm only terminates when the queue is empty. Now, assume that, at some point, the algorithm fails to visit a vertex v in G. Then the algorithm cannot have visited any of the vertices adjacent to v, or v would have been added to the *open* queue. But the graph is connected,

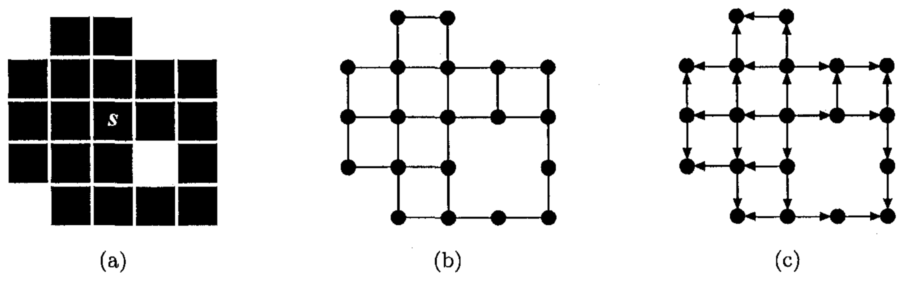


Figure 4.3: (a) Simple structure containing an interior hole. The seed component is marked with an s. (b) Adjacency graph for the structure. (c) Breadth-first assembly graph generated by Algorithm 4.1.

so there must be at least one vertex adjacent to v which is visited by the algorithm, and so v must have been added to the *open* queue. This is a contradiction, and hence each vertex is visited exactly once.

We now show that the directed graph G' is acyclic. Assume that the algorithm creates a cycle in the graph. By definition, the initial and terminal vertices for the cycle are the same. This means that the algorithm must have reached the same vertex twice, along different paths. But we showed above that each vertex is visited once, and so the algorithm cannot create a directed cycle. Therefore, the algorithm generates a graph which is directed and acyclic, and thus is an assembly graph.

The validity of the graph follows directly from Theorem 4.3. There is a directed edge between every adjacent pair of vertices in G', and the graph is acyclic, so, if all sink vertices in G' are exterior boundary vertices, then the graph is valid. Likewise, if an interior sink vertex exists, then the graph cannot be valid because there is at least one interior vertex that does not satisfy the disjoint-paths condition.

The time complexity of the algorithm is determined by the number of steps required to complete the breadth-first traversal of all vertices in the adjacency graph G. This breadth-first traversal has complexity $\Theta(|V| + |E|)$ in general. For our specific case, the adjacency graph has at least $|V| - 1$ edges. To determine an upper bound on the number of edges, we use an elementary result from graph theory, the Handshaking Lemma, which states that the number of edges in a graph is half of the sum of the degree of all vertices. In the adjacency graph, each vertex has degree at most four, and so the number of edges in the graph has an asymptotic upper bound of at most $4|V|/2 = 2|V|$. This gives an overall time complexity of at least $\Omega(|V| + |V|)$, and at most $O(|V| + 2|V|)$, which is in $\Theta(|V|)$. Since $|V| = |\mathcal{T}|$, this is also $\Theta(|\mathcal{T}|)$. □

For solid structures (without any interior holes), the breadth-first algorithm will always return a valid assembly graph, which can immediately be transformed into a set of pairwise ordering constraints (see Algorithm 4.2). There is also a large class of structures that contain

Algorithm 4.2 Valid Assembly Ordering for Target Structure

```
Input: Target structure $\mathcal{T}$
Output: Valid assembly ordering $(\mathcal{T}, \prec)$ or Failure if no valid ordering is found
1: $G' = (V, E') \leftarrow$ assembly graph for $\mathcal{T}$ generated by Algorithm 4.1
2: if $G'$ contains an interior sink vertex then
3:     return Failure
4: end if
5: $\prec \leftarrow \emptyset$
6: for each directed edge $(v_{(i,j)}, v_{(m,n)}) \in E'$ do
7:     $\prec \leftarrow \prec \cup \{(\textsc{Cell}(v_{(i,j)}), \textsc{Cell}(v_{(m,n)}))\}$   // add pairwise constraint
8: end for
9: return $(\mathcal{T}, \prec)$
```

interior holes, for which the breadth-first algorithm produces a valid graph. As an example, consider the simple 20-cell structure in Figure 4.3.[4] The structure encloses an interior hole (one cell in size); Figure 4.3(b) is the adjacency graph for the structure, and Figure 4.3(c) is the corresponding valid assembly graph generated by Algorithm 4.1.

Algorithm 4.1 is not complete. There are structures for which a valid assembly graph exists and the algorithm returns `Failure`. However, it is often possible, for simple structures that contain interior holes, to produce a valid graph by inspection, starting with the breadth-first result. We give an example of such a case in Chapter 6.

Once we have successfully generated a valid assembly graph, we use Algorithm 4.2 to transform the graph edges into a set of pairwise ordering constraints over the cells in the target structure $\mathcal{T}$. The notation $\textsc{Cell}(v_{(i,i)})$ in the algorithm description denotes the cell (i, j) in $\mathcal{T}$ that maps to vertex $v_{(i,j)}$.

Theorem 4.6 Algorithm 4.2 generates a valid assembly ordering for target structure $\mathcal{T}$, or returns `Failure`, in $\Theta(|\mathcal{T}|)$ time.

Proof. Algorithm 4.2 begins by generating an assembly graph G' using Algorithm 4.1, which requires $\Theta(|\mathcal{T}|)$ time by Theorem 4.5. If this initial step does not produce a valid graph, the algorithm returns `Failure` immediately. The final step in the algorithm is to produce the set of pairwise ordering constraints from the graph. The `for` loop adds one constraint to the set $\prec$ for every edge in G' (line 7). Since the graph is valid, and all required ordering constraints are encoded by the graph edges, it follows that the resulting ordering is valid. As shown previously, $\Theta(|\mathcal{T}|)$ time is required to to process all of the edges; this gives an overall time complexity of $\Theta(|\mathcal{T}|)$. □

As described, Algorithm 4.1 works outwards from the seed cell. However, if necessary, the algorithm could be modified to incorporate other restrictions or preferences, for example

[4]From this point onward, we will generally draw only the cells that form a structure; the existence of the grid will be implied.

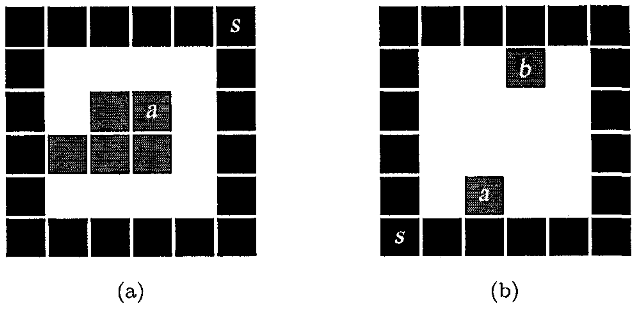


Figure 4.4: Examples of structures that cannot be assembled using local rules and a single seed only. For structure (a), there is a valid assembly ordering if the seed is moved from position s to position a. For structure (b), there is no valid ordering, regardless of the position of the seed.

that assembly should proceed in a north-to-south direction whenever possible. We emphasize that, regardless of the exact assembly graph produced, in *any* valid graph all interior vertices must satisfy the disjoint-paths condition.

We close with two example structures, shown in Figure 4.4, for which no valid assembly orderings exist. Cells a and b in the figures correspond to sink vertices in the respective breadth-first assembly graphs, and therefore represent the end of the flow of information about the state of the assembly process. Neither cell a nor b has an adjacent successor cell, and so assembly of the remainder of the structure is not impeded by their absence. More generally, there is no assembly ordering that enforces the constraints necessary to ensure that the exterior boundary remains open until components bind at a and b. That is, it is not possible in either case to produce an assembly graph where all of the interior vertices satisfy the disjoint-paths condition. For the structure in Figure 4.4(a), it is possible to avoid this problem by moving the seed vertex to position a. For the structure in Figure 4.4(b), the adjacency graph includes two non-seed vertices of degree one, and so moving the seed does not solve the problem. This is one drawback of purely local sensing.

Chapter 5

Rules and Labels

In this chapter, we describe a procedure for generating a set of local rules that, when executed by a group of assembly components, produce a specific target structure. First, we show that we can always generate such a rule set, as long as a valid assembly ordering exists for the target. This is an important point: it is possible to embed sufficient coordination information in the rule set to ensure that a structure is assembled *deterministically*, although the components move and interact randomly. Our initial solution will involve what we call a *worst-case* rule set, which assigns a unique label to every component in the final structure.

For targets of significant size, the memory and sensing demands (to store the rule table and to correctly identify all of the labels, respectively) of the worst-case rule set may be prohibitive, especially for physically very simple assembly components. To address this issue, we formally introduce the *Minimum Label Set* (MLS) problem, a combinatorial optimization problem which involves finding the minimum number of labels required to assemble a structure. We do not know of a polynomial time solution to the MLS problem, however we develop an iterative, randomized optimization algorithm which provides quantitatively 'good' solutions in a reasonable amount of time. Chapter 6 presents a series of experiments that characterize how well the optimization algorithm performs, for a variety of structures.

5.1 Consistent Rule Sets

Our overall goal is to generate a rule set that *a*) enables a group of components to assemble into one specific structure only, and *b*) requires a limited (ideally, minimal) number of labels. We begin by asking the following question: given a target structure $\mathcal{T}$, a valid assembly ordering $(\mathcal{T}, \prec)$, and some rule set $\mathcal{R}$, how can we determine if the rules in $\mathcal{R}$, when applied, actually produce $\mathcal{T}$ only, and no other structures? One possibility is to simulate the assembly process and observe the outcome – if we do this a sufficient number of times, we can then draw statistical conclusions about the rule set and the structure(s) it produces. This approach will not allow us to say with certainty, however, that a rule set *always* produces $\mathcal{T}$.

