Robotics and Autonomous Systems 121 (2019) 103240

Contents lists available at ScienceDirect

Robotics and Autonomous Systems

journal homepage: www.elsevier.com/locate/robot

A Compiler for Scalable Construction by the TERMES Robot Collective

Yawen Deng [a], Yiwen Hua [a], Nils Napp [b], Kirstin Petersen [a,*]

[a] Cornell University, Ithaca, NY 14850, USA
[b] University at Buffalo, Buffalo, NY 14260, USA

ARTICLE INFO

Article history:
Available online 18 July 2019

Keywords:
Multi-robot systems
Assembly
Construction
Autonomy
Compiler

ABSTRACT

The TERMES system is a robot collective capable of autonomous construction of 3D user-specified structures. A key component of the framework is an off-line compiler which takes in a structure blueprint and generates a directed map, in turn permitting an arbitrary number of robots to perform decentralized construction in a provably correct manner. In past work, this compiler was limited to a non-optimized search approach which scaled poorly with the structure size. Here, we first recast the process as a constraint satisfaction problem (CSP) to apply well-known optimizations for solving CSP and present new scalable compiler schemes and the ability to quickly generate provably correct maps (or find that none exist) of structures with up to 1 million bricks. We compare the performance of the compilers on a range of structures, and show how the completion time is related to the inter-dependencies between built locations. Second, we show how the transition probability between locations in the structure affect assembly time. While the exact solution for the expected completion time is difficult to compute, we evaluate different objective functions for the transition probabilities and show that these optimizations can drastically improve overall efficiency. This work represents an important step towards collective robotic construction of real-world structures.

1. Introduction

Autonomous robots have the potential to revolutionize the construction industry enabling rapid fabrication of inexpensive structures, novel designs, and construction in novel settings. Researchers and industrial specialists have proposed many solutions to these challenges, one of which involves collectives of autonomous mobile robots which can assemble structures much larger than the size of the individuals [1]. By focusing on distributed scalable coordination, such systems may deploy many robots to work efficiently in parallel and be tolerant to individual failures. Although robot collectives have received a lot of attention over the past couple of decades [2], most demonstrations are limited to controlled laboratory settings, relatively small assemblies, and/or small collectives. Open challenges range from scalable algorithms to capable, low-maintenance hardware. Here, we focus on the former, i.e. improving the algorithmic framework in terms of how it scales with the size of the structure. We present our results in the context of the TERMES system presented in previous literature [3–6], but our approach may generalize to other distributed construction systems.

The TERMES hardware consists of custom bricks and simple robots capable of climbing on, navigating, and adding bricks to the structure (Fig. 1A). Inspired by construction in social insects, the robots coordinate construction implicitly through their environment in a scalable manner. Despite this minimalistic approach the system has been shown to assemble 3D structures with provable guarantees, by relying on a combination of an off-line compiler and an onboard rule set. The compiler converts the structure blueprint to a 2D-map with assembly locations, the desired number of bricks at each location, and designated travel directions between locations (Fig. 1B). This map is given to an arbitrary number of robots, which follow these instructions and add material as determined by the onboard rule set which is dictated solely by the limitations of the robot platform used (Fig. 1C–D). The scalability of the TERMES and similar systems is determined by several factors, including (1) hardware cost and manufacturing complexity; (2) robot reliability and how likely failures are to disrupt system progress; (3) how the coordination mechanisms scale with the size of the collective; (4) how the compiler computation time scales with the size of the structure; and finally, and (5) how efficiently robots can reach the assembly frontier.

Fabrication and robot reliability (points 1–2) were addressed in [4]. The system was designed with minimalism in mind — co-design of robots, bricks, and algorithms resulted in a simple robot costing ~\$2K with a 1-week assembly time. The cost of the mechanics was brought down considerably in a subsequent paper [6]. To support reliability, the focus was not on achieving

* Corresponding author.
E-mail address: kirstin@cornell.edu (K. Petersen).

https://doi.org/10.1016/j.robot.2019.07.010
0921-8890/© 2019 Elsevier B.V. All rights reserved.

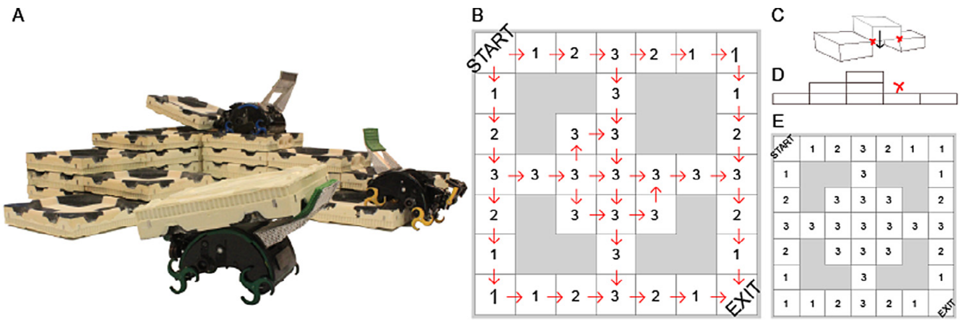


Fig. 1. (A) Photo of the TERMES system. (B) Example of the map generated by the compiler (top view). The digits indicate the number of bricks at each location; arrows how robots can transition between locations. System limitations include that bricks cannot be added in between others bricks (C, dimetric view), and that robots can climb at most one brick height between neighboring locations (D, side view). The set of structures which are compilable are not necessarily intuitive. (E) shows a structure which cannot be compiled, because the only way for a robot to complete the center would be an assembly move of type C.

perfect behavior, but rather to enable robots to recognize and fix errors before they propagated. Scalability of the collective (point 3) was addressed implicitly by relying on the structure as a shared physical database through which the robots can coordinate [3,5,6]. Here, we focus instead on improving the TERMES compiler to make it feasible to compile maps of large-scale structures (point 4). The work presented in this paper includes that of the conference paper presented at the International Symposium on Distributed Autonomous Robotic Systems (DARS) 2018 [7], with an additional contribution of showing how the transition probabilities between locations in the map affect structure assembly time, and how these can be optimized such that robots can complete the structure significantly faster (point 5).

First, we recast the compiler originally described in [3] (Section 3) as a backtracking solution to a constraint satisfaction problem (CSP) with pairwise, partial, and global constraint checking. We show that the original compiler scales poorly with the size of the structure (Section 4). By examining the behavior of the original search as a solution to a CSP, we are able to achieve significant improvements by formulating a new CSP that better exploits forward checking pairwise constraints during the backtracking search (Section 5). We then describe and prove an entirely new formulation for generating maps that is not based on search, but an iterative method that builds up feasible maps by considering locations in a breadth-first manner starting from the exit location (Section 6). We show the ability of the latter to compile structures with up to 1 million bricks in $\sim$1 min on commodity hardware. We compare the performance of these compilers on different sets of structures (Section 7), including unbuildable ones which are computationally intractable for search-based compilers. Finally, we show how, after the map has been compiled, construction speed may be improved simply by altering the transition probabilities between locations, with examples of a 2 order of magnitude improvement in completion time (Section 8).

