SCALABLE COMPILER FOR TERMES DISTRIBUTED ASSEMBLY SYSTEM

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

M.S.

by

Yawen Deng

May 2018

ABSTRACT

The TERMES system is a robot collective capable of autonomously constructing user-specified structures in three dimensions. The compiler is one of the key components that convert the goal structure into a directed map which an arbitrary number of robots can follow to perform decentralized construction. In previous work, the compiler was limited to brute force search which scales poorly with the size of structure. The purpose of this research is to enhance the scalability of the compiler so that it can be applied to very large-scale structures. Correspondingly, a new scalable compiler is presented, with the ability to generate directed maps for structures with up to 1 million stack of bricks. We further recast the old compiler as a constraint satisfaction problem and compare their performance on a range of structures. Results show that the new compiler has significant advantages over the old compiler as the size and complexity of the structures increase. We further developed an automated scheme for improving the transition probability between neighboring stack of bricks for efficient construction. This work represents an important step towards real-world deployment of robot collectives for construction.

BIOGRAPHICAL SKETCH

Yawen Deng was born in Taiyuan, China. After completing her schoolwork at Taiyuan Foreign Language School in Taiyuan in 2012, Yawen entered Zhejiang University in Hangzhou, China. During the summer of 2015, she attended University of California, Davis. She then went to Harvard University as a visiting student for 8 months. She received Bachelor of Engineering with a major in Energy and Environmental System Engineering in May 2016. In August 2016, She entered the mechanical engineering master of science program at Cornell University.

ACKNOWLEDGEMENTS

This project would not have been possible without the support of many people. Many thanks to my adviser, Kirstin H. Petersen, who always gives me advice whenever I had questions about my research or writing. Also thanks to my committee member, Ross A. Knepper, who offered guidance and support to my research. I would also like to thank professor Nils Napp from University at Buffalo, for his invaluable feedback in computer graphic and python implementation. Thanks to my teammate Yiwen Hua for being my teammate in TERMES project and contribute my thesis. Thanks all the members in the Cornell Collective Embedded Intelligence Lab for their support and help.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Autonomous robots have the potential to revolutionize the industry by performing high-precision, repetitive, and predictable tasks. Unlike human beings, autonomous robots do not experience fatigue and inaccuracies. Furthermore, the use of autonomous robots can improve the safety of workers and reduce accidents. Robots have the potential to work in extreme environment such as underwater, underground, the outer space, earthquake, and more. The focus here is on construction, where autonomous robots may enable rapid fabrication of inexpensive and potentially novel types of structures outside in highly unstructured environments.

1.1 Motivation

With expenditures reaching over 1,162 billion U.S. dollars, the United States is one of the largest construction markets worldwide[15]. The use of robots in the construction process may greatly minimize mistakes, which will further reduce the cost in terms of money and time. According to the UN habitat, about 1.6 billion people currently lack adequate housing and more than 60 million people flee their homes annually due to war, famine, etc[6]. Automated construction promises to lower the cost of construction and may lead to more affordable housing options. Up to 20% of worker injuries come from the construction sector[2]. By introducing robots to take over heavy loads and dangerous tasks, the health and safety of workers may be improved. It is predicted that by 2050 about 64% of the developing world and 86% of the developed world will be urbanized[7]. The demand of new city construction will exceed the amount of

human resources available and robots have the potential to fill this growing labor gap.

Collective of robots have the ability to efficiently assemble structures that are much larger than the size of the individuals. Robot collectives may also be tolerant to individual failures. A great natural example of how such collectives may operate include social insects such as ants, bees, and termites. These swarms are highly organized and can have millions of individuals. Each individual only has limited capacity, however, by working together they can achieve tasks beyond the reach of any individual. Although such robot collectives and swarms have received a lot of attention over the past few years, most demonstrations are limited to relatively small-scale assemblies and/or small scale collectives.

1.2 TERMES System

South African mound-building termites live in colonies with millions of workers and are capable of building mounds which are orders of magnitude larger than themselves. Inspired by termites, the TERMES system [11] consists of large collectives of robots capable of automated construction. The system comprises mobile robots and specialized passive building blocks; the robot can autonomously manipulate the blocks, build structures with them, and maneuver on these structures in 2.5D.

Fig.1.1 shows the pipeline of TERMES system. First, a user specifies a desired structure as well as start and exit site(s). A start site is where robots can enter the structure. An exit site is where the robots can leave the structure. Next, the off-line compiler generates traffic patterns that robots can follow to build

the structure with provable guarantees. Next, the system assigns an arbitrary number of robots for building activity. Robots attach bricks according to a rule set that guarantee no gaps which cannot be filled in later, and no cliffs which would leave the structure intraversable by future robots. The final step is to let the robots perform decentralized construction.

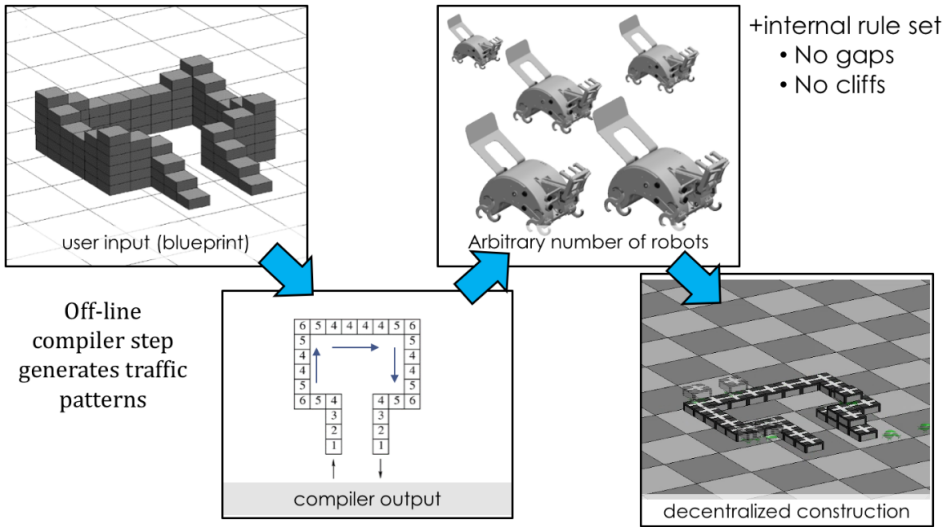


Figure 1.1: TERMES Pipeline. Adapted from "Distributed Multi-Robot Algorithms for the TERMES 3D Collective Construction System." by Justin Werfel, Kirstin Petersen, and Radhika Nagpal, *The International Journal of Robotics Research*, 27(3-4):463-479, 2008

The original TERMES system is a successful case of decentralized control and collective construction. However, it largely overlooked the performance of the path compiler. The old compiler was limited to brute force search which scales poorly with the size of structure making it very computationally expensive to generate paths for large-scale structures (even the size of a normal house), let alone check if large structures can be built. The motivation of this research is to enhance the scalability of the compiler so that it can be applied to very large-scale structures.

In addition to the compiler, robots have a uniform probability of choosing

any of the traversable neighboring sites in the structure, which leads to wasted trips where the robots exit the structure without finding a place to add a brick. In this research, we aim to optimize the transition probability for efficient construction.

Figure 1.2: TERMES robots and bricks. Adapted from [17]

1.3 Thesis Outline

Chapter 2 reviews related works of automated construction with special focus on collective construction robots in both research and industry area.

Chapter 3 introduces algorithmic framework of TERMES system. we describe the system pipeline and provide details of the compiler algorithms, robot building rules, and the challenges. A detailed problem formulation is also pro-

vided in this chapter.

Chapter 4 introduces the Constraint Satisfaction Problem (CSP). We recast the original compiler into a constraint satisfaction problem, and show a number of ways in which it can be tweaked to behave more efficiently.

Chapter 5 presents the Breadth First Disassembly (BFD) compiler, which is our main contribution, and a proof of correctness.

Chapter 6 shows the performance and evaluation of the BFD compiler. We implement BFD compiler, edge-CSP compiler, and location-CSP compiler in python and test them on a range of structures. We use three types of structures scaled in size.