We now show that it is possible to verify in polynomial time whether or not a candidate rule set $\mathcal{R}$ uniquely assembles a target structure $\mathcal{T}$ while respecting a particular assembly ordering. By *uniquely*, we mean that only one distinct label may appear at each position in the final structure. This restriction is necessary to avoid having to check a potentially exponential number of possible assembly sequences.

Definition 5.1 A rule set $\mathcal{R}$ is ***consistent*** for a target structure $\mathcal{T}$ under assembly ordering $(\mathcal{T}, \prec)$ if $\mathcal{R}$ produces $\mathcal{T}$ exactly, with only one possible label at each position in $\mathcal{T}$, and does not allow $(\mathcal{T}, \prec)$ to be violated, for any possible sequence of actions by the assembly components.[1]

Importantly, if a rule set is not consistent for a target $\mathcal{T}$, this does not necessarily imply that the rule set does not assemble the structure. We are concerned with generating constrained rule sets that *guarantee* assembly of the target – the space of rule sets that have this guarantee is considerably smaller that the space of rule sets that are statistically *able* to produce the desired structure with some non-zero probability. Without loss of generality, we also make the assumption that all the rules in $\mathcal{R}$ are used during the assembly process. It is possible for $\mathcal{R}$ to contain conflicting rules and yet to still be consistent, if none of the conflicting rules are ever applied. In this case, the extra rules are superfluous and can simply be removed.

The problem of determining whether a rule set is consistent may initially appear to require exponential time, given that the assembly components move randomly and can interact in a combinatorial number of ways. We show below that Algorithm 5.1 can be used to determine if a rule set $\mathcal{R}$ is consistent for a particular target $\mathcal{T}$ in a time that is polynomial in the size of $\mathcal{T}$ and $\mathcal{R}$.

Definition 5.2 The ***frontier*** of a structure $\mathcal{S}$ is the set of all grid cells adjacent to cells in $\mathcal{S}$.

Before describing Algorithm 5.1, we give an additional definition related to the assembly process. A rule r may *potentially* be applied at a cell (i, j) if there is a valid assembly sequence that allows the binding configuration for r to appear at (i, j). This is a local test that accounts for all possible ways in which the structure could have grown such that cell (i, j) lies on the frontier – that is, we evaluate whether a rule could be applied at (i, j) if any existing, adjacent predecessors were absent. For example, two rules, $(0, 0, 0, 4, 8)$ and $(0, 0, 3, 4, 8)$, may both potentially be applied at the same cell (i, j) but at different stages of

[1]The term *consistent* was originally introduced by Jones and Matarić in [13].

Algorithm 5.1 Consistent Rule Set Decision

```
Input: Target $\mathcal{T}$, assembly ordering $(\mathcal{T}, \prec)$ and rule set $\mathcal{R}$
Output: True if $\mathcal{R}$ is consistent for $\mathcal{T}$, False otherwise
1: $\bar{\mathcal{S}} \leftarrow \{((0,0),1)\}$
2: $\mathcal{K} \leftarrow$ {empty cells adjacent to $(0,0)$}  // initialize frontier
3: while $\mathcal{K} \neq \emptyset$ do
4:   $\mathcal{K}^* \leftarrow \emptyset$
5:   for each cell $(i,j) \in \mathcal{K}$ do
6:     if $\nexists\, r \in \mathcal{R}$ such that $r$ is potentially applicable at $(i,j)$ then
7:       continue
8:     else if $(i,j) \notin \mathcal{T}$ then
9:       return False
10:    else if $\exists\, r \in \mathcal{R}$ such that applying $r$ would violate $\prec$ then
11:      return False
12:    end if
13:    $r \leftarrow$ single rule from $\mathcal{R}$ applicable at $(i,j)$
14:    $\bar{\mathcal{S}} \leftarrow \bar{\mathcal{S}} \cup \{((i,j), \text{RESULTANT}(r))\}$  // apply rule
15:    $\mathcal{K}^* \leftarrow \mathcal{K}^* \cup$ {empty cells adjacent to $(i,j)$}  // grow new frontier
16:  end for
17:  $\mathcal{K} \leftarrow \mathcal{K}^*$
18: end while
19: if $|\bar{\mathcal{S}}| < |\mathcal{T}|$ then
20:   return False
21: end if
22: return True
```

the assembly process, depending on whether an adjacent component with label '3' is already bound to the south of (i,j).

Algorithm 5.1 operates by maintaining a set $\mathcal{K}$ of empty cells that form part of the frontier of the growing structure $\bar{\mathcal{S}}$. This set is updated on every iteration – it includes all of the empty, adjacent cells for which the local binding configuration has changed (as the result of a binding action). There is no need to check cells for which the local binding configuration is unchanged from the previous iteration (since all rules will already have been tested at those positions). At each step and for every position in $\mathcal{K}$, the algorithm determines if more than one rule can potentially be applied, or if there is a rule which would add a component not in $\mathcal{T}$. If either condition is true, the rule set is inconsistent. Otherwise, the algorithm iteratively adds components to the labelled structure $\bar{\mathcal{S}}$ until no further rules can be applied – if, at that time, $\bar{\mathcal{S}}$ has the same number of occupied cells as $\mathcal{T}$, then $\mathcal{R}$ is consistent for $\mathcal{T}$. The notation RESULTANT(r) denotes the resultant label specified by rule r.

Theorem 5.1 Algorithm 5.1 decides whether a rule set $\mathcal{R}$ is consistent for a target $\mathcal{T}$ under assembly ordering $(\mathcal{T}, \prec)$ in time $O(|\mathcal{R}||\mathcal{T}|)$.

Proof. We show that the algorithm returns `True` iff the assembly process always terminates with the desired target structure $\mathcal{T}$ and the assembly ordering is not violated at any stage.

Assume that $\mathcal{R}$ is not consistent for $\mathcal{T}$, and let $\bar{\mathcal{V}}$ be another labelled structure produced by $\mathcal{R}$. Choose a sequence in which blocks are added to $\bar{\mathcal{V}}$, and let (i, j) be the first position in the sequence where either the assembly ordering is violated or $(i, j) \notin \mathcal{T}$. We review each case in turn.

To violate the assembly ordering, a rule must be potentially applicable at cell (i, j), for some binding configuration in which one or more predecessors are missing. The algorithm will discover a witness for inconsistency in this case, since, when a rule is tried at a frontier cell, all possible combinations of adjacent predecessors are considered (line 10).

Likewise, if at any stage a component is able to bind at a cell (i, j) such that $(i, j) \notin \mathcal{T}$, the algorithm will discover a witness for the inconsistency (line 8), because every rule is tried at each untested frontier cell on each iteration.

Conversely, the algorithm only adds a block to the growing structure $\bar{\mathcal{S}}$ at position (i, j) when exactly one rule r is applicable (line 13), and only when the binding configuration for r includes all adjacent predecessors. The main for loop terminates when no additional rules can be applied at any cells on the frontier of $\bar{\mathcal{S}}$. If, at this point, $\bar{\mathcal{S}}$ contains fewer occupied cells than $\mathcal{T}$, then the rule set fails to assemble the entire target and the algorithm returns `False` (line 19). Otherwise, $\mathcal{R}$ is consistent for $\mathcal{T}$, and the algorithm returns `True`.

To determine the time complexity, we note that on each iteration of the `for` loop we try $|\mathcal{R}|$ rules at an empty cell (i, j). When a rule is applied at (i, j), no more than four adjacent cells are added to the set $\mathcal{K}$, which is empty at the start of each iteration. The loop runs once for every component we add to $\bar{\mathcal{S}}$, and hence we check at most $4|\mathcal{T}|$ frontier cells in total. This gives an overall time complexity of $O(|\mathcal{R}||\mathcal{T}|)$. □

5.2 From Assembly Ordering to Worst-Case Rule Set

We now turn to the problem of generating a consistent rule set for a specific target structure $\mathcal{T}$. Given a valid assembly ordering $(\mathcal{T}, \prec)$, it is always possible to generate such a rule set, in the following way. We begin by placing a component with a unique label at each position in $\mathcal{T}$, creating a labelled structure $\bar{\mathcal{S}}$. Then, as described by Algorithm 5.2, we step through the assembly ordering cell by cell, adding rules sequentially to the rule set. For a cell (i, j), we add a rule whose binding configuration is defined by the labels on components in adjacent predecessor cells (with non-predecessor entries set to '-'), and using the resultant label already assigned to the component in cell (i, j). This procedure (Algorithm 5.3) is the reverse of Algorithm 5.1 – we extract, from a labelled structure, the rules that would be required to produce it.

Algorithm 5.2 Rule Set from Labelled Structure

```
Input: Labelled structure $\bar{\mathcal{S}}$ and valid assembly ordering $(\mathcal{T}, \prec)$
Output: Rule set $\mathcal{R}$
 1: $\mathcal{R} \leftarrow \emptyset$
 2: for each cell $(i,j) \in \mathcal{T}$ do
 3:     for cell $(m,n)^d_{(i,j)}$ adjacent to $(i,j)$, $d \in \{n,e,s,w\}$ do
 4:         if $(m,n) \in \mathcal{T}$ and $(m,n) \prec (i,j)$ then
 5:             $l_d \leftarrow \bar{\mathcal{S}}(m,n)$
 6:         else
 7:             $l_d \leftarrow$ '_'
 8:         end if
 9:     end for
10:     $\mathcal{R} \leftarrow \mathcal{R} \cup (l_n, l_e, l_s, l_w, \bar{\mathcal{S}}(i,j))$    // add rule
11: end for
12: return $\mathcal{R}$
```

Theorem 5.2 Given a target structure $\mathcal{T}$ and a valid assembly ordering $(\mathcal{T}, \prec)$, Algorithm 5.3 generates a consistent rule set for $\mathcal{T}$ using exactly $|\mathcal{T}|$ labels. The algorithm can be implemented to run in $\Theta(|\mathcal{T}|)$ time.