2. Related work

Collective robotic construction can be achieved in a variety of ways, and examples include pre-programmed robots for functional structures [8,9], template-based construction [10], centralized controllers that allow for parallelism [11], communication-based coordination [12,13], and compiler-based systems [3,14].

Compilers for generating matter, which take high-level specifications and generate parallel assembly steps, are used in a variety of fields, e.g. digital materials [15], self-assembly, and modular robots [16]. In the construction setting, compilers must take into consideration the physical constraints of both building material and the robots that manipulate it. Constraints may exist both in mechanisms (e.g. the ability to traverse the structure) and perception/cognition (the ability to sense/remember the state of the surrounding structure). Broadly categorized, there are two ways to approach compilers [2]. The first is to define a set of sub-structures for which an assembly plan is known, and then to decompose new structures into combinations of those. The second is to compile based purely on the physical constraints of the system. Although the first method makes reasoning and guarantees easier, it also limits the set of structures (some structures that robots are physically capable of building cannot be compiled). The second method does not artificially restrict the set of buildable structures, but makes it hard to reason about what is buildable. In case of the latter, it is therefore critical that compilers can quickly assess whether or not a structure is buildable, or potentially come up with alternative solutions [5,17].

An example of the first approach include Seo et al. [14] who presented a compiler for 2D assembly of simply connected structures of floating bricks by boat-like robots, which decomposes structures into linear cells. Another example involves that of Lindsey et al. [11,18] who presented a compiler for assembly of strut structures by teams of quadcopters. The struts could be assembled into structurally stable cubes. Consequently, the compiler was designed to generate assembly rules for any structure which was decomposable into such special cubic structures. Both of these systems have a concise definition of the class of compilable structures.

The TERMES compiler is search-based and uses hardware limitations as constraints. As previously mentioned, this makes it harder to infer which structures are buildable. Fig. 1B and E shows structures which are buildable and unbuildable, respectively, despite the fact that they differ by only one location and despite the fact that it is possible for a robot to physically assemble each separate location. The issue is that there is no way to consistently order the assembly steps without violating the constraint shown in C. Currently, for TERMES-like constraints, there is no good specification for which structures have valid maps, other than when a map is found. This is especially problematic if the compiler used is slow and has a long runtime before failing. Here, we show that the compiler presented in [3] scales poorly with the size and complexity of the structure, and present an alternative compilation method, such that arbitrary structures can be compiled and checked quickly.

The second contribution of the paper concerns construction efficiency: i.e. after the offline compilation, how fast can the structure be completed by a given number of robots moving stochastically according to the map. The randomized execution model makes global state sharing unnecessary and thus makes concurrent execution between an arbitrary number of robot easy. However, it also introduces inefficiencies because of 1) physical bottlenecks which limits the number of robots that can simultaneously pass through a location and 2) construction order, i.e. the need for some actions to be completed before others can take place. Related work on optimizing assembly plans for TERMES focus on optimizing the map structure [17]. Here, we leave the original map in place and instead focus on optimizing the probabilities between different paths through the map. Past work on optimizing stochastic assembly policies under such spatial- and order-constrained scheduling is limited. In [19], the authors analyze stochastic assembly algorithms constrained both by assembly orders and by raw materials through chemical reaction models. In [20] the problem of optimizing transitions for material transport under spatio-temporal constraints is addressed, however, the transition probabilities are constrained to a relatively small parameterized model. Efficient spatial allocation of assembly robots have been shown in [21–23], with the ability to adapt to local failures and shape changes through space partitioning.

3. Problem formulation

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 locations, $l_{START} \in L$ and $l_{EXIT} \in L$, which correspond to the start and exit locations. In a structure, each location l has a target height $h_l \in \mathbb{N}$. We say that a path is *traversable* if each consecutive location differs in height by at most 1, which corresponds to the motion limitations of a TERMES robot.

In order to make a building plan for the TERMES system, we need to generate a directed graph on the vertex set L. To avoid the physical assembly constraint shown in Fig. 1C the graph needs to be acyclic and a location cannot have two opposing incoming edges. To ensure traversability, the graph must have the additional properties that for every $l \in L$ there is a directed, traversable path from l_{START} to reach l *and* for every $l \in L$ there is a directed, traversable path to reach l_{EXIT}. l_{START} has all outgoing edges; l_{EXIT} has all incoming edges.

In summary, the properties of a valid map are as follows:

Property 1: The map contains no cycles.

Property 2: The map contains no opposing incoming arrows.

Property 3: All locations can reach an exit on a traversable path that is consistent with the assigned edges.

Property 4: The start can reach all locations on a traversable path that is consistent with the assigned edges.

Properties 3 and 4 imply that, except for l_{START} and l_{EXIT} all locations must have directed edges that point both in- and outwards. We refer to this local check for Properties 3 and 4 as the sink/source-condition. We will reference these properties throughout the following sections.

4. Edge–CSP compiler

Past TERMES publications described a procedure for searching through the space of available assignments [3]. Here, we recast this compiler as a backtracking search to a CSP with pairwise, partial, and global constraint checking. The CSP problem consist of variables, domains (the possible values for each variable), and constraints (how variable assignments affect each other). The goal of backtracking search is to find an *assignment*, i.e. picking from each domain one value for each variable [24, Ch6].

In accordance with the compiler described in [3], we make variables correspond to edges between neighboring locations and give them a domain of the two possible edge directions. We refer to this compiler as an Edge-CSP compiler, further shown in Fig. 2A. The Edge-CSP tries to pick both a good variable ordering and a good domain ordering. The variable ordering is to pick variables that are adjacent to already assigned edges and as close to l_{START} as possible. The domains are ordered to first explore edges that point "away" from l_{START} in a breadth first manner. This choice is based on the observation that most edges in valid maps have this orientation.

We use three types of constraints. Binary constraints between edges that comply with Property 2. Constraints on partial assignments which check for cycles, i.e. Property 1, and checks that each location with fully assigned edges other than l_{START} and l_{EXIT} complies with the sink/source-condition. Constraints on the global assignment which checks Property 3–4, that every location can be reached from l_{START} and that l_{EXIT} can be reached from every location. The benefit of the binary checks is that constraints may be propagated forward to speed up the search using forward checking [24, Ch6]. We use the AC3 algorithm to do this [25]. Forward checking with the binary constraints enable a behavior equivalent to the "row rule" discussed in [3], i.e. a behavior that causes the structure to be built from one point outwards. An example of this is shown in Fig. 2A; if v_1 is fixed, v_2 and v_3 are as well. Reversely, the fixed value of v_{17} does not directly affect those around it.

Notice that this compiler does not take the height of the structure into consideration until the final global check. The search continues until all domain combinations have been tried, or have been eliminated early by a local or partial check. The total number of possible domain combinations scales as $O(2^n)$, where n corresponds to the number of edges between locations in the structure. However, early termination of partial assignments prunes the space significantly. In general, all backtracking search may work well on structures that have many feasible solutions, but will scale poorly with large structures that have only a few or no solutions, and where bad branches in the search tree cannot be pruned early.