Chapter 7 then discusses how the resulting directed map can be optimized for faster construction. Specifically, we alter the transition probability between neighboring stacks of bricks to efficiently guide the robots over the structure.

Chapter 8 summarizes and concludes this thesis.

1.4 Contribution

In this research, we focus on enhancing the scalability of TERMES compiler so that it is be feasible to compile paths for human-scale structures. The previous TERMES paper uses a search algorithm to assign travel directions, which we recast as a backtracking solution to a CSP with pairwise, partial, and global constraint checking. We show that, worst case, the time complexity of this search is $O(2^n)$, where n is the number of edges to be assigned. Although this may work

well on structures with many solutions, it will perform very poorly on structures with only a few or no solutions. We show simple methods for incremental improvements to this scheme and then implement a completely new compiler that works vastly better than the CSP. The new compiler is not based on search, but instead builds up a feasible solution in a breadth-first manner from all exits. We show that the new compiler has an average and worst-case time complexity of $O(n^2)$. To compare the compilers mentioned above, we implement them in Python, run them on the same computer, and evaluate their performance by runtime on a range of buildable/non-buildable structures. In addition to the compiler, transition probabilities between adjacent locations are altered for efficient construction; the efficiency is structure dependent, but we show a simple case in which the number of necessary steps taken by the robots is decreased by more than 30 times.

CHAPTER 2

RELATED WORK

Research in automated construction started in the 1980s and has evolved into two major thrusts: stationary and mobile assembly robots. This chapter focus on the mobile robots, with special attention on collective robots. The follow sections review industry application, scientific research of automated construction and related research on assembly compilers.

2.1 Industry Application

In construction, automation is challenging since the construction site and goal structure changes from day-to-day. Many companies use automated robots for onsite construction, but usually focus on short-term goals and explore practical options.

Self-driving cars have gained a lot of attention during the past few years and this application extends to construction. It requires a 3D map of the current site created by a laser scanner, drones, etc. Engineers plan the construction process based on the 3D map. By combining the real-time 3D map, the construction plan, and the location of the robots, the controller assigns tasks to each robot. Built Robotics developed such self-driving excavators for construction sites and a fully autonomous skid steer that can be remotely directed via a computer program to move around dirt. Komatsu[5] introduced their Intelligent Machine Control (IMC) with a D61i-23 dozer. The blade is automatically controlled according to 3D CAD construction data with the coordinates computed from design drawings. Caterpillar[1] developed a CAT Command technology

for addition to their mining equipment. This makes them able to do manual or semi-autonomous operation of their dozing and underground loading equipment, as well as complete autonomous operation of their haul trucks.

When it comes to bricklaying robots, Construction Robotics[3] designed a semi-automated mason (SAM) which is a combination of a conveyor belt, robotic arm, and concrete pump. It has the ability of placing between 300 and 400 bricks an hour. Fastbrick[4] robotics developed Hadrian X to complete brickwork at a lower cost and higher quality. This robot can handle different brick sizes as well as cut, grind, mill, and route the bricks to fit the structure before laying them down.

Such industrial robots work well for onsite construction, but either require human intervention or depend heavily on a single robot system which represents a single point of failure. The following section describe systems of many cooperating simple robots that have an easier time adapting to the circumstances at hand.

2.2 Scientific Research

Collective construction is the term used when more than one robot works together to complete the structure. Compared with stationary assembly, collective construction offers better parallelism, scalability, and error tolerance. Collective robots usually do not have pre-programmed actions or pre-assigned locations.

Researchers have implemented collective construction robots in many different ways. There are two major categories: ground robots and aerial robots.

Aerial robots are not limited by ground constraints. They can operate dynamically in space and assemble directly in the required position. Willmann, et al.[18] developed flying robots that lift building elements and drop them according to blueprint. Similarly, Lindsey et al.[9] proposed a system that in which quadrotor helicopters autonomously assemble 2.5D truss-like structures. However, to date, the majority of aerial robots still require a centralized controller and complete global sensing, typically provided by motion capture systems. Centralized controllers scale poorly with the size of the collective and represent a single point of failure. Further, it is difficult to set up full sensing and communication in every new construction site.

In contrast to aerial robots, ground robots are stable and capable of carrying heavy objects. Many such systems have been presented, ranging from complicated robots working in dedicated teams to do more complex assemblies, to simple robots building simple structures. A representative example of the former includes the work of Knepper et al.[8], where rather expensive robots works together in small teams to assemble furniture kits from IKEA. Knepper developed a geometric reasoning system that was capable of discovering the correct arrangement without knowing the final shape. There are also many examples of the latter. The Automatic Modular Assembly System(AMAS)[16] was one of the first such systems consisting of passive building blocks and assembler robot. Algorithms with large collectives were shown extensively in simulation, and a few, very simple robots and blocks were physically demonstrated. Rubenstein et al.[12] created a thousand-robot swarm that could form 2D preprogrammed shapes out of their own bodies. They developed a collective algorithm which was highly robust to robot variability and other typical error characteristics which occur in large-scale decentralized systems. coordination

of distributed construction robots have been achieved through gradients[16], [13], templates[10], and a combination of offline compilers and rule sets[17].

The TERMES system, which is the focus of this thesis, stands in contrast to these systems by focusing on simple climbing robots capable of building exact structures in 2.5D out of dedicated building material. It takes user-specified structures and generate low-level rules for robots to coordinate their construction activity. Robots use only local sensing and coordinate their activity via the shared environment[17].

CHAPTER 3

BACKGROUND

Since this thesis is based on the original TERMES compiler, this chapter describes the background of the research. The TERMES compiler and its requirements are introduced in section 3.1. Next, the robot rule set which determines when bricks can be placed is introduced in Section 3.2. Last but not least, the problem formulation of the TERMES compiler is presented in section 3.3.

3.1 TERMES Compiler

The compiler is one of the key component of the TERMES system. It converts the user-specified structure into a directed map similar to traffic patterns. Specifically, it generates a directed graph connecting reachable sites on the structure. An arbitrary number of robots can follow such a map to conduct the building activity. These traffic patterns prevent problems such as deadlock, unreachable areas, and incomplete construction. In order to successfully build the structure, the system must satisfy following requirements:

- Robots are physically capable of climbing up or down at most one brick height. An edge connecting two neighboring sites that has a height difference larger than 1 is considered untraversable.
- Robots cannot place bricks directly in between two other bricks (this is mechanically difficult to do), therefore the map cannot have arrows pointing towards each other in the same row or column. Furthermore, the map

cannot have cycles, as these will eventually also create unfillable gaps in the structure.

- There must be one start site where robots can enter the structure and start building activity. There must also be at least one exit site where robots can leave the structure after they place the brick, or find that there is nowhere to place one. Start and exit sites must be located on the external perimeter of the input structure. The height of start point and exit point must be 1 so that the robots can ascend/descend from there.
- For any site in the structure, there must exist at least one path that starts from the start site, pass through this site, and ends at an exit site. Each site must have at least one incoming arrow and at least one outgoing arrow. If it has no incoming arrow, it is not possible for robots to reach the site. If it has no outgoing arrow, it is not possible for robots to leave the site.

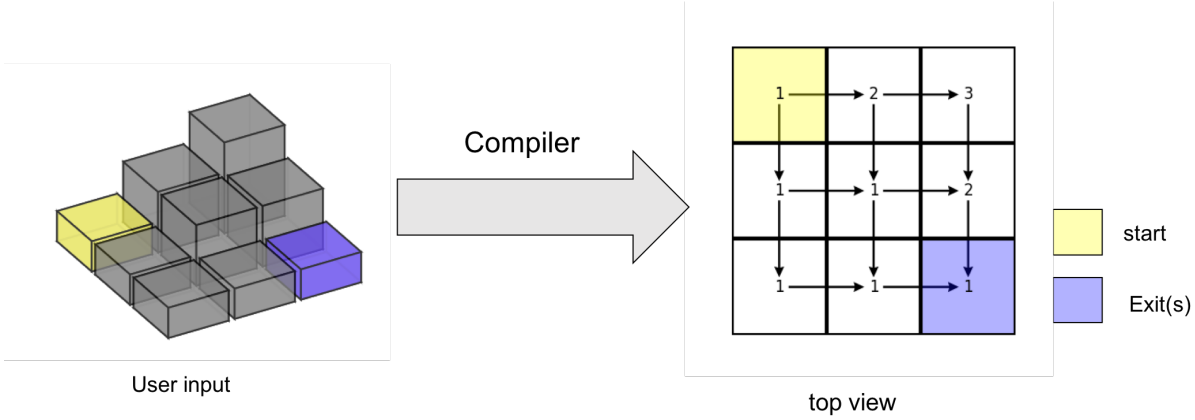


Figure 3.1: Sketch of how the compiler works. The compiler converts the user input to a directed map indicated by arrows on the plot. The numbers indicate the height of the stack of bricks at each site.