Proof. We first note that, because each label is used only once, the binding configuration for every rule must be unique (and thus there can be no conflicting rules). Now, assume that the rule set $\mathcal{R}$ produced by Algorithm 5.3 is not consistent for $\mathcal{T}$. Then one of the following must be true: $\mathcal{R}$ permits the assembly ordering to be violated, adds a component at a position not in $\mathcal{T}$, or fails to add a required component to $\mathcal{T}$. We handle each case in turn.

Let $r_1 \in \mathcal{R}$ be a rule which is potentially applicable at a position $(i,j) \in \mathcal{T}$ and which would violate the assembly ordering (if applied). There must be another rule $r_2 \in \mathcal{R}$ which is also potentially applicable at (i,j) and does not violate the ordering; this rule is added on line 10 of Algorithm 5.2 when position (i,j) is considered. Since both r_1 and r_2 are potentially applicable at (i,j), their binding configurations must share at least one label in the same binding direction (n, e, s or w). However, every label in $\bar{\mathcal{S}}$ is unique, so the algorithm cannot add two rules with the same label in the same binding direction. This is a contradiction, and so none of the rules in $\mathcal{R}$ are able to violate the ordering, regardless of the exact sequence of actions by the assembly components.

Likewise, let $r \in \mathcal{R}$ be a rule which would add a component at a position $(m,n) \notin \mathcal{T}$; r must be potentially applicable at (m,n). Since r was added by Algorithm 5.2, there must be another cell $(p,q) \in \mathcal{T}$ where r is also potentially applicable. However, every label in $\bar{\mathcal{S}}$ is unique, so two rules cannot share the same label in the same binding direction, unless the assembly ordering is violated. We know from the result above that this is not possible – therefore we again have a contradiction and the algorithm does not add a rule which would permit the binding of a component at a position not in $\mathcal{T}$.

Algorithm 5.3 Worst-Case Rule Set

```
Input: Target structure $\mathcal{T}$ and valid assembly ordering $(\mathcal{T}, \prec)$
Output: Consistent rule set $\mathcal{R}$
1: $\bar{\mathcal{S}} \leftarrow$ structure produced by placing a uniquely-labelled component at each cell in $\mathcal{T}$
2: $\mathcal{R} \leftarrow$ rule set generated by Algorithm 5.2 from $(\bar{\mathcal{S}}, (\mathcal{T}, \prec))$
3: return $(\bar{\mathcal{S}}, \mathcal{R})$
```

Finally, assume that Algorithm 5.3 fails to add a rule that allows a component to bind at a position that *is* in $\mathcal{T}$. Clearly, the algorithm adds one rule to $\mathcal{R}$ for each cell in $\mathcal{T}$, with that rule's binding configuration defined by adjacent predecessors. It follows that there exists a rule that will add every component in $\mathcal{T}$ to the growing structure, exactly when the adjacent predecessors are in place. By definition, the algorithm uses exactly $|\mathcal{T}|$ labels.

To derive the time complexity, we note that it is possible to find the adjacent predecessors of a cell in constant time using, for example, an adjacency list representation of the target $\mathcal{T}$. The algorithm considers each cell in $\mathcal{T}$ exactly once, for an overall time that is in $\Theta(|\mathcal{T}|)$. The adjacency list can also be generated in a time that is linear in the size of $\mathcal{T}$, if necessary. $\square$

The drawback of Algorithm 5.3 is that $|\mathcal{T}|$ labels and $|\mathcal{T}| - 1$ rules are always required for a $|\mathcal{T}|$-cell target structure. We call such a $|\mathcal{T}|$-label rule set the *worst-case* rule set for $\mathcal{T}$, and use this worst-case solution as the initial input to our optimization algorithm.

5.3 The Minimum Label Set Problem

We have established that, given any structure for which a valid assembly ordering exists, we are able to generate a consistent rule set which assembles the structure. Depending on the size of the input structure, however, our worst-case algorithm may require a prohibitively large number of labels. We therefore pose the following combinatorial optimization problems.

Definition 5.3 The (Full) Minimum Label Set Problem: Given a target structure $\mathcal{T}$, find a valid assembly ordering $(\mathcal{T}, \prec)$ and rule set $\mathcal{R}$ such that $\mathcal{R}$ is consistent for $\mathcal{T}$ under $(\mathcal{T}, \prec)$ and uses the minimum number of unique labels possible.

Definition 5.4 The Restricted Minimum Label Set Problem: Given a target structure $\mathcal{T}$ and a valid assembly ordering $(\mathcal{T}, \prec)$, find a rule set $\mathcal{R}$ such that $\mathcal{R}$ is consistent for $\mathcal{T}$ under $(\mathcal{T}, \prec)$ and uses the minimum number of unique labels possible.

The optimization cost function in both cases is simply the number of unique labels that appear in the rule set. We do not know of a polynomial time algorithm to exactly solve either the MLS problem or the Restricted MLS problem; however, we have shown that the

Algorithm 5.4 Randomized Contraction

```
Input: Target structure $\mathcal{T}$ and valid assembly ordering $(\mathcal{T}, \prec)$
Output: Optimized, consistent rule set $\mathcal{R}$
1: $\mathcal{R} \leftarrow$ worst-case, $|\mathcal{T}|$-label rule set generated by Algorithm 5.3
2: $\bar{\mathcal{S}} \leftarrow$ complete, labelled structure generated by Algorithm 5.3
3: while termination condition not met do
4:   $\bar{\mathcal{S}}^{\dagger} \leftarrow \bar{\mathcal{S}}$
5:   $(i, j) \leftarrow$ randomly selected non-seed cell in $\mathcal{T}$
6:   $\bar{\mathcal{S}}^{\dagger}(i, j) \leftarrow$ randomly selected label from interval $[2, \bar{\mathcal{S}}(i, j) - 1]$
7:   $\mathcal{R}^* \leftarrow$ rule set generated by Algorithm 5.2 from $(\bar{\mathcal{S}}^{\dagger}, (\mathcal{T}, \prec))$
8:   if two rules in $\mathcal{R}^*$ conflict then
9:     prune rule with larger resultant label from $\mathcal{R}^*$
10:  end if
11:  $\bar{\mathcal{S}}^* \leftarrow$ labelled structure produced by applying $\mathcal{R}^*$
12:  if $\mathcal{R}^*$ is consistent for $\mathcal{T}$ according to Algorithm 5.1 then
13:    prune any unused rules from $\mathcal{R}^*$
14:    $\mathcal{R} \leftarrow \mathcal{R}^*$
15:    $\bar{\mathcal{S}} \leftarrow \bar{\mathcal{S}}^*$
16:    re-enumerate labels in $\mathcal{R}$ and $\bar{\mathcal{S}}$ sequentially from '1'   // remove gaps
17:  end if
18: end while
19: return $\mathcal{R}$
```

Restricted MLS problem is in NP (see Algorithm 5.1). The full MLS problem, in particular, has a solution space that includes all valid assembly orderings for a target structure; this space is usually much larger than for the restricted case (where the assembly ordering is a priori specified). Further, the interaction between the assembly ordering and the rule set is complex, with different assembly orderings requiring (sometimes substantially) different numbers of labels.

Instead, we propose an algorithm called *randomized contraction* as an approximate solution for the Restricted MLS problem. This approach, described by Algorithm 5.4, attempts to iteratively reduce the number of labels appearing in a rule set $\mathcal{R}$. Each successful reduction (removal or one or more labels in a single step) is termed a *contraction*. Starting with a valid assembly ordering, we generate a consistent, worst-case rule set $\mathcal{R}$ and the associated labelled structure $\bar{\mathcal{S}}$ using Algorithm 5.3. Then, at each iteration, we randomly select a component in $\bar{\mathcal{S}}$, change the label on the component (again, randomly), and generate an updated rule set $\mathcal{R}^*$. If $\mathcal{R}^*$ is consistent for $\mathcal{T}$, the change is accepted, otherwise the change is rejected. At the end of each iteration, we re-enumerate the label set sequentially from '1', to avoid leaving gaps between the label values. This process continues until a user-defined termination condition is met, for example when a certain number of iterations have been completed.

In many cases, changing the label on a component will cause two conflicting rules to appear in the updated rule set $\mathcal{R}^*$. If this occurs, one of the rules must be removed. We choose to prune the rule with the larger resultant label. Pruning will frequently introduce a discontinuity in the rule set, preventing a portion of the structure from being completed because a required rule is missing. However, for a target that contains two or more isomorphic substructures which share identical edge-oriented assembly graphs, the worst-case rule set always includes disjoint subsets of rules which assemble the same shape using different labels. Only one of these disjoint subsets is required. If a pruning-induced discontinuity occurs within an isomorphism, we can sometimes eliminate the redundant subset(s) of rules (and any unique labels they contain) immediately. This process occurs automatically when the consistency check for the updated rule set is performed (line 12). The ability to naturally exploit redundancy is one attractive property of the contraction algorithm – it is not necessary to exhaustively search for isomorphisms beforehand.

