Scalable Compiler for the TERMES Distributed Assembly System

Yawen Deng[1], Yiwen Hua[1], Nils Napp[2], and Kirstin Petersen[1]

[1] Cornell University, Ithaca, NY 14850, USA,
kirstin@cornell.edu,
[2] University at Buffalo, Buffalo, NY 14260, USA,

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 recast the process as a constraint satisfaction problem 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 transition probability between locations in the structure may be altered to improve system 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 scalability of the algorithmic framework. We present our results in the context of the TERMES system presented in previous literature [3,4,5,6], but the 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. 1.A). 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

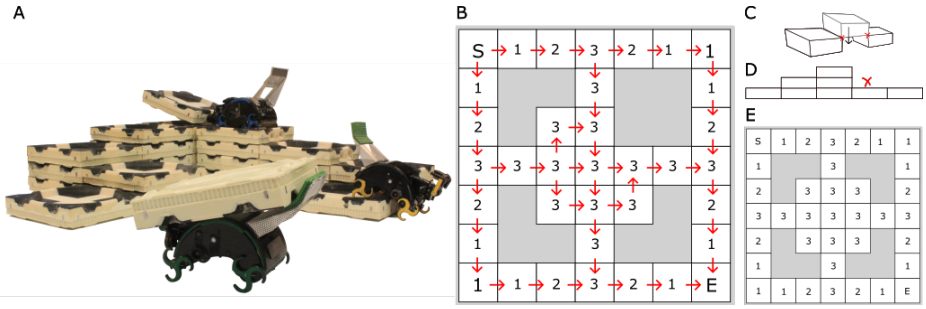

Fig. 1. A) Photo of the TERMES system. B) Example of the map generated by the compiler (top view). 'S' and 'E' denote start and exit locations respectively; digits the number of bricks at each location; arrows how robots can transition between locations. System limitations include that bricks cannot 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.

at each location, and designated travel directions between locations (Fig. 1.B). 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. 1.C-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.

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 minimalistic robot costing $\sim$\$2K with a 1-week assembly time. The cost of the mechanics was brought down considerably in a a subsequent paper [6]. The focus was not on perfect behavior, but rather to enable robots to recognize and fix errors before they propagated. 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 point 4, improving the TERMES compiler to make it feasible to compile maps of large-scale structures. We also briefly discuss point 5, i.e. how the transition probabilities between locations in the map can be optimized for faster progress.

We recast the compiler originally described in [3] (Sec. 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 (Sec. 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 (Sec. 5). Finally, we 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 (Sec. 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 (Sec. 7), including unbuildable ones which are computationally intractable for search-based compilers. We also show a method by which construction speed may be improved (Sec. 8).

2 Related Work

Decentralized robotic construction can be achieved in a variety of ways. Examples include pre-programmed robots for dedicated structures [7,8,9], template-based construction [10], centralized controllers [11] that allow for parallelism, 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. 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 [17,5].

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. Figure 1.B 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.

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. 1.C 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

The original compiler paper describes a procedure for searching through the space of available assignments [3]. 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 [19, 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

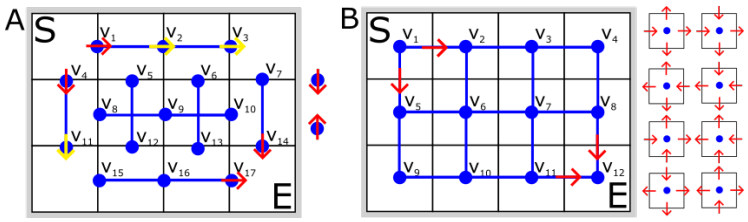

Fig. 2. Two versions of the CSP compiler applied to a 3 × 4 structure. A) In the Edge-CSP variables correspond to edges between locations. The domain for v_6 are shown as an example to the right of the structure. We can forward propagate the fixed variables, v_1 and v_4 shown in red, to fix v_2, v_3, 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.

Fig. 2.A. 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 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 [19, Ch6]. We use the AC3 algorithm to do this [20]. 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. 2.A; 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 Sec. 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. 2.B. 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.