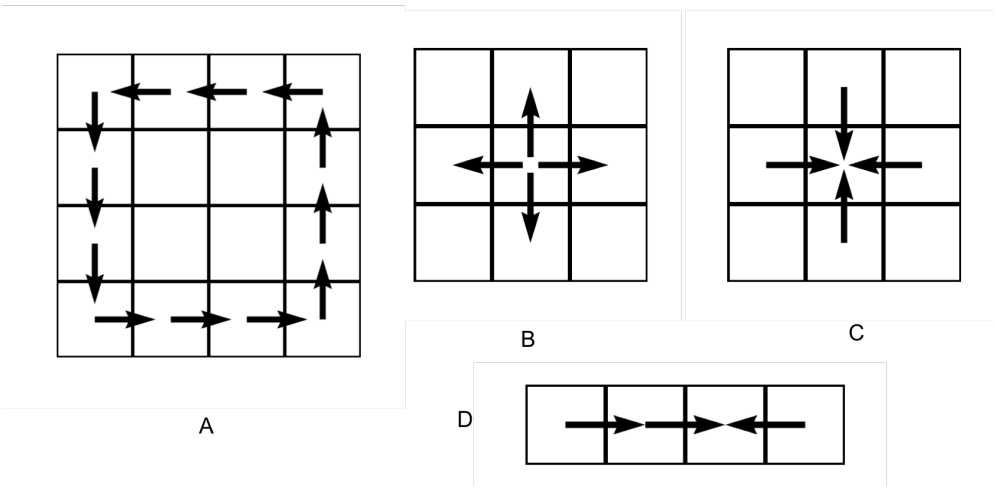


Figure 3.2: The compiler must comply with several restrictions. A) Cycles in the path are not allowed as they will lead to unfillable gaps. (B) If a site has no incoming arrow, it will be impossible for robots to reach this site. (C) If a site has no outgoing arrow, it will be impossible for robots to leave this site. (D) Arrows pointing towards each other may lead to unfillable gaps.

Note that a structure is considered unbuildable if it violates any of the restrictions mentioned above.

3.2 TERMES Robot Algorithm

Once receiving the directed map of the structure, an arbitrary number of robots move along the paths in the map and place bricks following a common behavioral program. This program does not change with the structures or the specific map generated. If robots were allowed to place bricks anywhere along the path, they would be likely to end up in situations that prevents future progress, e.g. by creating a cliff that other robots could not traverse over.

Algorithm 1: Building Algorithm[17]. H_i is the target height of site i on structure, h_i is the current height of site i. The robot's current location is marked as site 0. Parent sites of site 0 is sites that have outgoing arrows pointing to site i. Child sites of site 0 is sites that have incoming arrows pointing from site i. Next sites are reachable child sites from site 0.

Loop:
while structure is not complete **do**
    get a brick
    climb onto structure
    while on structure **do**
        move to any next site
        if holding a brick
            and $h_0 < H_0$
            and for all parent sites i: ($h_i > h_0$ **or** $h_i = H_i$)
            and for all child sites i: ($h_i = h_0$ **or** $|H_i - H_0| > 1$) **then**
            move to any next site
            attach brick at site just vacated
interrupt:
if robot close ahead **then**
    perform collision avoidance

Note that if one site has multiple children, the probability of going to each child site is uniformly distributed. In order to attach a brick to a site, the parent sites must either have reached their target height, or have a final height which is greater than the current site. If one of the parent sites does not satisfy these conditions, robots are not allowed to attach a brick there.

3.3 Problem Formulation

Based on the characteristics of the TERMES system, we state the problem as following: A structure consists of a finite set of locations L that each have integer x and y location, i.e. $(l_x, l_y) = l \in L$. Two locations $l, k \in L$ are said to be neighbors when either the x or y differ by one, but not when both are different. This type of neighbor relation corresponds to a distance of 1 with the Manhattan distance metric. A *path* is a sequence of locations $p = (l_1, l_2, .., l_N)$ such that consecutive locations are neighbors. We assume that all the locations for a structure are path connected, i.e. every location has a path to every other location. Disconnected structures can be treated as separate structures.

There are two special types of locations, $L_{start} \subset L$ and $L_{exit} \subset L$, which correspond to the start and exit locations. In a structure, each location has a target height $h \in \mathbb{N}$. We say that a path is *traversable* if each consecutive location differs in target height by at most one, i.e. in the completed structure such a path could be followed by a TERMES robot.

The goal of the compiler is to generate a directed, acyclic graph, on the vertex set L that has directed edges between neighbors, with the additional properties that: for every $l \in L$ there is a directed, traversable path from a start location to reach l, for every $l \in L$ there is a directed, traversable path to reach an exit, and that if bricks are only assembled after their predecessors have been assembled, this will never result in assembly operations that violate the physical adjacency constraints. To initialize, the compiler assigns outgoing arrows from L_{start} and ingoing arrows to L_{exit}.

CHAPTER 4

CONSTRAINT SATISFACTION PROBLEM

In this chapter, we recast the old compiler as backtracking search to a Constraint Satisfaction Problem (CSP) with pairwise, partial, and global constraint checking. Constraint Satisfaction Problems represent states as a vector of variable values. Each variable has a nonempty domain of possible values. It contains a set of constraints such that each constraint specifies the allowable combinations of values for some subset of the variables. A solution to a CSP is a complete assignment that satisfies all the constraints[14]. Section 4.1 introduces backtracking search for CSPs. Section 4.2 presents an ”Edge-CSP” compiler similar to the original compiler. An improved version of the Edge-CSP is introduced in section 4.3.

4.1 Backtracking Search for CSPs

Alg.2 shows the general formulation of backtracking search for CSPs. The backtracking search chooses values in the corresponding domain for variables one at a time in a depth-first search manner. It searches all possible combinations of values until it finds a solution. Worst case, it will search all combinations of values and find there is no solution.

Algorithm 2: General formulation of recursive CSP, from [14]. Backtracking is abbreviated BT.

```
1:  function BT-SEARCH(csp) returns a solution, or failure
        return RECURSIVE-BT({ }, csp)
2:  function RECURSIVE-BT(assignment, csp) returns a solution, or failure
3:      if assignment is complete then return assignment
4:      var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
5:      for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
6:          if value is consistent with assignment according to CONSTRAINTS[csp] then
                add {var = value} to assignment
7:              result ← RECURSIVE-BT(assignment, csp)
8:              if result ≠ failure then return result
9:              remove {var = value} from assignment
10:     return failure
```

4.2 Edge-CSP Compiler

The original compiler paper describes a search procedure for searching through the space of available assignments [17]. We recast the search as a backtracking search to a constraint satisfaction problem with pairwise, partial, and global constraints. Variables are defined as all edges connecting locations. Each variable has a domain of two values: ingoing and outgoing. Local constraints check if no neighboring edges pointing to the same location. Partial constraints check

the following conditions: 1) every assigned location, except for start and exit(s), must have both incoming and outgoing travel directions; 2) There cannot be cycles in the graph. The CSP checks, via a global constraint, if all locations are reachable from the start and that all exit(s) are reachable from those locations. This compiler does not take the height of the structure into consideration until the final global check. The search continues until all options have been tried, or no solution is found. Worst case, this amounts to $O(2^n)$ checks, where n corresponds to the number of edges.

In the original paper, it does not specify how to pick directions for edges. So we assume the choices are made randomly. We further optimize it by giving priority to direction that is pointing away from start.