The complexity of Algorithm 5.4 is dominated by time required to verify that the updated rule set is consistent, and thus a single iteration can be implemented to run in $O(|\mathcal{R}||\mathcal{T}|)$ time using Algorithm 5.1, since $|\mathcal{R}|$ is always less than $|\mathcal{T}|$. This is an upper bound – in the later stages of optimization (when there are fewer successful label updates), the consistency check will often fail very quickly.

We show the relationship between the algorithms presented in Chapters 4 and 5 as a flowchart in Figure 5.1. The flowchart indicates the steps required to go from a target structure to an optimized rule set for that structure.

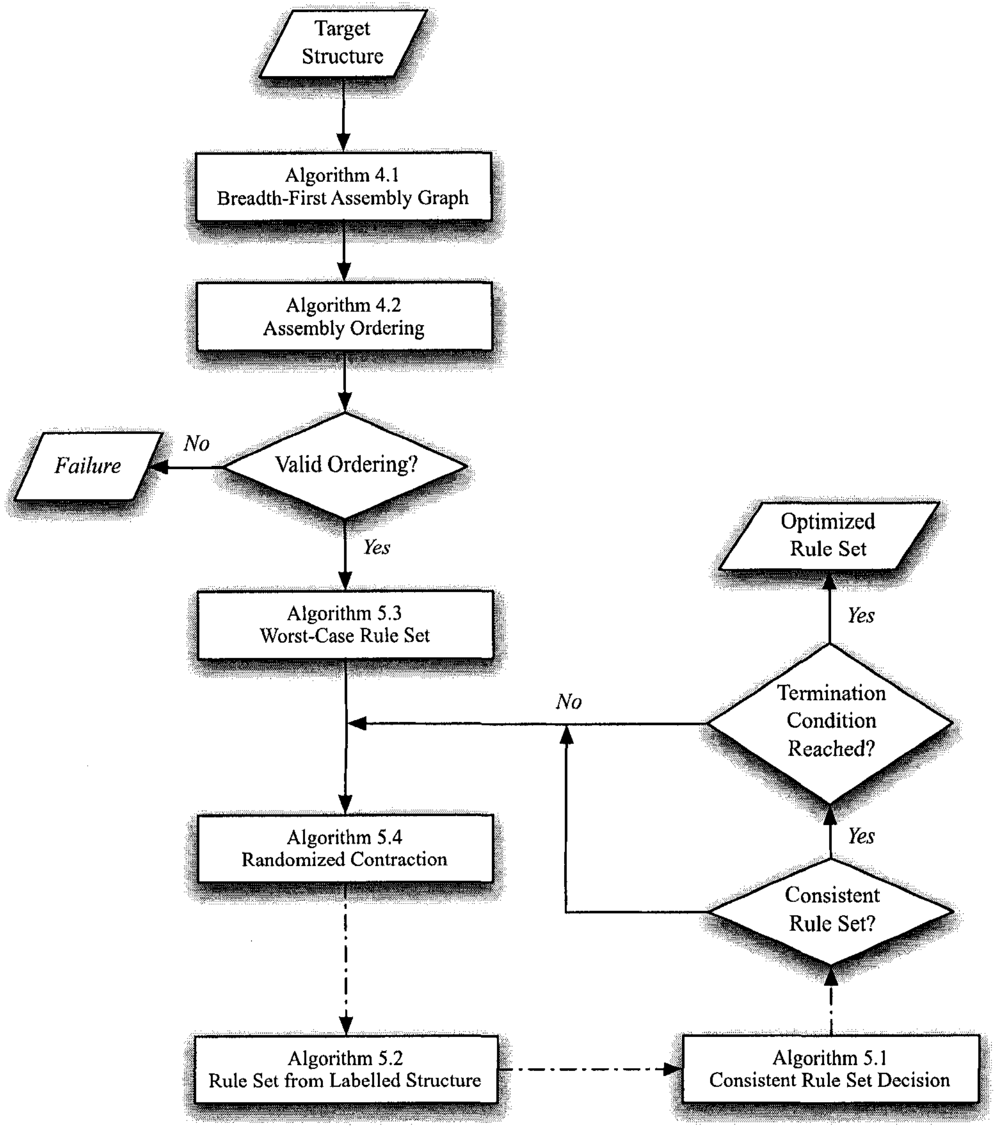

Target Structure
Algorithm 4.1 Breadth-First Assembly Graph
Algorithm 4.2 Assembly Ordering
Valid Ordering?
No
Failure
Yes
Algorithm 5.3 Worst-Case Rule Set
Algorithm 5.4 Randomized Contraction
Algorithm 5.2 Rule Set from Labelled Structure
Algorithm 5.1 Consistent Rule Set Decision
Consistent Rule Set?
Yes
Termination Condition Reached?
No
Yes
Optimized Rule Set

Figure 5.1: Algorithm flowchart, showing steps from (input) target structure to (output) optimized rule set. The dashed line from Algorithm 5.4 to Algorithm 5.2 and then to Algorithm 5.1 indicates that generating a rule set from the labelled structure and checking the rule set for consistency are both sub-steps in the randomized contraction algorithm.

Chapter 6

Experiments

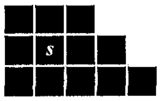
The previous three chapters form the main body of the thesis. We now present a series of experiments which characterize the performance of our randomized contraction algorithm. Our aim is to show that the algorithm is able to significantly reduce the number of labels required to assemble a variety of planar structures. To do so, we compare our optimization results with competing algorithms for a series of benchmark examples that have appeared previously in the literature. We also demonstrate that the contraction algorithm is able naturally to exploit redundancies due to structural isomorphisms and symmetries. We close the chapter with results for two larger and more complex structures.

6.1 Benchmarks

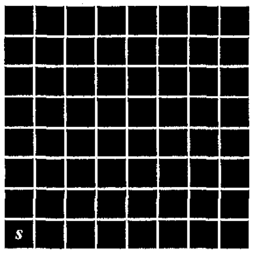
We have tested our optimization algorithm on the three benchmark structures that appear in [13] and that are examined in [14]. For these structures, our assembly model and the models presented in [13] and [14] are equivalent, and hence the performance of the corresponding algorithms may be compared directly, as is done in Table 6.1. Column eight of the table gives the number of iterations required to achieve our optimized result.

In all three cases, the randomized contraction algorithm generates rule sets which use fewer labels than the competing algorithms. For structures A (Figure 6.1(a)) and B (Figure 6.1(b)), it is possible to verify that our algorithm generates rule sets which are optimal by our criterion, using the minimum number of labels required to assemble these structures.[1] For structure A, we can prove that three labels are required by performing a simple exhaustive search, generating and testing rule sets for all possible two-labellings of the structure (of which there are only $2^{11} = 2048$, since the seed is always labelled '1'); all of the two-label rule sets fail our consistency check. We list the optimized, three-label rule set for structure A in Table 6.2.

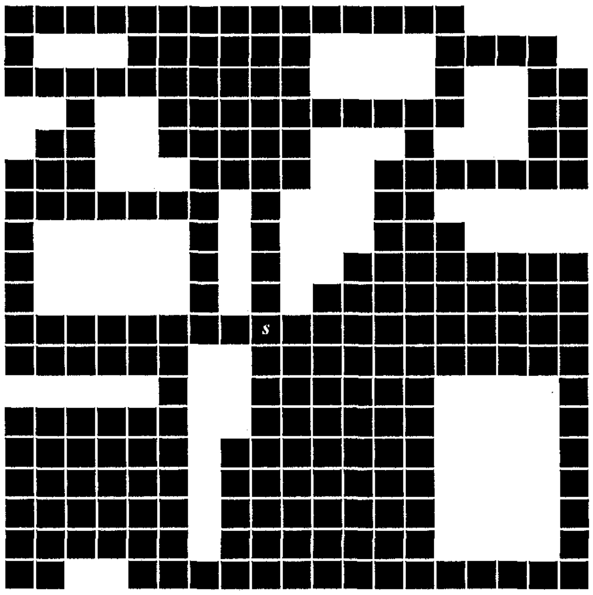
[1]That is, the minimum number of labels required for structures A and B, given the specific assembly orderings. We have exactly solved instances of the *restricted* minimum label set problem for these structures.



(a)



(b)



(c)

Figure 6.1: Benchmark structures from [13]. (a) Structure A, composed of 12 assembly components. (b) Structure B, composed of 64 assembly components. (c) Structure C, composed of 245 assembly components. The seed component in each structure is marked with an s.

Proving that the rule set for structure B uses the minimum number of labels is more complicated, because exhaustive search is infeasible. Our approach will be to show that components forming the left and bottom edges of the structure require unique labels which cannot appear at other positions. To begin, we will segment structure B as indicated in Figure 6.3(a), and describe the relationship between the components in each segment.

Consider the components in segments '1' and '2': each component has only a single predecessor and a single successor, according to the assembly graph in Figure 6.3(b). Assume that we attempt to use one of the labels from segment '1' or '2' for a component in segment '3'. Now assume that we choose a component in segment '3' with two adjacent predecessors (also in segment '3'), and assign a label from segment '1' or '2' to one of the predecessors. The rule set must contain a rule, used in segment '1' or '2', that *will now be potentially applicable* at a cell in segment '3'. Applying this rule in segment '3' will violate the assembly ordering, because the rule includes a single predecessor only. So we cannot reuse any of the labels from segment '1' or '2' for a component in segment '3' that has an adjacent successor. Likewise, we cannot reuse any of the labels from segment '1' or '2' for a component in segment '3' that lies on the exterior boundary of the structure – doing so could result in the addition of a component that is not part of the target. These two possibilities cover all of the positions in segment '3', and so segment '3' cannot share any labels with segments '1' or '2'.