Analyzing the compiler as a CSP shows that the binary constraints formulated on edges limits the amount of forward checking that can be done, since each row or column results in a disconnected component of constraint arcs. Furthermore, it is not possible to use the sink/source-condition to forward propagate because it cannot be expressed as a binary constraint. To address these shortcomings we formulate a more efficient CSP to solve the same problem in Section 5.

5. 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 as illustrated in Fig. 2B. Consequently we refer to this algorithm as a Location-CSP compiler. The benefit of this scheme is that it creates a fully connected graph, where constraints may more readily affect other variables. Note that like in the Edge-CSP, cycles and structure traversability is not checked until after partial or full assignment.

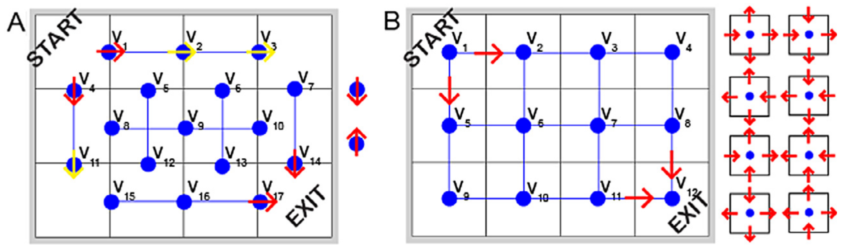


Fig. 2. Two versions of the CSP compiler applied to a 3 × 4 × 1 structure. (A) In the Edge-CSP variables correspond to edges between locations. The domain for v_6 are shown as an example to the right of the structure. We can forward propagate the fixed variables, v_1 and v_4 shown in red, to fix v_2, v_3, and v_{11} shown in yellow according to property 2. (B) In the Location-CSP variables correspond to all possible combinations of directions to and from the location. The domain for v_6 are shown as an example to the right of the structure. This scheme produces a fully connected graph in which all constraints affect each other.

6. BFD compiler

The final compiler is not based on search, but instead does an iterative assignment of the edge directions in a breadth-first manner starting from l_{EXIT}. Essentially, it evaluates if a location may serve as a drain (an exit-like location) for the intermediate structures where locations whose travel directions have been fully assigned were removed. We refer to this algorithm as a Breadth-First Disassembly (BFD) compiler. The process is shown in Fig. 3 and Alg. 1. Upon initialization, l_{EXIT} is added to the frontier list, $Q_{frontier}$. The compiler iteratively takes a location, l_0, from $Q_{frontier}$ and checks if it can serve as a drain. To serve as a drain, l_0 must have the following properties: (1) to comply with Property 2 it cannot be in between two unassigned locations, (2) it needs to have a traversable path to l_{EXIT} that only uses previously disassembled locations, and (3) it cannot cause a disconnect in the structure which would cause a violation of Property 4. If these statements are true l_0 is added to $Q_{visited}$, the edges to all neighbors are assigned as ingoing, and traversable neighbors are added to $Q_{frontier}$. The compiler continues to do this until $Q_{frontier}$ is empty or no solution is found.

The biggest overhead in the BFD compiler is the connectivity check which happens each time a location is tested as a viable drain. Note that the connectivity check takes the traversable height of the neighboring locations into account. We implement two versions of this check. (1) BFD$_0$: To check the connectivity, the compiler conducts a breadth-first search starting from l_{START} to count the number of reachable locations following unassigned edges. If this count is equal to the number of unvisited locations, l_0 may serve as a drain. This requires a complete check of all remaining locations ($L \setminus Q_{visited}$). (2) BFD: To speed up this process, we cache the connectivity computation by generating a spanning tree of unvisited locations. Removing leaves in the tree does not disconnect the graph, so the connectivity check can return an answer without having to traverse any nodes in the spanning tree. When the connectivity check is for a non-leaf node, we perform the original connectivity check. If l_0 does not disconnect the structure we add it to $Q_{visited}$ and recompute the spanning tree. To create a spanning tree that is likely to have leaf-nodes in $Q_{frontier}$, we add edges in breadth first manner beginning from l_{START} following traversable edges. In Section 7, we show that the second method speeds up the process significantly.

6.1. Proof of correctness

This proof refers to the Properties 1–4 of a valid map, described in Section 3 and Algorithm 1. The correctness proof is done by induction on the edges of visited locations for Properties 2–4. Property 1 follows from a gradient argument.

Algorithm 1 Pseudo code for the BFD Compiler which either returns a valid map, or identifies that no such map exists. l_0 denotes the current location in question and l_i its neighboring locations. $Q_{visited}$ is the set of visited locations which have been 'disassembled', i.e. fully determined; and $Q_{frontier}$ is the frontier, i.e. locations that have traversable paths to the exit and could potentially be disassembled next.

```
1:  initialize $Q_{frontier}$ and $Q_{visited}$ as empty
2:  initialize map to be an empty graph over the vertex set $L$
3:  add $L_{EXIT}$ to $Q_{frontier}$
4:  while $Q_{frontier}$ is not empty do
5:      remove $l_0$ from $Q_{frontier}$
6:      if $l_0$ is not in between two other unvisited sites (Property 2)
          and removing $l_0$ does not disconnect the structure (Properties 3-4) then
7:          Add $l_0$ to $Q_{visited}$
8:          for each unvisited neighboring site $l_i$ of $l_0$ do
9:              add edge ($l_i$ ,$l_0$) to map
10:             if $\exists$ traversable edge from $l_i$ to $l_v \in Q_{visited}$ then
11:                 add $l_i$ to $Q_{frontier}$
12: if $|Q_{visited}| = |L|$ then
13:     return map
14: else
15:     return False
```

Theorem 1 (*BFD–Compiler Correctness*). *When the BFD compiler completes successfully, it produces a valid map.*

Proof of Theorem 1.

Property 1: The edge assignment adds directions in such a way that the newly added directions point from unvisited locations into visited locations (Lines 7–9). By following such a direction (when it is traversable) a robot is brought one step closer to l_{exit}. Each location can be labeled with the steps left to l_{EXIT}. Since the paths in the map move down the label gradient, they cannot contain cycles as that would require a path where the label increases.

Properties 2-4: The induction hypothesis (IH) is that the edges of visited locations have Properties 2–4, as well as the two axillary properties: (Property 5) $\forall l_q \in Q_{frontier} \exists$ a traversable path to the exit in the assigned map; and (Property 6) $L \setminus Q_{visited}$ is traversably path connected, i.e. all unvisited locations have traversable paths from l_{START} that only move over other unvisited locations.