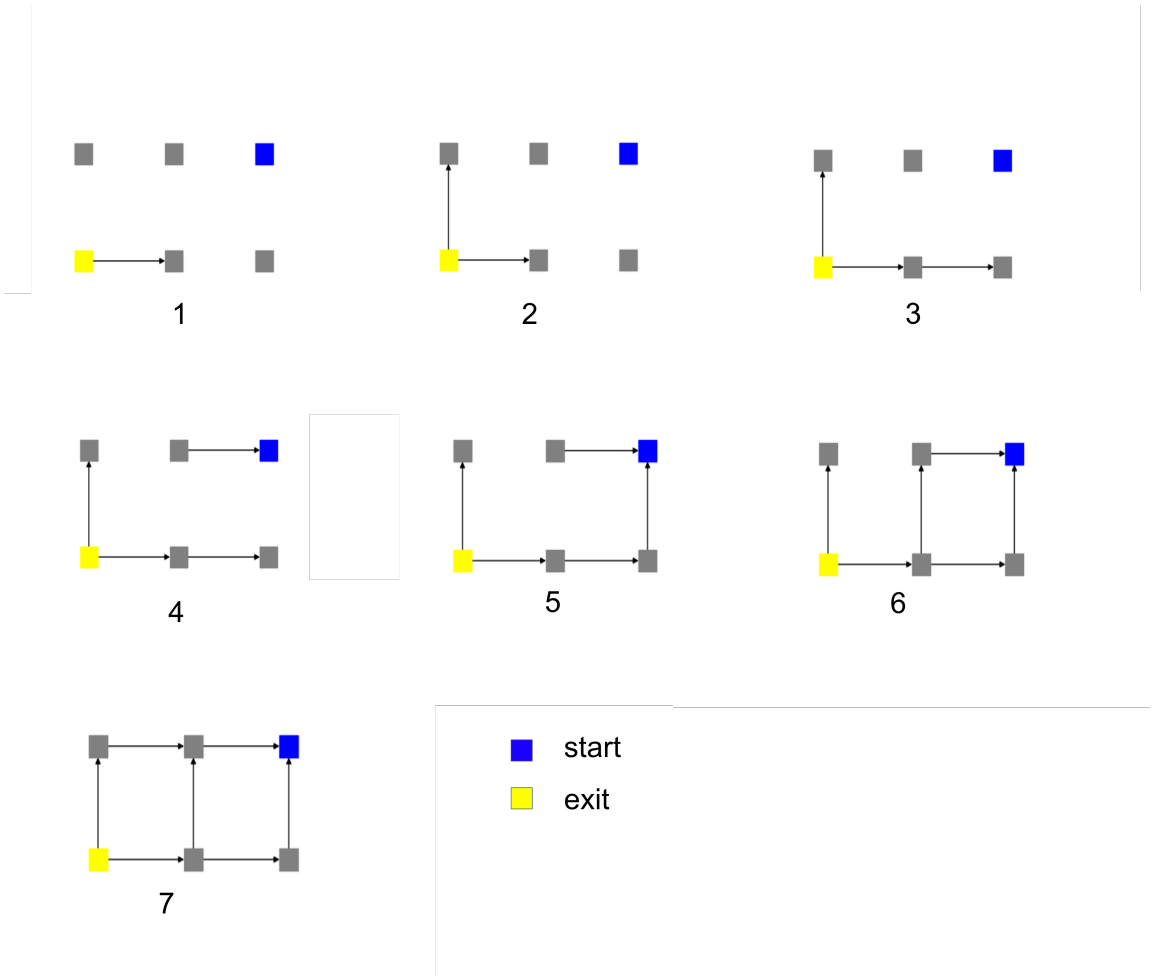


Figure 4.1: Example of the edge-CSP compiler on a 2×3 uniform height structure. Each square represents a location. Arrows represent edges and their directions. For each location, the compiler assigns one edge at a time.

4.3 Location-CSP Compiler

To speed up the backtracking search, we change the formulation of the CSP such that the variables become the locations and the domains include all combinations of travel directions on the 4 edges. The constraints are that the edge assignments of neighboring locations must be consistent (i.e. edges cannot have

arrows pointing towards each other). The benefit of this scheme is that it creates a fully connected graph, where a given constraint can more readily affect other variables. Similarly, this compiler does not take the height of the structure into consideration until the final global check. The search continues until all options have been tried, or no solution is found. Worst case, this amounts to $O(2^n)$ checks, where n corresponds to the number of edges.

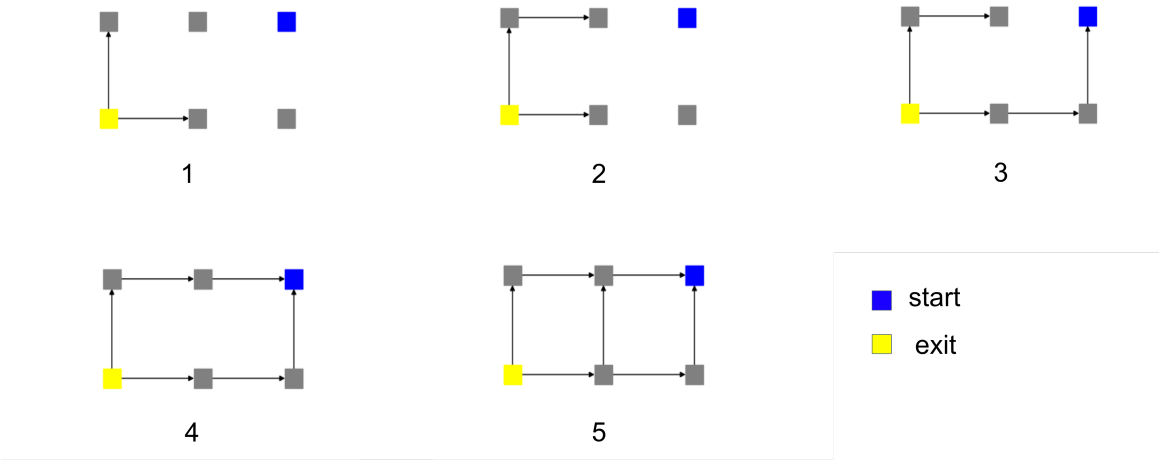


Figure 4.2: Example of the location-CSP compiler on a 2×3 uniform height structure. Each square represents a location. Arrows represent edges and their directions. For each location, the compiler assigns directions to all edges at the same time.

CHAPTER 5

BREADTH FIRST DISASSEMBLY COMPILER

In this chapter, the Breadth First Disassembly (BFD) Compiler is introduced. This compiler is not based on search, but instead does an iterative assignment of the travel directions in a breadth-first manner starting from all exit(s). Essentially, it evaluates if each location may serve as a drain for the structure, if locations whose travel directions have already been assigned were removed.

5.1 Algorithm

The process is shown in Fig. 5.1 and Alg.3. Upon initialization L_{exit} is fully defined and added to the visited list, $L_{visited}$. The neighbors to L_{exit} is enqueued in $Q_{frontier}$. The compiler recursively dequeues L_i and checks if it can serve as a drain. To serve as a drain, L_i 1) cannot be in between two unassigned locations, and 2) cannot cause a disconnect in the structure. To check the connectivity, the compiler conducts a breadth-first search starting from L_{start} to count the number of reachable locations. If this count is equal to the number of non-visited locations, L_i may serve as a drain. It is added to $L_{visited}$ and ingoing arrows are added to all reachable neighbors. The compiler continues to do this until $Q_{frontier}$ is empty or no solution is found. If successful, the compiler takes a final pass over the structure and assigns any edges that have not already been assigned, according to the restrictions mentioned above.

Algorithm 3: Pseudo code for the Breadth First Disassembly Compiler. This either returns a valid labeling of travel directions (arrows) between neighboring locations, or identifying that no such labeling exists for the given input. l_0 denotes the current location in question and l_i the neighboring locations, with final height of H_0 and H_i respectively. $Q_{visited}$ keeps track of locations which have been 'disassembled', i.e. fully determined; $Q_{frontier}$ keeps track of the frontier. Each location, l_i, has a distance property, d_i, indicating the distance to L_{exit}.