Components in segments '1' and '2' also cannot share any labels with each other. Segment '1' grows upwards, while segment '2' grows to the left – if we were to use a label from segment '1' at a position in segment '2', then there would be a (potentially applicable) rule that would allow a component to bind at a position in segment '3', when only a single predecessor was in place. The same would be true if we were to use a label from segment '2' at a position in segment '1'. The assembly ordering would be violated in either case.

Finally, the label on every component in segment '1' must be unique; any duplication would result in a 'rule loop', causing the segment to grow indefinitely. The same argument holds for the components in segment '2'. Therefore, we require six unique labels for segment '1' and six unique labels for segment '2', as well as the seed label. The remainder of the structure must use at least one additional unique label. This is exactly the number of labels in the optimized solution returned by the randomized contraction algorithm (6+6+1+1 = 14 labels), and is therefore the minimum number possible.

For the largest and most complex structure, C (Figure 6.1(c)), the contraction algorithm reduces the number of labels required by 68% and 60%, compared to [13] and [14] respectively.[2] Although structure C contains several interior holes, we are still able to generate a valid assembly ordering using the breadth-first algorithm. The plot in Figure 6.7 gives the number of unique labels as a function of optimization iteration for the structure. More than 90% of the successful label set contractions occur within the first 5,000 iterations. In this case, we ran the contraction algorithm until there was no further improvement (reduction in the label count) for 5,000 consecutive iterations.

Table 6.1 also lists the number of assembly steps required for each structure. This metric, introduced in [13], is the number of steps required to assemble the entire target, starting with the seed only, if all binding sites on the frontier of the growing structure are filled synchronously at every step. The number of assembly steps corresponds to the length of the longest directed path in the assembly graph, which is independent of the number of labels used. Entries in each table column are identical because [13] and [14] implicitly use assembly graphs with the same path lengths as our algorithm for structures A, B and C.

6.2 Isomorphic and Symmetric Structures

The contraction algorithm is able to naturally exploit label redundancies due to structural isomorphisms and symmetries. As the following examples illustrate, it is not necessary to attempt to identify isomorphic or symmetric substructures before running the algorithm – the redundancies are automatically discovered as part of the optimization process.

If a structure contains two or more isomorphic substructures, as defined in Chapter 3, that share identical, edge-oriented assembly subgraphs, redundancies exists which can be

[2]We note, however, that the purpose of Jones and Matarićs' original work was to show that consistent rule sets could be generated, and not explicitly to consider optimizing those rule sets.

	Transition Rule Set		Rectangular Partitioning		Randomized Contraction		
Structure	Labels	Steps	Labels	Steps	Labels	Iterations	Steps
A (12 components)	5	4	9	4	**3**	97	4
B (64 components)	26	14	16	14	**14**	1,344	14
C (245 components)	165	18	131	18	**52**	36,910	18

Table 6.1: Comparison of the randomized contraction algorithm with the transition rule set compiler (Jones and Matarić) and the rectangular partitioning algorithm (Li and Zhang) for structures A, B and C given in [13].

used to reduce the label count. Consider structure D, shown in Figure 6.4, which contains three isomorphic substructures (the upright 'T' shapes). Components a, b, c and d in the structure will initially have different labels. Assume that, during an iteration of the optimization algorithm, the label on component b is randomly changed so that it is the same as the label on component a, while the labels on components c and d remain the same. The updated rule set $\mathcal{R}^*$, generated by Algorithm 5.2, will contain two conflicting rules for the (a, c) and (b, d) pairs. If we assume also that the label on component d is larger than the label on component c, Algorithm 5.4 will prune the second rule. This breaks the chain of rules that led to the attachment of component d. However, when the algorithm applies $\mathcal{R}^*$, it immediately discovers that the subset of rules used to assemble the left substructure, including components a and c, *can also be used to assemble the central substructure*, including components b and d. The redundant rules for the central substructure are removed before the start of the next iteration. Optimization results for the structure are shown in Table 6.3.

The plot in Figure 6.7 shows the number of unique labels as a function of optimization iteration for structure D. Large decreases in the label count on iterations 41 (13 labels eliminated) and 577 (8 labels eliminated) are due to the contraction algorithm's discovery of the isomorphisms and the consequent rule and label pruning. Figure 6.4(b) shows the initial, worst-case labelling for structure D, while Figure 6.4(c) shows the optimized labelling. In the optimized case, all three isomorphic substructures share the same labels. By eliminating redundancies that exists in the form of isomorphisms, the contraction algorithm is often able to reduce the number of labels significantly within a single iteration. This is one reason why the algorithm works backwards from a complete, labelled structure and rule set, instead of attempting to generate a new rule set from the bottom up.

Symmetric substructures (mirrored horizontally or vertically in the complete structure) are also naturally optimized. Structure E, shown in Figure 6.5(a), contains both horizontally and vertically symmetric substructures. Figure 6.5(b) shows the initial, worst-case labelling for the structure, and Figure 6.5(c) shows the optimized labelling. Note that, in

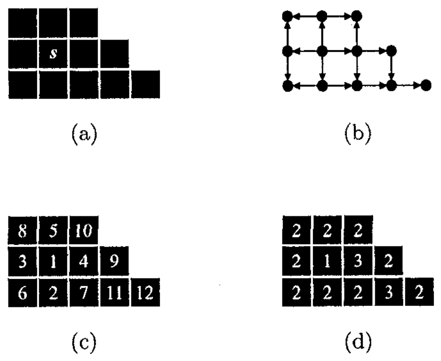


Figure 6.2: (a) Structure A, composed of 12 assembly components. The seed component is marked with an *s*. (b) Breadth-first assembly graph for structure A. (c) Worst-case labelling (12 labels). (d) An optimal labelling (given the assembly graph and associated ordering), generated by the randomized contraction algorithm (3 labels). The corresponding optimal rule set is listed in Table 6.2.

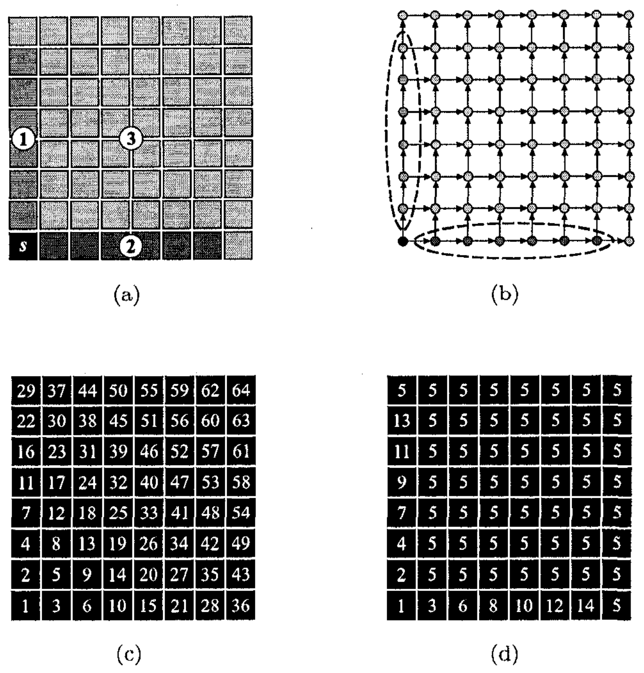


Figure 6.3: (a) Segmented view of structure B. Labels used to assemble sections '1' and '2' (medium grey and dark grey, respectively) cannot be used to assemble any portion of section '3' (light grey). (b) Breadth-first assembly graph for structure B. Vertices in sections '1' and '2' (highlighted by dashed ellipses) each have only a single adjacent predecessor; vertices in section '3' either have two adjacent predecessors or form part of the exterior boundary of the structure. (c) Worst-case labelling (64 labels). (d) An optimal labelling (given the assembly graph and associated ordering), generated by the randomized contraction algorithm (14 labels).

Rule Number	Expanded Format	Tuple Format
1	$\begin{pmatrix} & 1 & \\ - & \bullet & - \\ & - & \end{pmatrix} \Longrightarrow 2$	$(1,-,-,-,2)$
2	$\begin{pmatrix} & - & \\ - & \bullet & 1 \\ & - & \end{pmatrix} \Longrightarrow 2$	$(-,1,-,-,2)$
3	$\begin{pmatrix} & - & \\ - & \bullet & - \\ & 1 & \end{pmatrix} \Longrightarrow 2$	$(-,-,1,-,2)$
4	$\begin{pmatrix} & - & \\ 1 & \bullet & - \\ & - & \end{pmatrix} \Longrightarrow 3$	$(-,-,-,1,3)$
5	$\begin{pmatrix} & 2 & \\ - & \bullet & 2 \\ & - & \end{pmatrix} \Longrightarrow 2$	$(2,2,-,-,2)$
6	$\begin{pmatrix} & - & \\ - & \bullet & 2 \\ & 2 & \end{pmatrix} \Longrightarrow 2$	$(-,2,2,-,2)$
7	$\begin{pmatrix} & 2 & \\ 2 & \bullet & - \\ & - & \end{pmatrix} \Longrightarrow 3$	$(2,-,-,2,3)$
8	$\begin{pmatrix} & 3 & \\ 2 & \bullet & - \\ & - & \end{pmatrix} \Longrightarrow 2$	$(3,-,-,2,2)$
9	$\begin{pmatrix} & - & \\ 2 & \bullet & - \\ & 3 & \end{pmatrix} \Longrightarrow 2$	$(-,-,3,2,2)$
10	$\begin{pmatrix} & - & \\ 3 & \bullet & - \\ & - & \end{pmatrix} \Longrightarrow 2$	$(-,-,-,3,2)$

Table 6.2: Optimized rule set for benchmark structure A from [13]. The label set is $\mathcal{L} = \{1, 2, 3\}$. This is the minimum number of labels required to assemble A – no consistent rule set containing only two labels exists.