Base case: $Q_{frontier}$ has only l_{EXIT}. Properties 2–4 are true for the empty set, Property 5 is true because l_{EXIT} is path connected to

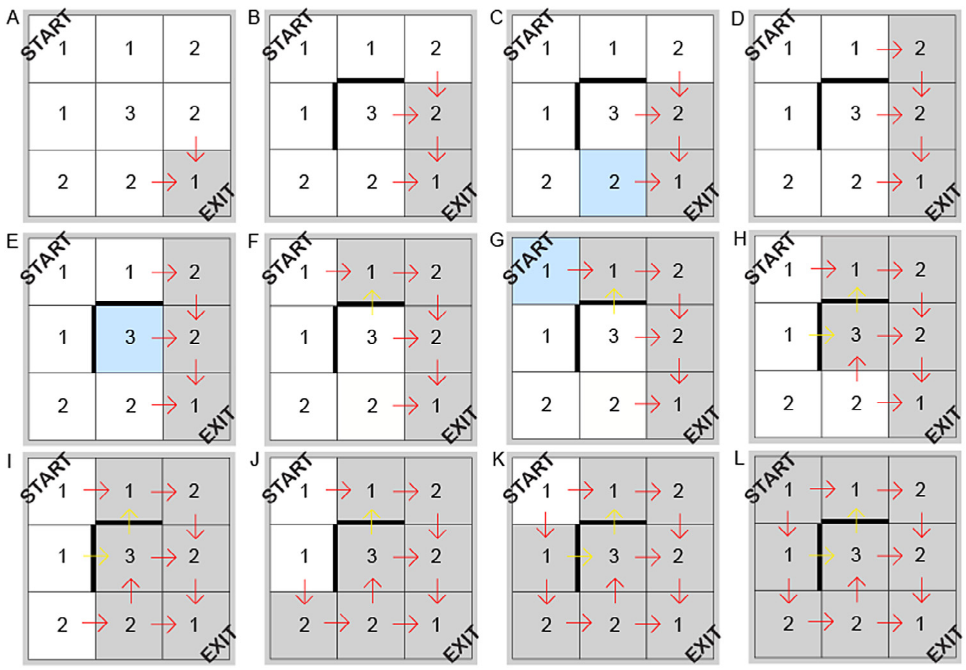


Fig. 3. BFD Compiler applied to a 3 × 3 structure. (A) Consider l_{START} to be (0,0) and l_{EXIT} to be (2,2); (B) the compiler removes (2,1); (C) (1,2) cannot be removed because this would cause a disconnected structure; (D) the compiler removes (2,0); (E) (1,1) cannot be removed because of Property 1. The compiler continues in the same manner until l_{START} has been removed at which point it returns a valid map. Notice that the yellow arrows do not count toward the traversability check, but are needed for the robot rule set.

itself, and Property 6 is correct because we assume that L is traversably connected.

Induction step: When adding another element l_0 to $Q_{visited}$, Property 2 is true because the new element would only have two opposing incoming directions if it had two unvisited neighbors. Property 3 is true, because when l_0 was added to $Q_{frontier}$ one of its edges was directed to a location in $Q_{visited}$ (Line 9) and by Property 5 in IH there is a directed path toward the exit. Property 4 is true because of Property 6 in IH, l_0 can be reached from l_{START} and l_i can be reached through l_0 after the new edge is added to the map (Line 9). Property 5 is true because of (Line 10–11) and Property 3 in IH. Property 6 is true because of the second condition in Line 6. □

Beyond proving that the compiler generates valid maps which work with the TERMES system, we also believe that the reverse is true; i.e. that the structure is unbuildable with the TERMES system if the compiler fails. The intuition for this is as follows. The compiler fails when $Q_{frontier}$ is empty and $|Q_{visited}| \neq |L|$. This happens when no more locations can be disassembled, either because they are not traversable from visited locations (Property 3) or because they are in between two other locations (Property 2). In other words, the structure formed by unvisited locations could not have been built because the last addition to the structure does not exist.

7. Comparison of compilers

We next evaluate how the runtime of the compilers scale with the number of locations for different types of structures (Fig. 4). These results are generated in a single process on a standard laptop (Intel(R) Core(TM) i7-4720HQ, CPU @ 2.60 GHz, quad core, 16G of RAM). Note that the compilers can handle a wide range of structure types, however, for the purposes of analysis, we focus only on square footprints in the following.

Fig. 4A shows the runtime of each compiler as the number of locations grow in a 1-height square structure. The Edge-CSP can compile such simple structures with 10,000 bricks in around 100 s; for scale, a standard U.S. family house contains around the same amount. As expected the Location-CSP does slightly better because the constraints propagate more readily. Notice that for small structures both BFD compilers compile about 10 times faster than the CSP compilers. The BFD$_0$ compiler converges to quadratic growth (slope 2 in log–log axis), and as the structure size approaches 100,000 locations the CSPs will start to outperform it. This happens because their domain-variable ordering is especially optimized for these simple square structures so that the first tried assignment during the search is usually correct. By adding the improved connectivity check, the BFD outperforms all other compilers (scaling almost linearly) and can easily compile structures with up to 1 million bricks (comparable to the number of bricks in the Great Pyramid of Giza according to egyptorigins.com). Similar results can be noted when we run the compilers on buildable structures with randomly generated height profiles up to 7 bricks tall (Fig. 4B).

Fig. 4C shows the runtime on unbuildable structures with randomly generated height profiles. The runtime of the Edge-CSP now varies significantly because the search only terminates early if it finds a locally checkable error. Such errors are more likely to be found with the Location-CSP compiler. The new BFD compilers show a similar scalability as before. Fig. 4D shows the runtime for unbuildable structures, also presented in Fig. 1D, which violate Property 2 with any consistent ordering. This structure is especially slow to search through, since ordering inconsistencies cannot be detected locally. Each internal raft has four connectors, and each of these may, from the raft's perspective, be either a sink or a source. If it is a sink it violates property 2, and as a result all possible source combinations are tried first. We halted compilations that exceeded 24 h of runtime, which is why both CSP compilers are only presented with a single data point. Notice again, how the BFD$_0$ compiler scale quadratically with the size of the structure, and the improved BFD compiler scales almost linearly.