Initialize:

  Add L_{exit} to $Q_{visited}$ and add neighboring sites of L_{exit} to $Q_{frontier}$.

  Set d_{exit} to 0; all others to Inf.

Loop:

while $Q_{frontier}$ is not empty **do**

  $l_0 = dequeue(Q_{frontier})$

  if l_0 is not in between two other unvisited sites

    and removing l_0 does not disconnect the structure

    and l_0 has at least one traversably connected neighboring site $\in Q_{visited}$

  then

    Add l_0 to $Q_{visited}$

    for each neighboring site l_i of l_0, $l_i \in Q_{visited}$ **do**

      if $|H_i - H_0| <= 1$ **then**

        add arrows from l_0 to l_i

        $d_0 = min(d_i + 1, d_0)$

    add unvisited neighboring locations to $Q_{frontier}$ if no in $Q_{frontier}$

if $length(Q_{visited})$ < number of locations in the structure **then**

  the structure is unbuildable

else

  while any locations have unlabeled edges **do**

    add arrows from higher d to lower d.

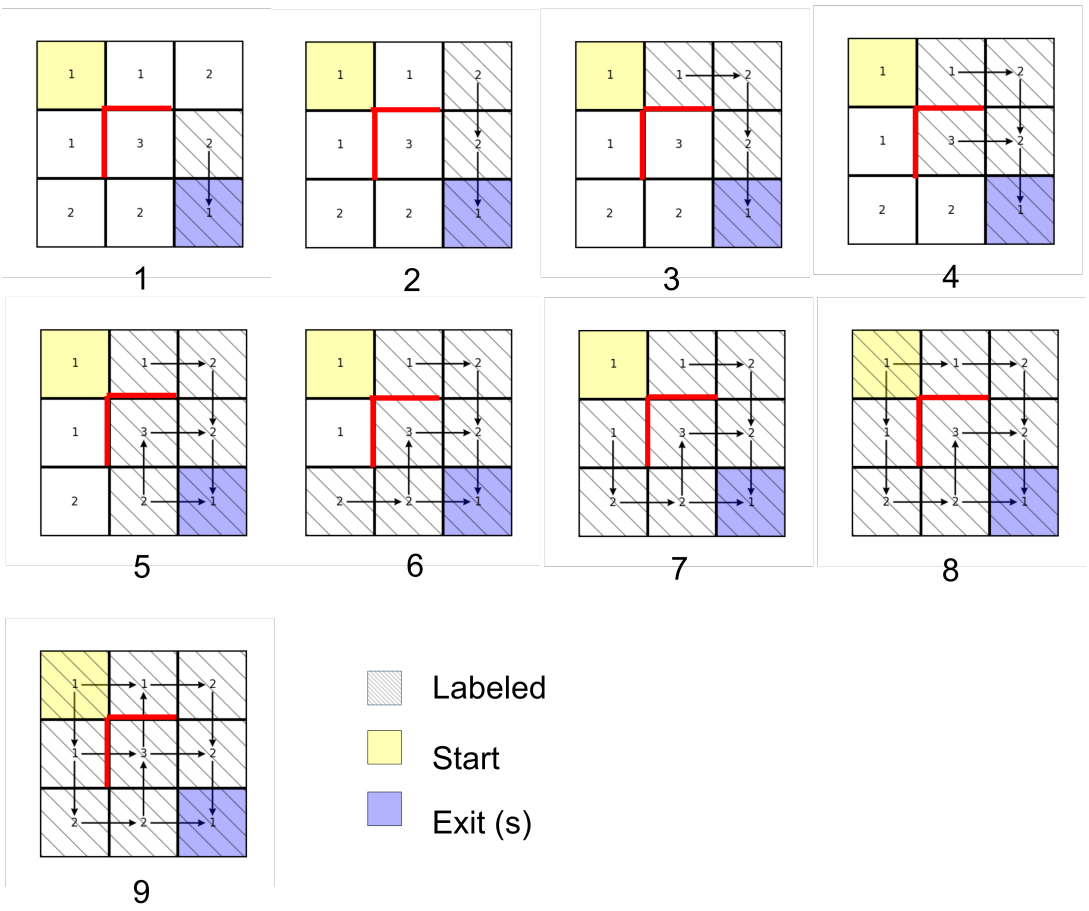


Figure 5.1: Example of the BFD compiler on a 3×3 structure. The compiler starts at the exit and removes locations in a breadth-first manner dependent on connectivity. Consider the start location to be (0,0) and the exit location to be (2,2): The compiler first removes (2,1) - arrows are added as bricks are removed. The compiler then tries to remove (1,2), but cannot as this would cause (1,1) to become disconnected. It then removes (2,0), and continues in the same manner until only the start location is left (returns success), or until no more locations can be removed (returns failure). Notice that the green arrows are optional and do not count towards the traversability check.

The connectivity check is further optimized. We compute a tree, in a breadth first manner starting from the start site, that connects all unvisited sites. When we want to remove another node, we can check if it is a leaf node in the tree and

then know that the rest stays connected without doing any extra work.

5.2 Proof of Correctness

There are several properties we need to maintain:

1. for every $l \in L$ there is a directed, traversable path from a start location to reach l, for every $l \in L$ there is a directed, traversable path to reach an exit
2. there are no cycles in the paths
3. there are no arrows which points towards each other in each row/column

The correctness proof of BFD compiler is by induction on a partition between the visited and unvisited locations.

Notation:
Traversable paths: paths are considered as traversable if target height of connected locations never have more than unit difference. Traversably connected: two locations are connected if they are adjacent, two locations are traversably connected if their height difference is no larger than 1.

Lemma 5.2.1. *During the partially assembled states, all visited sites can reach at least one exit via a traversable path.*

Proof. Base case: visited set is empty. All locations are considered as unvisited. Inductive Step: If in *nth* iteration, all visited sites can reach at least one exit via traversable paths. Then in the $(n + 1)th$ iteration, a new location from $Q_{frontier}$ is considered as visited if it is not in between two unvisited locations and does

not disconnect unvisited locations. The directions of edges that connects the new location and visited locations are marked from the new location to visited locations. Since all visited locations are traversably connected to at least one of the exit sites, the new location is traversably connected to visited locations. There is a traversable path from the new location to one of the exit sites. □

Lemma 5.2.2. *During the partially assembled states, all unvisited sites are traversably connected to the start site.*

Proof. Every time the compiler dequeues a new location, it conduct connectivity check. The connectivity checks if all unvisited sites are are traversably connected to start site. If not, this location cannot be visited. □

Lemma 5.2.3. *During the partially assembled states, there is no cycle in the directed paths that connect all visited locations.*

Proof. Base case: visited set is empty. All locations are considered as unvisited. Inductive Step: If in *nth* iteration, there is no cycle in the directed paths that connect all visited locations, then in $(n+1)th$ iteration, a new location from $Q_{frontier}$ is considered as visited or not. If it is not considered as visited, nothing changes. If it is considered as visited, arrows are assigned from the new location to its traversably connected visited neighboring sites. A cycle requires a traversable path from a visited location to itself. To form a cycle in the $(n+1)th$ iteration, the newly visited location must be in the cycle. It requires at least one arrow pointing from visited locations to the new location and at least one arrow pointing from the new location to visited locations. It contradicts with the rules we assign arrows. Therefore, there is no cycle in the directed paths that connect all visited locations. □

Lemma 5.2.4. *There are no arrows pointing towards each other in each row/column.*

Proof. From Alg.3 we know that the compiler only assigns arrows from newly visited locations to previously visited locations. If there are opposing arrows in a row/column, the first location being visited in this row/column is in the middle instead of at end. In other words, one of the locations that is in between two unvisited locations is visited. This causes contradiction. The compiler only visits a location if it is in between two unvisited locations and does not disconnect unvisited locations. Therefore, it is impossible to create opposing arrows.