Structure	Labels	Steps	Iterations
D (64 components)	**17**	16	1,704
E (65 components)	**14**	7	2,845

Table 6.3: *Optimization results for isomorphic structure D and symmetric structure E.*

the optimized case, many labels are shared at symmetric positions. Optimization results for the structure are shown in Table 6.3.

Although the exact position at which certain labels appear depends on the (random) order of selections made by the contraction algorithm, most labels can be utilized at positions in two or more symmetric substructures. This is because the mirroring of the edge-oriented assembly subgraph often allows labels used by a rule in one direction (for example, north) to also be used in the opposite direction (for example, south). Discovering symmetry typically requires a larger number of optimization iterations, however, since, unlike for the isomorphic case, label values at the symmetric positions must be reduced incrementally.

6.3 Complex Structures

We now consider two examples of more complex structures, including one structure that contains an interior hole. Our goal in this section is to demonstrate that the randomized contraction algorithm is flexible and scalable. For the experiments presented below, we ran the contraction algorithm until there was no further improvement for 5,000 consecutive iterations.

6.3.1 Structures with Interior Holes

We choose to give a result for structure F, shown in Figure 6.8(a), which contains an interior hole. This structure originally appeared in [15], as an example of a case that could be problematic for purely local assembly algorithms. The authors of [15] suggest that the ability to enforce a global ordering on the assembly process is important for these types of structures, in which the exterior shell may close before the interior is complete.

It is true that a simple breadth-first assembly ordering fails to enforce all of the spatiotemporal constraints that are necessary to guarantee successful assembly of structure F. However, the assembly graph in Figure 6.8(b) does represents a valid ordering, and thus we are able to guarantee successful assembly using local rules only. We produced this graph by inspection, starting with the (invalid) graph generated by Algorithm 4.2. In the graph, there is a single, directed path which includes the vertices bordering the interior hole.[3] The

[3]We slightly abuse the notion of a graph here, by referring to the embedding of the graph in the plane (since vertices do not really 'border' the interior hole). We do so only for convenience in this case, to easily identify the changes made to the original graph produced by Algorithm 4.2.

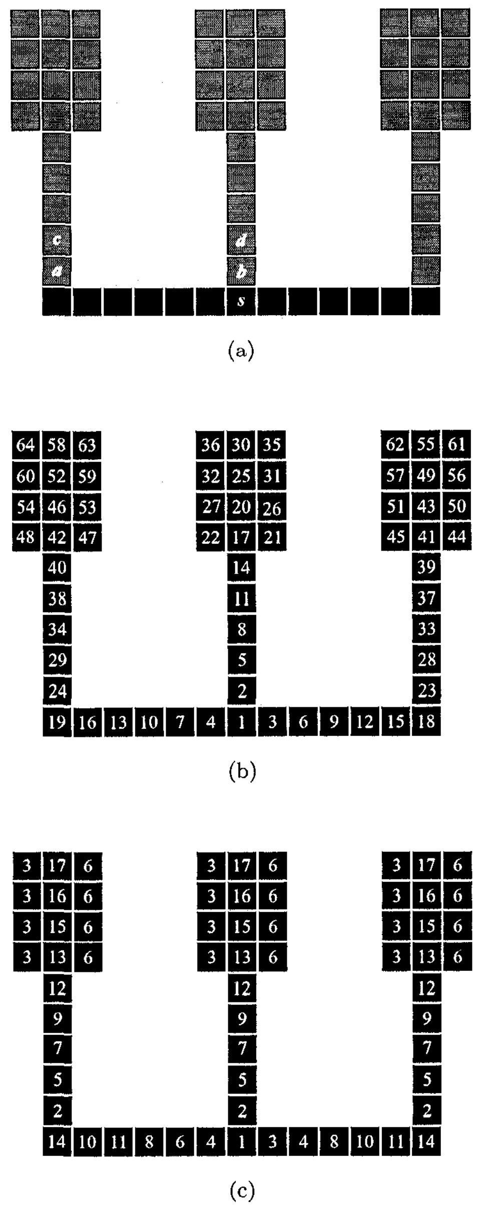


Figure 6.4: (a) Structure D, containing three isomorphic substructures, coloured medium grey. The seed component is marked with an *s*. (b) Worst-case labelling (64 labels). (c) Optimized labelling generated by the randomized contraction algorithm (17 labels). Note that all three isomorphic substructures share the same labelling.

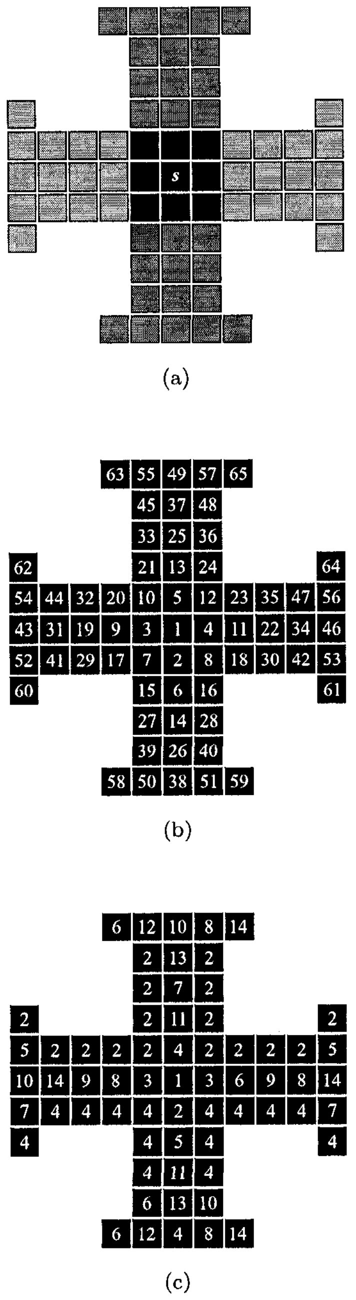


Figure 6.5: (a) Structure E, containing horizontally (light grey) and vertically (medium grey) symmetric substructures. The seed component is marked with an *s*. (b) Worst-case labelling (65 labels). (c) Optimized labelling generated by the Randomized Contraction algorithm (14 labels). Note that labels are often shared at symmetric positions.

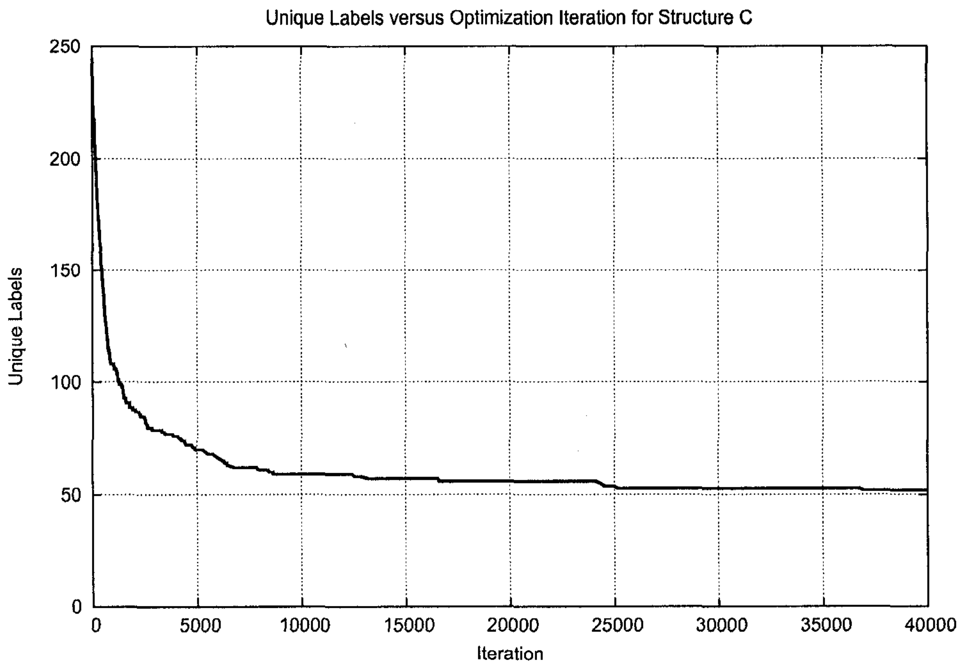


Figure 6.6: Unique labels versus optimization iteration for structure C, shown in Figure 6.1(c). More than 90% of the successful optimization steps are performed within the first 5,000 iterations.