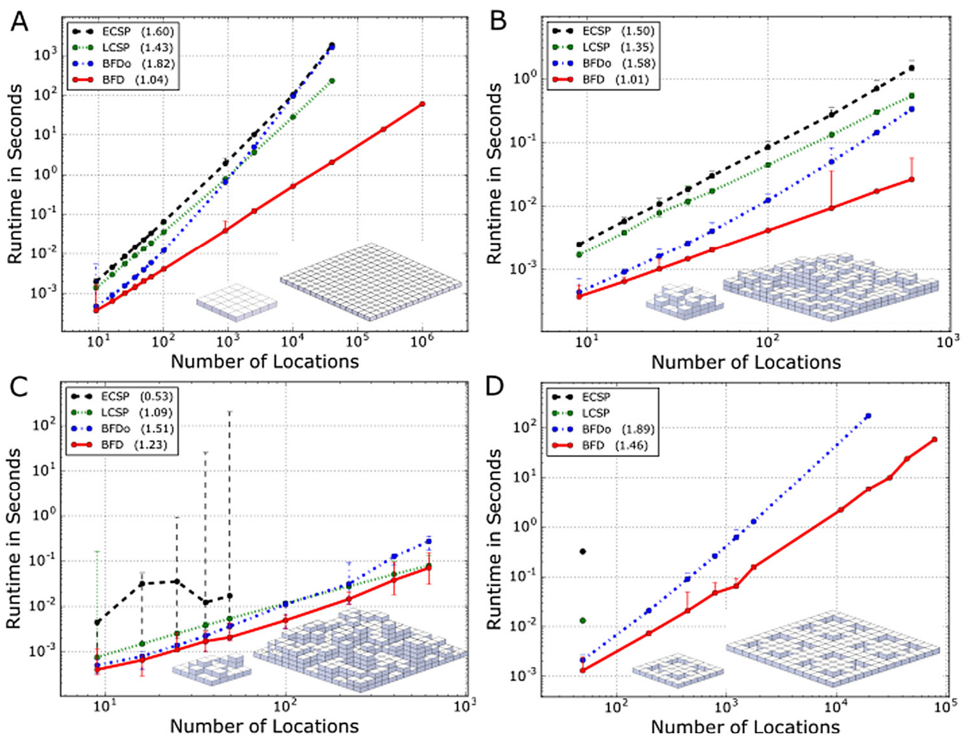


Fig. 4. Runtime of compilers versus the number of locations in different types of structures, including (A) square, buildable structures of height 1, (B) square, buildable structures of random height, (C) square, unbuildable structures of random height, and (D) unbuildable structures similar to that shown in Fig. 1D. Insets indicate how we scale the number of locations; marks annotate mean of 10 runs (in the case of random height structures, 10 different structures of the same number of locations were tested); error bars indicate maximum and minimum runtime; and the number in the parenthesis gives the slope of the best fit line for all data in the curve.

8. Transition probabilities

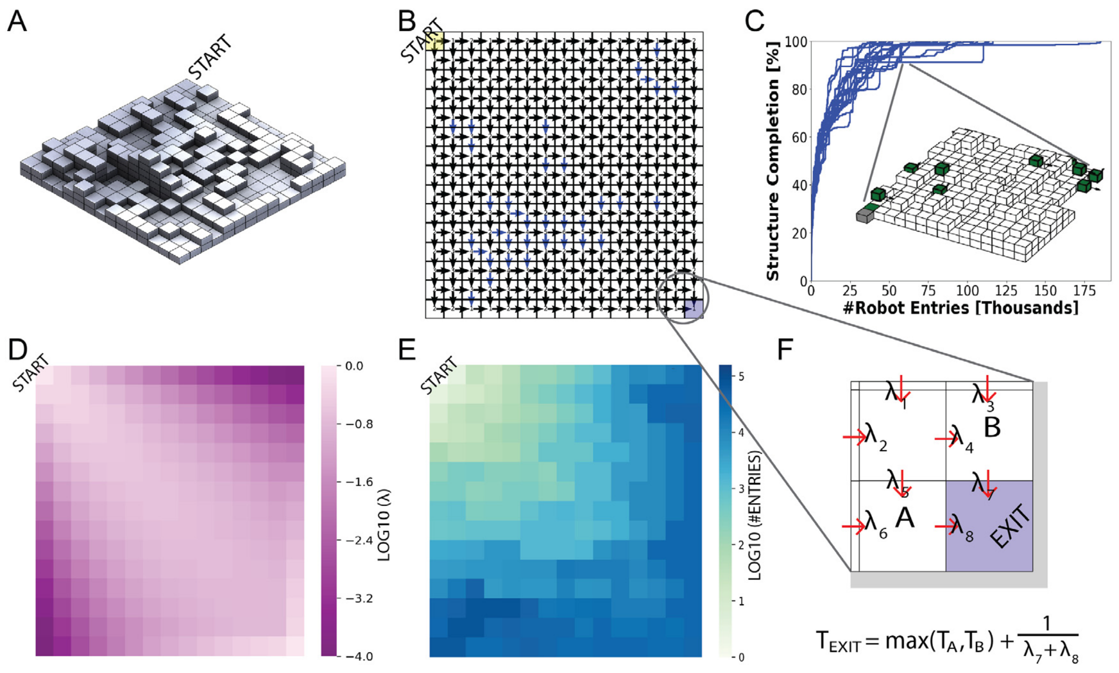
During the actual assembly of the structure, individual robots have no knowledge of the system assembly state and can therefore not navigate directly toward the construction frontier. Instead they move along the directed paths in the map at random, looking for open assembly locations. Once an open location is encountered, the internal rule set on the robot, based on restrictions shown in Fig. 1C–D, determines whether or not material can be added. To explain this rule set and how the combination of the map and rule set affect the construction progress, we first introduce several terms related to a location, l_i: (1) neighboring locations that lead to l_i are *parents* of l_i; (2) neighboring locations that lead from l_i are *children* of l_i; (3) the *visit rate* of l_i is the probability per unit time that a robot travels through it; and (4) the *assembly time* of l_i is average time it takes for the robots to assemble l_i.

The TERMES rule set is discussed in detail in previous papers [3,5]. To give an intuitive overview, the rules restrict robots from adding material to location l_i with height h_i, until (generally speaking) all parents and children are of similar height, if such specified by the structure blueprint. Parents of similar height ensures that robots never have to add a brick in between two others (Fig. 1C). Children of similar height ensures traversable paths (Fig. 1D). Because valid assembly steps are dependent on what bricks have already been placed, this leads to wasted trips; i.e. cases where the robot exits the structure before being able to deposit the brick it is carrying. The combination of the rule set and map, generally speaking, makes tall structures grow in forward propagating staircases starting from l_{START}.

Take as an example the structure shown in Fig. 5A, which has 406 bricks. To adhere with the map (Fig. 5B) and rule set, this structure must grow from the upper- and left-most edge toward the exit. The construction progress is plotted in blue for 10 simulated runs in Fig. 5C, where robots choose naively between children with equal probability. The visit rate is shown in Fig. 5D. Note that this plot is based purely on the directed travel paths, and does not take the structure height into consideration. With such uniform transition probability, robots are unlikely to stay by the upper- or left-most edge of the structure, and are therefore unlikely to assemble locations that must be in place before downstream locations can be filled in. The overflow of robots in the center is also likely to cause bottlenecks, which further slows down the assembly progress. In Fig. 5E, the plot of location assembly times clearly shows how the locations nearer the bottom- and right-most edge will require an excess of robots to file through the structure before they are completed. Analogous, the big vertical jumps in the traces in Fig. 5C indicate times at which a robot fills in a perimeter location which is holding everything else up. Worst case, structure completion may be exceedingly slow — one run requires almost 200,000 robot entries before placing the last brick in the 406-brick structure. In the following text we reason about how transition probabilities between locations affect the construction process, and explore the potential for optimization.