□

Lemma 5.2.5. *If the input structure is buildable, all locations in the structure will be added to $Q_{visited}$. Otherwise the structure is unbuildable.*

Proof. Assume the input structure is buildable and there are some locations in the structure are not added to $Q_{visited}$. For those locations, the times one is added to $Q_{frontier}$ is the same as the number of its neighboring sites. Every time dequeued from $Q_{frontier}$, one of the following situation must happen: 1) This location is in between two unvisited locations, 2) Visiting this location will disconnect unvisited locations. Since it is a buildable structure, each location must have at least two traversable edges, which means locations that are not added to $Q_{visited}$ must be added to $Q_{frontier}$ at least two times. The first situation can only happen when the first time a location is added to $Q_{frontier}$. The second time it is added to $Q_{frontier}$, two of its neighbors are already visited. It is not possible to have this location in between two unvisited location. The second situation can not happen during the last time a location is added to $Q_{frontier}$. If it is the last a location is added to $Q_{frontier}$, all its traversably connected neighbors are

visited. Visiting this location won't disconnect unvisited locations. Therefore, if the input structure is buildable, every locations in the structure will be added to $L_{visited}$. □

CHAPTER 6

EVALUATION

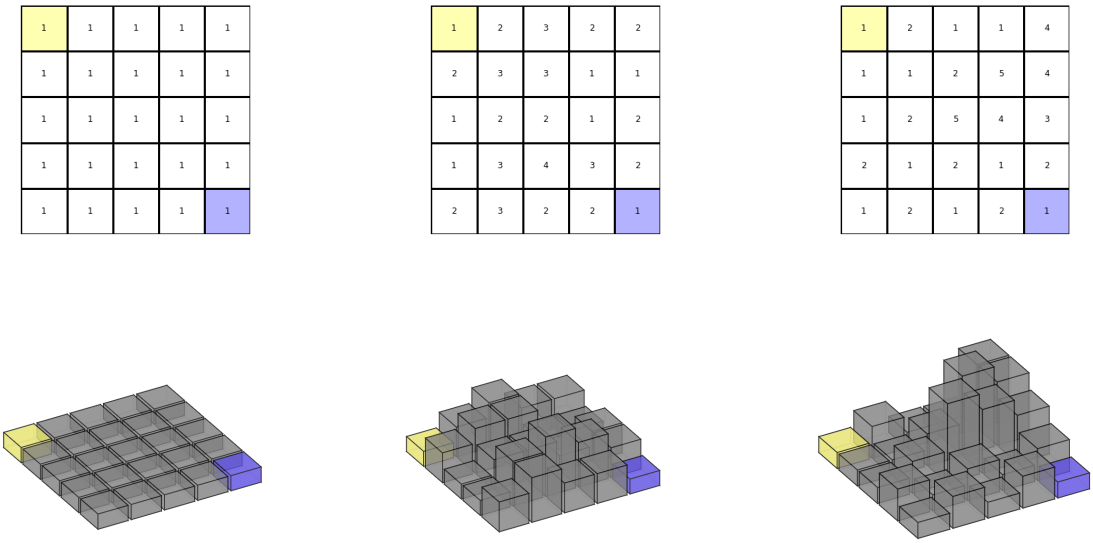
To test the performance of the compilers, the BFD compiler, the location-CSP compiler and the edge-CSP compiler were implemented in Python. All experimental tasks were performed a on a standard laptop, ThinkPad T580. We use runtime to evaluate their performance. The following details characterized the experiment.

6.1 Test Cases

Table 6.1 describes five compilers we implement. Three test structures are used to test the performance of the compilers. 1) A one-height square. It has the most parallel choices and feasible solutions. 2) A buildable random-height square. Such a structure requires the compiler to conduct height checks and it may be difficult to find a feasible solution comparing with case 1. 3) An unbuildable random-height square. The purpose of this test is to examine the performance of compilers to detect structures that are impossible to compile. For each test we grow these structures in size to see how the compilers scale with the number of locations. Fig.6.2 shows the way we scale up one-height squares. We use the same method to scale up the random squares. For each structure, we run the compiler 10 times and use maximum, minimum, and median runtime value.

Compiler	Description
BFD_NT	BFD compiler as described in section 5.1
BFD	BFD compiler as described in section 5.1 with optimized connectivity check.
LCSP	Location-CSP compiler as described in section 4.3
ECSP_RD	Edge-CSP compiler as described in section 4.2 with random ordering of domains.
ECSP	Edge-CSP compiler as described in section 4.2 with priority on assigning arrows outwards from start site

Table 6.1: Description of different compilers



(a) one-height squares

(b) buildable random squares

(c)unbuildable random squares

Figure 6.1: Base test structures. (a) shows the top view and 3D model of one-height square. (b) shows the top view and 3D model of random buildable square. (c) shows the top view and 3D model of random unbuildable structure. The start site is marked in yellow and exit site is marked in blue.

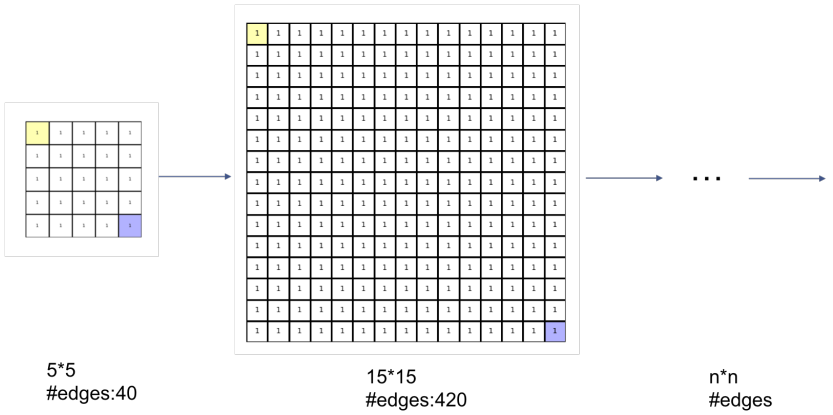


Figure 6.2: Scaling up squares

6.2 Results

Fig.6.3 is the log-log plot of runtime for one-height structures. The BFD compiler has the best performance while the Edge CSP_RD scales very poorly with the size of the structure. The Edge CSP and the Location CSP are expected to have relatively good performance since one-height square has many feasible solutions. Note that 10^4 locations corresponds to the approximate number of bricks in an average American house.

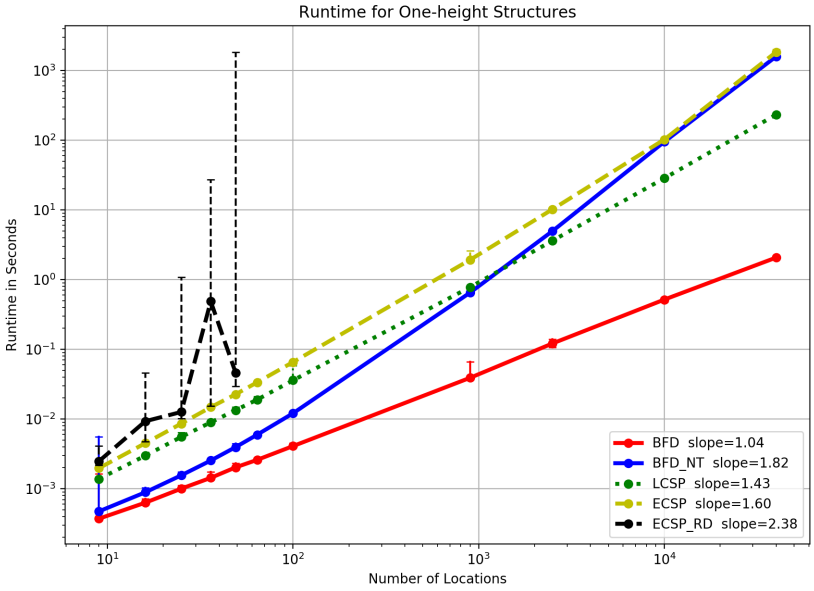


Figure 6.3: Log-log plot of different compilers on one height squares. The dot indicates the median value of the runtime for 10 trials; the error bars indicate the minimum and maximum value.

Fig.6.4 is the log-log plot of runtime for random buildable structures. The result is similar to the test of one-height square structures. The BFD compiler has the best performance while the Edge CSP_RD scales poorly with the size of structure. Note that in this case the BFD_NT has better performance than the Location CSP. This happens because random structures are likely to have less feasible solutions compared to one-height structures.