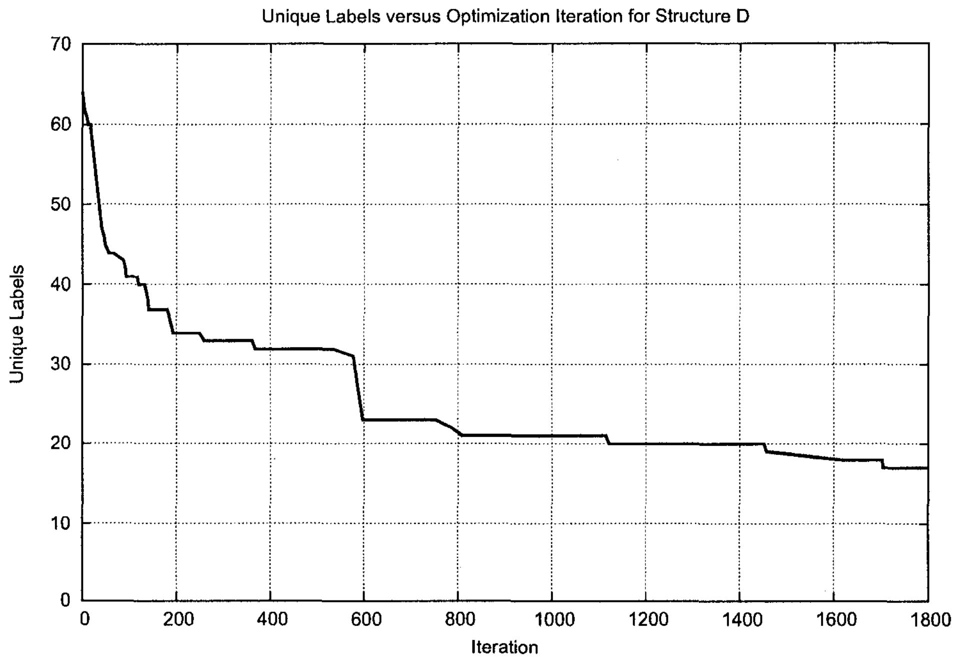


Figure 6.7: Unique labels versus optimization iteration for structure D, shown in Figure 6.4. Large decreases in the label count on iterations 41 and 577 are due to the discovery of the isomorphisms and the consequent rule and label pruning.

Structure	Labels	Steps	Iterations
F (162 components)	**51**	56	8,701
G (1,045 components)	**120**	121	83,838

Table 6.4: Optimization results for complex structures F and G.

existence of this path ensures that the exterior of the structure cannot close before the all of the interior components are in place. Our only modification to the graph produced by the breadth-first algorithm was to reorient 19 edges around the hole border, to produce the longer interior path. Also, we intentionally positioned the seed component as part of the exterior boundary, in order to make the problem more difficult. Optimization results are given in Table 6.4; the rule set contains 51 unique labels.

Our point here is to emphasize that there is a large class of structures which appear, at first glance, to require a *globally*-enforced assembly ordering, but which can in fact be assembled using local rules.

6.3.2 Larger Structures

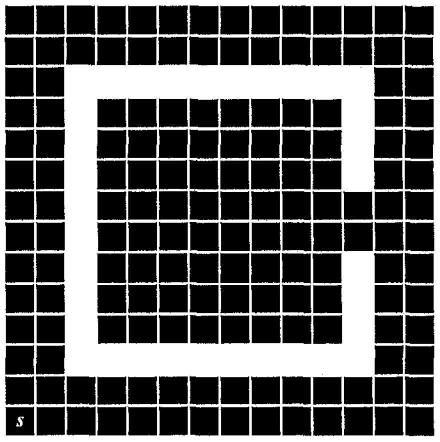
We give one final result, for a structure composed of more than one thousand individual components. Figure 6.9 shows a 1,045-component structure which can be assembled with an optimized rule set containing only 120 unique labels (Table 6.4). This is an 88.5% improvement over the worst-case label count, and an average of just 0.11 labels per component. Generating highly optimized rule sets for larger structures does require a comparative increase in the number of iterations of the contraction algorithm – a total of 83,838 iterations were required in this case.

6.4 Summary

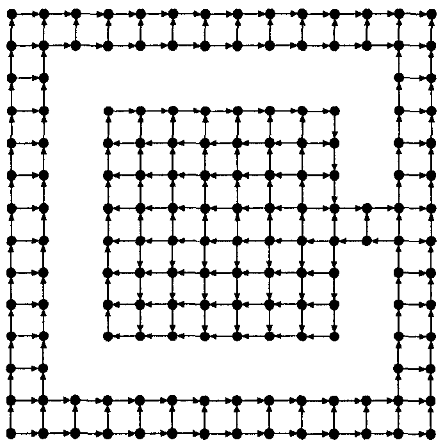
In this chapter, we examined the performance of our randomized contraction algorithm, for a variety of planar structures. We showed that for the benchmark structures A, B and C, the contraction algorithm generates rule sets which use fewer labels than competing algorithms. We also proved that the rule sets for structures A and B are optimal by our criterion, using the minimum number of labels possible.

For structures C through F, we are not able to determine the true minimum number of labels directly, and so it is more difficult to quantify the performance of the algorithm. However, we saw that in all these cases, significant reductions (of at least 68%) from the worst-case label counts were attained.

Our results for structures E and F demonstrated that the contraction algorithm is able to naturally exploit redundancies due to isomorphisms and symmetries, without the need to identify such substructures in advance. The result for structure G established that the algorithm is scalable, reducing the number of labels by more than 88% over the worst-case for this large structure.



(a)



(b)

Figure 6.8: (a) Structure F, composed of 162 assembly components. The seed component is marked with an *s*. (b) Assembly graph generated using the breadth-first ordering algorithm, followed by adjustment by inspection; the breath-first algorithm alone does not produce a valid ordering for this structure. Note that there is a single directed path which includes all vertices in the interior region that border the interior hole.

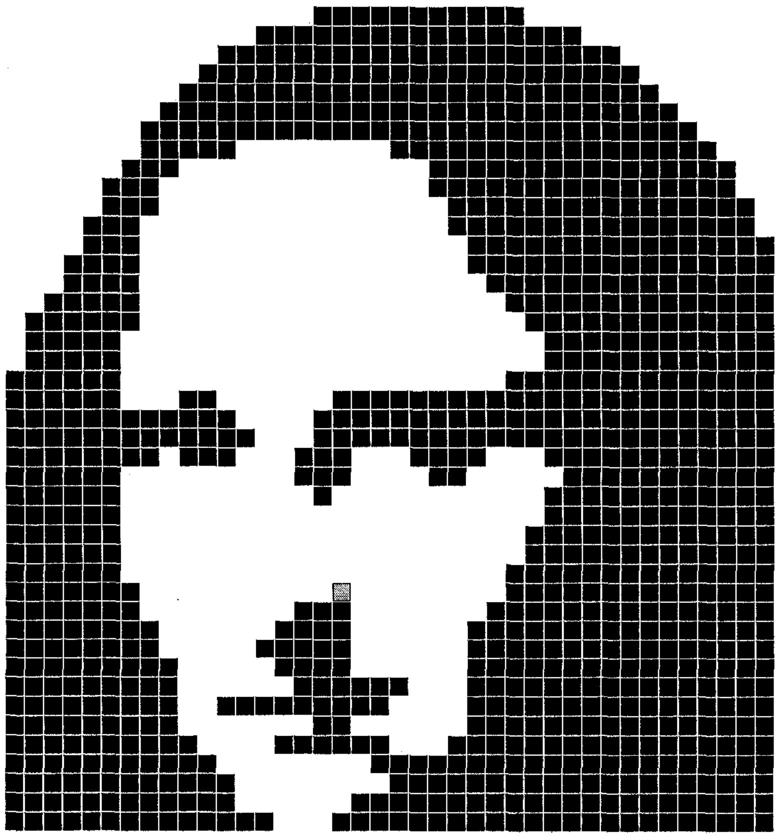


Figure 6.9: Structure G, Leonardo da Vinci's Mona Lisa, composed of 1,045 assembly components. The structure can be assembled using a rule set containing only 120 unique labels. The seed component is shown in light grey, near the centre of the figure.

Chapter 7

Conclusions and Future Work

This thesis explored distributed assembly from an algorithmic perspective. We presented a discretized, grid-based model for the assembly of planar structures by simple, reactive robots (assembly components). We then defined the spatiotemporal ordering constraints that must be satisfied to ensure coordinated assembly of a structure, and gave a procedure for encoding these constraints in a set of *local* assembly rules. We showed that it is possible to *guarantee* successful assembly using such a rule set, and in doing so solved an instance of the local-to-global problem introduced in Chapter 1.

Our final contribution was a randomized optimization algorithm for reducing the number of labels that appear in an assembly rule set, or equivalently for reducing the sensing demands placed on the assembly components. To characterize the performance of the optimization algorithm, we presented results from a series of experiments with a variety of simple and complex planar structures. These results demonstrated that our approach outperforms other algorithms found in the literature.

Our work is a further step towards developing a theoretical understanding of distributed assembly. Ultimately, we hope that our contributions will advance efforts to harness these processes and enable the synthesis of specific structures in an exact and reliable way.

There are a number of directions for future research. We restricted our discussion to planar structures, however our assembly model can readily be extended to three dimensions, simply by expanding the components' local sensing neighbourhood to include cells along a third axis. We have already begun to develop modified three-dimensional ordering and label optimization algorithms.

Our ordering algorithm generates valid outputs only when the breadth-first traversal of a structure's adjacency graph does not produce any interior, sink vertices. This means that the algorithm will not work for certain structures that contain interior holes with non-convex boundaries. We showed in Chapter 6 that it is sometimes possible to generate valid orderings by direct inspection. However, we believe that there is a complete, polynomial-time algorithm which will generate valid orderings for arbitrary structures. The algorithm should operate on the adjacency graph alone, once the exterior boundary vertices have been identified. That is, there is sufficient information contained in the (topological) adjacency graph to define the necessary ordering constraints, without considering the geometric embedding of the graph. We have developed a graph search algorithm (omitted from the thesis) to solve this general problem, however we have not yet verified its completeness.

An additional, important research problem is to show that the Minimum Label Set problem is NP-complete. We gave a polynomial time algorithm for verifying whether a rule set, which uses a certain number of labels, is consistent for a particular structure under a specific assembly ordering, and thus showed that the restricted MLS problem is in NP. Our conjecture is that a reduction from Adleman's Minimum Tile Types problem can be used to show NP-completeness in the general case, but this result needs to be proved.