8.1. Transition model

First, we model traversal and assembly as a Poisson splitting process, i.e. robots visit a location with a *rate* λ, and if the location has two children with a probability of β and $1-\beta$, the robot visits the two subsequent locations with rates $\beta\lambda$ and $(1-\beta)\lambda$ respectively. If a location has two parents the rates add up. Our experiments show that the completion time of the structure is strongly dependent on locations that have small visit rates. In Fig. 6 we analyze a simple $3 \times 3 \times 1$ structure and the distribution in assembly time as a function of the single splitting parameter, β. The assembly times are long if the splitting



$$T_{EXIT} = \max(T_A, T_B) + \frac{1}{\lambda_7 + \lambda_8}$$

Fig. 5. (A–B) 15 × 15 random height structure and its traversal map. (C) Construction progress as a function of robot entries for 20 simulated runs using maps with uniform transition probabilities. The inset shows a snapshot from the simulation, robots are shown in green. (D) Visit rates, λ, for each location in the structure, based on the map shown in B. (E) Mean assembly time for each location in the structure, based on the 20 simulated runs. (F) Sketch explaining how the exit location completion time, T_{EXIT}, depends on the completion time of the parent locations, T_A and T_B, and their transition probability, λ_7 and λ_8. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

parameter starves either the corner locations or the center of visit rates. The shape is asymmetric since the center receives rates from two parents, while starving either corner (small βs) affects the overall assembly time more dramatically.

Our goal is to choose splitting probabilities that minimize the expected assembly time of the last assembly location, T_{EXIT}. Since each location can only be assembled after its parent locations have been assembled, completion time, T, for each location can be written as the maximum of the parent assembly times plus the additional assembly time due to the limited rate of visiting the location in question (Fig. 5F). While a closed form expression for the assembly time due to rates is a simple exponential, closed form solutions for the maximum of two random variables requires integrating over their joint probabilities, for which we were unable to find an easy expression. Fig. 7 shows a sampled probability density function (PDF) of the assembly time for each location in a 5 × 5 × 1 structure. The data shows that the PDFs are heavy-tailed and that the PDF for two parent assembly times are not independent since they depend on a common ancestor, i.e. location (3,4) and (4,3) are not independently distributed, since they will both have a long tail if any of the location in the square (0,0) to (3,3) happened to have a long tail. Therefore, instead of trying to compute the actual assembly time to optimize the transition probabilities, we focus on finding visit rates λ_i for each location that produces small assembly times. We discuss this approach in the following subsection.

8.2. Optimization of transition probabilities

To formulate the optimization problem we assume that robots arrive at l_{START} with a rate of $\lambda_{START} = 1$. This means the visit rate for all other locations is between 0 and 1. We tested two different objective functions, one aiming for equally distributed visit rates ('equal-visit-rate') and one aiming to avoid visit rates below a certain threshold ('minimum-visit-rate'). We demonstrate our approach on the representative example structure shown in Fig. 5A–B. The rate of visiting location l_i, λ_i, with parent locations l_j and visit rates λ_j, is calculated as:

$$\lambda_i = \sum_{j=1}^{J} \lambda_j P_{ji} \tag{1}$$

where P_{ji} denotes the probability of choosing location l_i from l_j and J is the total number of parent locations. The choices for P_{ji} have the additional constraint that $\sum_I P_{ji} = 1$ and $P_{ji} \in [0, 1]$. We formulate the optimization problem by defining the visit rate for l_i, λ_i, and the transition probabilities, P_{ji}, as variables, and by expressing Eq. (1) and conditions on P_{ij} as quality constraints and bounds on the variables.

In Fig. 5D–E, we showed visit rates and assembly times for a map with uniform transition probability, which causes excess visits to the center of the structure leading to bottlenecks and wasted trips. This observation leads us to explore an equal-visit-rate objective. We partition the locations based on the number of steps it takes to reach l_{EXIT} (the distance along traversable paths in the map) and minimize the cost by:

$$Cost_{eq} = \sum_{dist} \sum_{I_{dist}} (\lambda_i - \mu_{dist})^2 \tag{2}$$

where I_{dist} is the set of locations that are equidistant from l_{EXIT}, and μ_{dist} is the mean visit rate of these. Using the previously formulated constraints and this objective we optimize over the transition probabilities P_{ij} and λ_i using sequential quadratic programming (SQP).[1] We test two maps: (i) one in which we include all edges in the map (Fig. 5B), and (ii) one in which we only include edges in the map which are permanently traversible (i.e. without the blue arrows in Fig. 5B). The second map is based on the intuition that paths which can be obstructed by locations that eventually reach a height difference above 1 restricts robot traversals later in the construction process.

[1] SciPy implementation of `optimize.minimize` with the `SLSQP` option.

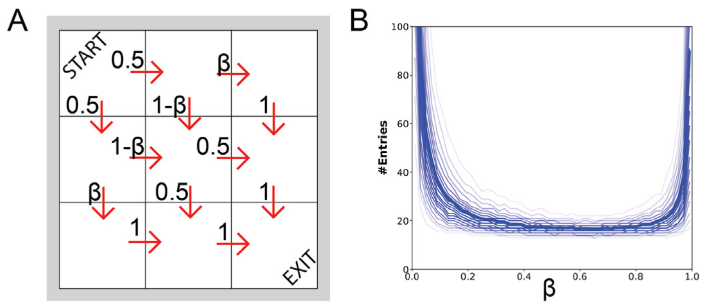


Fig. 6. The choice of transition probability may heavily affect completion time. (A) Example of transition probabilities in a 3 × 3 × 1 structure. (B) Effect of β on structure completion time over 10,000 simulated runs, expressed in robot entries to the structure.