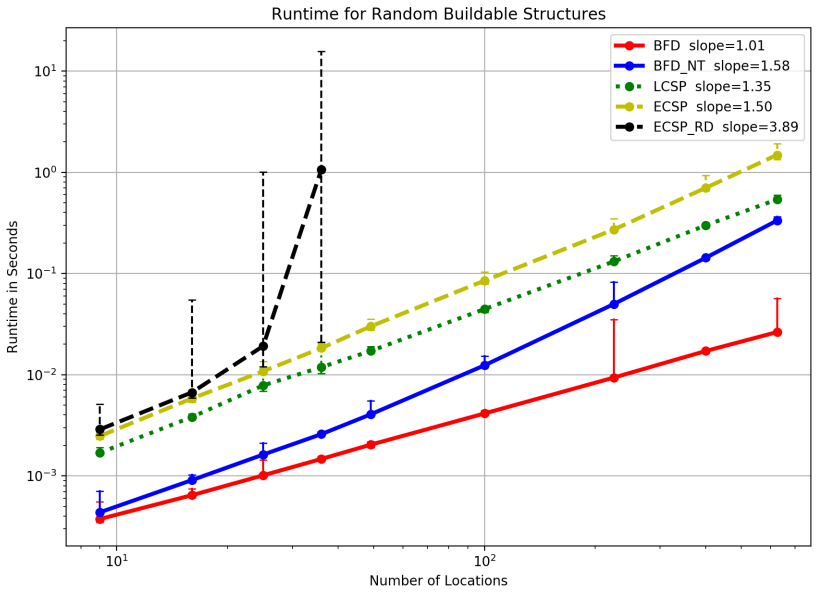


Figure 6.4: Results of different compilers on random buildable squares. The dot indicates the median value of runtime of the compilers on 10 random structures of similar size; the error bars indicate minimum and maximum values.

Fig.6.5 is the log-log plot of runtime for random unbuildable structures. Unbuildable structures which have no feasible solution may take a long time for the search-based algorithms to check unless there exist a local constraint which is impossible. As seen in the graph, the runtime fro ECSP and ECSP_RD grows rapidly with the size of structures. LCSP has better performance than ECSPs because the local check enable LCSP to stop early for certain branches during the backtracking search. However, the performance of LCSP will drop rapidly with the increase of structure size since the number of possible solutions also increases. We stop testing the Location CSP, the Edge CSP, and the Edge CSP_RD on 7×7 structures since it takes hours for those compilers to complete. The BFD and BFD_NT, however, have similar performance as the previous cases as they are not influenced by the number of possible solutions.

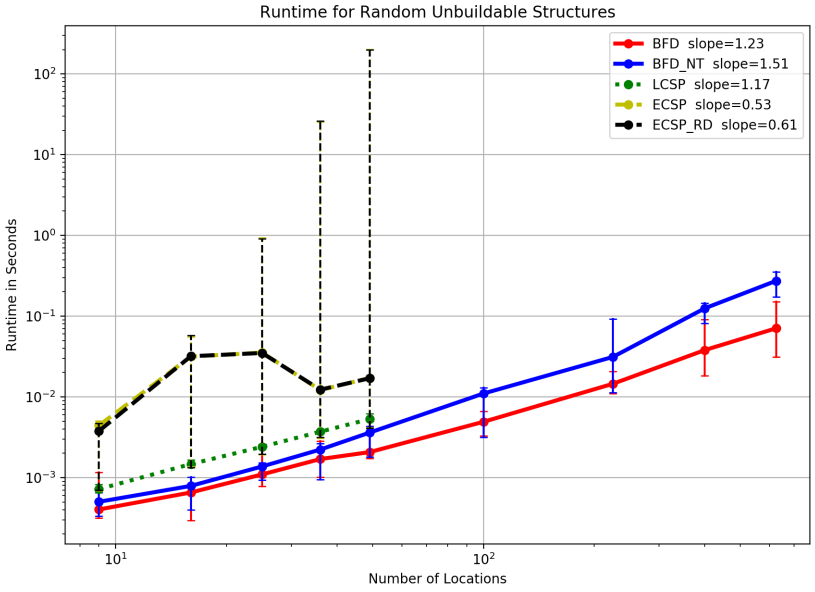


Figure 6.5: Results of different compilers on random unbuildable structures. The dot indicates the median value of runtime over 10 trials; the error bars indicate the minimum and maximum values.

CHAPTER 7

PROBABILITY OPTIMIZATION

In the standard maps, a robot on the structure has a uniform probability of choosing any of the traversable child locations. However, since the structure must expand from one point outwards this leads to many wasted trips where the robot exits the structure without finding a viable construction site. This is especially true for dense structures where many outer bricks must be inserted before the rest can be added. An example of such a structure is shown in Fig. 7.1.A-B. The probability, P_i, of finding a robot in a location l_i, with parent locations, P_j, is calculated as:

$$P_i = \sum_{j=1}^{J} P_j P_{j \to i} \tag{7.1}$$

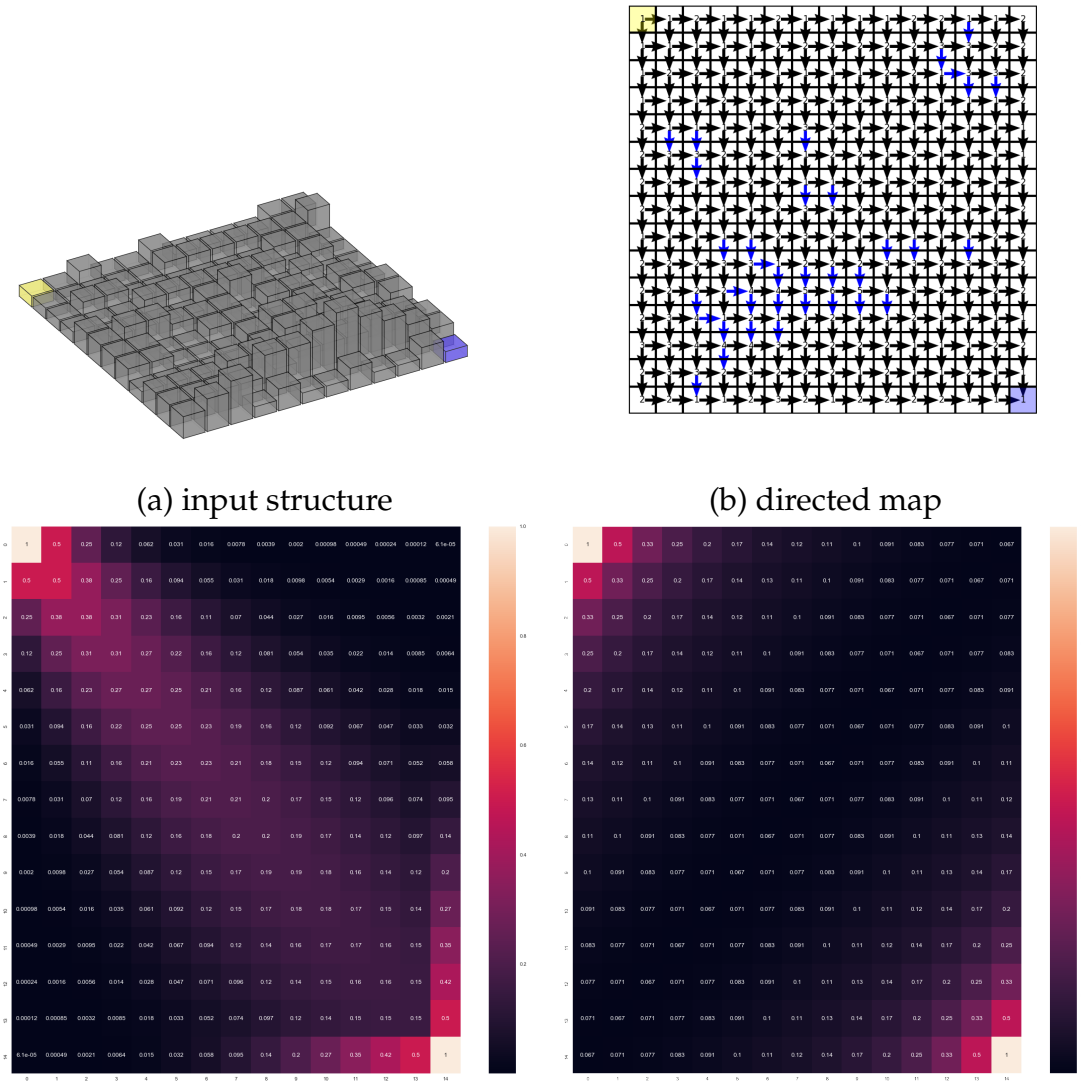
where $P_{j \to i}$ denotes the transition probability from l_j to l_i, and J the total number of parents. Figure 7.1.C shows an example of how uniform transition probabilities in this map cause the robots to be more likely in the center of the structure.