In a broader context, we believe that there are many results to be obtained by examining distributed assembly from an information-theoretic perspective. We saw in Chapters 4 and 5 that information about the state of the assembly process must propagate through a structure sequentially over time. In most situations, propagation is unlikely to be error-free – this is analogous to data transmission over a noisy communications channel, and can perhaps be modelled as such. Further, the Minimum Label Set problem is essentially a compression problem, where we seek to remove any redundancies in the label set; the MLS problem may therefore benefit from an information-theoretic analysis.

Finally, related to the above is the problem of determining the extent to which abstract models, such as the one presented here, can be used to describe real systems. Physical implementations must work in continuous (rather than discrete) environments, and must deal with noise, contamination and assembly errors. At present, only a handful of attempts have been made to build physical systems which are adequately described by these simple models. It is also not clear what level of fidelity is required from the models as we scale down to smaller dimensions (e.g. millimeters to nanometers) and up to thousands or tens of thousands of components.

The study of distributed assembly is still in its infancy – many interesting research problems remain to be solved. We believe there is much to be gained, both theoretically and pragmatically, by understanding how to engineer these systems. In fact, we expect that, with this understanding, distributed assembly will become a principal way in which many artifacts are built in the not-too-distant future.

Bibliography

[1] A. A. G. Requicha, "Nanorobots, NEMS, and nanoassembly," *Proceedings of the IEEE*, vol. 91, no. 11, pp. 1922–1933, November 2003.

[2] G. M. Whitesides and M. Boncheva, "Beyond molecules: Self-assembly of mesoscopic and macroscopic components," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 99, no. 8, pp. 4769–4774, April 2002.

[3] H. Wang, "Proving Theorems by Pattern Recognition II," *Bell Systems Techinical Journal*, no. 40, pp. 1–42, 1961.

[4] L. Adleman, "Towards a Mathematical Theory of Self-Assembly," Department of Computer Science, University of Southern California, Los Angeles, USA, Tech. Rep. 00-722, January 2000.

[5] P. Rothemund and E. Winfree, "The Program-size Complexity of Self-assembled Squares," in *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC'00)*, Portland, Oregon, USA, May 2000, pp. 459–468.

[6] L. Adleman, Q. Cheng, A. Goel, and M.-D. Huang, "Running Time and Program Size for Self-assembled Squares," in *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC'01)*, Hersonissos, Greece, July 2001, pp. 740–748.

[7] L. Adleman, Q. Cheng, A. Goel, M.-D. Huang, D. Kempe, P. M. de Espanés, and P. W. K. Rothemund, "Combinatorial Optimization Problems in Self-Assembly," in *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing (STOC'02)*, Montréal, Canada, May 2002, pp. 23–32.

[8] K. Saitou, "Self-Assembling Automata: A Model of Conformational Self-Assembly," in *Proceeding of Pacific Symposium on Biocomputing*, Maui, Hawaii, USA, January 1998, pp. 609–620.

[9] E. Klavins, "Directed Self-Assembly Using Graph Grammars," in *Foundations of Nanoscience: Self Assembled Architectures and Devices*, Snowbird, Utah, USA, April 2004.

[10] R. Ghrist and D. Lipsky, "Grammatical Self Assembly for Planar Tiles," in *Proceedings of the IEEE International Conference on MEMS, NANO, and Smart Systems (ICMENS'04)*, Banff, Canada, August 2004, pp. 205–211.

[11] E. Klavins, R. Ghrist, and D. Lipsky, "Graph Grammars for Self Assembling Robotic Systems," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'04)*, New Orleans, Louisianna, USA, April 2004, pp. 5293–5300.

[12] Y. Guo, G. Poulton, P. Valencia, and G. James, "Designing Self-Assembly for 2-Dimensional Building Blocks," in *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering, LNAI 2977*, G. D. M. S. et al., Ed. Springer Verlag, February 2004, pp. 75–89.

[13] C. V. Jones and M. J. Matarić, "From Local to Global Behavior in Intelligent Self-Assembly," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'03)*, Taipei, Taiwan, September 2003, pp. 721–726.

[14] G. Li and H. Zhang, "A Rectangular Partition Algorithm for Planar Self-Assembly," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05)*, Edmonton, Canada, August 2005, pp. 2324–2329.

[15] D. Arbuckle and A. A. G. Requicha, "Active Self-Assembly," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'04)*, vol. 1, New Orleans, USA, April 2004, pp. 896–901.

[16] ——, "Shape Restoration by Active Self-Assembly," in *Proceedings of the International Symposium on Robotics and Automation (ISRA'04)*, Queretaro, Mexico, August 2004.

[17] ——, "Self-repairing Self-assembled Structures," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'06)*, Orlando, Florida, USA, May 2006, pp. 4288–4290.

[18] A. Kondacs, "Biologically-Inspired Self-Assembly of Two-Dimensional Shapes Using Global-to-Local Compilation," in *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI '03)*, Acapulco, Mexico, August 2003, pp. 633–638.

[19] P. W. K. Rothemund, "Using lateral capillary forces to compute by self-assembly," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 97, no. 3, pp. 984–989, February 2000.

[20] J. H. Reif, T. LaBean, S. Sahu, H. Yan, and P. Yin, "Design, Simulation, and Experimental Demonstration of Self-Assembled DNA Nanostructures and DNA Motors," in *Computational Modeling and Simulation of Materials Conference (CIMTEC'04)*, Acireale, Sicily, Italy, 2004.

[21] J. Bishop, S. Burden, E. Klavins, R. Kreisberg, W. Malone, N. Napp, and T. Nguyen, "Programmable Parts: A Demonstration of the Grammatical Approach to Self-Organization," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05)*, Edmonton, Canada, August 2005, pp. 3684–3691.

[22] S. Burden, N. Napp, and E. Klavins, "The Statistical Dynamics of Programmed Robotic Self-Assembly," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'06)*, Orlando, Florida, USA, May 2006, pp. 1469–1476.

[23] R. A. Brooks, "Intelligence without representation," *Artificial Intelligence*, no. 47, pp. 139–159, 1991.

[24] C. R. Kube and H. Zhang, "Collective Robotic Intelligence," in *Proceedings of the Second International Conference on the Simulation of Adaptive Behaviour (SAB'92)*, J.-A. Meyer, H. L. Roitblat, and S. W. Wilson, Eds. Cambridge, Massachusetts, USA: MIT Press, 1993, pp. 460–468.

[25] ——, "Collective Robotics: From Social Insects to Robots," *Adaptive Behaviour*, vol. 2, no. 2, pp. 189–219, 1993.

[26] M. A. Lewis and G. A. Bekey, "The Behavioral Self-organization of Nanorobots Using Local Rules," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'92)*, vol. 2, July 1992, pp. 1333–1338.

[27] W.-M. Shen, C.-M. Chuong, and P. Will, "Simulating Self-Organization for Multi-Robot Systems," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'02)*, vol. 3, Lausanne, Switzerland, October 2002, pp. 2776–2781.

[28] J. Kelly and H. Zhang, "Combinatorial Optimization of Sensing for Rule-Based Planar Distributed Assembly," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'06)*, Beijing, China, October 2006, pp. 3728–3734.

[29] G. Theraulaz and E. Bonabeau, "A Brief History of Stigmergy," *Artificial Life*, vol. 5, no. 2, pp. 97–116, 1999.

[30] R. Beckers, O. E. Holland, and J. L. Deneubourg, "From Local Actions to Global Tasks: Stigmergy and Collective Robotics," in *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, R. A. Brooks and P. Maes, Eds., Artificial Life IV. Cambridge, Massachusetts, USA: MIT Press, 1994, pp. 181–189.

[31] Z. Mason, "Programming with Stigmergy: Using Swarms for Construction," in *Proceedings of the Eighth International Conference on Artificial Life (ICAL'02)*, Sydney, Australia, December 2002, pp. 371–374.

[32] E. Bonabeau, G. Theraulaz, E. Arpin, and E. Sardet, "The Building Behavior of Lattice Swarms," in *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, R. A. Brooks and P. Maes, Eds., Artificial Life IV. Cambridge, Massachusetts, USA: MIT Press, 1994, pp. 307–312.

[33] G. Theraulaz and E. Bonabeau, "Coordination in Distributed Building," *Science*, vol. 269, no. 5224, pp. 686–688, August 1994.

[34] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, 1999.

[35] E. Bonabeau, S. Guéin, D. Snyers, P. Kuntz, and G. Theraulaz, "Three-dimensional architecture grown by simple 'stigmergic' agents," *Biosystems*, vol. 56, no. 1, pp. 13–32, 2000.

[36] J. Werfel, Y. Bar-Yam, and R. Nagpal, "Building Patterned Structures with Robot Swarms," in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI '05)*, Edinburgh, Scotland, August 2005, pp. 1495–1502.

[37] J. Wawerla, G. S. Sukhatme, and M. J. Matarić, "Collective Construction with Multiple Robots," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'02)*, vol. 3, Lausanne, Switzerland, October 2002.

[38] J. Werfel, Y. Bar-Yam, D. Rus, and R. Nagpal, "Distributed Construction by Mobile Robots with Enhanced Building Blocks," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'06)*, Orlando, Florida, USA, May 2006, pp. 2787–2794.

[39] C. Parker, H. Zhang, and C. R. Kube, "Blind Bulldozing: Multiple Robot Nest Construction," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'3)*, vol. 2, Las Vegas, Nevada, USA, October 2003, pp. 2010–2015.

[40] D. B. West, *Introduction to Graph Theory*, 2nd ed. Prentice Hall, 2000.

[41] S. S. Epp, *Discrete Mathematics with Applications*, 3rd ed. Brooks Cole, December 2003.