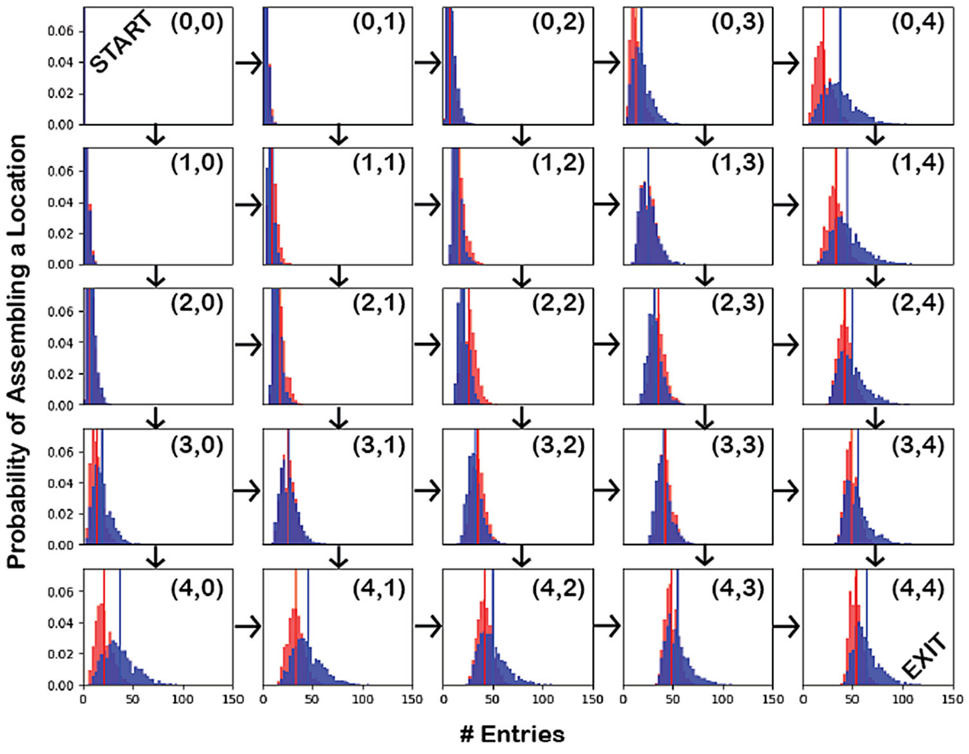


Fig. 7. Probability density function of assembly time for each location in a 5 × 5 × 1 structure, measured in robot entries and generated through 5000 simulated runs. Blue bars show the PDF for a uniform transition probability map; red bars for a transition probability map which was optimized according to a minimum-visit-rate. The vertical lines show their respective average. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The results are shown in Fig. 8. Optimizing transition probabilities according to equal visit rate makes the system perform significantly better than it does with a uniform transition probability map. The maximum completion time out of 20 simulated runs was ∼15,000 robot entries when all edges were taken into consideration, and ∼20,000 robot entries when only permanent edges were included in the optimization (Fig. 8A). The latter generally performs worse than the former (Fig. 8B–C). By analyzing the plot of assembly times, we suspect this decrease in performance occurs because individual low rate assignments have a disproportionate effect on the overall assembly time, and by enforcing equal visit rate for all locations in I_{dist}, we give equal weight to variations of λ_i that have minimal effect on the overall assembly time. For example, the visit rates for a good assignment is shown in Fig. 8C. In it, the top left corner locations have vastly different rates within an I_{dist} set, but still enable critical locations in the middle to have roughly equal visit rates.

In order to optimize assembly time while taking the structure height into consideration, we instead propose the following minimum-visiting-rate constraint:

$$Cost_{min} = \sum_{I} e^{\alpha(m-\lambda_i)} \tag{3}$$

where m is a minimum visit rate threshold defined for the entire structure and α is scaling factor of how aggressively this minimum value is enforced during optimization. When the visit rate is bigger than the minimum it adds little to the overall cost, but locations that have smaller visit rates are heavily penalized. For a given structure, we computed m to be the smallest visit rate obtained during the equal-visit-rate optimization when all edges are present, i.e. the smallest value that should be achievable by every location under ideal conditions.

The results are shown in Fig. 9. The minimum-visit-rate constraint is able to achieve significantly better performance than the equal-visit-rate constraint. By taking location heights and non-traversable edges into account, it allows other locations to have unequal visit rates in order to feed locations that are below the minimum visit rate m. This objective achieves equal visit rates for

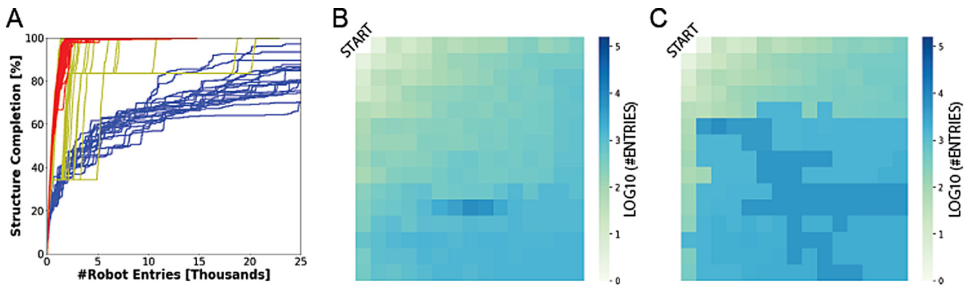


Fig. 8. Optimization of transition probabilities based on uniform visit rates of locations that are equidistant from l_{EXIT} on all (i) and permanent (ii) edges in the map. These results are based on the 15 × 15 random height structure shown in Fig. 5A–B. (A) System performance with maps of uniform transition probability (blue), and transition probabilities optimized according to (i) (red) and (ii) (yellow). (B–C) Location assembly time as an average of 20 simulated runs, for (i) and (ii), respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

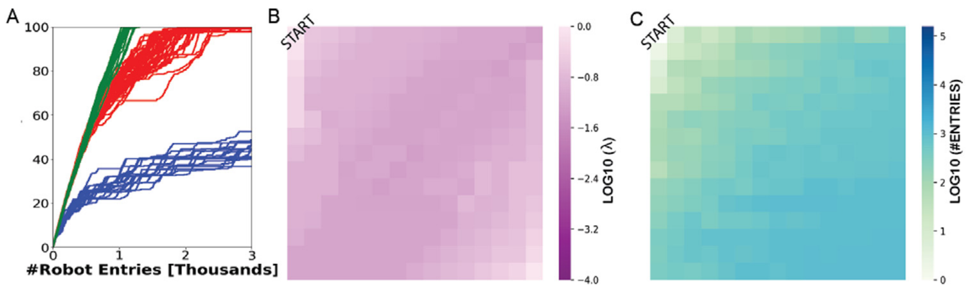


Fig. 9. (A) Optimization of transition probabilities based on a minimum-visit-rate constraint (green), as compared to an equal-visit-rate constraint (red), and uniform probabilities (blue). (B–C) Visit rate and assembly time per location respectively, based on an average of 20 simulated runs for the minimum-visit-rate constraint optimization. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the widest part of the structure and penalizes any other locations with a visit rate less than m. With this new optimization, the 406-brick structure is finished with an average of 1200 robot entries. In other words, every third robot entering the structure is able to deposit a brick. Notice also, that the worst case completion time is much less than for any of the other optimization schemes.

The tradeoff of assembly times between different locations can be seen in Fig. 7. While the overall assembly time for location (5,5) of the optimized transition probabilities decreases, optimization does increase the assembly time of the center location (2,2). Basically, the optimized probabilities re-allocate visit rate from the center to low rate locations in the corners. Removing these low rate locations decreases the mean assembly time, and also makes the assembly times more tightly clustered, reducing long outliers.

On a final note, it is clear that the idea of exploiting parallelism in construction schemes that have a single point of entry heavily depends on the optimization of transition probabilities. We can compare our approach with one that produces a single path through the structure such that the structure grows in a sequential manner and that every robot that enters can deposit a brick, in terms of the parallelism offered. With the minimum-visit-rate constraint optimization, you can achieve an increase in performance over this single-path approach if you can place more than just three robots with bricks at the construction frontier at any one point in time, which is true for most large scale structures. Another benefit of multi-path structures is that robots can take shortcuts to the frontier as opposed to having to travel over every location first.

9. Conclusion and future work

In summary we have presented work to address the scalability of the TERMES compiler, and demonstrated a BFD compiler which scales better than quadratic with the number of locations in the structure independent of whether or not that structure is buildable. We demonstrated this on structures with up to 1 million bricks, which were compiled on commodity hardware in minutes. We have further shown an approach by which the transition probabilities between locations in the generated map can be improved for markedly faster construction speed without added hardware complexity. Future work will involve development of metrics by which to evaluate the compiled maps, especially in terms of the parallelism they offer, and compilers which can suggest modifications to make unbuildable structures buildable.