By using the connected graph of parent and child nodes, we can optimize the transition probabilities to more effectively spread the flow of robots over the structure. We base this optimization on the distance in the graph to the exit site. For locations with the same distance, we need to minimize the difference of P_i. We use Sequential Least Square Programming (SLQSP) minimization to find improved transition values. We formulate the problem as follows:

$$
\begin{aligned}
\underset{P_{j\to i}}{\text{minimize}} \quad & \sum_{I_d \subseteq I} \text{Var}(P_{I_d}) \\
\text{subject to} \quad & P_{start} = 1 \\
& P_i = \sum_j P_j P_{j\to i},\ i \in I, j \in I_{p(i)}, \\
& \sum_k P_{i\to k} = 1,\ i \in I, k \in I_{c(i)}, \\
& P_i > 0, P_i \leq 1,\ i \in I
\end{aligned}
\tag{7.2}
$$

I donates a set of all sites on structure. I_d donates sites that are d distance away exits. $I_{p(i)}$ donates parent sites of site i. $I_{c(i)}$ donates child sites of site i. Fig. 7.1 shows an example of how this affects the probability of robots moving over the structures. To test the difference in construction speed, we run 5 simulated robots and count the number of steps between locations until structure completion. With uniform transition probabilities, the system completes the structure in 718,611 steps (run once). With improved transition probabilities, the structure is completed in an average of 44,322 steps out of 10 runs - indicating over 100 times speedup. This speedup will of course be structure dependent.

Note, that this scheme only considers flat structures, in the future it may be further optimized by considering the bulk of bricks that needs to be placed in different locations.



(a) input structure

(b) directed map

(c) probability map before optimization

(d) probability map after optimization

Figure 7.1: Example of transition probability optimization for a 15×15 random structure. (a) is the 3D model of input structure, the height of each sites are randomly generated. It is a 15 × 15 structure consists of 406 bricks. (b) is the direct map generated by compiler. Notice that travel directions which are intraversable in the final structure are colored in blue. (c) is the probability map before optimization. The probability of reaching anti-diagonal corner is 6.1e-5. (d) is the probability map after optimization. The probability of reaching anti-diagonal corner is increased to 0.067.

CHAPTER 8

CONCLUSION

8.1 Conclusion

In this research, my main contribution is BFD compiler, a scalable compiler for the TERMES system.

We first recast the original compiler as a backtracking solution to a Constraint Satisfaction Problem with pairwise, partial, and global constraint checking in which the variables are the edges between locations, and the domains are the travel directions. We refer to this as an Edge CSP compiler. We improved on this by creating a Location CSP compiler, where the variables are the locations in the structure, which sped up the backtracking search. The worst case time complexity for the Edge CSP and the Location CSP is $O(2^n)$. We then developed a new compiler, named BFD compiler, that is not based on search, but instead builds up a feasible solution in a breadth-first manner from all exits. The new compiler has $O(n^2)$ time complexity for average and worst case. We implemented the Edge CSP compiler, the Location CSP compiler, and the BFD compiler in Python and compared their performance on a range of structures. As expected, the BFD compiler became especially advantageous as the size and complexity of the structure increases. It is also superior when it comes to detecting unbuildable structures. We showed the ability of this compiler to work on a structure of up to 1 million bricks, comparable to the number of bricks in the Great Pyramid of Giza. As a secondary contribution, we altered the transition probabilities between adjacent locations to make construction more efficient. This work represents an important step towards real-world deployment

of robot collectives for construction.

8.2 Future Work

I imagine several ways in which this work could progress. The current compilers were evaluated with respect to their runtime, however, they have not been evaluated with respect to the quality of the solution they find. For a buildable structure with more than one feasible solution, different compilers might output different solutions. As a next step we could design an evaluation method to compare those solutions and pass on only that which permits the fastest and safest way to success. Furthermore, when finding that the desired structure is unbuildable, we could automatically give the user suggestions on minimal modifications by which it could become buildable. Furthermore, as previously mentioned, the current transition probability optimization does not take the height of the structure into consideration; this could certainly be optimized in a second iteration.

APPENDIX A

CHAPTER 1 OF APPENDIX

A.1 Generating Random Structures

Given the size of the structure, we assign heights based on a breadth-first expansion. The algorithm starts from all exits and start at the same time and then expands to their neighbors. The heights of the start site and exit site(s) are 1. When expanding from one site to its neighbors, we assign the heights of the neighbors based on the probability in Table A.1. The height of neighbors can be one brick higher, one brick lower, or the same as the height of the current site. Note that this algorithm works for any structure, but here we only generate squares.

Size	P(+1)	p(0)	p(-1)
$< 10 \times 10$	0.4	0.2	0.4
$< 20 \times 20$	0.3	0.4	0.3
$< 25 \times 25$	0.2	0.6	0.2
$< 30 \times 30$	0.1	0.8	0.1

Table A.1: This table shows the probability of generating random heights for square structures. $P(+1)$ indicates the probability of increasing the height by one. $P(0)$ indicates the probability of staying the same height. $P(-1)$ indicates the probability of lowering the height by one brick.

BIBLIOGRAPHY

[1] Caterpillar. https://www.caterpillar.com.

[2] Commonly used statistics. https://www.osha.gov/oshstats/commonstats.html 2. Worlds population increasingly urban with more than half living in urban ar- eas (2014).

[3] Construction robotics. http://www.construction-robotics.com.

[4] Fastbrick robotics. fastbrick robotics.

[5] Komatsu ltd. https://home.komatsu/en/.

[6] World cities report 2016. http://wcr.unhabitat.org/wp-content/uploads/sites/16/2016/05/WCR- Full-Report-2016.pdf.

[7] Open-air computers, Oct 2012. https://www.economist.com/news/special-report/21564998-cities-are-turning-vast-data-factories-open-air-computers.

[8] Ross A Knepper, Todd Layton, John Romanishin, and Daniela Rus. Ikeabot: An autonomous multi-robot coordinated furniture assembly system. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 855–862. IEEE, 2013.

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

[10] Nils Napp and Radhika Nagpal. Distributed amorphous ramp construction in unstructured environments. *Robotica*, 32(2):279–290, 2014.

[11] Kirstin Petersen, Radhika Nagpal, and Justin Werfel. Termes: An autonomous robotic system for three-dimensional collective construction. *Proc. Robotics: Science & Systems VII*, 2011.

[12] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.

[13] Michael Rubenstein and Wei-Min Shen. Scalable self-assembly and self-repair in a collective of robots. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 1484–1489. IEEE, 2009.

[14] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[15] Statista. U.s. construction - total value of new construction 2017. https://www.statista.com/statistics/184341/total-value-of-new-construction-put-in-place-in-the-us-from-1999/.

[16] Yuzuru Terada and Satoshi Murata. Automatic modular assembly system and its distributed control. *The International Journal of Robotics Research*, 27(3-4):445–462, 2008.

[17] Justin Werfel, Kirstin Petersen, and Radhika Nagpal. Designing collective behavior in a termite-inspired robot construction team. *Science*, 343(6172):754–758, 2014.

[18] Jan Willmann, Federico Augugliaro, Thomas Cadalbert, Raffaello D'Andrea, Fabio Gramazio, and Matthias Kohler. Aerial robotic construction towards a new field of architectural research. *International journal of architectural computing*, 10(3):439–459, 2012.