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 an 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) it cannot be in between two unassigned locations (Property 2), 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 this process up, 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 Sec. 7, we show that the second method speeds up the process significantly.

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, 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 ∃ traversable edge from l_i to l_v ∈ Q_visited then
11:                 add l_i to Q_frontier
12: if |Q_visited| = |L| then
13:     return map
14: else
15:     return False
```

6.1 Proof of correctness

This proof refers to the Properties 1-4 of a valid map, described in Sec. 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.

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

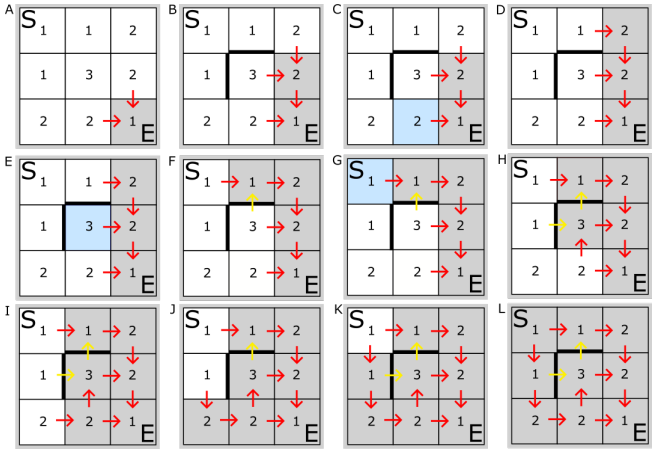

Fig. 3. BFD Compiler applied to a 3 × 3 structure. A) Consider l_{start} to be (0,0) and l_{exit} to be (2,2); B) the compiler removes (2,1); C) (1,2) cannot be removed because this would cause a disconnected structure; D) the compiler removes (2,0); E) (1,1) cannot be removed because of Property 1. 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 towards the traversability check, but are needed for the robot rule set.

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 compilers generate 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 @ 2.60GHz, quad core, 16G of RAM).

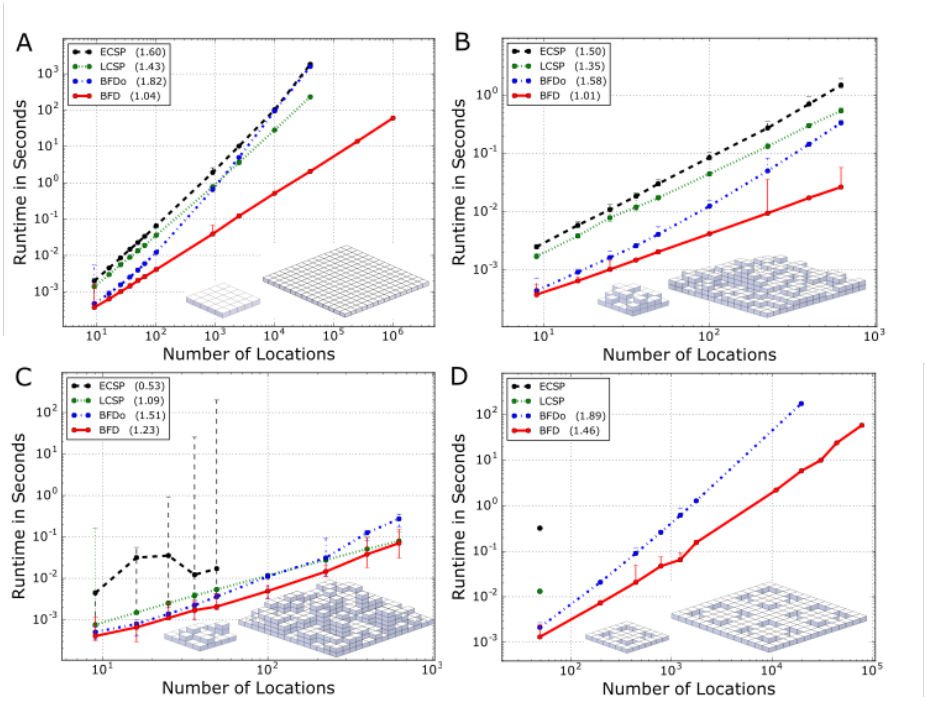

Fig. 4. Runtime of compilers versus the number of locations in different types of structures, including A) square, buildable structures of height 1, B) square, buildable structures of random height, C) square, unbuildable structures of random height, and D) unbuildable structures similar to that shown in Fig. 1.D. 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.

Fig. 4.A 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, 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. 4.B).

Fig. 4.C 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. 4.D shows the runtime for unbuildable structures, also presented in Fig. 1.D, 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 hours 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.

8 Improving Transition Probabilities

During construction individual robots do not know the assembly state and must guess which path to take to find an assembly site. In the *maps* presented in [3], a robot in l_i has a uniform probability of choosing any of the traversable neighboring locations. This approach can lead to wasted trips where a robot exits the structure without finding a place to assemble the brick it is carrying. This is especially true for dense structures where a few key bricks must be inserted before the rest can be added. A representative example of such a structure is the random height 15×15 structure shown in the inset in Fig. 4.B. The *map* generated for this structure is shown in Fig. 5.A. The probability, P_i, of finding a robot in a location l_i, with parent locations, l_j and probabilities P_j, is calculated as:

$$P_i = \sum_{j=1}^{J} P_j P_{ji} \tag{1}$$

where P_{ji} denotes the transition probability from l_j to l_i, and J the total number of parent locations. Figure 5.B shows an example of how uniform transition probabilities cause robots to visit in the center of the structure during most trips, since there are many combinations of choices that reach those locations. This results in both wasted trips, where no bricks can be added, and bottlenecks where all robots try to file through the same area.

By using the 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 l_{exit}. For locations with the same distance, we need to minimize the difference of P_i. We use Sequential Least Square Programming (SLSQP) minimization to find improved transition probabilities P_{ji}. The constraints correspond to 1) $P_{start} = 1$, 2) $P_{min} > 0$, and 3) outgoing edge directions from a location must sum to 1. Fig. 5.C shows how improved transition values change the distribution of robots over the locations in the structures. To test the difference in construction speed, we simulated a 5-robot team and counted the total number of steps until structure completion. With uniform transition probabilities, the robots complete the structure in 2,216,370 steps (we ran this once). With improved transition probabilities, the structure

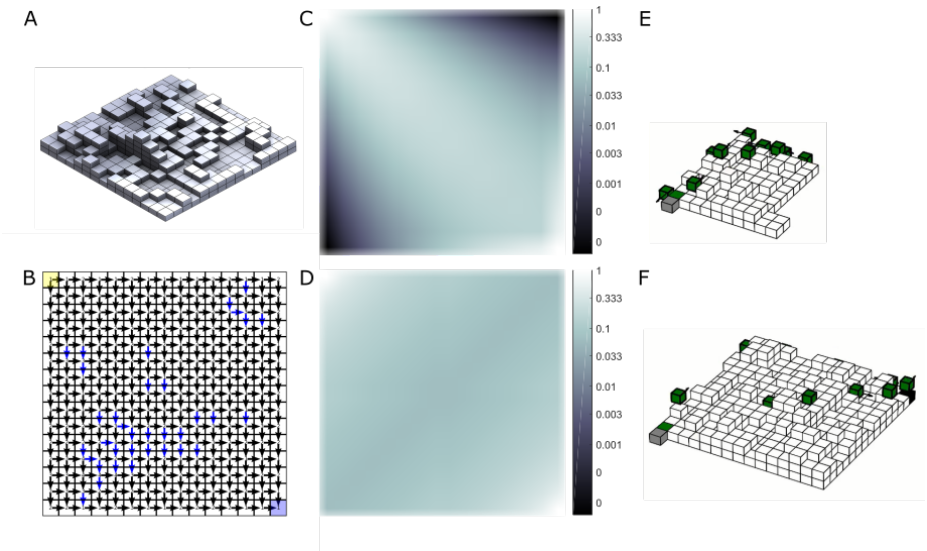

Fig. 5. Improving the transition probabilities between locations in the map helps speed up construction. A) Example 15 × 15 location structure with a random height profile and a total of 406 bricks. This structure is also shown in the inset in Fig. 4.B. B) Structure map. The start and exit location is colored in yellow and blue respectively. In this map all edges are directed towards the exit (right or down respectively). Non-traversable edges in the final structure are colored in blue. C-D) The probability of a robot to traverse over a location when all children from every node have uniform transition probabilities, and when the probabilities are improved according to the distance to the exit. E-F) Snapshots from the simulation showing how improved transition probabilities speed up construction significantly (bricks are shown in white; robots in green).

is completed in an average of 68,877 (averaged over 10 trials), indicating a speed-up of more than 30 times. Note, that this increases is structure dependent, and that the approach presented here only considers locations not then number of remaining bricks. In the future these *maps* may be further optimized by considering the bulk of bricks that needs to be placed in different locations.

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 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. The transition probabilities may further be optimized by taking the height of the structure into consideration.