Acknowledgments

This work was supported by the GETTYLAB, and partially supported by the National Science Foundation grant #1846340.

References

[1] K. Petersen, R. Nagpal, Complex design by simple robots, Archit. Des. (2017) 44–49.
[2] K.H. Petersen, N. Napp, R. Stuart-Smith, D. Rus, M. Kovac, A review of collective robotic construction, Sci. Robot. 4 (2019).
[3] J. Werfel, K. Petersen, R. Nagpal, Designing collective behavior in a termite-inspired robot construction team., Science 343 (2014) 754–758.
[4] K. Petersen, R. Nagpal, J. Werfel, TERMES: An autonomous robotic system for three-dimensional collective construction, in: Robotics: Science and Systems Conference VII, 2011.
[5] J. Werfel, K. Petersen, R. Nagpal, Distributed multi-robot algorithms for the TERMES 3D collective construction system, in: Modular Robotics Workshop, IEEE Intl. Conference on Robots and Systems (IROS), 2011.
[6] Y. Hua, Y. Deng, K. Petersen, Robots building bridges, not walls, in: IEEE International Workshops on Foundations and Applications of Self* Systems, 2018.
[7] Y. Deng, Y. Hua, N. Napp, K. Petersen, Scalable compiler for the TERMES distributed assembly system, in: Distributed Autonomous Robotic Systems, Springer, 2019, pp. 125–138.
[8] M.S.D. Silva, V. Thangavelu, W. Gosrich, N. Napp, Autonomous adaptive modification of unstructured environments, Robot.: Sci. Syst. (2018).

[9] N. Napp, R. Nagpal, Robotic construction of arbitrary shapes with amorphous materials, in: Proceedings - IEEE International Conference on Robotics and Automation, 2014, pp. 438–444.

[10] T. Soleymani, V. Trianni, M. Bonani, F. Mondada, M. Dorigo, Autonomous construction with compliant building material, in: Intelligent Autonomous Systems, Vol. 13, Springer, 2016, pp. 1371–1388.

[11] V. Lindsey, Q. Mellinger, D. Kumar, Construction of cubic structures with quadrotor teams, in: Robotics: Science and Systems VII, 2011.

[12] C. Jones, M.J. Mataric, Toward a multi-robot coordination formalism, Technical Report, DTIC Document, 2004.

[13] M. Rubenstein, A. Cornejo, R. Nagpal, Programmable self-assembly in a thousand-robot swarm, Science 345 (2014) 795–799.

[14] J. Seo, M. Yim, V. Kumar, Assembly planning for planar structures of a brick wall pattern with rectangular modular robots, in: 2013 IEEE International Conference on Automation Science and Engineering (CASE), 2013, pp. 1016–1021, http://dx.doi.org/10.1109/CoASE.2013.6653996.

[15] C. Coulais, E. Teomy, K. d. Reus, Y. Shokef, M. van Hecke, Combinatorial design of textured mechanical metamaterials, Nature 535 (2016) 529.

[16] T. Tucci, B. Piranda, J. Bourgeois, A distributed self-assembly planning algorithm for modular robot, in: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, International Foundation for Autonomous Agents and Multiagent Systems, 2018, pp. 550–558.

[17] T.S. Kumar, S.J. Jung, S. Koenig, A tree-based algorithm for construction robots, in: ICAPS, 2014.

[18] V. Lindsey, Q. Mellinger, D. Kumar, Construction with quadrotor teams, Auton. Robots 33 (2012) 323–336.

[19] Loïc Matthey, Spring Berman, Vijay Kumar, Stochastic strategies for a swarm robotic assembly system, in: 2009 IEEE International Conference on Robotics and Automation, IEEE, 2009, pp. 1953–1958.

[20] Nils Napp, Eric Klavins, Load balancing for multi-robot construction, in: 2011 IEEE International Conference on Robotics and Automation, IEEE, 2011.

[21] Mac Schwager, Jean-Jacques Slotine, Daniela Rus, Decentralized, adaptive control for coverage with networked robots, in: Proceedings 2007 IEEE International Conference on Robotics and Automation, IEEE, 2007, pp. 3289–3294.

[22] Marco Pavone, Emilio Frazzoli, Francesco Bullo, Distributed policies for equitable partitioning: theory and applications, in: 2008 47th IEEE Conference on Decision and Control, IEEE, 2008, pp. 4191–4197.

[23] David Stein, T Ryan Schoen, Daniela Rus, Constraint-aware coordinated construction of generic structures, in: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2011, pp. 4803–4810.

[24] S.J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Pearson Education Limited, Malaysia, 2016.

[25] A.K. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1977) 99–118.

Yawen Deng completed her M.Sc. in Mechanical Engineering at Cornell University in 2018, with a focus on planning and control for autonomous construction robots. Before Cornell, she was a visiting fellow with Harvard University for 7 months and at UC Davis for 2 months. She graduated from Chu Kochen Honors College, Zhejiang University, China in June 2016. Her interests cover Computer Science, Energy and Environmental Engineering, and Operation Research, with a skill set across machine learning, Bayesian statistics, databases, algorithms, and website design.

Email: dengyawen159@gmail.com, website: http://yawendeng.com/

Yiwen Hua recently joined Leia Inc. as a Computer Visions Engineer. Before this, he completed his M.Sc. in Mechanical Engineering at Cornell University, 2018, with a focus on design of robotic mechanisms for autonomous construction, and a B.Sc. in Mechanical Engineering at Penn State, 2016. His interests cover robotics, systems control, autonomous driving, product design and development, as well as investments and consumer electronics.

email: owen.hua930531@hotmail.com

Nils Napp is an Assistant Professor of Computer Science and Engineering at the University at Buffalo. He completed a Ph.D. at the University of Washington in 2011, and a postdoc at Harvard's Wyss Institute in 2014. His research focuses on algorithms at the intersection of collaborative behavior and physical embodiment. When taking inspiration from natural systems, the central question is why they work well, and if those insights can be applied to engineered systems. Since machines often outperform biology in both communication and computation, the results typically look different from either their biological inspiration or traditional robots.

email: nnapp@buffalo.edu, website: https://cse.buffalo.edu/~nnapp

Kirstin Petersen is an Assistant Professor in Electrical and Computer Engineering at Cornell University. She completed a Ph.D. at Harvard's Wyss Institute in 2014, and a postdoc with the Max Planck Institute for Intelligent Systems in 2016. Her research involves design and coordination of large robot collectives able to achieve complex behaviors beyond the reach of single robot systems, and corresponding studies of how social insects to so in nature. Major topics include swarm intelligence, embodied intelligence, autonomous construction, human-swarm interaction, soft robots, entomology, and bio-cyber physical systems.

email: kirstin@cornell.edu, website: http://cei.ece.cornell.edu