Acknowledgements
This work was supported by the Getty Labs.

References

1. K. Petersen and R. Nagpal, "Complex Design by Simple Robots," *Architectural Design*, pp. 44–49, 2017.
2. M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics: a review from the swarm engineering perspective," *Swarm Intelligence*, vol. 7, no. 1, pp. 1–41, 2013.
3. J. Werfel, K. Petersen, and R. Nagpal, "Designing collective behavior in a termite-inspired robot construction team." *Science*, vol. 343, no. 6172, pp. 754–8, 2014.
4. K. Petersen, R. Nagpal, and J. Werfel, "TERMES: An autonomous robotic system for three-dimensional collective construction," *Robotics: Science and Systems Conference VII*, 2011.
5. J. Werfel, K. Petersen, and R. Nagpal, "Distributed multi-robot algorithms for the TERMES 3D collective construction system," in *In Modular Robotics Workshop, IEEE Intl. Conference on Robots and Systems (IROS).*, 2011.
6. Y. Hua, Y. Deng, and K. Petersen, "Robots Building Bridges, Not Walls," in *IEEE International Workshops on Foundations and Applications of Self* Systems*, 2018.
7. M. S. D. Silva, V. Thangavelu, W. Gosrich, and N. Napp, "Autonomous Adaptive Modification of Unstructured Environments," *Robotics: Science and Systems*, 2018.
8. N. Napp and R. Nagpal, "Robotic construction of arbitrary shapes with amorphous materials," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 438–444, 2014.
9. K. Stoy and R. Nagpal, "Self-reconfiguration using directed growth," *Distributed autonomous robotic systems 6*, pp. 3–12, 2007.
10. T. Soleymani, V. Trianni, M. Bonani, F. Mondada, and M. Dorigo, "Autonomous construction with compliant building material," in *Intelligent Autonomous Systems 13*. Springer, 2016, pp. 1371–1388.
11. V. Lindsey, Q., Mellinger, D., Kumar, "Construction of Cubic Structures with Quadrotor Teams," *Robotics: Science and Systems VII*, 2011.
12. C. Jones and M. J. Mataric, "Toward a multi-robot coordination formalism," DTIC Document, Tech. Rep., 2004.
13. M. Rubenstein, A. Cornejo, and R. Nagpal, "Programmable self-assembly in a thousand-robot swarm," *Science*, vol. 345, no. 6198, pp. 795–799, 2014.
14. J. Seo, M. Yim, and 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)*, Aug 2013, pp. 1016–1021.
15. C. Coulais, E. Teomy, K. de Reus, Y. Shokef, and M. van Hecke, "Combinatorial design of textured mechanical metamaterials," *Nature*, vol. 535, no. 7613, p. 529, 2016.
16. T. Tucci, B. Piranda, and J. Bourgeois, "A Distributed Self-Assembly Planning Algorithm for Modular Robots," 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, and S. Koenig, "A tree-based algorithm for construction robots." in *ICAPS*, 2014.
18. V. Lindsey, Q., Mellinger, D., Kumar, "Construction with quadrotor teams," *Autonomous Robots*, vol. 33, no. 3, pp. 323–336, 2012.
19. S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.
20. A. K. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, no. 1, pp. 99 – 118, 1977.